BABEŞ-BOLYAI UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Agent Based Pattern Recognition

PhD Thesis

PhD student: Radu D. Găceanu
Scientific supervisor UBB: Prof. Dr. Horia F. Pop
Scientific supervisor ELTE: Assoc. Prof. Dr. Habil. László Kozma

2012

Acknowledgments

I would like to express my gratitude to my supervisors, Prof. Dr. Horia F. Pop and Assoc. Prof. Dr. Habil. László Kozma for their guidance, suggestions and support. I would also like to thank to all my collaborators.

The author would like to thank for the financial support provided from programs co-financed by The Sectorial Operational Programme Human Resources Development, Contract POSDRU 6/1.5/S/3 "Doctoral studies: through science towards society".

List of publications

[CDG07] C. Chira, D. Dumitrescu, and **R. D. Găceanu**. Stigmergic agent systems for solving NP-hard problems. *Studia Informatica*, Special Issue KEPT-2007: Knowledge Engineering: Principles and Techniques (June 2007):177–184, June 2007. **(indexed MathSciNet, Zentralblatt MATH, EBSCO Publishing).**

[GP10] **R. D. Găceanu** and H. F. Pop. An adaptive fuzzy agent clustering algorithm for search engines. In *MACS2010: Proceedings of the 8th Joint Conference on Mathematics and Computer Science*, pages 185–196. Komarno, Slovakia, 2010. **(indexed MathematicalReviews).**

[GP11a] **R. D. Găceanu** and H. F. Pop. A context-aware ASM-based clustering algorithm. *Studia Universitatis Babes-Bolyai Series Informatica*, LVI(2):55–61, 2011. **(indexed MathSciNet, Zentralblatt MATH, EBSCO Publishing).**

[GO11] **R. D. Găceanu** and G. Orbán. Using rsl to describe the stock exchange domain. In *microCAD International Scientific Conference.* University of Miskolc, Hungary, 31 March – 1 April 2011. **(International conference).**

[GP11b] **R. D. Găceanu** and H. F. Pop. A fuzzy clustering algorithm for dynamic environments. In *KEPT2011: Knowledge Engineering Principles and Techniques, Selected Papers, Eds: M. Frentiu, H.F. Pop, S. Motogna*, pages 119–130. Babes-Bolyai University, Cluj-Napoca, Romania, July 4–6 2011. **(ISI — Conference Proceedings Citation Index).**

[GP11c] **R. D. Găceanu** and H. F. Pop. An incremental ASM-based fuzzy clustering algorithm. In *Informatics'2011, Slovakia, i'11:Proceedings of the Eleventh International Conference on Informatics, Informatics 2011, Eds: V. Novitzká, Štefan Hudák*, pages 198–204. Slovak Society for Applied Cybernetics and Informatics, Rožňava, Slovakia, November 16–18 2011. **(indexed MathematicalReviews).**

[Găc11] **Radu D. Găceanu**. A bio-inspired fuzzy agent clustering algorithm for search engines. *Procedia Computer Science*, 7(0):305 – 307, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11), 10.1016/j.procs.2011.09.060. http://www.sciencedirect.com/science/article/pii/S187705091100620X. **(ISI — Conference Proceedings Citation Index).**

[CCGa] Gabriela Czibula, Istvan Czibula, and **Radu D. Găceanu**. Intelligent data structures selection using neural networks. *Knowledge and Information Systems*, Springer London, pages 1–22. 10.1007/s10115-011-0468-3. **(ISI — Science Citation Index Expanded)** **IF = 2.0** (2010)

[GP12] R. D. Găceanu and H. F. Pop. An incremental approach to the set covering problem. *Studia Universitatis Babes-Bolyai Series Informatica*, LVIII(2), 2012. — under review. **(indexed MathSciNet, Zentralblatt MATH, EBSCO Publishing).**

[CCGb] Gabriela Czibula, Istvan Czibula, and **Radu D. Găceanu**. A Support Vector Machine Model For Intelligent Selection of Data Representations. *Applied Soft Computing* — under review. **(ISI — Science Citation Index Expanded)** **IF = 2.1** (2010)

Contents

List of Figures

List of Tables

Introduction

This work is the result of my research in the field of Pattern Recognition, particularly Agent Based Pattern Recognition, research conducted under the supervision of both Prof. Dr. Horia F. Pop (starting from 2008) and of Assoc. Prof. Dr. Habil. László Kozma (starting from 2009).

The research topic is about using several types of software agents in pattern recognition. We will investigate in the thesis the use of agents in NP-hard optimization problems as well as in hybrid data analysis.

The rapid growth of data comes with the natural need for extracting and analysing meaningful information and knowledge from this data. This information and knowledge could be used in different applications, ranging from fraud detection, to production control, market basket analysis, customer analytics and so on. Data analysis can be viewed as a step forward in the information technology evolution. It is the process of inspecting, transforming, and modelling data with the goal of uncovering patterns, associations and anomalies and thus support decision making.

An important step in data mining is pattern recognition which deals with assigning a label to a given input data. Classification and clustering are examples of pattern recognition. Classification and clustering can be applied in many fields like in marketing (for finding groups of customers with similar behaviour), biology (classification of plants and animals given their features), fraud detection, and document classification.

Classification is the process of assigning a label to a piece of input data based, for example, on a predefined model. Since the class label of each training data item is provided, classification is a supervised learning problem. On the other hand, clustering is an unsupervised learning problem and it deals with finding a structure in a collection of unlabelled data. Classification and clustering together with a general overview of data analysis are presented in Chapter 1.

In both classification and clustering object data belonging to the same class or cluster have to be similar with each other and items from different classes or clusters have to be as dissimilar as possible. This implies a great deal of imprecision and uncertainty and a way to handle this is by using soft computing methods. Soft computing deals with imprecision and uncertainty in the attempt to achieve robustness and low cost solutions. This multidisciplinary field was introduced by Lotfi A. Zadeh and its main goal is to develop intelligent systems and to solve mathematically unmodelled problems [Zad97, CMR^{+}07, VSP09].

Soft Computing opens the possibility of solving complex problems for which a mathematical model is not available. Moreover, it introduces human knowledge like cognition, recognition, learning into the field of computing. This opens the way for constructing intelligent, autonomous, self-tuning systems.

In order to design autonomous and intelligent systems software agents are employed. An agent is an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. An agent that always tries to optimize an appropriate performance measure is called a rational agent. Agents exhibit several characteristics ([SP04, Ser06]) from which the most interesting one is self-organization. It is the capability of an entity to organize and improve its behaviour without being guided or managed. Agents seldom reside alone in the environment. Instead they coexist and interact forming multi-agent systems. Chapter 1 presents several types of interactions in a multi-agent system.

The thesis is structured in four chapters as follows.

Chapter 1, **Theoretical background**, introduces the field of data analysis and agent-based data analysis. Data analysis is becoming increasingly popular, due to the rapid growth of data amounts and the natural need for extracting meaningful information and knowledge from this data. The information and knowledge could be used in different applications ranging from intrusion detection systems, to production control, pattern recognition and so on. Data analysis can be viewed as a step forward in the information technology evolution. This chapter presents some of the most important problems in data analysis like clustering and classification and also the use of software agents.

The Chapters 2, 3 and 4 contain our original contribution in the field of agent-based pattern recognition. For each original approach that we propose, we outline possibilities for improvement and future research directions. Chapter 5 outlines the conclusions of the thesis.

Chapter 2, **Contributions to NP optimization problems**, begins with a short overview of NP completeness and NP optimization problems in Section 2.1. The rest of the chapter is entirely original and presents our contribution to NP optimization problems, focusing on two well-known NP-hard problems: Travelling Salesman Problem (TSP) and Set Covering Problem (SCP). In Section 2.1 a short overview of NP completeness is made. In Section 2.2 the travelling salesman problem is approached using the stigmergic agent model. The Stigmergic Agent System (SAS) combines the strengths of Multi-agent Systems (MAS) and Ant Colony Systems (ACS). Stigmergy provides a general mechanism that relates individual and colony level behaviours: individual behaviour modifies the environment, which in turn modifies the behaviour of other individuals. The stigmergic agent mechanism employs several agents able to interoperate in order to solve problems by using both direct communication and indirect (stigmergic) communication. The algorithm was evaluated on several standard datasets outlining the potential of the method. In Section 2.3 the soft agent model is introduced. A soft agent is an intelligent agent that may deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both. This new agent model is used in Section 2.4 where a new incremental clustering approach to the Set Covering Problem is presented. Experiments on standard datasets suggest that the approach is promising. Section 2.5 outlines the conclusions of the chapter and indicates future research directions.

Chapter 3, **New approaches to unsupervised learning**, begins with a short overview of various agent-based clustering approaches in Section 3.1. The rest of the chapter is entirely original and presents our contribution to agent-based clustering, particularly in two main directions: ASM-based batch clustering and incremental clustering [EKS$^+$98, Kam10, LKC02, LLLH10, DL11]. We are focusing on developing clustering algorithms that allow the discovery and analysis of hybrid data. In Section 3.1 a short overview of various agent-based clustering approaches is presented. We approach the idea of agent-based cluster analysis in Section 3.2. Each data is represented by an agent placed in a two dimensional grid. The agents will group themselves into clusters by making simple moves according to some local environment information and the parameters are selected and adjusted adaptively. This behaviour based on ASM (Ant Sleeping Model [CXC04]) where an agent may be either in an active state or in a sleeping state. In order to avoid the agents being trapped in local minima, they are also able to directly communicate with each other. Furthermore, the agent moves are expressed by fuzzy IF-THEN rules and hence hybridization with a classical clustering algorithm is needless. The proposed fuzzy ASM-based clustering algorithm is presented in Section 3.2.1. In this model data items to be clustered are represented by agents that are able to react according to the changes in the environment, namely the number of neighbouring agents. However a change in the data item itself is not handled at runtime. An extension to a context-aware system would be beneficial in many practical situations. In general, context-aware systems could greatly change the way we interact with the world — they could anticipate our needs and advice us when taking some decisions. In a changing environment context-awareness is undoubtedly beneficial. Such systems could make much more relevant recommendations and support decision making. An extension to a context-aware approach is presented in Section 3.2.2. Case studies for both approaches including experiments on standard datasets [Iri88, Win91] are presented in Section 3.2.3. The idea behind incremental clustering is that it is possible to consider one instance at a time and assign it to one of the already built clusters without significantly affecting the already existing structures. Section 3.3 presents an incremental clustering approach based on ASM. In incremental clustering only the cluster representations need to be kept in memory so not the entire dataset and thus the space requirements for such an algorithm are very small. Whenever a new instance is considered an incremental clustering algorithm would basically try to assign it to one of the already exiting clusters. Such a process is not very complex and therefore the time requirements for an incremental clustering algorithm are also small. The fuzziness of the approach allows the discovery of hybrid data. Experimental evaluation on standard datasets [Iri88, Win91] are presented in Section 3.3.3. Section 3.4 outlines the conclusions of the chapter and indicates some research directions that will be followed.

Chapter 4, **New supervised learning approaches to software development**, is entirely original and it focuses on the problem of dynamically selecting, using supervised learning approaches, the most suitable representation for an abstract data type, according to the software system's current execution context. In this direction, a neural network approach and a support vector machine approach are proposed. Selecting and creating the appropriate data structure for implementing an abstract data type (ADT) can greatly influence the performance of a software system. It is not a trivial problem for a software developer, as it is hard to anticipate all the usage scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Due to this fact, the software system may choose the appropriate data representation, at runtime, based on the effective data usage pattern. This dynamic selection can be achieved using machine learning techniques, which can assure complex and adaptive systems development. In this chapter we approach the problem of dynamically selecting, using supervised learning approaches, the most suitable representation for an abstract data type according to the software system's current execution context. In this direction, a neural network model and a support vector machine model are proposed. The considered problem arises from practical needs, it has a major importance for software developers. Improper use of data structures in software applications leads to performance degradation and high memory consumption. These problems can be avoided by properly selecting data structures for implementing ADTs, according to the nature of the manipulated data. In Section 4.1 the problem of dynamic data structure selection is presented. It is explained that this is a complex problem because each particular data structure is usually more efficient for some operations and less efficient for others and that is why a static analysis for choosing the best representation can be inappropriate, as the performed operations can not be statically predicted. A practical example is presented and an experiment is performed in order to motivate our approach. In Section 4.2 we present our first proposal of using supervised learning for dynamically selecting the implementation of an abstract data type from the software system, based on its current execution context. For this purpose, a neural network model will be used. In fact, selecting the most appropriate implementation of an abstract data type is equivalent to predicting, based on the current execution context, the type and the number of operations performed on the ADT, on a certain execution scenario. In Section 4.3 we evaluate the accuracy of the technique proposed in Section 4.2, i.e. the ANN model's prediction accuracy. Starting from a data set given at [For10], we have simulated an experiment for selecting the most appropriate data structure for implementing the *List* ADT. Experimental results suggest that our approach provides optimized data structure selection and reduces the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario. Section 4.4 presents a comparison to related work. In Section 4.5 the problem of data representation selection problem (DRSP) is approached using support vector machines. Computational experiments from Section 4.6 confirm a good performance of the proposed model and indicates the potential of our proposal. The advantages of our approach in comparison with similar approaches are also emphasized in Section 4.7.

Chapter 5, **Conclusions**, draws the conclusions of the thesis.

The original contributions introduced by this thesis are contained in Chapters 2, 3 and 4 and they are as follows:

- A stigmergic agent system algorithm for solving the travelling salesman problem (Section 2.2) [CDG07].
- A new model for software agents: the soft agent model (Section 2.3) [GP12].
- An incremental clustering algorithm for solving the set covering problem (Section 2.4) [GP12].

- Experimental evaluation of both algorithms on standard datasets (Section 2.2 and Section 2.4) [CDG07, GP12].
- A fuzzy ASM-based clustering algorithm (Section 3.2.1) [GP10, Găc11].
- A context-aware fuzzy clustering algorithm (Section 3.2.2) [GP11a, GP11b].
- An incremental fuzzy clustering algorithm (Section 3.3) [GP11c].
- Experimental evaluation of the algorithms on standard datasets (Section 3.2.3 and Section 3.3.3) [GP10, Găc11, GP11a, GP11b, GP11c].
- The discovery and analysis of hybrid data (Section 3.2.3 and Section 3.3.3) [GP11a, GP11b, GP11c].
- The applicability of the fuzzy ASM-based methods in clustering web search results (Section 3.2.3) [GP10, Găc11].
- A supervised learning approach for the dynamic selection of abstract data types implementations during the execution of a software system, in order to increase the system's efficiency (Section 4.2) [CCGa, CCGb].
- A neural networks approach to the considered problem (Section 4.2.2) [CCGa].
- Accuracy evaluation of the proposed neural network based technique on a case study (Section 4.3) [CCGa].
- A support vector machines approach to the considered problem (Section 4.5.3) [CCGb].
- Accuracy evaluation of the proposed support vector machine based technique on a case study (Section 4.6) [CCGb].
- A comparison of the advantages of the proposed supervised learning approaches to DRSP with existing similar approaches (Section 4.4 and Section 4.7) [CCGa, CCGb].

Chapter 1

Theoretical background

Data analysis is becoming increasingly popular due to the rapid growth of data amounts and the natural need for extracting meaningful information and knowledge from this data. Various applications ranging from production control to intrusion detection systems and pattern recognition may benefit from the extracted information and learned knowledge. Data analysis may be seen as a step forward in the evolution of information technology. In this chapter some of the most important problems in pattern recognition are presented, namely, clustering and classification. The field of soft computing is briefly presented in Section 1.3 and the topic of multi agent interactions is presented in Section 1.4.

1.1 Data analysis and data mining

The process of gathering, modelling and transforming data in the attempt to extract relevant information that may support decision making is called data analysis. Data mining focuses on modelling predictive rather than purely descriptive purposes and it is a particular technique of data analysis. Business intelligence refers to data analysis techniques applied mainly in analysing business data aiming to increase decision making support. Data analysis may be divided into exploratory data analysis and confirmatory data analysis. Exploratory data analysis deals with discovering new features in the data. On the other hand, confirmatory data analysis deals with confirming or falsifying given hypotheses. However there are several data analysis varieties. Predictive analytics, for example, applies statistical models in forecasting future events. Text analytics focuses on applying various techniques to extract and classify information contained in sources of textual data.

Data being analysed could come from various sources like databases, information repositories, data streams etc. Data warehouses represent an emerging architecture which includes the following processes: data cleaning, integration, and online analytical processing. Online analytical processing techniques allow users to analyse information from different perspectives involving the following operations: consolidation, drill-down, slice and dice. Classical classification and clustering methods may further help in data processing and information extraction. However, data sources to be analysed go beyond databases or data warehouses. Huge volumes of data may come from data streams or even from the world wide web. Efficient data analysis methods are extremely important in these situations. The huge amounts of data together with the need of extracting meaningful information from the data, is commonly known as "the data rich but information poor problem" [HK06]. In order to be able to cope with the increasing volumes of data coming from various sources, data analysis methods to be sought need to be scalable, efficient and robust. Data analysis lies at the confluence of several disciplines like databases, statistics, pattern recognition, neural networks, machine learning, image processing, information retrieval and signal processing. So, data analysis is a highly interdisciplinary domain and a natural step forward in the evolution of computer science.

Intelligent data analysis (IDA) is an emerging discipline applied in many fields [Mar09, HK06, GO11] ranging from marketing, finance, stock exchange, insurance to www document classification, biology and so on. Traditional data analysis is becoming less able to deal with the challenges raised by the increasing volumes of data and intelligent methods need to be employed in order to handle consequent difficulties. Thus intelligent data analysis deals with the development of intelligent systems for data analysis. The concepts of interest in this area include: classical statistical models, neural networks, fuzzy systems, evolutionary computation, cluster analysis, intelligent agents, expert systems, and rough sets.

1.2 Pattern recognition

Pattern recognition is the problem of assigning a label to a given input data. According to the type of the involved learning procedure, algorithms in pattern recognition may be categorized in supervised learning, unsupervised learning and semi-supervised learning algorithms. In the following we will refer to clustering which is an unsupervised learning problem and to classifications which is a supervised learning problem.

1.2.1 Cluster analysis

According to [Mar09], clustering is "the most important unsupervised learning problem". Given a collection of unlabelled data the goal of a clustering process is to find a structure in the considered dataset. In [Mar09] clustering is defined as "the process of organizing objects into groups whose members are similar in some way". So a cluster is a group of objects which are "similar" between them and are "dissimilar" to the objects belonging to other clusters [Mar09].

A good clustering algorithm should satisfy several requirements as follows [Mar09]:

- scalability, i.e., it should still work properly when applied to large datasets
- ability to handle different types of features, namely, continuous, binary, categorical, ordinal and ratio-scaled
- tolerance to high dimensional data
- independence with respect to the cluster shape
- minimal domain knowledge requirements, i.e., not too many parameters that need to be initialized
- tolerance to noise
- tolerance to outliers
- insensitivity to the order in which data items are read
- interoperability
- usability.

There are many fields in which clustering algorithms may be applied [Mar09, HK06]:

- marketing: given a dataset containing information about customers and their shopping history, find groups of customers that have a similar behaviour
- biology: given a dataset containing information about plants or animals, classify plants or animals into groups sharing similar features
- insurance: given a dataset containing information about policy holders, identify groups of persons having high claim costs, i.e., fraud detection
- earthquake studies: given a dataset of observed earthquake epicenters, the goal is to identify dangerous zones
- world wide web: identify similar documents or websites; given weblog data, identify groups where access patterns are alike.

Clustering can also be used for outlier detection [AZAY10]. A strange data value that stands out, an extreme value, a point which is far away from the other data points or, in general, a data item which is not like the rest of the data items in some sense is commonly known as an outlier [AY01]. An outlier may appear as an extreme value or a peculiar combination of the values in multivariate data. An important intelligent data analysis task is the handling of such anomalous data items because of the significant influence of outliers on the analysis result.

Careful outlier analysis may lead to the discovery of interesting phenomena from the application domain point of view. Nevertheless it is often the case that outliers are no more than measurement errors. So simply rejecting all outliers may case important information loss and inaccurate data analysis results. For example, in fraud detection, suspicious credit card transactions may indeed be fraudulent, but could also only be looking suspicious, but actually be legitimate ones.

Originated as a branch of statistics, clustering tools based on algorithms like k-means or k-medoids have been built into several packages for statistical analysis software like S-Plus, SPSS, SAS etc [HKT01]. Being an unsupervised learning problem, clustering does neither rely on any predefined classes nor on training examples. So in clustering, learning is observation-based rather than example-based. Continuous efforts are being done in order to develop more efficient, robust and scalable clustering analysis techniques.

Clustering methods could be classified in partitioning methods, hierarchical methods, density-based methods, grid-based methods and model-based methods. In the following each class of clustering algorithms will briefly described.

A partitioning clustering algorithm constructs k clusters from the given dataset, where $k <= n$ and n is the number of data items, i.e., objects to be clustered. Given the number of clusters, k, a partitioning clustering algorithm starts by creating k initial clusters. Afterwards, it tries to improve the partitioning quality by moving objects from one cluster to another such that objects which are closely related are in the same cluster and objects which are different from each other are located in different clusters.

There are many variations, but two of the most popular would be:

- k-means
- k-medoids.

In k-means, the representative of a cluster c is the mean value of the data items from c. In the k-medoids clustering algorithm the cluster representative is one of the data items located close to the cluster center.

Both k-means and k-medoids work by minimizing the squared error:

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - \mu_i||^2$$

where $(x_1, x_2, ..., x_n)$ are the data items to be clustered, $C = \{C_1, C_2, ..., C_k\}$ denotes the set of k clusters $(k \leq n)$ and μ_i is the mean of points in C_i.

Partitioning clustering methods are a good choice when the number of clusters in known in advance and when the cluster shape is spherical. However, when the dataset size grows and when the data items are complex, other partitioning methods are needed.

Hierarchical clustering methods create a hierarchy of clusters by repetitive merges or splits, and the whole process may be visualized in a dendrogram. Based on the way the clusters are created (by merging or splitting) hierarchical clustering is either agglomerative or divisive. In the divisive (top-down) approach the algorithm starts from one cluster containing the whole dataset and at each step the clusters are split into smaller ones based on some criterion. The process ends when a certain condition is met or when there is only one data item left in each cluster. In the agglomerative (bottom-up) approach the algorithm starts by encapsulating each data item with a cluster and at each step clusters are merged based on some criterion. The whole process ends when some condition is met or when there is only one cluster left containing the whole dataset.

A major drawback of hierarchical methods is that there is no possibility for undoing either a merge or a split operation so when an erroneous decision has been made there is no way for correction. On the other hand, this situation may also lead to lower computational costs.

The hierarchical clustering quality may be improved in the following ways:

- analyse object "linkages" at each hierarchical partitioning, like in Chameleon [KHK99]. Chameleon uses a sparse graph in which the nodes represent the data items to be clustered and the edges have associated weights which represent the similarities between the data items. Its key feature is that it takes into account not only interconnectivity but also closeness when deciding if two clusters are similar or not.

- perform a micro-clustering using an agglomerative approach and then group micro-clusters into macro-clusters using another clustering method as in BIRCH [ZRL97].

Density-based methods construct clusters by adding data items as long as the number of items in the neighbourhood (the density) exceeds a previously established threshold. In other words, a cluster is valid if given a data item from a cluster the neighbourhood of a certain radius contains at least a certain minimum number of data items. Such a method may be used to filter out outliers and discover clusters having any shape. DBSCAN [Est09] is a density-based clustering algorithm which works well if the density within the clusters is uniform. OPTICS [ABKS99] is another density-based algorithm which addresses the varying density issue from DBSCAN by creating an ordering of the database representing its density density-based clusters.

Grid-based methods perform a grid representation of the object space. This class of methods is especially appropriate when dealing with large data sets. Clustering operations are performed on the grid structure and among the advantages of this method is the fast processing time, which is dependent on the number of cells in each dimension from the grid space rather than on the number of objects. STING [WYM97] is an example of a grid-based method.

Model-based clustering algorithms consider a model for each cluster and the task of the algorithm is to find the best fit of the given data with respect to the considered model [HKT01]. The number of clusters is estimated using statistical methods. For example, based on statistical modelling, the EM (expectation-maximization) algorithm performs either expectation or maximization operations at each step. The COBWEB algorithm uses probabilistic concepts as cluster models and incrementally organizes the data items into a classification tree. Self-organizing feature map (SOM) is a neural network-based algorithm that perform a mapping of the data from high dimensions into a two or three dimension feature map.

1.2.2 Classification

Classification is the process of assigning a piece of input data (instance or data item) described by a vector of features to a given category (class). Initially a classifier is built based on a given dataset. This step is the training (learning) phase and at this point a classifier is built by learning from a training dataset containing a set of data items with their features and the associated class labels. Because the class label of each training data item is provided, classification is a supervised learning problem. More formally, classification may be seen as learning a mapping, $y = f(X)$, such that given a data item from X the class label y can be predicted. This model is then used for classification.

The performance or the quality of a classifier can be evaluated by computing its accuracy, i.e., the percentage of test set data items that are correctly classified. Other performance measures could be the computation speed, robustness (the ability to handle noisy or missing data), scalability and interpretability (the level of understanding and insight that is provided by the classifier) [HK06].

1.2.2.1 Decision tree induction

Decisions trees are perhaps the most straightforward and intuitive way of representing a classification process and for this reason they have been chosen for introducing this topic. Decision tree induction is the process of learning decision trees from training datasets. In a decision tree each non-leaf node represents a test on a feature, branches denote the outcome of a certain test, and leaf nodes contain the class labels. An example of a decision tree may be seen in Figure 1.2 taken from [HK06]. The concept of "buys computer" is represented in this decision tree, i.e., decide whether a given customer is likely to buy a computer.

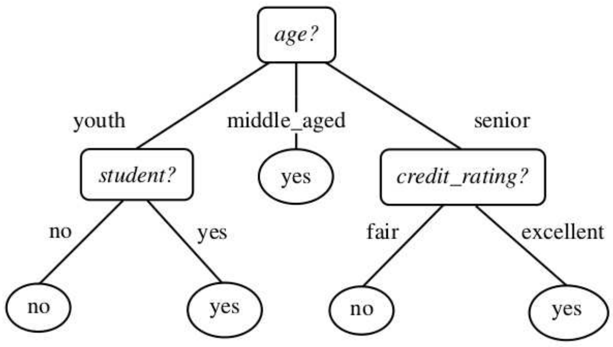


Figure 1.1: A decision tree for the concept *buys_computer*, indicating whether a customer is likely to purchase a computer. Each internal (non-leaf) node represents a test on an feature. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*). Image source: [HK06].

The classification process takes place in the following way: given a data item with no associated class label, test the data item's features with respect to an already built decision tree. By doing so a path from the root node of the decision tree to a leaf is constructed. The class prediction of the given data item is contained in the leaf node that was reached. It is important to remark that decision trees may be transformed into classification rules. One of the advantages of decision tree classifiers is that they do not require any parameter setting or domain knowledge, making them suitable for exploratory knowledge discovery [HK06]. Another advantage is their ability to handle high dimensional data. Also the tree representation is intuitive and easily understandable. According to [HK06] decision tree classifiers have good accuracy. During tree construction, various feature selection methods are used in order to select the feature that leads to the best data partitioning. When building decision trees, some branches may represent noise or may denote outliers from the training dataset and hence tree pruning is used for removing these branches. A drawback of decision trees is scalability since in general the dataset is memory resident. A general approach for decision trees construction is presented in Algorithm 1.2.1.

Algorithm 1.2.1 Generate decision tree

```
1: Create a node $N$
2: Check for particular cases
3: Find the feature $a_{best}$ that best divides the training data
4: Label the node $N$ with $a_{best}$
5: Recurse on the split data and add the new nodes as children of node $N$
```

A particular case is when all data items from the dataset are of the same class C. In this case the node N is returned with label C. Another particular case is when the feature list is empty and in this particular case the node N is labelled by the majority class, C, from the dataset.

An important step of the considered algorithm is choosing the splitting feature a_{best} that best divides the input data. Decision tree algorithms differ from each other mainly in two aspects: the feature selection process and pruning mechanism. In a feature selection process the goal is to select the splitting criterion that best partitions a given dataset, D, of class-labelled training data items into separate partitions. In the ideal case the resulted partitions are pure, i.e., all data items from a partition belong to the same class. Some of the mostly used feature selection measures (splitting rules) are information gain, gain ratio, and gini index [HK06].

Let D be a training dataset of class-labelled data items and m the number of classes, $C_i, i = \overline{1, m}$.

The $ID3$ algorithm uses information gain as its feature selection measure by determining at each step the attribute having the highest information gain and grouping data items according to the value of the selected attribute [HK06].

Definition 1.2.1 *The expected information needed for classifying a data item in D (the entropy of D) is given by*

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{1.1}$$

where:

- *p_i denotes the probability that a data item from the dataset belongs to the class C_i*
- *$p_i \approx \frac{|C_i|}{|D|}$.*

Definition 1.2.2 *The expected information needed for partitioning a dataset D given an attribute A is*

$$Info_A(D) = \sum_{j=1}^{v} \frac{\mid D_j \mid}{\mid D \mid} Info(D_j), \tag{1.2}$$

where:

- *the term $\frac{|D_j|}{|D|}$ acts as the weight of the j^{th} partition*
- *A is an attribute having v possible values: $a_1, a_2, \ldots, a_v$.*

Remark 1.2.1 *The goal is to find the attribute A from which $Info_A(D)$ is minimal and hence the purity of the obtained partitions is maximal.*

Definition 1.2.3 *The information gain is*

$$Gain_A(D) = Info(D) - Info_A(D) \tag{1.3}$$

Remark 1.2.2 *The information gain, $Gain_A(D)$, shows how much information is obtained by splitting on the attribute A.*

Remark 1.2.3 *The goal is to find the attribute A for which $Gain_A(D)$ is maximal.*

Remark 1.2.4 *Information gain favours attributes having a large number of values.*

The $C4.5$ algorithm attempts to overcome the drawback from Remark 1.2.4 and uses the gain ratio measure (Definition 1.2.4) for feature selection [HK06].

Definition 1.2.4 *The gain ratio measure is*

$$GainRatio_A(D) = \frac{Gain_A(D)}{SplitInfo_A(D)}, \tag{1.4}$$

where:

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{\mid D_j \mid}{\mid D \mid} \log_2\left(\frac{\mid D_j \mid}{\mid D \mid}\right) \tag{1.5}$$

Remark 1.2.5 *The goal is to split on the attribute A having the maximal gain ratio.*

Remark 1.2.6 *A drawback of the gain ratio measure is that special constraints need to be considered when $SplitInfo_A(D) \to 0$.*

The gini index measure is used in $CART$ and it measures the impurity of D [HK06].

Definition 1.2.5 *The gini index is*

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \tag{1.6}$$

where:

- *p_i is the probability that a data item from D belongs to the class C_i*
- *$p_i \approx \frac{|C_i|}{|D|}$*
- *D is a dataset or a partition of a dataset.*

Definition 1.2.6 *The weighed gini index is*

$$Gini_A(D) = \sum_{k=1}^{v} \frac{\mid D_k \mid}{\mid D \mid} Gini(D_k), \tag{1.7}$$

where:

- *A is the splitting attribute*
- *v is the number of partitions obtained by splitting on A*

Definition 1.2.7 *The reduction in impurity is*

$$\Delta Gini(A) = Gini(D) - Gini_A(D) \tag{1.8}$$

Remark 1.2.7 *The goal is to maximize the reduction in impurity $\Delta Gini(A)$.*

1.2.2.2 Other conventional classification methods

This section will briefly describe some other classical classification methods namely Bayesian classification and rule based classification.

Bayesian classification Bayesian classification is a statistical classification method which computes the probability that a given data item belongs to a particular class thus predicting class memberships [BG08].

Theorem 1 (Bayes' theorem) . *Let us denote by $C_j, j = \overline{1, J}$ the possible classes and by $P(C_j \mid X_1, X_2, \ldots, X_p)$ the posterior probability of belonging to the class C_j given the features $X_1, X_2, \ldots, X_p$ then*

$$P(C_j \mid X_1, X_2, \ldots, X_p) = \frac{P(X_1, X_2, \ldots, X_p \mid C_j) \cdot P(C_j)}{\sum_j P(X_1, X_2, \ldots, X_p \mid C_j)},$$

where:

- *$P(X_1, X_2, \ldots, X_p \mid C_j)$ denotes the probability of an item with individual characteristics (features) $X_1, X_2, \ldots, X_p$ belonging to the class C_j*
- *$P(C_j)$ denotes the unconditional prior probability of belonging to the class C_j.*

Remark 1.2.8 *The conditional class probabilities are exhaustive, that is, an item $X_1, X_2, \ldots, X_p$ has to belong to one of the J classes, i.e.,*

$$\sum_{j=1}^{J} P(C_j \mid X_1, X_2, \ldots, X_p) = 1. \tag{1.9}$$

Remark 1.2.9 *When the number of data items is low and the number of classes is high there is an increased probability of having*

$$P(X_1, X_2, \ldots, X_p \mid C_j) = 0 \tag{1.10}$$

for most classes.

One way to overcome the drawback mentioned in 1.2.9 is to make the classifier "naïve", i.e., to consider the features $X_1, X_2, \ldots, X_p$ as being independent from each other. Then the probabilities are computed as follows

$$P(X_1, X_2, \ldots, X_p \mid C_j) = \prod_{k=1}^{p} P(X_k \mid C_j). \tag{1.11}$$

Of course the independence assumption is very simplistic since the features $X_1, X_2, \ldots, X_p$ are very likely to be correlated and in this sense this classification method is "naïve" [BG08].

Rule based classification In rule based classifiers the set of rules may either be generated from decision trees or they may be induced from the training dataset [HK06]. These classifiers represent each classes by DNFs (disjunctive normal forms). A k-DNF expression is of the form:

$$(X_1 \wedge X_2 \wedge \cdots \wedge X_n) \vee (X_{n+1} \wedge X_{n+2} \wedge \ldots X_{2n}) \vee \cdots \vee (X_{(k-1)n+1} \wedge X_{(k-1)n+2} \wedge \cdots \wedge X_{kn}), \tag{1.12}$$

The goal of the rule based classifier is to construct the smallest set of rules such that consistency with respect to training data is preserved. A large number of rules is an indication of attempting to remember the training set, as opposed to discovering the assumptions that govern it [Kot07]. A general pseudo-code for rule-based classifiers is presented in Algorithm 1.2.2.

Algorithm 1.2.2 Learn rules

```
1: $RuleSet \leftarrow \emptyset$
2: for all classes c do
3:   repeat
3:     $Rule \leftarrow Find\_Best\_Rule(c)$
3:     Remove items covered by $Rule$
4:   until $termination\_condition$
5:   $RuleSet \leftarrow RuleSet \cup \{Rule\}$
6: end for
```

The *Find_Best_Rule* procedure finds the "best" rule for the considered class c, from the given training set. Usually rules are constructed in a general-to-specific fashion. The algorithm starts with an empty rule and then repetitively appends attribute tests to it as a conjunction to the already constructed rule antecedent. With each attribute test addition the rule should cover more data items from the current class c. The process repeats until the rule reaches an acceptable quality.

1.2.2.3 Alternate classification methods

In this section, other classification methods like neural networks and support vector machines are described.

Neural networks A neural network is composed of several connected input and output units each connection having an associated weight. During learning, the network modifies the weights in the attempt to predict which is the class label corresponding to the given input data items.

In general neural networks require long time training and some parameter setting like the topology or structure of the network. Also the learned weights together with the hidden units in the network are difficult to be interpreted [HK06]. Some of the advantages of neural networks are as follows [Zha00, MK91]:

- high tolerance to noisy data
- good performance in classifying unknown patterns
- can be used when there is little knowledge regarding any relationship among attributes and classes
- well-suited for continuous-valued inputs and outputs, as opposed to most decision tree based algorithms
- are parallel — may lead to faster computation
- self-adaptive to the data without external intervention or any kind of specification
- are universal function approximators, that is, neural networks are able to approximate any function with arbitrary accuracy
- are non-linear models — can model complex, real-world relationships
- can estimate posterior probabilities — provide the basis for establishing classification rule and performing statistical analysis.

A neural network classifier may be considered as a mapping

$$F : R^d \to R^M, \tag{1.13}$$

where x represents the input of the network and the output y is the classification result. The goal of the network is to minimize an error function.

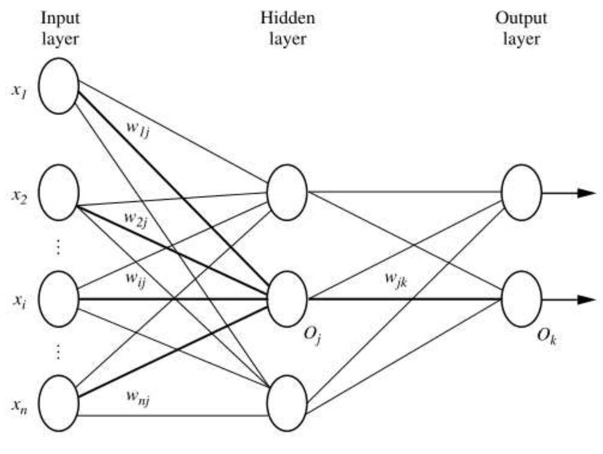


Figure 1.2: A two-layer feed-forward neural network. Image source: [HK06].

Maybe the most popular neural network algorithm is backpropagation where a multilayer feed-forward neural network is used. The algorithm repetitively learns a set of weights in the attempt to predict the class label corresponding to the input data items. A multilayer feed-forward neural network is shown in Figure 1.2.2.3. Neural networks are able to approximate any function provided that there are enough training samples given and enough hidden units available [HK06].

Support vector machines Given a training dataset of the form $(x_i, y_i), i = \overline{1, l}$ where $x_i \in R^n$ are instances and $y \in \{1, -1\}^l$ are labels, the support vector machine (SVM) [HCL00] searches for the solution of an optimization problem as follows:

$$\min_{w,b,\xi} \frac{1}{2}w^T w + C\sum_{i=1}^{l}\xi_i \tag{1.14}$$

$$subject\ to\ y_i(w^T\phi(x_i)+b) \geq 1-\xi_i, \tag{1.15}$$

$$\xi_i \geq 0. \tag{1.16}$$

The function ϕ maps the training vectors x_i into a higher dimensional space which may be infinite. Support vector machines find in this higher dimensional space a maximal margin linear separating hyperplane $C > 0$ is called the error term's penalty parameter. $K(x_i, y_i) = \phi(x_i^T\phi(x_j))$ is called kernel function. One of the most popular kernel function is RBF (the radial basis function)

$$K(x_i, y_i) = exp(-\gamma \parallel x_i - x_j \parallel^2), \gamma > 0, \tag{1.17}$$

where γ is a kernel parameter. This kernel non-linearly maps samples into a higher dimensional space so it can handle non-linear relationships between attributes and class labels [HCL00].

Even though training time can be slow, SVMs are very accurate, not too prone to overfitting and are able to model complex, non-linear relationships.

1.3 Soft computing

Proposed by Lotfi A. Zadeh [Bla94b], soft computing [Zad94] is a multidisciplinary field that deals with imprecision, uncertainty, approximation, and partial truth in order to achieve robustness and low cost solutions. The main objective of soft computing is the development of intelligent systems and solving non-linear and mathematically complicated to model problems [Zad97]. The main advantages of soft computing are:

- it opens the possibility of solving complex problems, in which mathematical models are not available
- it introduces the human knowledge such as cognition, recognition, understanding, learning, and others into the field of computing and hence opens the way for constructing intelligent, autonomous, self-tuning systems.

Soft computing comprises, but is not limited to, the following components: fuzzy systems, neural networks, swarm intelligence, evolutionary computing. Fuzzy sets [Zad65] represent a mathematical theory for modelling imprecision and they are central to soft computing. They were introduced by Zadeh, having a major success initially in Japan and China and then in the whole world [Bla94a].

The elements of a fuzzy set have a certain membership degree with respect to the set as opposed to classical (crisp) set theory where an element either belongs or not to the set.

Definition 1.3.1 *Let X be the universe of discourse. A fuzzy set A is a function*

$$A : X \to [0, 1], \tag{1.18}$$

where:

$$A(x)\ is\ the\ membership\ degree\ of\ x\ to\ A, \forall x \in X. \tag{1.19}$$

Remark 1.3.1 *In the classical set theory an element either belongs or not to the set. This can be denoted with a value from the set {0,1}. In the fuzzy set theory the membership degree of an element to the set is a value from the [0,1] interval.*

Remark 1.3.2 *A crisp set can be seen as a special case of a fuzzy set as follows:*

$$A(x) = \begin{cases} 1 & if x \in A \\ 0 & if x \in X \setminus A. \end{cases}$$

The attempts of philosophers and logicians to go beyond the classical two-valued logic have been motivated by the need to represent and conduct reasoning on reality. Those attempts to expand the two-valued logic into a more realistic and flexible logic have started very early in history. Aristotle presented the problem of having propositions that may not be true or false [KY95] arguing that propositions on future events would not have any truth values; the way to resolve their truth values is to wait until the future becomes present.

But the propositions about future events are not the only propositions with problematic truth values. In fact, the truth values of many propositions may be inherently indeterminate due to uncertainty. This uncertainty may be due to measurement limitations or it can also be caused by the intrinsic vagueness of the linguistic hedges of natural languages when used in logical reasoning.

So multi-valued logics have been invented in order to capture the uncertainty in truth values. Initially a three-valued logic appeared where a proposition had a truth-value of half, in addition to the classically known zero and one. Afterwards, the three-valued logic concept was generalized into the n-valued logic concept, where n can be any number. Hence, in an n-valued logic, the degree of truth of a proposition can be any one of n possible numbers in the interval [0, 1]. Lukasiewicz was the first one to propose a series of n-valued logics for which $n \geq 2$. In the literature, the n-valued logic of Lukasiewicz is usually denoted as L_n, where $2 \leq n \leq \infty$. Hence, L_∞ is the infinite valued logic, which is obviously isomorphic to fuzzy set theory as L_2 is the two-valued logic, which is isomorphic to the crisp set theory [KY95].

Fuzzy logic can be viewed as an extension of the Boolean logic. It's important to remark that basically any theory could be fuzzified by replacing the classical sets with fuzzy sets. Fuzzy sets could be applied in modelling inexact behaviour, which was not very convenient via the classical set theory. While the classical set theory deals with well defined objects, the fuzzy set theory deals with objects which have a certain membership degree.

For example the set of all tall persons includes persons who are very tall, tall and less tall. If the set of tall persons is a classical set, each person either belongs or not belongs to it; but if it is a fuzzy set, different persons can belong to it with a different degree (very tall persons with a high degree, less tall persons with a low degree). While the human mind can easily decide what tall means, it is a quite difficult task for a computer to be programmed like this. Fuzzy logic prescribes rules by which abstract notions like tall could be transposed in algorithms. Fuzzy sets are used with success in multiple domains and currently have a large industrial applicability.

It is often the case that related linguistic concepts like short, medium, tall are represented by fuzzy sets which define the states of a fuzzy variable. Fuzzy variables allow gradual state transitions and, hence, one can naturally deal with uncertainties and human perceptions. Traditional variables (also called crisp variables) are not able to deal with uncertainty, with vagueness. Even though states as defined by crisp sets are mathematically correct, they are not realistic because, for example, a measurement my fall close to the border between two states of a crisp variable. In this case only one state is considered relevant, in spite of the inevitable imprecision and uncertainty involved in this decision. At the border, where uncertainty is maximal, it would be normal to consider the measurement as evidence for both states. Even this extreme situation is missed in the crisp case. Fuzzy variables on the other hand gracefully capture such measurement uncertainties which makes them more practical in real life scenarios than crisp sets.

Definition 1.3.2 *Fuzzy IF-THEN rules are conditional statements that comprise fuzzy logic:*

$$if\ x\ is\ A\ then\ y\ is\ B \tag{1.20}$$

where:

- *A and B are fuzzy variables*
- *the if-part of the rule is called antecedent or premise*
- *the then-part is called consequent or conclusion.*

Fuzzy logic is suitable in the following situations[KS05]:

- controller analysis and design for chemical and other kind of processes
- parameter estimation in nonlinear systems
- heuristics based systems
- where conventional approaches are difficult or expensive to implement
- where the system is complicated to be modelled because of the presence of uncertainties.

There are two views regarding fuzzy logic among scientists [KY95]:

- the traditional view — according to which uncertainty should be avoid by any means
- an alternative view — according to which science cannot avoid dealing with uncertainty.

The traditional view promotes the idea that science should struggle for certainty in any of its aspects (precision, specificity, sharpness, consistency, etc.) and consequently, uncertainty (imprecision, nonspecificity, vagueness, inconsistency, etc.) should be considered as unscientific [KY95]. According to [KY95], when constructing a model, our goal is to maximize its usefulness and it is argued that this goal is actually related to the three key characteristics of any system model: complexity, credibility, and uncertainty. This relationship is not as yet fully understood. However it may be observed that by allowing more uncertainty in a system one obtains a reduction in complexity and an increased credibility of the model. The level of uncertainty that should be allowed in a system is an open issue.

An important aspect is to distinguish between probability theory and fuzzy set theory and in order to do that we have to understand the type of uncertainty that each of them describes and processes. The uncertainty described by probability is randomness. The uncertainty described by fuzzy set theory is fuzziness.

Probability theory deals with the prediction of a future event based on the information currently available. On the other hand, fuzzy set theory deals with concepts and status of variables, rather than future events.

Zadeh advances the idea that probability theory should be based on fuzzy logic instead of the being based on bivalent logic [Zad02]. The most important argument is the impossibility to operate with information based on perception in the case of probability theory. Information based on perceptions has a rather descriptive aspect and consists of propositions from the natural language like: "They are is tall", "They usually drink tea at about 5 o'clock". The inability of probability theory to operate with information based on perception is a considerable limitation since perceptions are central in human cognition.

To enable probability theory to deal with problems of this kind, a restructuring of probability theory is necessary by replacing boolean logic with fuzzy logic. The idea is that fuzzy logic is actually the logic of perceptions, while boolean logic is the logic of measurements. In [Zad02], Zadeh introduces the concept of perception-based probability theory. A major difference between probability theory and perception-based probability theory is that in the former only likelihood is a matter of degree. On the other hand, in perception-based probability theory, everything (especially truth and possibility) is allowed to be a matter of degree.

In complex environmental systems, several types of uncertainties may arise for various reasons like high variability in space and time of some physical, chemical or biological processes involved but also man-induced uncertainties. Such uncertainties are difficult to be modelled by classical theories, but fuzzy logic can be applied in such situations. There are many applications of fuzzy logic [SP00, PSHD96, PS96, FMPS00, SGT$^+$02, DP95, DP98], but actually "is there need for fuzzy logic"? This is the question that Zadeh addresses in [Zad08]. There were and maybe still are many people who are reluctant to fuzzy logic. As it is very well explained, fuzzy logic is not fuzzy, it is actually a precise logic of imprecision and approximate reasoning. It is concluded that the progress from bivalent to fuzzy logic is a natural step forward and an important evolution of science. This is because the real world is a fuzzy one so in order to deal with a fuzzy reality, fuzzy logic is needed. According to Zadeh, in future years, fuzzy logic is likely to grow in acceptance.

1.4 Multi agent interactions

A multi agent system (MAS) is a system composed of several interacting agents. Multi-agent systems may be used for solving problems which are difficult or impossible for an individual agent or a monolithic system to solve. Communication is crucial in MAS. In general direct communication is assumed in a classical MAS and in this case we deal with intelligent agents. But communication could also be done indirectly, through the environment. In this case we deal with formations of simple creatures like ant colonies or bird flocks which collectively lead to the emergence of intelligent global behaviour, to what is known as swarm intelligence.

1.4.1 Direct agent interactions

An agent is an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [SP04]. An agent that always tries to optimize an appropriate performance measure is called a rational agent. Such a definition of a rational agent is fairly general and can include human agents (having eyes as sensors, hands as actuators), robotic agents (having cameras as sensors, wheels as actuators), or software agents (having a graphical user interface as sensor and as actuator). According to [SP04, Ser06] agents exhibit the following characteristics:

- autonomy
- reactivity
- pro-activity
- sociability
- intelligence
- mobility
- self-organization.

The most attractive property is self-organization, that is, the ability to improve the its behaviour without external influence or guidance.

Usually agents coexist and interact forming Multi-agent Systems (MAS).

Definition 1.4.1 *In computer science, a multi agent system (MAS) is a system composed of several interacting agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system.*

The precise nature of the agents is not clearly established. MAS may also include human agents. Examples of multi agent systems include human organizations and society in general.

Remark 1.4.1 *In a multi agent system, agents can be software agents, robots and also humans.*

Remark 1.4.2 *As a consequence to Remark 1.4.1 it follows that there is no single agent system.*

A multi agent system has the following advantages:

- it is inherently a distributed architecture and thus critical failures and performance bottlenecks are avoided
- allows interconnection of legacy systems
- problems like task allocation, team planning, complex phenomena simulation are naturally modelled in terms of interacting agents
- efficiently operates with information from spatially distributed sources

- suitable in situations where knowledge is distributed temporally and spatially
- enhances system performance at least in the following aspects of computational efficiency, robustness, reliability, and extensibility.

Some of the important things that need to be taken into consideration when dealing with a MAS are: communication between agents, collaboration and coordination [Ser06]. The interactions between agents in a MAS can be either cooperative or selfish [Ser06, Woo99, RN02]. The information exchange is done using Agent Communication Languages like KQML [FLM97] and FIPA ACL [fIPA].

MAS include several topics of research as follows [Ser06]:

- beliefs, desires, and intentions (BDI)
- cooperation and coordination
- organization
- communication
- distributed problem solving
- multi agent learning.

Examples of agent based applications include:

- planners that schedule our meetings
- personal assistants that send us reminders
- web ranking tools
- automated form fillers
- intelligent download managers.

It has been a matter of debate on which domain to fit the agents — software engineering or artificial intelligence. It has been claimed that "Intelligent agents are ninety-nine percent computer science and one percent AI" [Etz96]. But such discussions imply a certain degree of interest from researchers in both software engineering and artificial intelligence which eventually leads to a better development this area of research.

1.4.1.1 A formal agent model

An agent is an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [SP04]. The abstract view of this definition is best expressed by Figure 1.3.

A formalization of the abstract view of an agent based on [Woo09] is presented in the following. We start doing so by considering the environment as a finite set E of states:

$$E = \{e, e', \ldots\}. \tag{1.21}$$

Remark 1.4.3 *Any continuous environment may be modelled by a discrete environment with any accuracy level.*

The set of possible actions an agent may choose from is

$$Ac = \{\alpha, \alpha', \ldots\}. \tag{1.22}$$

Definition 1.4.2 *A run, r, of an agent in an environment a sequence of state transitions:*

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{u-1}} e_u \tag{1.23}$$

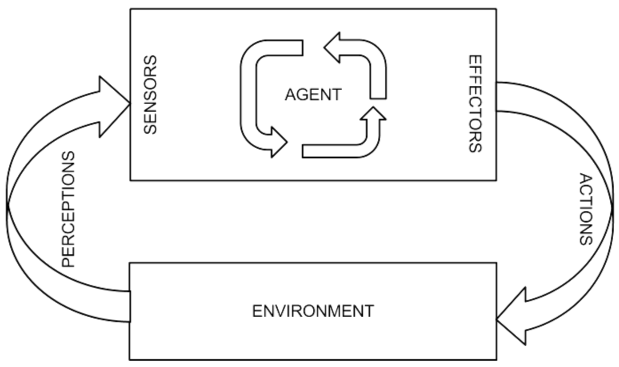


Figure 1.3: An agent perceiving its environment. The agent takes sensory input from the environment, and outputs actions that modify the environment.

Definition 1.4.3 *An agent in the environment is:*

$$Ag : \mathcal{R}^E \to Ac \tag{1.24}$$

where $\mathcal{R}^E$ is the subset of all possible runs r that end with an environment state.

Thus agents decide upon the action to perform based on his history in the environment.

Certain types of agents decide what to do without reference to their history. They base their decision making entirely on the present, with no reference at all to the past. We will call such agents purely reactive, since they simply respond directly to their environment.

Definition 1.4.4 *A purely reactive agent is*

$$Agr : E \to Ac. \tag{1.25}$$

Remark 1.4.4 *Every purely reactive agent has an equivalent standard agent.*

In order to be able to construct agent this abstract agent model needs to be refined by breaking it down into subsystems. In that sense the agent's decision function may be divided in two subsystems as shown in Figure 1.4.

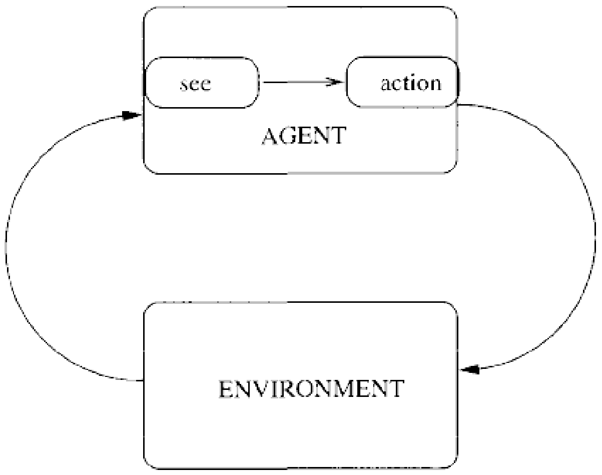


Figure 1.4: Perception and action subsystems. Image source: [Woo09]

The *see* block captures the environmental information and transforms it into perceptions. The set of all possible perceptions is denoted with Per. Then *see* is defined as a function

$$see : E \to Per \tag{1.26}$$

which maps environmental states to perceptions.

In a similar manner the *action* block is defined as a function

$$action : Per^* \to Ac \tag{1.27}$$

which maps sequences of percepts to actions.

Remark 1.4.5 *An agent Ag may be seen as a pair $Ag = (see, action)$, where see and action are functions as considered above.*

Definition 1.4.5 *An agent with state is a triple*

$$Ag = (see, action, next), \tag{1.28}$$

where:

$$see : E \to Per, \tag{1.29}$$

$$action : I \to Ac, \tag{1.30}$$

$$next : I \times Per \to I. \tag{1.31}$$

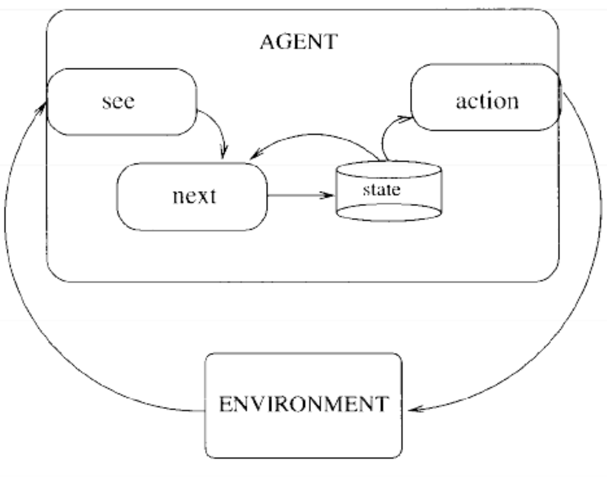


Figure 1.5: Agents with state. Image source: [Woo09].

As shown in Figure 1.5, agents that maintain state use an internal data store of the environmental sates. We denote by I set the of internal sates. The *next* function maps an internal states and percepts to new updated internal states and the *action* function uses these updated states.

1.4.2 Indirect agent interactions

Ant colony optimization (ACO) [DS04] is a nature-inspired metaheuristic that addresses combinatorial optimization (CO) problems [PS82]. Some inherently hard problems can be addressed using metaheuristics [BR03]. Examples of metaheuristics are: ACO, tabu search, simulated annealing and evolutionary computation. It is important to note that these are approximation algorithms, i.e., they are used for obtaining good enough solutions in an acceptable amount of time [Glo89, Glo90, KJV83].

Definition 1.4.6 *A combinatorial optimization problem problem is an optimization problem in which, given:*

- *a set S of candidate solutions (search space) and*
- *a function $f : S \to \mathbb{R}^+$*

where the function f (objective function) assigns a positive cost value to each solution from S, the goal is one of the following:

- *find a solution of minimum cost value*
- *a good enough solution in a reasonable amount of time (the metaheuristic case).*

The ACO metaheuristic is composed from different algorithms in which several cooperative agents attempt to simulate real ants behaviour. Initially ants wander randomly in order to find food, but they leave pheromone trails in their way. If another ant finds the trail it will likely follow it rather than continue its random path, thus reinforcing the trail. Over time however, pheromone trails tend to evaporate and thus, in time, longer paths will tend to have a lower pheromone level. When an ant needs to choose which path to follow, the path having a larger pheromone quantity has a higher probability to be chosen. As a result, ants will converge to a shorter path which should be a near-optimum solution to the considered problem. So the use of indirect communication enabled by the pheromones allows ants to find the shortest path between their nest and the place where the food is located. This behaviour observed at real-life colonies of ants is the inspiration for the use of artificial ant colonies in solving CO problems.

The ACO approach addresses optimization problems by repeating the steps from bellow [DB05]:

- construct candidate solutions using a pheromone model
- use candidate solutions in order to update the pheromone model such that future solutions are improved

Several ACO algorithms have been proposed, but in the following we will describe the MAX-MIN, Ant System and Ant Colony System algorithms from [Dor07]. Problems addressed by ACO are often modelled as finding the shortest path in a weighted graph.

In the Ant System approach the values of the pheromones are updated at each iteration by all the m ants when a solution is built. The pheromone τ_{ij}, associated with the edge joining vertex i and vertex j, is updated as follows:

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{1.32}$$

where:

- ρ is the evaporation rate
- m is the number of ants
- $\Delta\tau_{ij}^{k}$ is a quantity measure of the pheromone laid by ant k on edge (i,j).

When located on vertex i, ant k and will move to vertex j with the following probability:

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{c_{ij}\epsilon N(s^p)} \tau_{ij}^{\alpha}\cdot\eta_{ij}^{\beta}} & if c_{ij} \epsilon N(s^p) \\ 0 & otherwise, \end{cases} \tag{1.33}$$

where:

- s^p is the partial solution

- $N(s^p)$ is the set of available components composed of the edges (i,l) where l is a city not yet visited by the ant k
- η_{ij} is the heuristic information
- $\eta_{ij} = \frac{1}{d_{ij}}$, where d_{ij} is the weight of the edges (i,j) .
- α and β control the importance between the heuristic information and the pheromone.

The the MAX-MIN algorithm is an improvement of the Ant System algorithm by allowing pheromone updates to be made only by the best ant. Also the the pheromone value is bound. The update rule is the following:

$$\tau_{ij} \leftarrow \left[(1-\rho)\cdot\tau_{ij} + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{max}}, \tag{1.34}$$

where:

- τ_{max} is the upper bound of the pheromone
- τ_{min} is the lower bound of the pheromone
- $\Delta\tau_{ij}^{best} = \begin{cases} 1/L_{best} & if\ (i,j)\ belongs\ to\ the\ best\ path \\ 0 & otherwise, \end{cases}$
- L_{best} is length of the path followed by the best ant

The operator $[x]_a^b$ is defined as:

$$[x]_a^b = \begin{cases} a & if x > a \\ b & if x < b \\ x & otherwise; \end{cases} \tag{1.35}$$

The pheromone's lower and upper bounds, τ_{min} and τ_{max}, they are typically obtained empirically and tuned on the considered problem.

The pheromone update is usually offline, i.e., it is performed by one of the ants when a solution is obtained. In addition to that, the Ant Colony System (ACS) introduces the idea of local pheromone update which is applied by each ant to the last traversed edge only:

$$\tau_{ij} = (1-\varphi)\cdot\tau_{ij} + \varphi\cdot\tau_0, \tag{1.36}$$

where:

- $\varphi\epsilon(0,1]$ is called pheromone decay coefficient
- τ_0 is the initial pheromone value.

The main advantage of this local update rule is that it decreases the chances that more ants construct the same solution within one iteration. The offline update, similarly to MAX-MIN algorithm, is applied by only one ant at the end of each iteration. The offline update rule in this case is:

$$\tau_{ij} \leftarrow \begin{cases} (1-\rho)\cdot\tau_{ij} + \rho\cdot\Delta\tau_{ij} & if\ (i,j)\ belongs\ to\ the\ best\ path \\ \tau_{ij} & otherwise \end{cases} \tag{1.37}$$

where $\tau_{ij} = 1/L_{best}$ — as in the MAX-MIN algorithm.

Chapter 2

Contributions to NP optimization problems

The chapter begins with a short overview of NP completeness and NP optimization problems in Section 2.1. The rest of the chapter is entirely original and presents our contribution to NP optimization problems, focusing on two well-known NP-hard problems: Travelling Salesman Problem (TSP) and Set Covering Problem (SCP).

The approaches presented in this chapter represent original works published in [CDG07, GP12].

The chapter is structured as follows. In Section 2.1 a short overview of NP completeness is made. In Section 2.2 the travelling salesman problem is approached using the stigmergic agent model. The Stigmergic Agent System (SAS) combines the strengths of Multi-agent Systems (MAS) and Ant Colony Systems (ACS). Stigmergy provides a general mechanism that relates individual and colony level behaviours: individual behaviour modifies the environment, which in turn modifies the behaviour of other individuals. The stigmergic agent mechanism employs several agents able to interoperate in order to solve problems by using both direct communication and indirect (stigmergic) communication. The algorithm was evaluated on several standard datasets outlining the potential of the method. In Section 2.3 the soft agent model is introduced. A soft agent is an intelligent agent that has to deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both. This new agent model is used in Section 2.4 where a new incremental clustering approach to the Set Covering Problem is presented. Experiments on standard datasets suggest that the approach is promising. Section 2.5 outlines the conclusions of the chapter and indicates future research directions.

The original contributions of this chapter are:

- A stigmergic agent system algorithm for solving the travelling salesman problem (Section 2.2) [CDG07].
- A new model for software agents: the soft agent model (Section 2.3) [GP12].
- An incremental clustering algorithm for solving the set covering problem (Section 2.4) [GP12].
- Experimental evaluation of both algorithms on standard datasets (Section 2.2 and Section 2.4) [CDG07, GP12].

2.1 NP-completeness

Problems for which the required time for solving a problem from this class using any nowadays available algorithm increases very fast as the problem size grows are called NP-complete. Nevertheless it is still necessary to somehow deal with these problems. In many situations approximation algorithms are used in order to address NP-complete problems [CLRS09].

Definition 2.1.1 *A decision problem L is called NP-complete if:*

1. $L \in NP$ and

2. $L^{'} \leq p\ L, \forall L^{'} \in NP$,

where $L^{'} \leq p\ L$ means that $L^{'}$ is reducible to L in polynomial time.

Definition 2.1.2 *A problem is NP-hard if it satisfies only the second property from Definition 2.1.1 and not necessarily the first one.*

Perhaps the most interesting aspect of NP-completeness is the property that if a single NP-complete problem can be solved in polynomial time then there an algorithm solvable in polynomial time for all NP-complete problems, i.e., $P = NP$ (where P is the class of problems solvable in polynomial time and NP is the class of problems solvable in nondeterministic polynomial time). The $P \neq NP$ is the most interesting open problems in the theory of computer science [CLRS09].

There are many examples of NP optimization problems [coNop] and we will discuss about two of them in the next sections, namely the Travelling Salesman Problem (TSP) and The Set Covering Problem (SCP).

2.2 Stigmergic agents

In [CDG07] a Stigmergic Agent System (SAS) combining the strengths of Ant Colony Systems and Multi-Agent Systems concepts is proposed. The agents from the SAS are using both direct and indirect (stigmergic) communication. Stigmergy occurs as a result of individuals interacting with and changing an environment [DS04]. Stigmergy was originally discovered and named in 1959 by Grasse, a French biologist studying ants and termites. Grasse was intrigued by the idea that these simple creatures were able to build such complex structures. The ants are not directly communicating with each other and have no plans, organization or control built into their brains or genes. Nevertheless, ants lay pheromones during pursuits for food, thus changing the environment. Even though ants are not able to directly communicate with each other, they do communicate however — indirectly — through pheromones.

Stigmergy provides a general mechanism that relates individual and colony level behaviours: individual behaviour modifies the environment, which in turn modifies the behaviour of other individuals.

The SAS mechanism employs several agents able to interoperate on the following two levels in order to solve problems:

- direct communication: agents are able to exchange different types of messages for knowledge sharing and direct interoperation support; the knowledge exchanged refers to both local and global information
- indirect (stigmergic) communication: agents have the ability to produce pheromone trails that influence future decisions of other agents within the system.

The initial population of active agents has no knowledge of the environment characteristics. Each path followed by an agent is associated with a possible solution for a given problem. Each agent leaves pheromone trails along the followed path and is able to communicate to the other agents of the system the knowledge it has about the environment after a complete path is created.By using direct communication the risk of getting trapped in local optima is lower. In order to better express the idea, the Stigmergic Agent System (SAS) algorithm for solving the Travelling Salesman Problem (TSP) from [CDG07] is given in Algorithm 2.2.1.

The Travelling Salesman Problem (TSP) is finding the cheapest round-trip route that visits each city exactly once and then returns to the starting city, given a list of cities, the cost of moving from one city to another and the starting city. An equivalent formulation in terms of graph theory is: given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or length of that road), find a Hamiltonian cycle with the least weight [DDC99].

Algorithm 2.2.1 SAS for TSP

```
1: @ initialize noOfAgents, stigmergyLevel, startingCity, knowledge base
2: while true do
3:     LaunchAgents (noOfAgents, stigmergyLevel, startingCity)
4:     @ wait until all agents finish execution
5:     @ handle best solution found - a global update rule is applied, that is, update the pheromone
       level and store the best solution found so far
6: end while
```

Algorithm 2.2.2 LaunchAgents

```
1: for i = 0 to noOfAgents do
2:     Agent agent = createAgent(stigmergyLevel, startingCity)
3:     @ execute the agent's behaviour
4: end for
```

The procedure starts by setting algorithm parameters such as the number of agents, the stigmergy level of the agents, the starting city — the knowledge base of the system in general. The process runs until certain conditions are met. At the first step agents with the given parameters are launched. Once their task is completed, the best found solution is compared with the best already known solution (if any) and a global update is performed. For updating the pheromone level the following local update rule (see [CMP06]) is used:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \rho\frac{1}{n * L^+}, \tag{2.1}$$

where τ_{ij} represents the stigmergy level of the edge (i, j) at moment t, ρ is the evaporation level and L^+ is the cost of the best tour.

The global update rule is similar:

$$\tau_{ij} = (1-\rho)\tau_{ij}(t) + \rho\Delta\tau_{ij}(t), \tag{2.2}$$

where $\Delta\tau_{ij}(t)$ is the inverse cost of the best tour.

Agents are autonomous entities meaning that they can choose to ignore the path communicated by the system and proactively choose another city to explore. This is crucial for the SAS success since only using a purely stigmergic approach the solution of a problem could be trapped into a local optimum.

The algorithm allows stigmergic selection of the next city based on the probability (see [CMP06]):

$$p_{ij}^k = \frac{\tau_{iu}(t)[\eta_{iu}(t)]^\beta}{\sum_{o \epsilon J_i^k} \tau_{io}(t)[\eta_{io}(t)]^\beta}, \tag{2.3}$$

where J_i^k represents the unvisited neighbours of node i by agent k, $\eta_{io}(t)$ is *visibility* and denotes the inverse of the distance from node i to node o and β shows what is more important between the

Algorithm 2.2.3 AgentBehavior

```
1: while a solution is not found do
2:     @ proactively determine if the next city should be chosen stigmergically or using direct commu-
       nication
3:     @ if the agent decides to behave stigmergically then the next city to be visited is chosen using
       standard ACS; otherwise the next city to be visited is chosen using direct communication with
       the other agents
4:     @ handle best solution so far - a local update rule is applied, that is, update the pheromone
       level
5: end while
```

cost of the edge and the pheromone level.

Using direct communication agents can proactively choose another city to explore. So at a certain point in time if an agent decides that it should use direct communication it can ask the other agents if they have already visited a certain city. This way an unexplored city can be identified and the agent can autonomously choose it as its next move (regardless of pheromone trail intensities). Cooperating proactive agents capable of both direct and stigmergic communication provide a robust way to find a solution greatly reducing the risk of being trapped into local minima.

SAS algorithm for solving TSP is compared to standard Ant Colony System (ACS) model. In the ACS algorithm the values of the parameters were chosen as follows: $\beta = 5$, $\rho = 0.5$.

Table 2.2 presents comparative results of the proposed SAS algorithm and the ACS model for solving some instances of TSP taken from [Ins].

Problem	**Number of Agents**	**Number of Generations**	**Best Known Solution**	**ACS Result**	**SAS Result**
swiss42	3	30	1273	1589	**1546**
swiss42	5	30	1273	1539	**1517**
swiss42	10	30	1273	1491	**1489**
swiss42	15	30	1273	1472	**1470**
swiss42	20	30	1273	1472	**1439**
bays29	3	30	2020	2312	**2312**
bays29	5	30	2020	2288	**2225**
bays29	10	30	2020	2288	**2209**
bays29	15	30	2020	2288	**2202**
bays29	20	30	2020	2288	**2177**
gr120	3	30	6942	12271	**11999**
gr120	5	30	6942	12220	**10339**
gr120	10	30	6942	9571	**9548**
gr120	15	30	6942	9488	**9488**
gr120	20	30	6942	9488	**8668**

Table 2.1: Comparative testing results

Numerical experiments suggest a beneficial use of direct and stigmergic communication in cooperative multi-agent systems for addressing combinatorial optimization problems.

One of the major properties of an agent is autonomy and this allows agents to take the initiative and choose a certain path regardless of the communicated or stigmergic information. Agents can lead the way to the shortest path in a proactive way ensuring that the entire solution space is explored. Agents can demonstrate reactivity and respond to changes that occur in the environment by choosing the path to follow based on both pheromone trails and directly communicated information. Using a purely stigmergic approach the solution of a problem could fall into a local optimum, but due to direct communication ability of the agents they can proactively break out of the local optima and continue to explore the search space. In conclusion the proposed approach is a powerful optimization technique that combines the advantages of two models: Ant Colony Systems and Multi-Agent Systems. Interoperation between agents is based on both indirect communication — given by pheromone levels — and direct knowledge sharing, greatly reducing the risk of falling into the trap of local minima.

2.3 Soft agents

2.3.1 General agent models

An architecture based on agents using indirect communication is proposed in [Ste90]. This approach is employed for the following scenario [Woo09]. The task is to collect samples of a certain type of precious rock from a remote planet. The rock samples are normally grouped in certain spots, but neither their location is known nor a map of the search space is available. Several autonomous vehicles are available for search space exploration and rock collection and it is assumed that the terrain contains obstacles that also prevent any communication between the vehicles.

The solution uses two mechanisms introduced by Steels. The first is a gradient field. In order for the agents to know in which direction the mother ship lies, the mother ship generates a radio signal. The signal will of course weaken as the distance from the source increases so in order to find the direction of the mother ship, an agents should travel "up the gradient" of the strength of the signal.

The second mechanism enables information exchange between agents. Since direct communication is not possible, Steels adopted an indirect communication mechanism. The idea is that agents will carry "radioactive crumbs", which can be detected, picked up or dropped by other agents passing by. This simple mechanism leads to a rather complex form of agent cooperation.

There are obvious advantages to such an approach like [Woo09]:

- simplicity
- economy
- low computational cost
- robustness
- elegance.

However there are some fundamental, unsolved problems as in any purely reactive architecture [Woo09]:

- if agents do not employ models of their environment, then they need enough information in their local environment for being able to choose a proper action to be performed
- since purely reactive agents choose actions only based on local information then other surrounding relevant information may be not taken into account
- there seems to be no obvious way for designing purely reactive agents that learn from experience, and improve their performance over time
- purely reactive systems advertise the fact that the overall behaviour emerges from the interaction with the environment, but this implies hard to engineer software components for fulfilling a given task
- while effective agents having a small number of internal rules can be generated, it is much more difficult to build agents with predefined complex environmental situations handling mechanisms.

In [HSP08] social relationships are modelled using a fuzzy-agent model. Social relationships like friendship are governed by the so called "proximity principle" according to which two individuals are more likely to become friends if they are similar to each other. But concepts like similarity, friendship and proximity are rather abstract and difficult to define in a fixed and precise manner. This in why, in their agent based model, the authors have chosen to model these concepts using fuzzy logic.

In [CXC04] an ant-based clustering algorithm using the ASM (Ants Sleeping Model) model is presented. In this approach, an ant may be in any one of the two possible states on a 2D grid: active or sleeping. When the ant's fitness is low, it has a higher probability to wake up and start searching for a more comfortable position for sleeping. When it finds a position where its fitness is high enough the ant has a higher probability to stop there and switch to sleeping state.

Based on ASM, the authors present an Adaptive Artificial Ants Clustering Algorithm (A^4C) [CXC04] in which each artificial ant is a simple agent representing an individual data object. According to Chen et al. "the whole ant group dynamically self-organizes into distinctive, independent subgroups within which highly similar ants are closely connected. The result of data objects clustering is therefore achieved." [CXC04]. However, by using local information only the risk of getting trapped into local optimum solutions exists.

In [CDG07] a Stigmergic Agent System (SAS) combining the strengths of Ant Colony Systems and Multi-Agent Systems concepts is proposed. The agents from the SAS are using both direct and indirect communication. By using direct communication the risk of getting trapped in local optima is lower. However, as showed in [SCCK04], most ant-based algorithms can be used only in a first phase of the clustering process because of the high number of clusters that are usually produced. In a second phase a k-means-like algorithm is often used.

In [SCCK04], an algorithm in which the behaviour of the artificial ants is governed by fuzzy IF-THEN rules is presented. According the authors, their approach does not need the number of clusters as an input parameter which is the general case in ant-based clustering algorithms. However, in their approach, the dataset does not need any initial partitioning. The ants are capable to make their own decisions about picking up items. Hence the two phases of the classical ant-based clustering algorithm are merged into one, and k-means becomes superfluous.

In the approaches from [GP10, GP11a] fuzzy agents are employed for solving the clustering problem. Agent moves are expressed by fuzzy IF-THEN rules and hence hybridization with a classical clustering algorithm is needless.

2.3.2 The soft agent model

A soft agent is an intelligent agent that has to deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both. Figure 2.1 gives an abstract view of a soft agent.

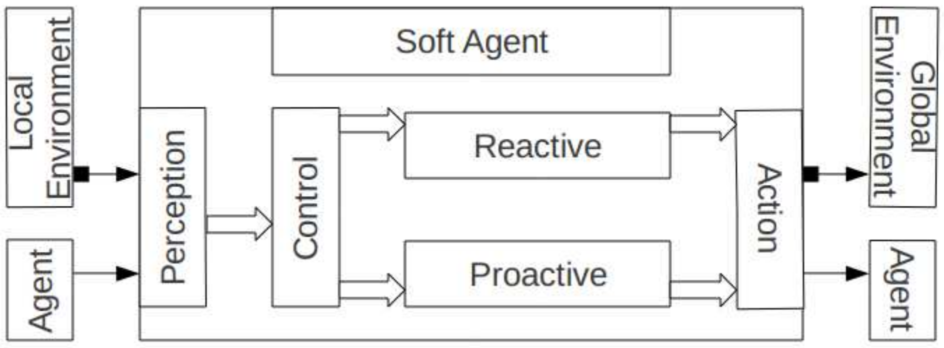


Figure 2.1: A soft agent senses its local environment and acts on the global environment.

An important property of the soft agents is that they only can sense their local environment. They can communicate with remote agents, but their vision is limited to a local neighbourhood and they maintain little information about their state. Nevertheless they can act on the global environment.

As seen in Figure 2.1, a soft agent perceives its local environment though the *Perception* layer. This layer is also responsible for listening to messages from other, possibly remote, agents. The *Controller* layer is responsible for deciding which of the following layers should have control over the agent. The control layer can be implemented as a set of control rules which can also act as a filter suppressing information from sensors. The reactive layer provides an immediate response to changes that occur in the local environment. Roughly speaking, it implements a mapping *situation* $\rightarrow$ *action*. The *Procative* layer achieves the agent's proactive behaviour, it ensures that the agent reaches its goal. The *Action* layer is responsible for executing the selected action on the environment (local or global) and with dispatching messages to other agents.

We will consider that the environment may be in any of the states given by:

$$S = \{s_1, s_2, \ldots, s_n\} \tag{2.4}$$

Let us remark that a simplifying assumption has been made for the environment being considered discrete. But any continuous environment can be modelled by a discrete environment to any desired degree of accuracy [Woo09].

The set of actions an agent may choose from is given by:

$$A = \{a_1, a_2, \ldots, a_n\} \tag{2.5}$$

At time t the environment is in some state s_t, and the agent picks an action $a_t \in A(s_t)$ to be executed. As a result of this action, the agent receives a reward $r_{t+1} \in \mathbb{R}$ and the environment reaches a new state s_{t+1}. Given the new state, the agent chooses another action to be executed and so on.

Thus the interaction between the agent and its environment is a sequence of environment state-reward pairs and actions:

$$h : (s_0, r_0) \xrightarrow{a_0} (s_1, r_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} (s_u, r_u) \tag{2.6}$$

In order to represent the effect of an agent's actions over the environment, the following a state transformer function is considered:

$$env : S \times \mathbb{R} \times A \to \mathcal{P}(S \times \mathbb{R}) \tag{2.7}$$

According to this definition environments are assumed to be history dependent. This means that the next state of an environment is determined by the action chosen by the agent given the current state, the reward and also by the earlier actions made by the agent. This behaviour is thus non-deterministic. In other words, the result of performing an action given a state is uncertain.

So the environment may be formally written as a triple $Env = \langle S, (s_0, r_0), env \rangle$, where S represent the set of all possible states of the environment, s_0 is the starting state, r_0 is the initial reward and env is the state transformer function.

We introduce the abstract soft agent model which as a function which assigns actions to sequences of state-reward pairs:

$$agent : (S \times \mathbb{R})^* \to A \tag{2.8}$$

So, roughly speaking, a soft agent chooses its next action based on its previous experience, i.e., previous environment states.

If $agent : (S \times \mathbb{R})^* \to A$ is an agent, $env : S \times \mathbb{R} \times A \to \mathcal{P}(S \times \mathbb{R})$ is an environment, s_0 is the initial state and r_0 is the initial reward then the sequence:

$$h : (s_0, r_0) \xrightarrow{a_0} (s_1, r_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} (s_u, r_u) \tag{2.9}$$

is a possible history of the given agent in the given environment if and only if the following conditions hold:

1. $\forall u \in \mathbb{N}, a_u = agent(((s_0, r_0), (s_1, r_1), \ldots, (s_u, r_u)))$
2. $\forall u \in \mathbb{N}$ *such that* $u > 0, (s_u, r_u) \in env(s_{u-1}, r_{u-1}, a_{u-1})$

The set of all such possible histories will be denoted with H.

Now that we defined the abstract agent model we may go further in describing the main function of each of its layers as shown in Figure 2.1.

The *Perception* layer has two main input channels: one for the information received from the local environment and the other for the messages received from other agents. The function *see* will map environment states to percepts and the function *listen* will map messages received from other agents to percepts. Let us denote with *Per* the set of percepts and with M the set of messages and then the functions *see* and *listen* are:

$$see : S \to Per \tag{2.10}$$

$$listen : M \to Per \tag{2.11}$$

The perceptions are sent further to the *Control* layer which is responsible for deciding which is the layer that should take control over the agent — the *Reactive* layer or the *Proactive* layer. The *Control* layer could also be used as a filter for sensory information. This layer may be implemented as a set of control rules like:

$$if\ perception\ instanceof(M)\ then\ activate\ proactive\ layer \tag{2.12}$$

The rules could be updated at runtime through messages from other agents or from a supervising authority ensuring thus an adaptive behaviour of the activation layers.

The *Reactive* layer is responsible for handling sensory data received from the local environment. From a highly abstract point of view, the *Reactive* layer could be seen as an agent with states [Woo09], like in Figure 2.2

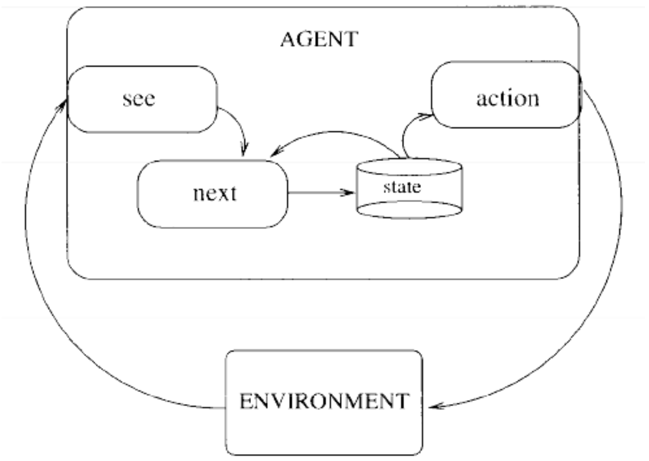


Figure 2.2: A state agent. Image source: [Woo09].

The *see* function from Figure 2.2 is the one defined in relation (2.10). The action-selection function is:

$$action : I \rightarrow A, \tag{2.13}$$

where I be the set of all internal states of the agent. The function *next*

$$next : I \times Per \rightarrow I. \tag{2.14}$$

is a mapping from internal states and percepts to internal states.

The behaviour of a state-based agent can be summarized as follows [Woo09]. The agent starts in some initial internal state i_0; it then observes its environment state s, and generates a percept $see(s)$. The internal state of the agent is then updated via the *next* function, becoming set to $next(i_0, see(s))$. The action selected by the agent is then $action(next(i_0, see(s))$. This action is then performed, and the agent enters another cycle, perceiving the world via *see*, updating its state via *next*, and choosing an action to perform via *action*.

In the soft agent case the *next* function may be replaced by a fuzzy inference system [GC10]. The process of fuzzy inference implies formulating a mapping from a given input to an output using fuzzy logic elements like membership functions, logical operators, fuzzy IF-THEN rules [Mat].

Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modelling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously) fuzzy systems [Mat].

In the following the fuzzy inference process will be explained though an example of the two input, one output, three rule tipping problem taken from [Mat]. The basic tipping problem is the following:

given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?

The example uses a process which is slightly different from Mamdani's fuzzy inference method [MA75] — among the first control systems built using fuzzy set theory where intended to control a steam engine and boiler combination by synthesizing a set of linguistic control rules obtained from experienced human operators. [Mat].

The basic structure of the example is shown is Figure 2.3.

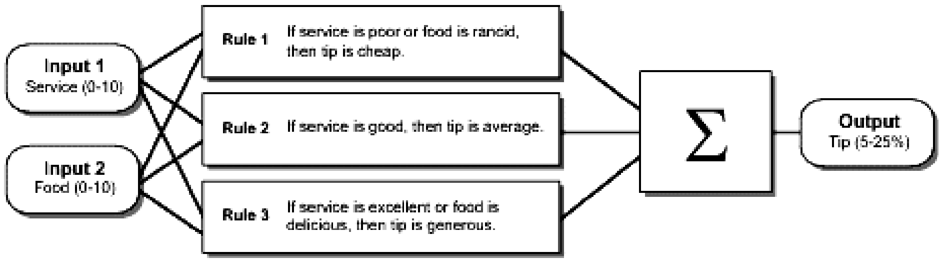

Input 1
Service (0-10)
Input 2
Food (0-10)
Rule 1
If service is poor or food is rancid, then tip is cheap.
Rule 2
If service is good, then tip is average.
Rule 3
If service is excellent or food is delicious, then tip is generous.
Σ
Output
Tip (5-25%)

Figure 2.3: Dinner for two. Two inputs, one output, three rules. Image source: [Mat].

Information flows from left to right, from two inputs to a single output. The parallel nature of the rules is one of the more important aspects of fuzzy logic systems. Instead of sharp switching between modes based on breakpoints, logic flows smoothly from regions where the system's behaviour is dominated by either one rule or another.

The fuzzy inference process takes place in the following steps: fuzzification of the input variables, application of the fuzzy operator (AND or OR) in the antecedent, implication from the antecedent to the consequent, aggregation of the consequents across the rules and defuzzification. The process will be briefly described bellow, step by step.

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. After the inputs are fuzzified, one knows the degree to which each part of the antecedent is satisfied for each rule. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. Every rule may have a weight (a number between 0 and 1), which is applied to the number given by the antecedent. Generally, this weight is 1 (as it is for this example) and thus has no effect at all on the implication process. After proper weighting has been assigned to each rule, the implication method is implemented. A consequent is a fuzzy set represented by a membership function, which weights appropriately the linguistic characteristics that are attributed to it. The input for the implication process is a single number given by the antecedent, and the output is a fuzzy set. The process described so far (taken from [Mat]) can be viewed in Figure 2.4.

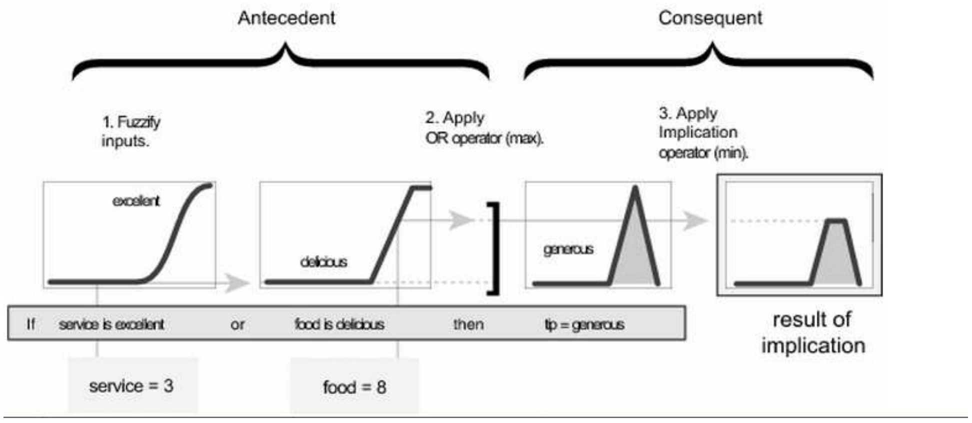


Figure 2.4: The first input is $service = 3$ and the second input is $food = 8$. The result of the implication process is a fuzzy set. Image source: [Mat].

Because decisions are based on the testing of all of the rules in a FIS, the rules must be combined in some manner in order to make a decision. Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once for each output variable, just prior to the fifth and final step, defuzzification. The input of the aggregation process is the list of truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable. So the input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single number. Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of area under the curve [Mat]. The entire inference process is best described by Figure 2.5.

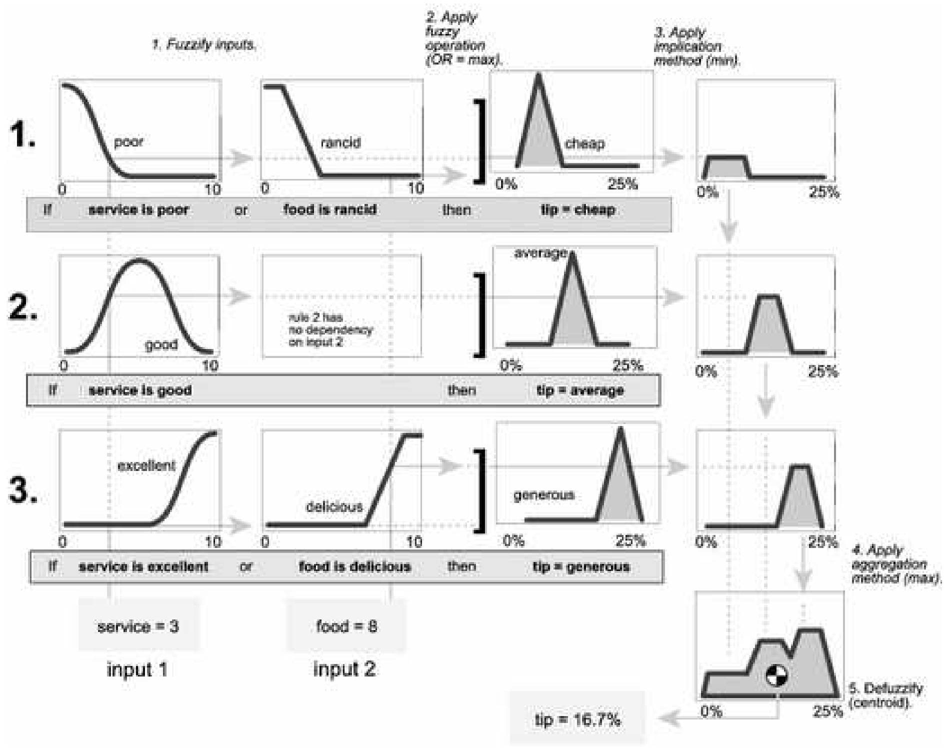

1. Fuzzify inputs.
2. Apply fuzzy operation (OR = max).
3. Apply implication method (min).
1.
poor
rancid
cheap
If service is poor or food is rancid then tip = cheap
2.
average
good
rule 2 has no dependency on input 2
If service is good then tip = average
3.
excellent
delicious
generous
If service is excellent or food is delicious then tip = generous
4. Apply aggregation method (max).
service = 3
input 1
food = 8
input 2
5. Defuzzify (centroid).
tip = 16.7%

Figure 2.5: The fuzzy inference process. Image source: [Mat].

While the *Reactive* layer is responsible for the actions done by the agent as a response to changes in the local environment, the *Proactive* layer is the one that ensures us that the agent does what is suppose to do. And now the question is how to specify the task that the agent has to carry out. Since writing a program to be executed by the agent is out of question because there may be some unforeseen situations in which only an agent could be able to respond accordingly. It may do so if we told him what to do instead of telling him how to do a certain task. Agents are told what to do in an indirect manner, through a performance measure. In the case of soft agents fitness functions are associated to states of the environment and the agent has to maximize its fitness. The fitness function is defined in the following way:

$$F : H \to \mathbb{R}, \tag{2.15}$$

where H is the set of histories. So a fitness function associates a real value to every history.

The task that an agent has to accomplish is to maximize its fitness so an optimum is then reached for:

$$\arg\max_{agent} \sum_{h \in H(agent, Env)} F(h) P(h|agent, Env), \tag{2.16}$$

where $P(h|agent, Env)$ denotes the probability that the history h occurs when the *agent* is placed in environment *Env*.

While the fitness function evaluates what is good on a long term, the reward function evaluates the quality of a certain state. So the reward function shows which would be the good and the bad actions to be taken from a certain state. The reward function could be used to modify an agent's policy π, i.e, the agent's behaviour. Thus the reward function is defined as

$$Q : S \times A \to \mathbb{R} \tag{2.17}$$

and maps a real value, a reward, to a state-action pair.

2.3.3 Possible extensions and applications

By bypassing the reactive layer and sending state-reward pairs to the *Proactive* layer the soft agent may be used as a reinforcement learning agent as discussed in Section 2.3.3.1. Applications in Multi-objective optimization problems are discussed in Section 2.3.3.2.

2.3.3.1 Reinforcement learning

Reinforcement learning is a formal computational model inspired by the way in which animals acquire complex behaviour: they learn to obtain rewards and to avoid punishments. A learning agent repeatedly observes the state of its environment and then chooses which actions to perform. Performing an action changes the state of the environment and the agent receives a numeric reward. The agent must learn how to choose actions in order to maximize the total reward on a long term. There are two main approaches for this:

- learning the utility function
- learning the reward (action-value) function

We will focus on the second approach, that is, learning the reward function. This is also known as Q-learning [Wat89].

A soft agent has a reward function $Q : S \times A \to \mathbb{R}$. This function is part of the *Proactive* layer. In order to deal with a purely reinforcement learning agent, the *Control* layer should be programmed to only send the state-reward pairs to the *Proactive* layer and not to the *Reactive* layer.

Each time the state has changes new values are calculated for each combination of a state $s \in S$, and action $a \in A$ so the the algorithm basically consists in the following update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t)(1 - \alpha(s_t, a_t)) + \alpha(s_t, a_t)[R(s_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})], \tag{2.18}$$

where $R(s_{t+1})$ is the reward observed from s_t, $\alpha(s_t, a_t)$ is the learning rate and $0 \leq \gamma < 1$ is the discount factor. A learning rate equal to 0 will make the agent not learn anything, while a learning rate equal to 1 would make the agent consider only the most recent information. A discount factor equal to 0 will make the agent "opportunistic" by considering only current rewards. The convergence of the algorithm was presented by Watkins and Dayan [WD92].

2.3.3.2 Multi-objective optimization

When an optimization problem involves more than one objective function, the task of finding one (or more) optimum solution(s), is known as the Multi-Objective Optimization Problem (MOOP). In problems characterized by more than one conflicting objective, there is no single optimum solution; instead there exists a set of solutions which are all optimal, called the Optimal Pareto front. A general multi-objective optimization problem is defined as follows (minimization case) [NMM06]:

$$\begin{array}{ll} min & O(x) = [o_1(x), o_2(x), \ldots, o_M(x)] \\ subject\ to & E(x) = [e_1(x), e_2(x), \ldots, e_L(x)] \geq 0 \\ & x_i^{(L)} \leq x_i \leq x_i^{(U)}, i = 1 \ldots N, \end{array} \tag{2.19}$$

where $x = (x_1, x_2, \ldots, x_N)$ is an array of N decision variables, M is the number of objectives o_i, L is the number of constraints e_j and $x_i^{(L)}$ and $x_i^{(U)}$ are respectively the lower and upper bound for each decision variable x_i. Two different solutions are compared using the concept of dominance, which induces a strict partial order in the objective space O. Here a solution a is said to dominate a solution b if it is better or equal in all objectives and better in at least one objective . For the minimization case we have [NMM06]:

$$O(a) \prec O(b) \; iff \left\{ \begin{array}{ll} o_i(a) \leq o_i(b) & \forall i \in 1, \ldots, M \\ \exists j \in 1, \ldots, M & o_j(a) < o_j(b) \end{array} \right. \tag{2.20}$$

In many situations it is actually the case that multi-objective problems need to deal with two conflicting objectives: maximizing profit and minimizing the cost of a product, maximizing performance and minimizing fuel consumption of a vehicle and so on. A soft agent could be employed for solving a two-objective optimization problem in the following way: one objective could be assigned to the *Reactive* layer and the other one to the *Proactive* layer. Actually an agent's behaviour is optimal if it finds a right balance between its two stances: reactive and pro-active. So the task of the agent is to minimize the function $O(x) = [o_1(x), o_2(x)]$ where $o1$ is the function performed by the *Reactive* layer and $o2$ is the fitness function of the *Proactive* layer. For multiple objectives more agents could be launched.

2.4 A new approach to the set covering problem

The set covering problem is a classical problem in computer science and complexity theory and it serves as a model for many real-world applications especially in the resource allocation area. In an environment where the demands that need to be covered change over time, special methods are needed that adapt to such changes. We reformulate the set covering problem as a clustering problem where the within cluster sum of squared errors to be minimized corresponds to the cost associated to a certain set covering that needs to be minimal. We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to be included in a certain cluster in the attempt to either maximize its cover or minimize the cost. We have introduced the soft agent model in order to encapsulate this behaviour. Initial tests suggest the potential of our approach.

2.4.1 SCP overview

The Set Covering Problem (SCP) is a classical problem in computer science and complexity theory and it serves as a model for many applications in the real world like: facility location problem, airline crew scheduling, resource allocation, assembly line balancing, vehicle routing, information retrieval etc. Let us consider a set X and a family F of subsets of X such that every element from X belongs to at least one subset from F. The set covering problem is the problem of finding a minimum number of subsets from F (or subsets of minimum cost) such that their union is the set X.

The set covering problem is NP-hard and it has been addressed in many ways over time [GP08, CSM$^+$11, CM01, BC96, AAA03, AM10]. A straightforward solution is the greedy approximation algorithm [CLRS09]. This method selects at each step a set from F that covers most of the still uncovered elements. In [BC96] the set covering problem is addressed using a genetic algorithm by using an n-bit binary string as the chromosome structure, where n is the number of columns from the SCP dataset. In order to mark that column i is in the solution the i^{th} is set to 1. The authors designed heuristic operators that transform invalid solutions (obtained after applying genetic operators) to valid ones. The binary tournament selection was chosen as the method for parent selection and for crossover the authors propose a so-called fusion operator taking into account both the structure and the relative fitnesses of the parents. A variable mutation rate specified in [BC96] is used arguing that the genetic algorithm is more effective. Computational experiments on a large set of randomly generated problems show that the genetic algorithm based approach is capable of producing high quality solutions. In

[AAA03] the online version of SCP is considered. In the online version an adversary gives elements to the algorithm from one by one. Whenever a new element is arriving, the algorithm has to cover it. The elements of X and the members of F are known in advance to the algorithm, but the set $X^{'} \subseteq X$ of elements given by the adversary isn't and the task is to minimize the total cost of the sets chosen by the algorithm. In [CSM+11] the set covering problem is addressed using an ant colony optimization algorithm together with a new transition rule. The authors have also used a look-ahead mechanism for constraint consistency checking such that new elements are added to the solution if they do not produce conflicts with the next element to be chosen. In [AM10] a clustered variant of the SCP is defined. In the Clustered-SCP the subsets are partitioned into k clusters and a fixed cost is associated to each cluster. So the objective is to find a cover that minimizes the sum of subsets costs plus the sum of fixed cluster costs.

The classical set covering problem can be formulated as a clustering problem where the within cluster sum of squared errors to be minimized corresponds to the cost associated to a certain set covering that needs to be minimal. We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to be included in a certain cluster in the attempt to either maximize its cover or minimize the cost.

In machine learning, clustering is an example of unsupervised learning because it does not rely on predefined classes and class-labelled training examples. So it could be said that clustering is a form of learning by observation, as opposed of being a form of learning by examples. In data analysis, efforts have been conducted on finding efficient cluster analysis methods for large datasets. The main requirements for a good clustering algorithm would be the scalability of the method, its effectiveness in clustering complex data shapes and types, dealing with high-dimensional data, and handling mixed data, i.e., both numerical and categorical data.

There is a great variety of clustering algorithms to choose from each with its own strengths and weaknesses. In [GP11c] an incremental clustering algorithm is presented. Incremental clustering algorithms in general do not rely on the in-memory dataset and they build the solution gradually, with every new incoming data item. The idea behind is that it is possible to consider one instance at a time and assign it to existing clusters without significantly affecting the already existing structures. Only the cluster representations need to be kept in memory so not the entire dataset and thus the space requirements for such an algorithm are very small. Whenever a new instance is considered an incremental clustering algorithm would basically try to assign it to one of the already exiting clusters. Such a process is not very complex and therefore the time requirements for an incremental clustering algorithm are also small.

Let us consider a set X and a family of subsets of X

$$F = S_1, S_2, \ldots, S_n. \tag{2.21}$$

The classical set covering problem is the problem of finding a minimum cardinality $J \subseteq 1, \ldots, n$ such that

$$\cup_{j \in J} S_j = X. \tag{2.22}$$

The set covering problem is NP-hard [CLRS09] and it can be approached by the following greedy approximation algorithm:

Algorithm 2.4.1 SCP-Greedy

```
1: $U \leftarrow X$
2: $C \leftarrow \emptyset$
3: while $U \neq \emptyset$ do
4:     find $S_j$ such that $\mid S_j \cap U \mid$ is maximal, where $j \in J$
5:     $U \leftarrow U - S_j$
6:     $C \leftarrow C \cup \{S_j\}$
7: end while
```

The algorithm iteratively selects the set S_j which covers the most still uncovered elements from X. The set U contains the still uncovered elements and at the beginning it is equal to X. At each step the algorithm is choosing a set S_j that covers most of the elements that are left uncovered up to this point. The set S_j is added to C and in the end C will contain a subfamily of sets that cover X. The algorithm finishes when U is empty.

The minimum cost set covering problem considers a cost c_j for each S_j and the problem is to find a cover for X and to minimize the sum of costs $\sum_{j \in J} c_j$.

2.4.2 Incremental SCP

In our model the input is an $m \times n$ incidence matrix A, where $m = \mid X \mid$ and each column corresponds to a set S_j with $j \in \{1, \ldots, n\}$. Each column j has a corresponding cost $c_j > 0$. We say that a column j covers a row i if $a_{ij} = 1$. Let x_j be a binary decision variable which has the value 1 if column j is chosen and 0 otherwise. Then the set covering problem can be defined as minimize (2.23) subject to (2.24) [CSM+11].

$$f(x) = \sum_{j=1}^{n} c_j x_j, \tag{2.23}$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1, \forall i = \overline{1, n}, \tag{2.24}$$

$$x_j = \begin{cases} 1, & if\ column \text{ j } is\ chosen \\ 0, & otherwise. \end{cases} \tag{2.25}$$

Clustering can be seen as the problem of finding "meaningful" groups in data and a way to do this is by minimizing a certain objective function. The set covering problem can be formulated as a clustering problem in the following way: assign columns from A to clusters such that the function from (2.23) is minimized and the cluster is valid, i.e., the relation (2.24) holds.

We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm considers one instance at a time and assigns it to one of the existing clusters without significantly affecting the already existing structures. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to join a certain cluster in the attempt to either maximize the cluster cover or minimize its the cost. These two objectives are rather conflicting and this brings a great deal of imprecision and uncertainty in the whole reasoning process. That is why we have employed *soft agents* in this matter. A *soft agent* is an intelligent agent that has to deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both as explained in Section 2.3.2.

For the set covering problem we define the fitness of an agent a_i given a cluster c_k in the following way:

Definition 2.4.1 *The fitness of an agent a_i given a cluster c_k is:*

$$f(a_i, c_k) = \frac{\mid rows(a_i) - rows(c_k) \mid}{m}, \tag{2.26}$$

where:

- *$rows(a_i)$ denotes the set of rows covered by agent a_i,*
- *$rows(c_k)$ denotes set of rows covered by cluster c_k,*

- *m is the total number of rows.*

The algorithm considers one column at a time and encapsulates it in a agent. In the first part an initial valid cluster will be built using a greedy approach (a cluster is valid if it covers the considered set). After this step every newly considered agent will decide weather to try to maximize the cover of one of the existing clusters or to minimize the cost using the following control function: $f^{\lambda}(a_i, c_k)$. The parameter λ is increasing in time thus leading to agents that act upon minimizing the cost rather than maximize the cover. A pseudo-code of the algorithm is sketched in Algorithm 2.4.2.

Algorithm 2.4.2 Incremental SCP

```
1: initialize parameters
2: find a proper cluster $c_0$ by randomly selecting agents
3: $C \leftarrow C \cup \{c_0\}$
4: while $condition()$ do
5:    $U \leftarrow U \cup \{createAgent()\}$
6:    while $U \neq \emptyset$ do
7:       if $reactive(a_i)$ then
8:          assign $a_i$ to a non-valid cluster $c_k$ or create a new cluster
9:       else
10:         $\{S_j, c_k\} \leftarrow tryReplace(a_i)$
11:         $U \leftarrow U \cup S_j$
12:      end if
13:   end while
14:   $U \leftarrow U \cup discardWorseCluster()$
15:   update parameters
16: end while
```

The algorithm continuously receives new items (columns) to be clustered and an agent encapsulates each item. The agent is placed in the collection U of unclustered items. Starting from line 6 the algorithm repetitively considers agents from the collection U. The agent decides to behave in a reactive or proactive manner based on its control function. If it decides to behave reactively, i.e., maximize the cover then the agent attempts to find a non-valid cluster to be included in. If no such cluster is found then a new cluster is created containing this agent. The new cluster is added to C and a_i is removed from U. A cluster is valid if all the rows are covered (2.24). If on the other hand the agent wants to minimize the cost then it will try to replace agents from a valid cluster c_k. If the replace operation took place then the set of replaced agents S_j is added to the unclustered collection. After all agents from U have been added to some clusters the cluster with the worse cost is discarded and its agents are added to the unclustered collection. Parameters like λ or acceptable cost threshold may be updated at this point. The whole process starts over from line 4 and ends when some condition is met (like a good-enough solution is found or a certain number of iterations have been completed).

Initial tests have been done on the *scp*410 dataset from the $OR-Library$ [Bea] which is a collection of test data sets for a variety of operation research (OR) problems, including SCP. The *scp*410 dataset has 200 rows and 1000 columns in the following format: number of rows (m), number of columns (n), the cost of each column $c(j), j = 1, \dots, n$ and then for each row $i (i = 1, \dots, m)$: the number of columns which cover row i followed by a list of the columns which cover row i.

We have obtained near-optimum solutions as shown in Figure 2.6. Reference [CSM+11] mentions that the optimum cost is 514. After 15000 iterations we have obtained the cost 561. By one iteration we mean the execution of the loop from *line* 4 of Algorithm 2.4.2. At each iteration we read one new agent until all agents have been read. The *scp*410 dataset may produce at most 1000 agents. At iteration 1000 we have already obtained the cost 774 so at the beginning the algorithm finds good solutions fast. Unfortunately, in time, it slows down. We are confident that a careful analysis of the implementation will lead to a decrease in the running time. Tests on other datasets from [Bea] are ongoing.

We have developed an incremental clustering algorithm in order to address the set covering prob-

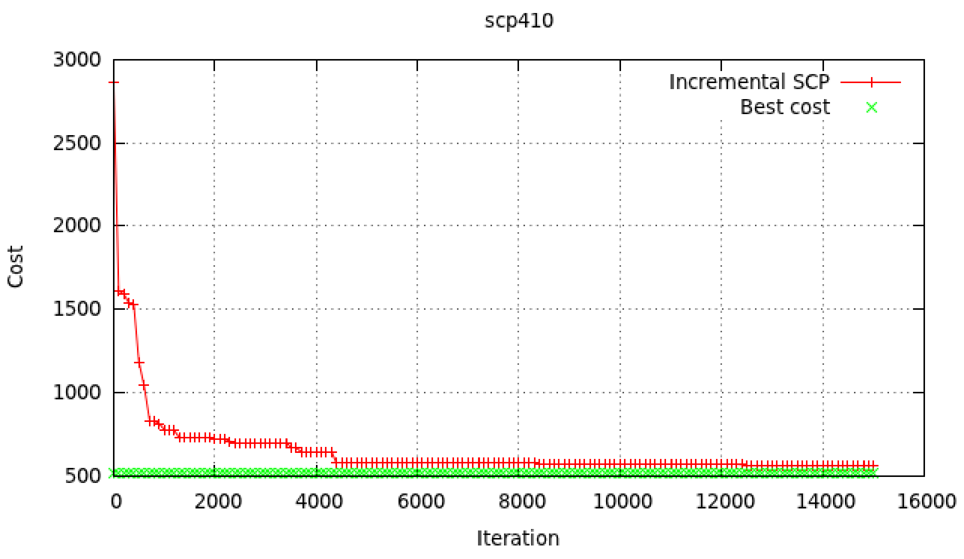


Figure 2.6: Execusion on scp410.

lem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to join a certain cluster in the attempt to either maximize the cluster cover or minimize its the cost. We have used soft agents in order to deal with the two conflicting objectives: maximize cover and minimize cost. As in any approximation algorithm an optimal solution is not guaranteed to be found, the purpose being to find reasonably good solutions fast enough. Ongoing tests on large datasets [Bea] suggest promising results and lead us to also consider similar problems like the set partitioning problem.

2.5 Conclusions and future work

In this chapter we have presented our contributions to NP optimization problems namely to the travelling salesman problem and to the set covering problem. These approaches have been presented in our original papers [CDG07, GP12].

We have seen that the proposed SAS approach for solving TSP is a powerful optimization technique that combines the advantages of two models: Ant Colony Systems and Multi-Agent Systems. Interoperation between agents is based on both indirect communication — given by pheromone levels — and direct knowledge sharing, greatly reducing the risk of falling into the trap of local minima. Experimental results on standard datasets outline the advantage of the approach over the classical Ant Colony Systems. We have also developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to join a certain cluster in the attempt to either maximize the cluster cover or minimize its the cost. We have used soft agents in order to deal with the two conflicting objectives: maximize cover and minimize cost. As in any approximation algorithm an optimal solution is not guaranteed to be found, the purpose being to find reasonably good solutions fast enough. Ongoing tests on large datasets [Bea] suggest promising results.

For each original approach proposed in this chapter we have emphasized improvement possibilities and possible future extensions.

As future research directions we intend to improve the approaches presented in this chapter, to extend the evaluation of the proposed techniques and to investigate and develop other computational models for addressing NP-hard problems.

Ongoing research focuses on numerical experiments to demonstrate the robustness of the proposed model. The SAS method has to be further refined in terms of types of messages that agents can directly exchange. Furthermore, other metaheuristics are investigated with the aim of identifying additional potentially beneficial hybrid models.

Chapter 3

New approaches to unsupervised learning

This chapter begins with a short overview of various agent-based clustering approaches in Section 3.1. The rest of the chapter is entirely original and presents our contribution to agent-based clustering, particularly in two main directions: ASM-based batch clustering and incremental clustering. We are focusing on developing clustering algorithms that allow the discovery and analysis of hybrid data.

The unsupervised learning approaches presented in this chapter are original works published in [GP10, GP11a, GP11b, Găc11, GP11c].

The chapter is structured as follows. In Section 3.1 a short overview of various agent-based clustering approaches is presented. We approach the idea of agent-based cluster analysis in Section 3.2. Each data item is represented by an agent placed in a two dimensional grid. The agents will group themselves into clusters by performing simple moves using some local environment information, the parameters being selected and updated adaptively. This behaviour based on ASM (Ant Sleeping Model) where an agent may be either in an active state or in a sleeping state. In order to avoid the agents being trapped in local minima, they are also able to directly communicate with each other. Furthermore, the agent moves are expressed by fuzzy IF-THEN rules and hence hybridization with a classical clustering algorithm is needless. The proposed fuzzy ASM-based clustering algorithm is presented in Section 3.2.1. In this model data items to be clustered are represented by agents that are able to react according to the changes in the environment, namely the number of neighbouring agents. However a change in the data item itself is not handled at runtime. An extension to a context-aware system would be beneficial in many practical situations. In general, context-aware systems could greatly change the way we interact with the world — they could anticipate our needs and advice us when taking some decisions. In a changing environment context-awareness is undoubtedly beneficial. Such systems could make much more relevant recommendations and support decision making. An extension to a context-aware approach is presented in Section 3.2.2. Case studies for both approaches including experiments on standard datasets [Iri88, Win91] are presented in Section 3.2.3. The idea behind incremental clustering is that it is possible to consider one instance at a time and assign it to existing clusters without significantly affecting the already existing structures. Section 3.3 presents an incremental clustering approach based on ASM. In incremental clustering only the cluster representations need to be kept in memory so not the entire dataset and thus the space requirements for such an algorithm are very small. Whenever a new instance is considered an incremental clustering algorithm would basically try to assign it to one of the already exiting clusters. Such a process is not very complex and therefore the time requirements for an incremental clustering algorithm are also small. The fuzziness of the approach allows the discovery of hybrid data. Experimental evaluation on standard datasets [Iri88, Win91] are presented in Section 3.3.3. Section 3.4 outlines the conclusions of the chapter and indicates some research directions that will be followed.

The original contributions of this chapter are:

- A fuzzy ASM-based clustering algorithm (Section 3.2.1) [GP10, Găc11].
- A context-aware fuzzy clustering algorithm (Section 3.2.2) [GP11a, GP11b].

- An incremental fuzzy clustering algorithm (Section 3.3) [GP11c].
- Experimental evaluation of the algorithms on standard datasets (Section 3.2.3 and Section 3.3.3) [GP10, Găc11, GP11a, GP11b, GP11c].
- The discovery and analysis of hybrid data (Section 3.2.3 and Section 3.3.3) [GP11a, GP11b, GP11c].
- The applicability of the fuzzy ASM-based methods in clustering web search results (Section 3.2.3) [GP10, Găc11].

3.1 Agent-based unsupervised learning

Several clustering algorithms exist each with its own strengths and weaknesses. Some algorithms need an initial estimation of the number of clusters (k-means, fuzzy c-means) while others could often be too slow (agglomerative hierarchical clustering algorithms). Ant-based clustering algorithms often require hybridization with a classical clustering algorithm such as k-means [SCCK04].

In [CXC04] the authors present an ant-based clustering algorithm. It is based on the ASM (Ants Sleeping Model) approach. In the ASM model an ant may be in any one of the following two possible states: active and sleeping. When the ant's fitness is low, the probability of switching to active state is higher. If this happens the ant will leave its position and it will start to search for a more comfortable position for sleeping, i.e., a position where its fitness is high enough. When a position which increases the ant's fitness has been located it has a higher probability to switch to sleeping state and stay at that position until the environment becomes less hospitable again.

Based on ASM, the authors present an Adaptive Artificial Ants Clustering Algorithm (A^4C) [CXC04] in which every ant is a simple agent that corresponds to an individual data item. "The whole ant group dynamically self-organizes into distinctive, independent subgroups within which highly similar ants are closely connected. The result of data objects clustering is therefore achieved." [CXC04]. However, by using local information only the risk of getting trapped into local optimum solutions exists.

Another approach to data clustering with ants is the one from the BM [DGF$^+$91] and LF [LF94] models which use the ants' behaviour of piling corpses. In this case ants pick up the corpses and put them in piles preferring larger piles. Items can be moved from one pile to another. This model has a rather high computational time cost because of the idle moves required until an item is found.

In [CDG07] a Stigmergic Agent System (SAS) combining the strengths of Ant Colony Systems and Multi-Agent Systems concepts is proposed. The agents from the SAS are using both direct and indirect communication. By using direct communication the risk of getting trapped in local optima is lower. However, as showed in [SCCK04], most ant-based algorithms can be used only in a first phase of the clustering process because of the high number of clusters that are usually produced. In a second phase a k-means-like algorithm is often used.

In [SCCK04], an algorithm in which the behaviour of the artificial ants is governed by fuzzy IF-THEN rules is presented. As in any typical ant-based clustering algorithm, the advantage is that an initial data partitioning is not required and also the number of clusters does not need to be known a prori. The ants are capable to make their own decisions about picking up items and then the two phases of the classical ant-based clustering algorithm are merged into one, which makes k-means superfluous [SCCK04].

In [GP11a], the clustering problem is approached by the idea of context-aware ASM agents. The agents are able to detect changes in the environment and adjust their moves accordingly. The advantage of this approach is that it enables the ants to communicate directly like in [CDG07] therefore breaking the neighbourhood boundaries and thus decreasing the chance of ants to get trapped in local minima. The fuzzy IF-THEN rules governing the agents' movements are also allowing the agents to go beyond the neighbourhood limits; for example in the case of two very different (VD) agents it makes no point to keep them in a reachability distance; and it makes sense that two very different (VD) agents should move further away from each other than two different (D) agents do. Thus the system behaves more naturally. The agents are able to adapt their movements if changes in the environment would occur and this is an important feature in real-time systems, wireless sensor networks or data streams.

The skeleton of our approach [GP10] is based on the ASM-like algorithm from [CXC04] embellished with features from [CDG07] and [SCCK04]. In ASM (Ants Sleeping Model), an ant may be either in active state or in sleeping state. When the artificial ant's fitness is low, it has a higher probability to wake up and stay in active state. It will thus leave its original position to search for a more secure and comfortable position to sleep. When an ant locates a comfortable and secure position, it has a higher probability to sleep unless the surrounding environment becomes less hospitable and activates it again. Ants use only local information in order to switch between states and in time the whole group dynamically self organizes into formations within which similar ants are close to each other and therefore clustering is achieved. The definitions 3.1.1 – 3.1.6 from bellow are taken from [CXC04].

Definition 3.1.1 *The grid in ASM is a two-dimensional array*

$$G(x,y)\epsilon Z^{+}\bigcup\{0\}, \tag{3.1}$$

of all positions

$$(x,y)\epsilon[0..2\lceil\sqrt{n}\rceil-1]^{2}, \tag{3.2}$$

such that:

$$G(x,y)=\begin{cases} i & \textit{if there is an agent labelled } i \textit{ at position } (x,y)\\ 0 & \textit{otherwise}\end{cases} \tag{3.3}$$

where n is the number of agents.

Remark 3.1.1 *The grid size depends on the number of agents and thus we avoid both grid overcrowding and allocating a grid which consumes too many resources.*

Remark 3.1.2 *The ASM uses a grid topologically equivalent to a sphere grid and hence all cells are equal to each other with respect to the number of neighbours. So the cells from the margins have the same number of neighbours as the cells from the interior of the grid, i.e., one can jump from one of the northmost positions to one of the southmost positions with one step.*

Definition 3.1.2 *In ASM, each agent represents one data item. Let an agent represent a data item by using $agent_i$ to represent the i^{th} agent, and n be the number of agents. The position of an agent is represented by (x_i,y_i), namely $G(agent_i)=G(x_i,y_i)=i$*

Definition 3.1.3 *The neighbourhood of an agent is*

$$N(agent_i)=N(x_i,y_i)=\{(x,y)mod(2\lceil\sqrt{n}\rceil)|\mid x-x_i\mid\leq s_x,\mid y-y_i\mid\leq s_y\} \tag{3.4}$$

Definition 3.1.4 *The set of empty positions in the neighbourhood is*

$$L(agent_i)=L(x_i,y_i)=\{(x,y)|(x,y)\epsilon N(agent_i),G(x,y)=0\} \tag{3.5}$$

s_x and s_y are the vision limits in the horizontal and vertical direction respectively.

Definition 3.1.5 *The fitness of an agent is*

$$f(agent_i)=\frac{1}{(2s_x+1)(2s_y+1)}\sum_{agent_j\epsilon N(agent_i)}\frac{\alpha^2}{\alpha^2+d(agent_i,agent_j)^2}, \tag{3.6}$$

$$\alpha=\frac{1}{n(n-1)}\sum_{i=1}^{n}\sum_{j=1}^{n}d(agent_i,agent_j), \tag{3.7}$$

where:

- $f(agent_i)$ *represents the current fitness of agent* i.
- α *is the average distance between the agents.*

Remark 3.1.3 *The distance between two agents,* $d(agent_i, agent_j)$, *is the Euclidean distance between the two agents from the grid.*

Definition 3.1.6 *The activation probability is*

$$p_a(agent_i) = cos^{\lambda}(\frac{\pi}{2} f(agent_i)), \tag{3.8}$$

where $\lambda \epsilon R^+$ *is a parameter, and can be called agents' activation pressure.*

Remark 3.1.4 *Function* $p_a(agent_i)$ *represents the probability of the activation of the agent by the surroundings. If the fitness is low then the probability of activation is high so the agent is probably going to wake up, move and search for a better place to sleep. Conversely if the fitness is high then the probability is low so most probably the agent will stay and sleep.*

The algorithm from [CXC04] starts by randomly placing the agents on the grid in active state. The agents start to perform simple moves on the grid. Whenever it arrives in a new location the agents updates its fitness and activation probability (defined above). If the activation probability is low then the chances that the agent will stop wandering are higher. If this is the case then the agent will switch to sleeping state and unless the environment becomes less hospitable it will continue to stay in sleeping state at that position. The agents's fitness is related to the number of similar individuals in its neighbourhood. With increasing number of iterations, such movements gradually increase this leading to a clustering in data.

3.2 ASM-based clustering

In ASM (Ants Sleeping Model), an agent located on a two-dimensional grid may be in any of the following states: active or sleeping. When the agent's fitness is low, it has an increased probability to become active and start searching for a more comfortable position, where its fitness is increased. When such a position in located, the agent has an increased probability to move in a sleeping sate until the local environment becomes less hospitable and activates it again. At the beginning of every considered ASM-based clustering approach the agents are randomly placed on the grid in active state. Whenever an agent move to a new position, it will update its fitness f_{disim} and probability p_a and based on this information it can decide whether it should continue moving or not. While the p_a is high the agent is likely to stay active and continue to explore the grid. If the current p_a becomes small, the agent has a lower probability to keep exploring the grid so it may stop at the current position and switch to sleeping state. With increasing number of iterations, such movements gradually increase, eventually, making similar agents gathered within a small area and different types of agents located in separated areas. Thus, the corresponding data items are clustered.

In our ASM approaches we use Definition 3.2.1 for the fitness of an agent.

Definition 3.2.1 *The dissimilarity-based fitness of an agent is*

$$f(agent_i) = \frac{1}{v} \sum_{a_j \epsilon N(a_i)} \frac{\alpha^2}{\alpha^2 + disim(a_i, a_j) de(a_i, a_j)}, \tag{3.9}$$

$$\tag{3.10}$$

$$v = \frac{1}{(2s_x + 1)(2s_y + 1)} \tag{3.11}$$

$$\alpha = \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j=1}^{n} de(a_i, a_j), \tag{3.12}$$

where:

- *a_i and a_j respectively denote agent$_i$ and agent$_j$,*
- *$de(a_i, a_j)$ represents the Euclidean distance between the agents on the grid,*
- *$disim(a_i, a_j)$ denotes the dissimilarity between the two agents.*

The activation probability p_a is the same as the one from Definition 3.1.6.

3.2.1 Fuzzy ASM-based clustering

The agents decide upon the way they move on the grid based on their similarity with the neighbours, using fuzzy IF-THEN rules. Thus two agents can be similar (S), different (D), very different (VD). If two agents are similar they would get closer to each other. If they are different or very different they will get away from each other. The number of steps they do each time they move depend on the similarity level. So if the agents are VD they would jump many steps away from each other; if they are D they would jump less steps away from each other. In the end the ants which are S will be in the same cluster. The parameter α is the average distance between agents and this changes at each step further influencing the fitness function. The parameter λ influences the agents' activation pressure and it may decrease over time. The parameter t is used for the termination condition which could be something like $t < t_{max}$. The parameters s_x, s_y, the agent's vision limits, may also be updated in some situations.

Algorithm 3.2.1 Fuzzy ASM Clustering

```
1: initialize parameters $\alpha, \lambda, t, s_x, s_y$
2: for all agent do
3:    @ place agent at randomly selected site on the grid
4: end for
5: while not termination do
6:    for all agent do
7:       @ compute agents fitness and activate probability $p_a$ according to the definitions from above
8:       $r \leftarrow random(0,1)$
9:       if $r < p_a$ then
10:         @ activate agent and move to a site in the neighbourhood based on the similarity with the
            neighbours using fuzzy IF-THEN rules
11:      else
12:         @ stay at current site and sleep
13:      end if
14:   end for
15:   @ adaptively update parameters $\alpha, \lambda, t, s_x, s_y$
16: end while
```

In the following, the fuzzy sets S, D, VD are defined:

$$S, D, VD : X \rightarrow [0, 1] \tag{3.13}$$

$$S(x) = \begin{cases} 1 & , x \in [0, SD1] \\ (SD1 - x) + 1 & , x \in [SD1, SD2] \\ 0 & , otherwise \end{cases} \tag{3.14}$$

$$D(x) = \begin{cases} (x - SD2) + 1 & , x \in [SD1, SD2] \\ 1 & , x \in [SD2, VD1] \\ (VD1 - x) + 1 & , x \in [VD1, VD2] \\ 0 & , otherwise \end{cases} \tag{3.15}$$

$$VD(x) = \begin{cases} (x - VD2) + 1 & , x \in [VD1, VD2] \\ 1 & , x > VD2 \\ 0 & , otherwise \end{cases} \tag{3.16}$$

The limits $SD1$, $SD2$, $VD1$, $VD2$ are application specific.

In the Figure 3.2.1 a graphical representation for a fuzzy variable $Similarity$ is shown. The fuzzy sets S, D and VD corresponding respectively to the linguistic concepts $Similar$, $Different$ and $VeryDifferent$ are called the *states* of the fuzzy variable called $Similarity$.

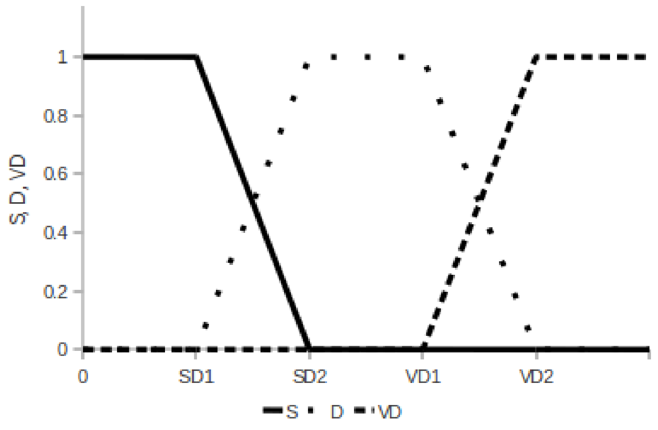


Figure 3.1: Similarity

3.2.2 Context-aware ASM-based clustering

The skeleton of this approach is based on the ASM-like algorithm from [CXC04] embellished with features from [CDG07, GP10, SCCK04].

The agents decide upon the way they move on the grid based on their similarity with the neighbours, using fuzzy IF-THEN rules. Thus two agents can be similar (S), different (D), very different (VD). If two agents are similar or very similar they would get closer to each other. If they are different or very different they will get away from each other. The number of steps they do each time they move depend on the similarity level. So if the agents are VD they would jump many steps away from each other; if they are D they would jump less steps away from each other. In the end the ants which are S will be in the same cluster. The similarity computation considers the actual structure of the data or the data density from the agent's neighbourhood; a bigger change from one agent to another translates into a certain similarity which then affects the agent's movement on the grid.

Whenever it is supposed to make a move the agent can pro-actively decide to either move based on the similarity with its neighbours, like described above, or to move based on direct communication with other agents, located possibly outside its neighbourhood.

The parameter α is the average distance between agents and this changes at each step further influencing the fitness function. The parameter λ influences the agents' activation pressure and it may decrease over time. The parameter t is used for the termination condition which could be something like $t < t_{max}$. The parameters s_x, s_y, the agent's vision limits may also be updated in some situations.

The approach from [CXC04] allows a high liberty to the ants' movements but only in their neighbourhood. The advantage of our approach is that it enables the ants to communicate directly like in [CDG07] therefore breaking the neighbourhood boundaries and thus decreasing the chance of ants

Algorithm 3.2.2 Clustering

```
1: @ initialize parameters $\alpha, \lambda, t, s_x, s_y$
2: for all agent do
3:     @ place agent at randomly selected site on the grid
4: end for
5: while not termination do
6:     for all agent do
7:         @ compute agents fitness and activation probability $p_a$ according to Definition 3.2.1 and
           Definition 3.1.6
8:         r ← random (0,1)
9:         if $r < p_a$ then
10:            @ activate agent and adapatively move based on the context to a site in the neighbourhood
               using fuzzy IF-THEN rules
11:        else
               @ stay at current site and sleep
12:        end if
13:    end for
14:    @ update parameters $\alpha, \lambda, t, s_x, s_y$
15: end while
```

to get trapped in local minima. The fuzzy IF-THEN rules governing the agents' movements are also allowing the agents to go beyond the neighbourhood limits; for example in the case of two very different (VD) agents it makes no point to keep them in a reachability distance; and it makes sense that two very different (VD) agents should move further away from each other than two different (D) agents do. Thus the system behaves more naturally. Compared to [GP10] the agents are able to adapt their movements if changes in the environment would occur which is an important feature in real-time systems, wireless sensor networks or data streams. As in any non-deterministic algorithm, the way of processing the data is not fixed. This leads to slightly different results from one execution to another as it will be seen i the experiments. However we should mention that our goal here is to obtain a certain quality of the clustering (a certain overall fitness of the agents), so it is unimportant for us which two items are misclassified. Another drawback of the approach is that there is an agent encapsulating every item being clustered. While this is an advantage compared to other methods, as shown in [CXC04], we can imagine that in very large datasets the approach could become impractical in terms of memory consumption.

The algorithm we have presented is based on the adaptive ASM approach from [CXC04]. The major improvement is that, instead of moving the agents at a randomly selected site, we are letting the agents choose the best location. Agents can directly communicate with each other — similar to the approach from [CDG07]. In [SCCK04], the fuzzy IF-THEN rules are used for deciding if the agents are picking up or dropping an item. In our model we are using the fuzzy rules for deciding upon the direction and length of the movement. Compared to [GP10] the agents are able to adapt their movements if changes in the environment would occur.

3.2.2.1 Methodology

Let us consider the proposed clustering algorithm. Before it can be applied to cluster items in a given dataset, one needs to preprocess the data.

The preprocessing phase is a process that is usually necessary and it takes place before any data mining operation. This preprocessing step is necessary because it is often the case that data comes from multiple heterogeneous sources which increases the chance of receiving low-quality data, i.e., noisy, missing or inconsistent data.

Several data preprocessing techniques exist like data cleaning, data integration, data transformation and data reduction. Data cleaning involves transformations to correct wrong data and is used in general for removing noise and inconsistencies in data. Data integration is used for merging multiple sources of data into a single data store, such as a data warehouse while data transformations, like normalization can improve the accuracy and efficiency of the data mining algorithms that consider distance measurements. Data reduction leads to a reduced size of data by aggregating, eliminating redundant features, or clustering [HK06]. Nevertheless, data preprocessing should be used with caution because it could destroy interesting features in data, maybe exactly what the data analyst is looking for. So from the above we only use data normalization for scaling the attributes in the range $[0, 1]$. Missing attributes values are filled in with 0 so it could be considered that some data cleaning is also performed, but this would be all it is done in terms of destructive actions over the data.

Normalization is particularly useful for algorithms like clustering because it helps preventing attributes with large ranges from outweighing attributes with smaller ranges. As a normalization method, the min-max normalization method is chosen because it preserves the relationships between the original data values as opposed to z-score normalization, and normalization by decimal scaling [HK06].

For scaling the values in the $[0, 1]$ interval the following formula is used:

$$A_k \leftarrow \frac{A_k - min_A}{max_A - min_A}, \forall k = \overline{1, |A_k|}, \exists i = \overline{1, |\mathcal{X}|} : A_k \in X_i \tag{3.17}$$

where $\mathcal{X} = \{X_i | i = \overline{1, |\mathcal{X}|}\}$ is the dataset with items to be clustered, A_k denotes the value of the k^{th} item in the considered attribute A, min_A and max_A are the minimum and, respectively, the maximum values over A.

The next step is to compute the disimilarity $\mathcal{D}$ between all items in the dataset $\mathcal{X}$:

$$\mathcal{D} = \{d_{i,j} | i, j = \overline{1, |\mathcal{X}|}\} \tag{3.18}$$

where $d_{i,j}$ denotes the disimilarity between the items X_i and X_j from the dataset $\mathcal{X}$. The disimilarity $d_{i,j}$ between X_i and X_j could be the Euclidean distance between the two items:

$$de(X_i, X_j) = \sqrt{\sum_{k=1}^{m} (X_{i_k} - X_{j_k})^2} \tag{3.19}$$

where $m = |X_i| = |X_j|$.

The disimilarity $\mathcal{D}$ between all items is sent to the clustering component. The clustering component first creates a grid according to the given definitions and then the population of ants is created, one ant for each data item. The process executes as described in the given algorithm leading to a clustering in data as explained. The grid containing the clustered data, i.e., ants can be sent to further components or can be visualized and interpreted by the data analyst.

3.2.2.2 The advantages of the proposed approach

The BM [DGF$^+$91] and LF [LF94] models both separate ants from the objects being clustered, which increases computational cost. As noticed in [CXC04], ants carrying isolated data items may fall into a infinite loop because they are unable to ever finding a proper location to drop down the isolated data items. This leads to an even higher computational time consumption. The approach from [SCCK04] also keeps the ant and item to be clustered separated. The advantage of the ASM model is that ants directly represent the data items to be clustered. The ants move based on the similarity with the neighbouring fellows eventually forming distinctive groups and hence the corresponding data items are clustered.

The approach from [CXC04] allows a high liberty to the ants' movements but only in their neighbourhood. The advantage of our approach is that it enables the ants to communicate directly like in [CDG07] therefore breaking the neighbourhood boundaries and thus decreasing the chance of ants to get trapped in local minima. The fuzzy IF-THEN rules governing the agents' movements are also allowing the agents to go beyond the neighbourhood limits; for example in the case of two very different

(VD) agents it makes no point to keep them in a reachability distance; and it makes sense that two very different (VD) agents should move further away from each other than two different (D) agents do. Thus the system behaves more naturally, it becomes closer to human behaviour. Because, even if we are not aware of it, humans make decisions based on such if-then statements: if the weather is fine then I might go out, if the forecast for the weekend is bad weather then I will not plan a trip in the mountains. So if we want to have systems which mimic the human behaviour then fuzzy inference is a way towards this.

In the first model the agents represent the data items to be clustered and a change in the data item itself is not be handled at runtime. The extension to a context-aware dynamic system is be beneficial in many practical situations.

In general, context-aware systems could greatly change the way we interact with the world they could anticipate our needs and advice us when taking some decisions. In a changing environment context-awareness is undoubtly beneficial. The nowadays technology is able to support such systems — devices with GPS are present in the lives of many of us. So a system can be aware about the location, it may know where the user is going, but it could also be aware of the user's preferences or it could infer such preferences from a user's activity. Such systems could make much more relevant recommendations and support decision making.

In the current approach each data item is encapsulated by an agent. While this is an advantage compared to the BM and LF models [DGF$^+$91, LF94] since it reduces computational cost, it also comes with a drawback: clearly if the data set has many items then the memory consumption is high. An idea to overcome this issue would be to use a similar approach to the one from [KHK99]. Here the clustering algorithm uses a sparse graph in which nodes represent data items and edges have weights corresponding to similarities between data items. This representation enables Chameleon [KHK99] to scale well and to successfully use datasets that are available in similarity space only and not in metric spaces.

3.2.3 Case studies

In this chapter computational experiments showing the potential of the proposed method are presented. In the first case study a custom dataset is considered and comparison with the k-means clustering is done suggesting the strength of the proposed algorithm. In the second case study the algorithm is tested on a larger dataset. Comments regarding the performance together with idea for further improvements are presented. The third case study is presenting a possible application of this clustering approach in a real-life scenario — clustering web search results [GP10].

3.2.3.1 Synthetic dataset

In order to evaluate the proposed algorithm the following custom dataset was considered:

Test dataset

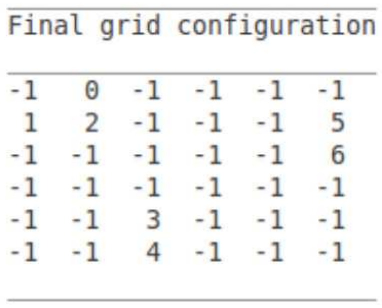
Id	A1	A2	A3	A4
0	0.11	0.11	0.12	0.13
1	0.12	0.12	0.14	0.11
2	0.11	0.11	0.11	0.11
3	0.41	0.41	0.41	0.42
4	0.43	0.43	0.41	0.41
5	0.81	0.81	0.82	0.82
6	0.81	0.81	0.83	0.81

The algorithm was evaluated against this dataset and the following clusters were obtained:

- $Cluster0$ (0, 1, 2)
- $Cluster1$ (3, 4)

- *Cluster*2 (5, 6).

In Figure 3.2 the final position of the agents can be seen. The agents are labelled with values from 0 to 6, -1 denotes and empty position.

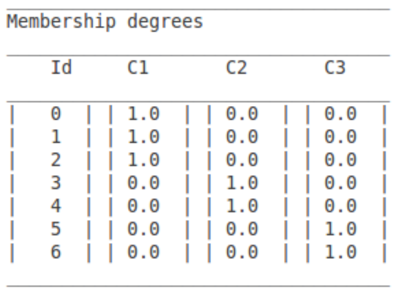


```
Final grid configuration

-1   0  -1  -1  -1  -1
 1   2  -1  -1  -1   5
-1  -1  -1  -1  -1   6
-1  -1  -1  -1  -1  -1
-1  -1   3  -1  -1  -1
-1  -1   4  -1  -1  -1
```

Figure 3.2: Final grid configuration - synthetic dataset

In Figure 3.3 one can see the membership degree of each agent to each cluster. In this test case, because the points from different clusters are very well separated, the membership degree of each ant to the clusters is either 0 or 1. Of course in reality it is not always the case to have so clearly separated points to cluster and in such cases the membership degrees of the points will belong to the interval $[0, 1]$ as opposed of belonging to the set $\{0, 1\}$.



```
Membership degrees

    Id     C1         C2         C3

|   0  | | 1.0  | | 0.0  | | 0.0  |
|   1  | | 1.0  | | 0.0  | | 0.0  |
|   2  | | 1.0  | | 0.0  | | 0.0  |
|   3  | | 0.0  | | 1.0  | | 0.0  |
|   4  | | 0.0  | | 1.0  | | 0.0  |
|   5  | | 0.0  | | 0.0  | | 1.0  |
|   6  | | 0.0  | | 0.0  | | 1.0  |
```

Figure 3.3: Membership degrees - synthetic dataset

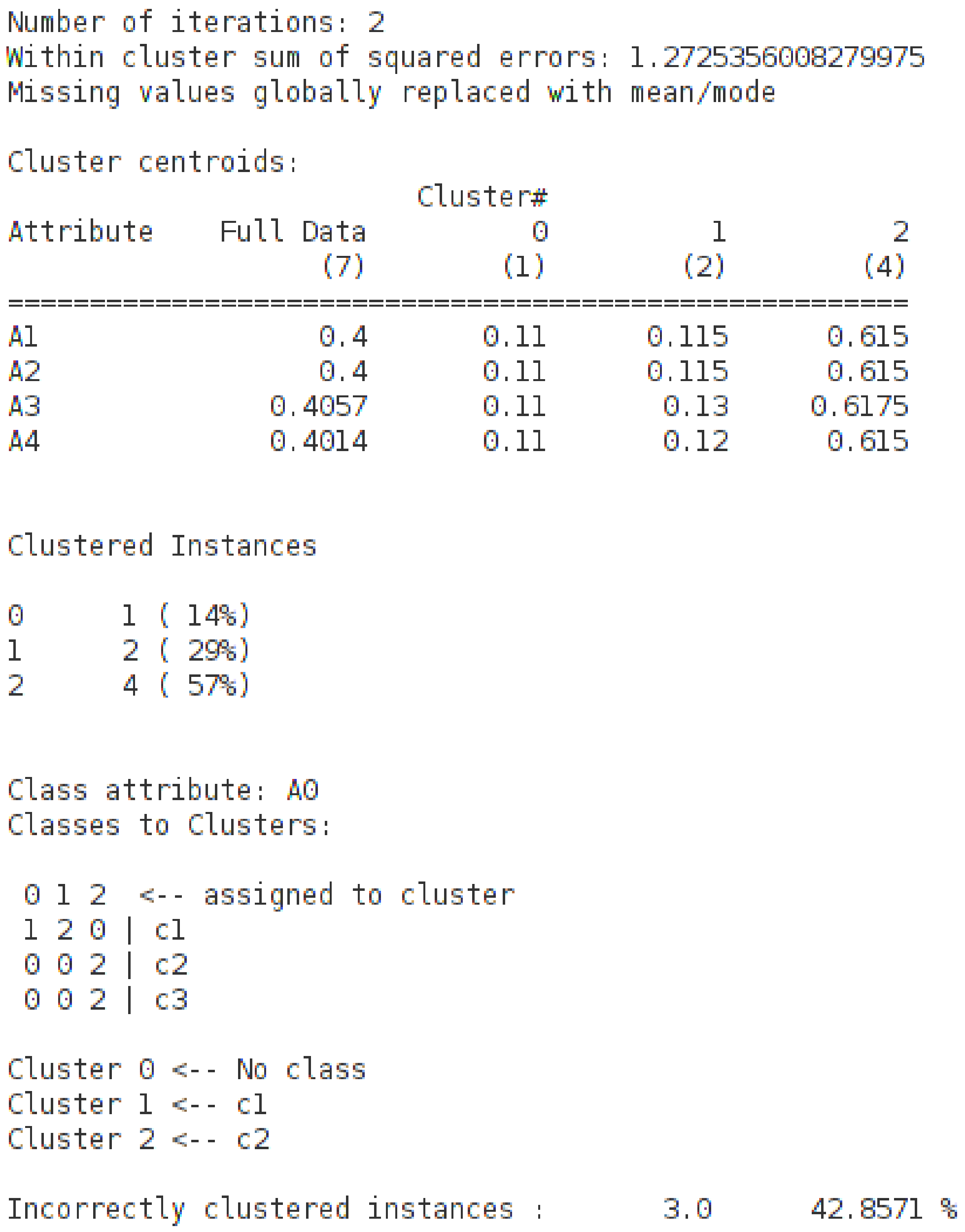
In Figure 3.4 the reported result from Weka [HFH$^+$09] on the same dataset is shown. As it will be seen the k-means algorithm has 3 misclassifications while our algorithm has no misclassifications in this test case. Further comments are superfluous for this test case.

3.2.3.2 Iris dataset

In order to test the algorithm in a real-world scenario, the Iris dataset was considered [Iri88]. The data set contains three classes of 50 instances each, each class referring to a type of iris plant. There are four attributes plus the class: sepal length in *cm*, sepal width in *cm*, petal length in *cm*, petal width in *cm*, class (Iris Setosa, Iris Versicolour, Iris Virginica). This dataset is appropriate for rather testing classification, but it was preferred for clustering too because the class attribute is given and hence there a way to evaluate the algorithm. So apparently it would be ideal for the algorithm to produce three clusters of 50 instances, the three clusters corresponding to the given three classes.

The last two attributes (petal length in cm and petal width in *cm*) are highly correlated according to [Iri88]. We do not dismiss any of these attributes though because as explained in the methodology we would like to keep as much of the data unchanged. We do however scale the data to the interval $[0, 1]$. At this point the clustering process can be started as described in the corresponding section and a result is given. In the final grid configuration (Figure 3.5) some columns from the middle, where a bigger space can be seen, have been removed in order for the grid to fit into the page.

As it can be seen from Figure 3.5, all the agents have been assigned to clusters. As it will be seen in Figure 3.6 they actually belong to the clusters in a certain degree.



```
Number of iterations: 2
Within cluster sum of squared errors: 1.2725356008279975
Missing values globally replaced with mean/mode

Cluster centroids:
                         Cluster#
Attribute    Full Data          0          1          2
                   (7)        (1)        (2)        (4)
=======================================================
A1                 0.4       0.11      0.115      0.615
A2                 0.4       0.11      0.115      0.615
A3              0.4057       0.11       0.13     0.6175
A4              0.4014       0.11       0.12      0.615


Clustered Instances

0      1 ( 14%)
1      2 ( 29%)
2      4 ( 57%)


Class attribute: A0
Classes to Clusters:

 0 1 2  <-- assigned to cluster
 1 2 0 | c1
 0 0 2 | c2
 0 0 2 | c3

Cluster 0 <-- No class
Cluster 1 <-- c1
Cluster 2 <-- c2

Incorrectly clustered instances :       3.0      42.8571 %
```

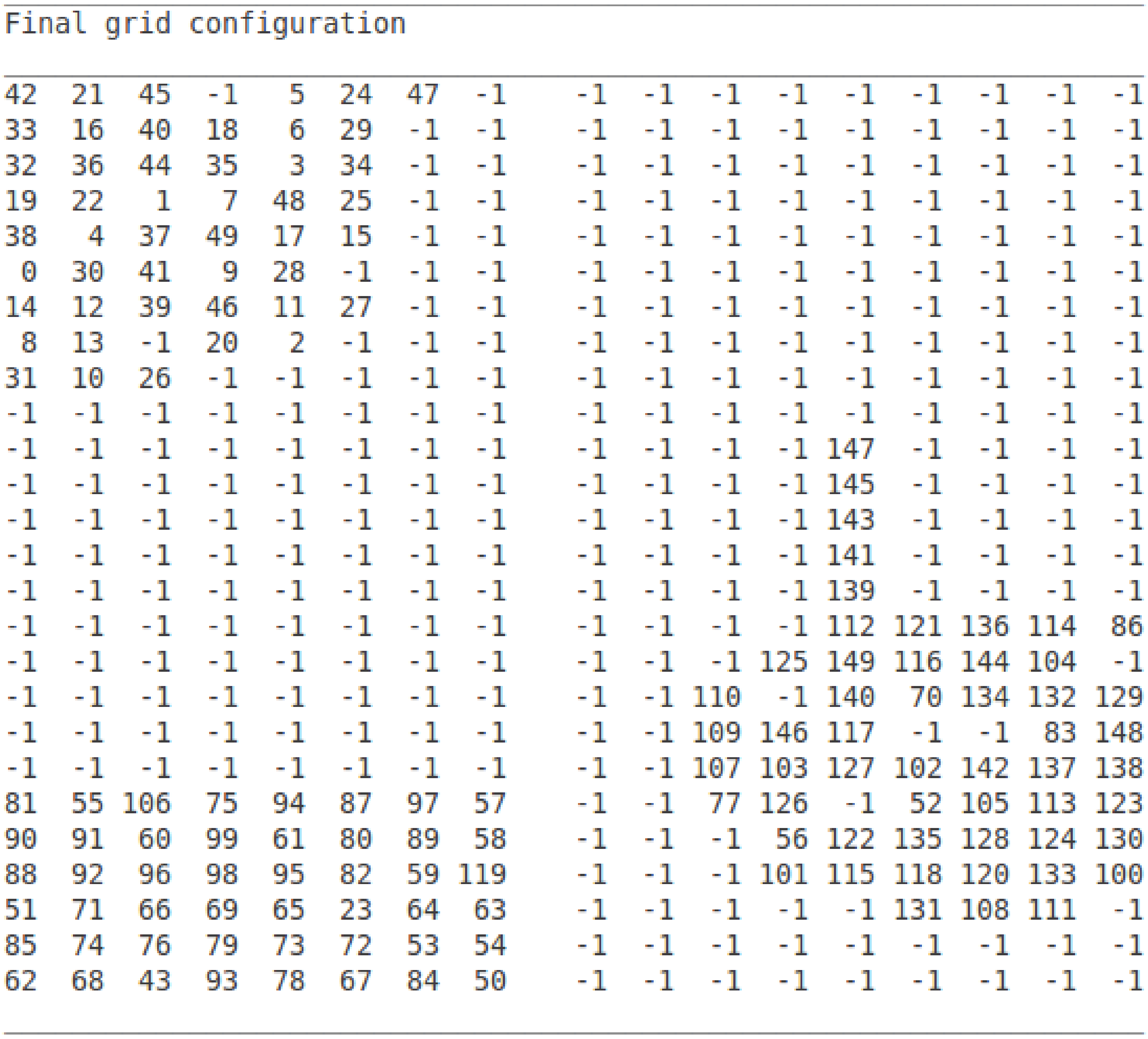
Figure 3.4: Classical approach result - synthetic dataset

According to the Iris dataset [Iri88], items ranging from 0 to 49 belong to the first class, items ranging from 50 to 99 belong to the second class and items ranging from 100 to 149 belong to the third class. So from the grid table it appears that the following clusters contain some misclassifications:

- $Cluster1$ (items 0 – 49): no misclassifications
- $Cluster2$ (items 50 – 99): 106, 119, 23, 43
- $Cluster3$ (items 100 – 149): 86, 70, 83, 52, 56

So it appears that the algorithm has misclassified 9 items.

Let us check each item in more detail. In order to do this deeper analysis we first need a representative for each cluster. We try to simulate the real-life process in which the data analyst would point such representatives with the mouse. Of course that if he deals with a high density cluster then he normally can only make a rough approximation. We can refine his choice by proposing an item in the neighbourhood which has the highest fitness. So what we do in this test is to randomly choose a candidate representative from each cluster and then replace him with the best fitted agent from a



```
Final grid configuration

 42  21  45  -1   5  24  47  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 33  16  40  18   6  29  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 32  36  44  35   3  34  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 19  22   1   7  48  25  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 38   4  37  49  17  15  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
  0  30  41   9  28  -1  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 14  12  39  46  11  27  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
  8  13  -1  20   2  -1  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 31  10  26  -1  -1  -1  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 147  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 145  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 143  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 141  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 139  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1  -1 112 121 136 114  86
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1  -1 125 149 116 144 104  -1
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1 110  -1 140  70 134 132 129
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1 109 146 117  -1  -1  83 148
 -1  -1  -1  -1  -1  -1  -1  -1    -1  -1 107 103 127 102 142 137 138
 81  55 106  75  94  87  97  57    -1  -1  77 126  -1  52 105 113 123
 90  91  60  99  61  80  89  58    -1  -1  -1  56 122 135 128 124 130
 88  92  96  98  95  82  59 119    -1  -1  -1 101 115 118 120 133 100
 51  71  66  69  65  23  64  63    -1  -1  -1  -1  -1 131 108 111  -1
 85  74  76  79  73  72  53  54    -1  -1  -1  -1  -1  -1  -1  -1  -1
 62  68  43  93  78  67  84  50    -1  -1  -1  -1  -1  -1  -1  -1  -1
```

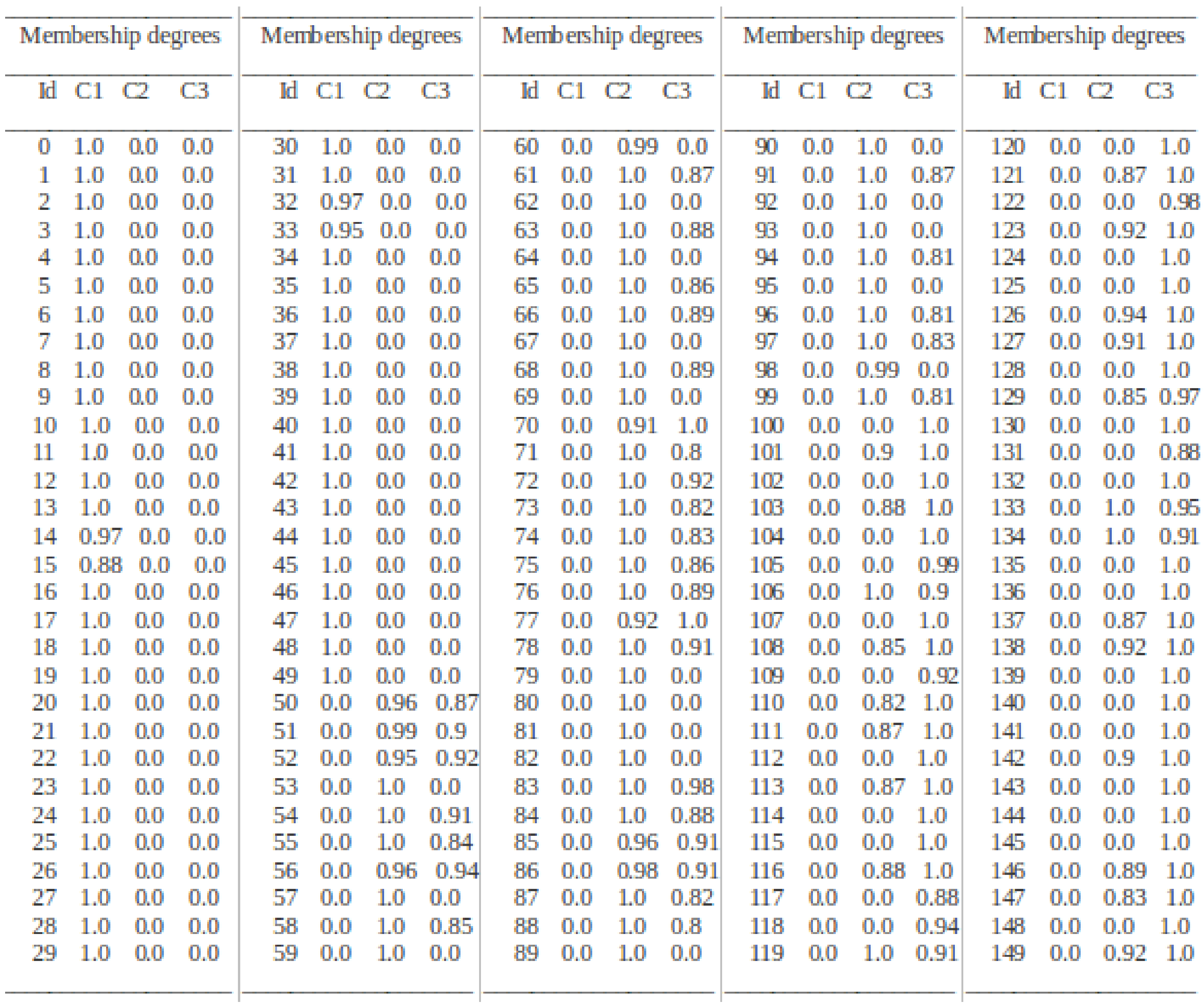
Figure 3.5: Final grid configuration - Iris dataset

certain radius; a radius $r = 2$ was considered for this case study. The agents 34, 90, 120 resulted from this process as final representatives for clusters 1, 2 and 3 respectively.

In the following the similarity between each of the above reported misclassification and the corresponding representative is given. The membership degree to each cluster is also considered. As in the first test case, the Euclidean distance between items has been used as a similarity metric. So a small similarity value i.e. distance between two items means that the two items are similar, should stay together.

Cluster2 — RepresentativeId (90)				
MisclassificationId	Similarity	C1	C2	C3
106	0.20	0.0	1.0	0.9
119	0.18	0.0	1.0	0.91
23	0.50	1.0	0.0	0.0
43	0.51	1.0	0.0	0.0

From the table Cluster2 — RepresentativeId (90) it can be seen that items 23 and 43 are clearly misclassifications. Both of them should surely belong to $Cluster1$ as they have a membership degree of 1 to this cluster and 0 to the other clusters. Moreover, their similarity value of 0.50 and 0.51 respectively



Membership degrees			
Id	C1	C2	C3
0	1.0	0.0	0.0
1	1.0	0.0	0.0
2	1.0	0.0	0.0
3	1.0	0.0	0.0
4	1.0	0.0	0.0
5	1.0	0.0	0.0
6	1.0	0.0	0.0
7	1.0	0.0	0.0
8	1.0	0.0	0.0
9	1.0	0.0	0.0
10	1.0	0.0	0.0
11	1.0	0.0	0.0
12	1.0	0.0	0.0
13	1.0	0.0	0.0
14	0.97	0.0	0.0
15	0.88	0.0	0.0
16	1.0	0.0	0.0
17	1.0	0.0	0.0
18	1.0	0.0	0.0
19	1.0	0.0	0.0
20	1.0	0.0	0.0
21	1.0	0.0	0.0
22	1.0	0.0	0.0
23	1.0	0.0	0.0
24	1.0	0.0	0.0
25	1.0	0.0	0.0
26	1.0	0.0	0.0
27	1.0	0.0	0.0
28	1.0	0.0	0.0
29	1.0	0.0	0.0
30	1.0	0.0	0.0
31	1.0	0.0	0.0
32	0.97	0.0	0.0
33	0.95	0.0	0.0
34	1.0	0.0	0.0
35	1.0	0.0	0.0
36	1.0	0.0	0.0
37	1.0	0.0	0.0
38	1.0	0.0	0.0
39	1.0	0.0	0.0
40	1.0	0.0	0.0
41	1.0	0.0	0.0
42	1.0	0.0	0.0
43	1.0	0.0	0.0
44	1.0	0.0	0.0
45	1.0	0.0	0.0
46	1.0	0.0	0.0
47	1.0	0.0	0.0
48	1.0	0.0	0.0
49	1.0	0.0	0.0
50	0.0	0.96	0.87
51	0.0	0.99	0.9
52	0.0	0.95	0.92
53	0.0	1.0	0.0
54	0.0	1.0	0.91
55	0.0	1.0	0.84
56	0.0	0.96	0.94
57	0.0	1.0	0.0
58	0.0	1.0	0.85
59	0.0	1.0	0.0
60	0.0	0.99	0.0
61	0.0	1.0	0.87
62	0.0	1.0	0.0
63	0.0	1.0	0.88
64	0.0	1.0	0.0
65	0.0	1.0	0.86
66	0.0	1.0	0.89
67	0.0	1.0	0.0
68	0.0	1.0	0.89
69	0.0	1.0	0.0
70	0.0	0.91	1.0
71	0.0	1.0	0.8
72	0.0	1.0	0.92
73	0.0	1.0	0.82
74	0.0	1.0	0.83
75	0.0	1.0	0.86
76	0.0	1.0	0.89
77	0.0	0.92	1.0
78	0.0	1.0	0.91
79	0.0	1.0	0.0
80	0.0	1.0	0.0
81	0.0	1.0	0.0
82	0.0	1.0	0.0
83	0.0	1.0	0.98
84	0.0	1.0	0.88
85	0.0	0.96	0.91
86	0.0	0.98	0.91
87	0.0	1.0	0.82
88	0.0	1.0	0.8
89	0.0	1.0	0.0
90	0.0	1.0	0.0
91	0.0	1.0	0.87
92	0.0	1.0	0.0
93	0.0	1.0	0.0
94	0.0	1.0	0.81
95	0.0	1.0	0.0
96	0.0	1.0	0.81
97	0.0	1.0	0.83
98	0.0	0.99	0.0
99	0.0	1.0	0.81
100	0.0	0.0	1.0
101	0.0	0.9	1.0
102	0.0	0.0	1.0
103	0.0	0.88	1.0
104	0.0	0.0	1.0
105	0.0	0.0	0.99
106	0.0	1.0	0.9
107	0.0	0.0	1.0
108	0.0	0.85	1.0
109	0.0	0.0	0.92
110	0.0	0.82	1.0
111	0.0	0.87	1.0
112	0.0	0.0	1.0
113	0.0	0.87	1.0
114	0.0	0.0	1.0
115	0.0	0.0	1.0
116	0.0	0.88	1.0
117	0.0	0.0	0.88
118	0.0	0.0	0.94
119	0.0	1.0	0.91
120	0.0	0.0	1.0
121	0.0	0.87	1.0
122	0.0	0.0	0.98
123	0.0	0.92	1.0
124	0.0	0.0	1.0
125	0.0	0.0	1.0
126	0.0	0.94	1.0
127	0.0	0.91	1.0
128	0.0	0.0	1.0
129	0.0	0.85	0.97
130	0.0	0.0	1.0
131	0.0	0.0	0.88
132	0.0	0.0	1.0
133	0.0	1.0	0.95
134	0.0	1.0	0.91
135	0.0	0.0	1.0
136	0.0	0.0	1.0
137	0.0	0.87	1.0
138	0.0	0.92	1.0
139	0.0	0.0	1.0
140	0.0	0.0	1.0
141	0.0	0.0	1.0
142	0.0	0.9	1.0
143	0.0	0.0	1.0
144	0.0	0.0	1.0
145	0.0	0.0	1.0
146	0.0	0.89	1.0
147	0.0	0.83	1.0
148	0.0	0.0	1.0
149	0.0	0.92	1.0

Figure 3.6: Membership degrees - Iris dataset

to the representative item 90 should have made these agents to reject each other; because such similarity values would make them D ($Different$) in terms of the considered fuzzy sets. The considered limit values for the fuzzy sets described in a previous section are: $SD1(0.2), SD2(0.4), VD1(0.6), VD2(0.7)$.

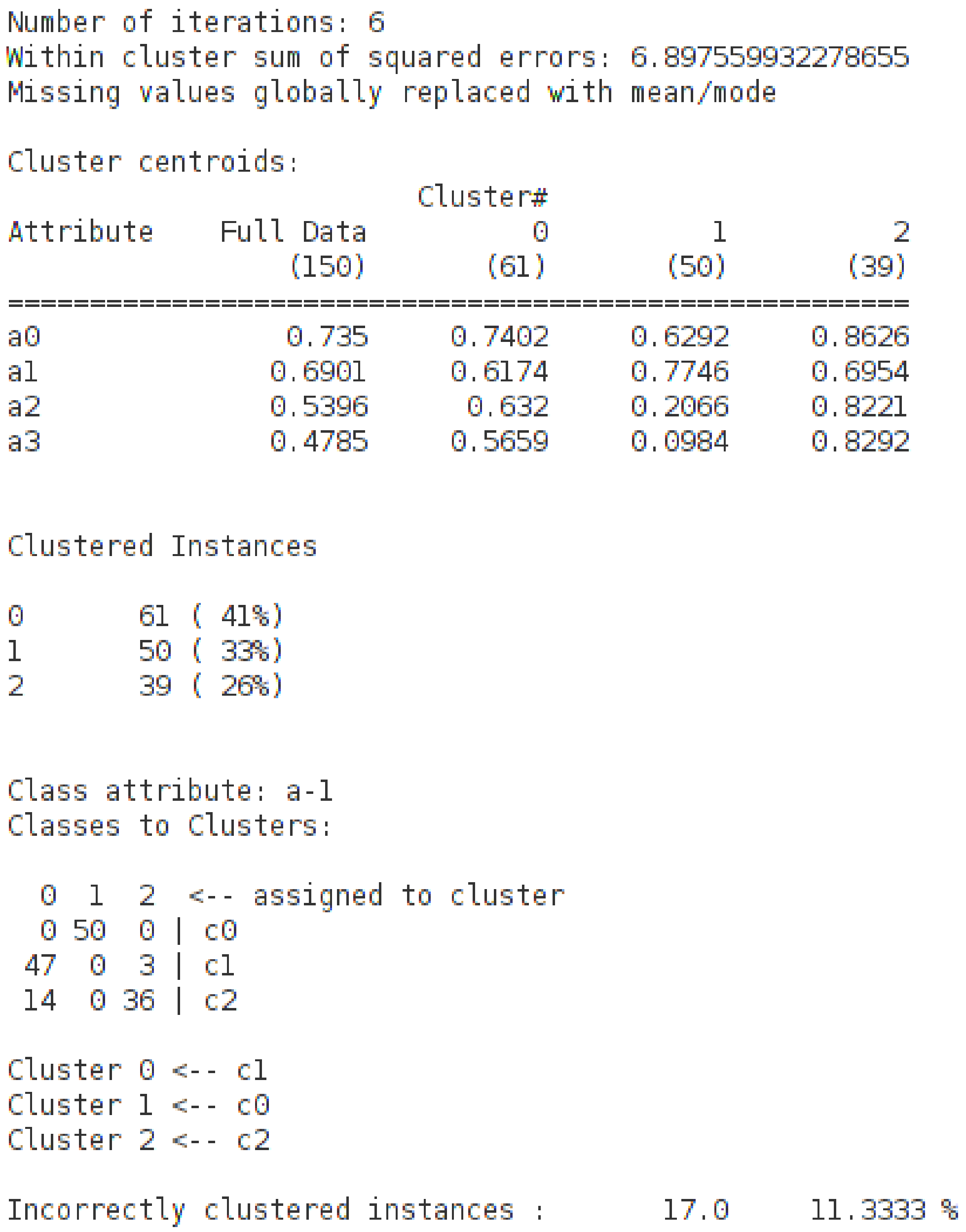
On the other hand, it is unclear why should items 106 and 119 be considered misclassifications. According to our similarity measures they have a 0.20 and a 0.18 similarity with the representative item 20. This makes them S ($Similar$) with this item. The membership degree with $Cluster2$ suggests that these items belong in this cluster. However the membership degree with $Cluster3$ is also high. The highest membership degree is with $Cluster2$ though and because of this it could be claimed that the items are actually correctly classified with respect to the considered metric. However we believe that items 106 and 119 cannot be considered to strictly belong either $Cluster2$ or $Cluster3$ as they are clearly at the border of the two clusters so they belong to both. In this case we also believe that they should not be regarded as misclassifications.

Cluster3 — RepresentativeId (120)				
MisclassificationId	Similarity	C1	C2	C3
86	0.34	0.0	0.98	0.91
70	0.26	0.0	0.91	1.0
83	0.32	0.0	1.0	0.98
52	0.32	0.0	0.95	0.92
56	0.31	0.0	0.96	0.94

Like items 106 and 119, the items from the table Cluster3 — RepresentativeId (120) should also not count as misclassifications for the same reasons from above. This leaves only items 23 and 43 as misclassifications.

A final remark regarding the membership degrees table. It may be observed that starting with item 50 a lot of instances have high membership degrees to both *Cluster*2 and *Cluster*3. This suggests that the members of these two classes are not linearly separable as mentioned in [Iri88] so a clustering process could also merge these two clusters and hence report only two clusters instead of three.

In Figure 3.7 we will list the output of the k-means clustering algorithm of the Iris dataset [Iri88] from Weka [HFH+09]. As it can be seen, 17 instances are reported as misclassified so our proposed model delivers better results than k-means for a real-world dataset.



```
Number of iterations: 6
Within cluster sum of squared errors: 6.897559932278655
Missing values globally replaced with mean/mode

Cluster centroids:
                          Cluster#
Attribute    Full Data          0          1          2
                 (150)       (61)       (50)       (39)
=======================================================
a0               0.735     0.7402     0.6292     0.8626
a1              0.6901     0.6174     0.7746     0.6954
a2              0.5396      0.632     0.2066     0.8221
a3              0.4785     0.5659     0.0984     0.8292


Clustered Instances

0       61 ( 41%)
1       50 ( 33%)
2       39 ( 26%)


Class attribute: a-1
Classes to Clusters:

  0  1  2  <-- assigned to cluster
  0 50  0 | c0
 47  0  3 | c1
 14  0 36 | c2

Cluster 0 <-- c1
Cluster 1 <-- c0
Cluster 2 <-- c2

Incorrectly clustered instances :       17.0     11.3333 %
```

Figure 3.7: Classical approach result — Iris dataset

A final remark regarding the membership degrees table. It may be observed that starting with item 50 a lot of instances have high membership degrees to both *Cluster*2 and *Cluster*3. This suggests that the members of these two classes are not linearly separable as mentioned in [Iri88] so a clustering process could also merge these two clusters and hence report only two clusters instead of three.

3.2.3.3 Clustering web search results

This case study was presented in [GP10]. An initial version of the clustering algorithm presented here was applied for clustering web search results showing the applicability in real life of this research. In general, web search engines respond to queries by returning a list of links to web pages that are considered relevant. These queries are often ambiguous or too general for accurately expressing the users information need. Thus most of the search results are not really relevant. So users are often browsing through a long list of items in order to find what they are actually looking for. And hence the idea to cluster web search results so that the output would be a list of labelled clusters.

In the other test case the Euclidean distance was used to measure the (di)similarity between items. In this test case the following formula is used:

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t \tag{3.20}$$

$$idf_t = log\frac{N}{df_t} \tag{3.21}$$

where N is the total number of documents in the collection, df_t (document frequency) is the number of documents in the collection containing the term t. Thus the *idf* (inverse document frequency) of a seldom term is high, while as the *idf* of an often term is low. So, $tfidf$ (term frequency inverse document frequency) assigns to term t a weight in document d that is:

- highest when given a small number of documents, t has many occurrences within them
- lower when either given a document d, the term t occurs few times or when it occurs in many documents
- lowest when t occurs in almost all documents.

So documents may be seen as a vector of components corresponding to each term in the dictionary, together with the associated weight (given by $tfidf$). This weight is equal to zero for dictionary terms that do not appear in the document.

In order to evaluate the algorithm for clustering web search results [GP10] we have made a Java application containing the following module: a web crawler, a weighing module and a clustering module.

The Web crawler browses the World Wide Web in a methodical, automated manner with the purpose of parsing and indexing the HTML pages. To explore the Web graph the breadth-first algorithm is used. The crawler receives as input a set of starting pages and it extracts the text and the links. The weighting component has two parts: a $MySQL$ procedure and a Java thread that executes the procedure at a given interval of time. Using the formulas from relations (3.20) and (3.21) we apply a stored procedure to update the token's weights. When performing a normal search, the dot product between the query vector and documents from the index is computed and the documents are returned in decreasing order. A matrix of document similarities is given to the clustering component. The clustering component uses the Algorithm 3.2.1 and outputs the clusters and the document ids from each cluster. For example, suppose we are searching for the word “mouse”. We take only the first 5 search results and send them to the clustering component:

The clustering component will output the following clusters:

- Cluster0: 0, 2, 4
- Cluster1: 1, 3.

This result was found after 46 iterations (the grid configuration has stabilized) and the computation took less than 1 second. We have used the following initial values for the parameters: $s_x = s_y = \lambda = 2$. The value of the parameter α is computed according to relation (3.12). The parameter t is used for the termination condition. In our approach the termination condition is simply $t < t_{max}$ and we have considered $t_{max} = 100$, i.e., the algorithm iterates 100 times. If we look at the first cluster, we see

Table 3.1: First 5 search results

Document id	Site
0	http://computer.howstuffworks.com/mouse.htm
1	http://en.wikipedia.org/wiki/Mouse
2	http://www.newegg.com/
3	http://animal.discovery.com/
4	http://computer.howstuffworks.com/share-redirect? type=facebook&cid=1106

that we have documents containing information about the "mouse" from computing and if we look at the second cluster we see that we have documents containing information about "mouse" — the animal. So one would have the possibility to browse through the desired cluster of search results only. We consider that this is a good example for showing the applicability of our methods in real life.

3.2.4 Discussion

The BM [DGF$^+$91] and LF [LF94] models both separate ants from the objects being clustered, which increases computational cost, as noticed in [CXC04]. Also the following unfortunate situation could occur: ants which carry isolated data items may get trapped into infinite loops since they will never find a proper location to drop down the isolated data items which will lead to a higher amount of computational time consumption. The advantage of the ASM model is that ants directly represent the objects to be clustered. The results reported in [CXC04] show that the LF model needs one million iterations in order to achieve satisfying results, while the ASM model needs five thousand iterations. By using direct communication and fuzzy if-then rules we have managed to reduce the number of necessary iterations from the order of thousands to the order of hundreds. As being still in an experimental phase, there is a penalty at the level of each iteration for the moment, but further work is continuously done in order to improve this aspect.

The approach from [SCCK04] keeps the ant and item to be clustered separated and the behaviour of the artificial ants is governed by fuzzy if-rules — in their approach the rules are used in deciding weather or not to pick-up an item as opposed to our approach where the fuzzy rules are governing the movement strategy of each individual ant. In order to evaluate the algorithm they define a classification error which strongly penalizes a solution in which a wrong number of clusters has been obtained. The results are provided in terms of the classification error defined there and they were found after one million iterations. Also, unlike our approach, they consider each item as uniquely belonging to a cluster which, as we have seen in the second case study, is in our opinion not the best approach.

So by using direct communication and fuzzy if-then rules we have managed to significantly improve the number of necessary iterations. An important achievement over the considered approaches are the use fuzzy memberships because they allow us to have a better view over the data. As we have seen along the case studies, this is especially important when dealing with non-linearly separable classes where we can spot items belonging in a high degree to both classes avoiding wrong classifications or misjudgements over such items. This makes our method more robust than the considered approaches.

The approach from [CXC04] allows a high liberty to the ants' movements but only in their neighbourhood. The advantage of our approach is that it enables the ants to communicate directly like in [CDG07] therefore breaking the neighbourhood boundaries and thus decreasing the chance of ants to get trapped in local minima. The fuzzy IF-THEN rules governing the agents' movements are also allowing the agents to go beyond the neighbourhood limits; for example in the case of two very different (VD) agents it makes no point to keep them in a reachability distance; and it makes sense that two very different (VD) agents should move further away from each other than two different (D) agents do. Thus the system behaves more naturally. Compared to [GP10] the agents are able to adapt their movements if changes in the environment would occur which is an important feature in real-time systems, wireless sensor networks or data streams. As in any non-deterministic algorithm, the way of processing the data is not fixed. This leads to slightly different results from one execution to another — for example in our experiment we had two misclassifications, namely items 23 and 43, but at another execution some other items could be misclassified. However we should mention that our goal here is to obtain a certain quality of the clustering (a certain overall fitness of the agents), so it is unimportant for us which two items are misclassified. Another drawback of the approach is that there is an agent for each item being clustered. While this is an advantage compared to other methods, as shown in [CXC04], we can imagine that in very large datasets the approach could become impractical in terms of memory consumption.

3.3 Incremental clustering

The idea behind incremental clustering is that it is possible to consider one instance at a time and assign it to existing clusters without significantly affecting the already existing structures. This chapter presents an incremental clustering approach based on ASM.

3.3.1 General considerations

The idea behind incremental clustering is that it is possible to consider one instance at a time and assign it to existing clusters without significantly affecting the already existing structures. The incremental approach to clustering is also applicable in online situations like wireless sensor networks or data streams. Ongoing research is done in the area sensor data and data stream mining [SdLFdCG09, HZK$^+$09, GKS09]. In [SdLFdCG09], a new approach to novelty detection in data streams is presented. The ability to detect new concepts is an important aspect in machine learning systems. The approach presented in this paper [SdLFdCG09] takes novelty detection beyond one-class classification, by detecting emerging cohesive and representative clusters of examples, and then further by merging similar concepts. The proposed method goes in the direction of constructing a class structure that aims at reproducing the real one in an unsupervised continuous learning fashion. The paper [HZK$^+$09] presents a general approach for context-aware adaptive mining of data streams that tries to dynamically and autonomously adjust data stream mining parameters according to changes in context and situations. Data stream processing adaptation to variations of data rates and resource availability is crucial for consistency and continuity of running applications like health care systems. In [GKS09] a new data model called Spatio-Temporal Sensor Graphs (STSG), which is designed to model sensor data on a graph by allowing the edges and nodes to be modelled as time series of measurement data is presented. It is shown how this model could be applied in finding patterns like growing hotspots in sensor data. The case studies and the related study show that the presented model is less memory expensive.

In incremental clustering only the cluster representations need to be kept in memory so not the entire dataset and thus the space requirements for such an algorithm are very small. Whenever a new instance is considered an incremental clustering algorithm would basically try to assign it to one of the already exiting clusters. Such a process is not very complex and therefore the time requirements for an incremental clustering algorithm are also small.

In [GP11a], the clustering problem is approached by the idea of context-aware ASM agents. The agents are able to detect changes in the environment and adjust their moves accordingly. Unlike the agents from the classical ASM model [CXC04] the agents from [GP11a] are able to communicate directly therefore breaking the neighbourhood boundaries and thus decreasing the chance of ants to get trapped in local minima. The agents are able to adapt their movements if changes in the environment would occur. However only changes in the features of already existing data items are handled. So clustering new incoming data items is not an issues in the approach from [GP11a]. In this sense the incremental approach from this paper represents a natural step forward.

In [NGS11] an interesting incremental clustering approach is presented similar to the incremental DBSCAN algorithm [EKS$^+$98]. In the incremental version of the DBSCAN clustering algorithm data items can be added to existing clusters, one item at a time. The main difference of the approach from [NGS11] is that it adds groups of items to existing clusters. The data points to be added are initially clustered using the DBSCAN algorithm [EpKSX96] and the resulted clusters are merged with already existing ones. In other words, instead of adding data items incrementally the algorithm is adding clusters incrementally.

Kamble uses in [Kam10] a genetic algorithm for incrementally cluster data. The genetic algorithm [Gol89] uses and manipulates a population of potential solutions in the attempt to obtain the optimal solutions and a generation is completed when every individual from the population has performed the genetic operators. The individuals from the resulted population will be better fitted with respect to the objective function. The algorithm from [Kam10] works in metric spaces and it is a density-based approach as the key idea is that for each element of a cluster the number of items in the neighbourhood need to be above a certain threshold.

Lee et al. present in [LKC02] a method to implicitly resolve ambiguities using an incremental clustering in Korean to English cross language information retrieval. In their approach a query in Korean is first translated into English using a Korean-English dictionary and then documents are retrieved for the translated query terms. Query-oriented document clusters are incrementally created for the top ranked retrieved documents and the weight of each retrieved document is recomputed based on the created clusters. Considering their experiments the authors conclude that their method outperforms the monolingual retrieval.

In [CCFM97] an incremental clustering model is considered where the goal is to efficiently attempt to maintain clusters of small diameter as new items to be clustered are arriving. The dual clustering problem where clusters have a fixed diameter and the goal is to minimize the number of clusters is also considered. The authors focus their study in performing detailed running time complexity analysis of their approach compared to greedy algorithms and to static clustering approaches.

In [LLLH10] an incremental clustering for trajectories is presented. Due to their sequential nature, trajectory data (or moving objects) are often received incrementally as flows of data from various possible sources like GPS. Most of the existing trajectory clustering algorithms are developed for static datasets, but, as the authors remark, such static approach are not suitable when frequent reclustering is needed and when huge amounts of trajectory data are accumulated constantly and needs immediate processing. The proposed incremental approach contains two parts: an online micro-cluster maintenance and an offline macro-cluster creation. Authors explain that micro-clusters are used to store compact summaries of similar trajectory line segments, which take much smaller space than raw trajectories. Incremental online update takes place on micro-clusters. Then macro-clustering is performed on the set of micro-clusters. Authors argue that since the number of micro-clusters is smaller than that of original trajectories, the macro-clusters generation is efficient.

In [DL11] a method for training set compression by using incremental clustering is proposed. The size of the training set can greatly influence the performance of a classifier because it is difficult to be stored in memory and to process it. So reducing the size of a training set is undoubtedly beneficial for a classification process. Training set compression is a method for reducing the size of the training set without having a negative impact on the classification accuracy. It attempts to eliminate redundancy in a dataset. Sequential leader clustering [Har75] is a classical clustering algorithm that may be used to eliminate redundancy. The samples to be discarded are considered redundant if they are close to one of the samples in the clusters. Hence the process of clustering may also be viewed as a subsampling process because clustering guarantees that all the samples in a group are separated at a certain distance which is greater than a minimal threshold and in the approach from [DL11], this subsampling based compression is introduced.

Our incremental clustering approach is based on the ASM-like algorithm from [CXC04]. In the ASM model, due to the need for security, the ants are constantly choosing a more comfortable environment to sleep in. The ants feel comfortable among individuals having similar characteristics. In ASM, each data item is represented by an agent, and his purpose is to search for a comfortable position for sleeping in his surrounding environment. While he doesn't find a suitable position to have a rest, he will actively move around to search for it and stop when he finds one; when he is not satisfied with his current position, he becomes active again.

If the fitness is low then the probability of activation is high so the agent is probably going to wake up, move and search for a better place to sleep. Conversely if the fitness is high then the probability is low so most probably the agent will stay and sleep.

3.3.2 Incremental ASM approach

At the beginning of the algorithm from [CXC04], the agents are randomly scattered on the grid in active state. They randomly move on the grid. In each loop, after the agent moves to a new position, it will recalculate its current fitness f and probability p_a so as to decide whether it needs to continue moving. The same p_a as in [CXC04] is used:

Definition 3.3.1 *The activation probability of $agent_i$ is:*

$$p_a(agent_i) = cos^{\lambda}(\frac{\pi}{2} f(agent_i)) \tag{3.22}$$

Remark 3.3.1 *If the current p_a is small, the agent has a lower probability of continuing to move on the grid and higher probability of stopping at the current location and taking a rest. Otherwise the agent will stay in active state and continue moving. The agent's fitness is related to its similarity with other agents in its neighbourhood.*

Remark 3.3.2 *For computing the fitness the Definition 3.2.1 is used.*

With increasing number of iterations, such movements gradually increase, eventually, making similar agents gathered within a small area and different types of agents located in separated areas. Thus, the corresponding data items are clustered.

A high level pseudo-code of our incremental clustering algorithm is presented in Algorithm 3.3.1.

Algorithm 3.3.1 Algorithm clustering

```
1: @ initialize parameters $\alpha, \lambda, t, s_x, s_y$
2: @ initialize an agent $a_0$ for the first arrived item and create a cluster $c_0$ containing agent $a_0$
3: while condition do
4:   for all item i do
5:     @ create an agent $a_i$ and randomly place it on the grid
6:     j ← random{0, i}
7:     if $isSimilar(a_i, a_j)$ then
8:       $groupSimilar(a_i, a_j)$
9:     else
10:      $add(U, a_i)$ // U — the set of unclustered agents
11:    end if
12:    if $hasElements(U)$ then
13:      $a_{asm} \leftarrow getRandomAgent(U)$
14:      $tryActivate(a_{asm})$
15:    end if
16:  end for
17:  @ adaptively update parameters
18: end while
```

The algorithm continuously receives new items to be clustered. For the first such item, a new agent is created that encapsulates this item. As in the classical ASM model, one agent per item will be allocated. A new cluster is created (the first one) and the agent is added to this cluster. In the following, whenever a new item i arrives, a new agent, a_i, is created. This agent is randomly placed on the grid and contacts an already existing agent from the grid. If they are similar then they are grouped together otherwise the agent a_i is added to the set of unclustered agents, U. The $groupSimilar(a_i, a_j)$ procedure groups together two agents, i.e., the agent a_i moves towards the agent a_j on the grid and also it is added to the a_j's cluster. If a_j is not already in a cluster then a new cluster containing both agents is created. Whenever a new cluster is created we try to merge it with an already existing cluster. The merging is performed based on the similarity between the cluster representatives. The representative of a cluster is either the first item that was added to the cluster or an item pointed out, i.e., manually chosen by the data analyst. The similarity between two agents will be discussed immediately. To the cluster representatives we will come back in the experiments section.

In the final step of the for loop we test if the set of unclustered agents is not empty and if it contains elements then we try to activate a randomly extracted agent from there. In $tryActivate(a_{asm})$ the agent will follow an ASM-like clustering process similar to the one from [GP11a]. If an agent finds a similar fellow then it will call the $groupSimilar(a_i, a_j)$ procedure.

In [GP11a], the agents decide upon the way they move on the grid according to their similarity with the neighbours, using fuzzy IF-THEN rules. Thus two agents can be similar (S), different (D), very different (VD). If two agents are similar or very similar they would get closer to each other. If they are different or very different they will get away from each other. The number of steps they do each time they move depend on the similarity level. So if the agents are VD they would jump many steps away from each other; if they are D they would jump less steps away from each other. In the end the ants which are S will be in the same cluster. The similarity computation is taking into account the actual structure of the data or the data density from the agent's neighbourhood; a bigger change from one agent to another translates into a certain similarity which then affects the agent's movement on the grid. A graphical representation of a fuzzy variable $Similarity$ is shown in figure 3.8.

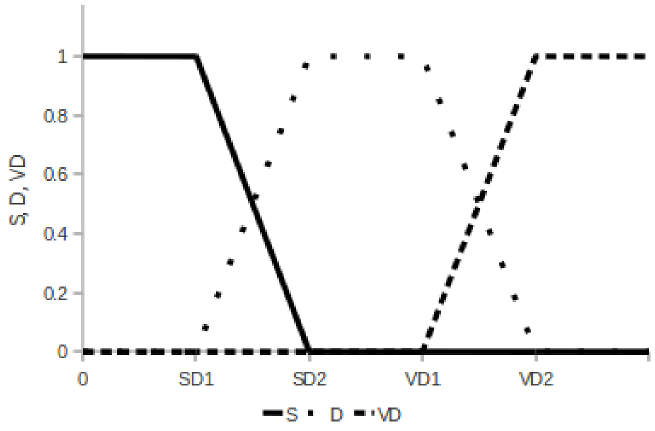


Figure 3.8: The fuzzy sets S, D and VD corresponding respectively to the linguistic concepts $Similar$, $Different$ and $VeryDifferent$ are called the *states* of the fuzzy variable $Similarity$. The limits $SD1$, $SD2$, $VD1$, $VD2$ are application specific.

The parameter α is the average distance between agents and this changes at each step further influencing the fitness function. The parameter λ influences the agents' activation pressure and it may decrease over time. The parameter t is used for the termination condition which could be something like $t < t_{max}$. The parameters s_x, s_y, the agent's vision limits may also be updated in some situations.

3.3.3 Experiments

In order to test the algorithm in a real-world scenario, the Iris dataset [Iri88] was considered for a first test case. The data set contains 3 classes of 50 instances each, each class referring to a type of iris plant. There are 4 attributes plus the class: sepal length in cm, sepal width in cm, petal length in cm, petal width in cm, class (Iris Setosa, Iris Versicolour, Iris Virginica). This dataset is appropriate for rather testing classification, but it was preferred for clustering too because the class attribute is given and hence there is a way to evaluate the algorithm. So apparently it would be ideal for the algorithm to produce 3 clusters of 50 instances, the 3 clusters corresponding to the given 3 classes.

The last 2 attributes (petal length in cm and petal width in cm) are highly correlated according to [Iri88]. However we do not dismiss any of these attributes because we would like to keep as much of the data unchanged. We do however scale the data to the interval $[0, 1]$. At this point the clustering process can be started. In the final grid configuration one would clearly see the clustered agents. Due to space limitations we can't show here the final grid configuration, but we will immediately summarize the results. Besides the final grid configuration a membership table is also produced. The membership table shows the membership degree of each agent to the clusters. Relevant parts of the membership table will also be explained immediately.

According to the Iris dataset [Iri88], items ranging from 0 to 49 belong to the first class, items ranging from 50 to 99 belong to the second class and items ranging from 100 to 149 belong to the third class. So from the grid table it appears that the following clusters contain some misclassifications:

- $Cluster1$ (items 0 – 49): no misclassifications
- $Cluster2$ (items 50 – 99): 106, 119, 133, 134
- $Cluster3$ (items 100 – 149): 70, 77

So it appears that the algorithm has misclassified 6 items.

Let us check each item in more detail. In order to do this deeper analysis we have to check out the membership degrees table. In the membership table one can see in what degree does each item belong to the clusters. This membership is computed with respect to the cluster representatives. The representative of a cluster is either the first item from that cluster or an item designated by the data analyst. In case of cluster merging if any of the representatives is designated by data analyst then this will be the representative of the new cluster; otherwise the representative of the cluster with the greatest number of elements will be chosen as the new representative. The agents 45, 95, 145 resulted from this process as final representatives for clusters 1, 2 and 3 respectively.

In the following, the similarity between each of the above reported misclassification and the corresponding representative is given. The membership degree to each cluster is also considered. The Euclidean distance between items has been used as a similarity metric. So a small similarity value i.e. distance between two items means that the two items are similar, should stay together.

MisclassificationId	Similarity	C1	C2	C3
106	0.25	0	0.95	0.84
119	0.24	0	0.96	0.83
133	0.19	0	1	0.89
134	0.24	0	0.96	0.82

Table 3.2: Cluster2 — RepresentativeId (95)

From the table 3.2 it can be seen that the similarities between the considered items and the representative lie between 0.19 and 0.25. This makes them S ($Similar$) with this item. The membership degree with $Cluster2$ suggests that these items belong to this cluster. However the membership degree with $Cluster3$ is also high. The highest membership degree is with $Cluster2$ though and because of this it could be claimed that the items are actually correctly classified with respect to the considered metric. However we believe that these items cannot be considered to strictly belong either to $Cluster2$ or to $Cluster3$ as they are clearly at the border of the two clusters so they belong to both. In this case we also believe that they should not be regarded as misclassifications.

Let us check the reported misclassifications from the third cluster.

MisclassificationId	Similarity	C1	C2	C3
70	0.22	0.0	0.94	0.98
77	0.24	0.0	0.95	0.96

Table 3.3: Cluster3 — RepresentativeId (145)

Like the items from table 3.2, the items from the table 3.3 should also not count as misclassifications for the same reasons from above. So, arguably, the algorithm has a 100% classification accuracy for the considered dataset.

It may be observed that starting with item 50 a lot of instances have high membership degrees to both $Cluster2$ and $Cluster3$. This suggests that the members of these two classes are not linearly separable as mentioned in [Iri88] so a clustering process could also merge these two clusters and hence report only two clusters instead of three.

For the second case study the wine dataset [Win91] was considered. This dataset contains the results of a chemical analysis of wines grown in the same region in Italy but derived from three different wine growers. The analysis determined the quantities of 13 constituents found in each of the three types of wines. In [Win91] it is mentioned that the initial dataset had 30 attributes. So the current dataset has 13 attributes plus the class. There are 178 instances grouped in three classes corresponding to the three wine growers. Items ranging from 1 to 59 belong to the first class, items from 60 to 130 belong to the second class and items from 131 to 178 belong to the third class.

The same procedure as the one form the first case study was applied and the following misclassifications resulted from the final grid configuration:

- $Cluster1$ (items 0 – 58): 63, 65, 66, 73, 78, 95, 98
- $Cluster2$ (items 59 – 129): 130, 134
- $Cluster3$ (items 130 – 177): 83

From the above it appears that the algorithm has misclassified 10 items. Moreover, the following items have not been classified, i.e., they remained in the collection U of unclustered agents: 59, 110, 121, 123, 124. So it appears that there are 15 classification errors. But let us check out everything in more detail.

The agents 17, 89, 166 are the final representatives for clusters 1, 2 and 3 respectively.

In the following, the similarity between each of the above reported misclassification and the corresponding representative is given. The membership degree to each cluster is also considered. The Euclidean distance between items has been used as a similarity metric. So a small similarity value i.e. distance between two items means that the two items are similar, should stay together.

MisclassificationId	Similarity	C1	C2	C3
63	0.63	0.87	0.83	0
65	0.42	1	0.99	0
66	0.64	0.86	0.83	0
73	0.61	0.89	0	0
78	0.69	0.81	0	0
95	0.69	0.81	0	0
98	0.51	0.99	0.8	0

Table 3.4: Cluster1 — RepresentativeId (17)

From the table 3.4 it can be seen that the similarities between the considered items and the representative are bellow 0.7. This makes them still S ($Similar$) with this item. The membership degree with $Cluster1$ suggests that these items belong to this cluster. However the membership degree with $Cluster2$ is also high. The highest membership degree is with $Cluster1$ though and because of this it could be claimed that the items are actually correctly classified with respect to the considered metric.

Let us check the reported misclassifications from the second cluster.

Cluster2 — RepresentativeId (89)				
MisclassificationId	Similarity	C1	C2	C3
130	0.76	0	0	0
134	0.68	0	0.82	0.8

Table 3.5: Cluster2 — RepresentativeId (89)

In table 3.5, it can be seen that item 134 has a high membership degree to both $Cluster2$ and $Cluster3$. Again, the highest membership is with $Cluster2$, the cluster in which it was classified. So it could be considered that it is correctly classified with respect to the considered metric. According to the same criterion, item 130 should not belong to any of the clusters, it should be labelled as an outlier. However it is incorrectly classified to $Cluster2$.

Let us check the reported misclassifications from the third cluster.

MisclassificationId	Similarity	C1	C2	C3
83	0.6	0	0.81	0.9

Table 3.6: Cluster3 — RepresentativeId (166)

In table 3.6, it can be seen that item 83 has a high membership degree to both $Cluster2$ and $Cluster3$, but the highest membership is with $Cluster3$, the cluster in which it was classified. Again it could be considered that it is correctly classified with respect to the considered metric. So up to this point only the item 130 was incorrectly classified to $Cluster2$.

Let us now analyse the items from the collection U of unclustered items.

ItemId	SimC1	C1	Sim C2	C2	Sim C3	C3
59	1.06	0	0.75	0	1.08	0
110	0.91	0	0.93	0	1.11	0
121	0.73	0	0.96	0	1.2	0
123	1	0	0.9	0	0.99	0
124	0.94	0	0.86	0	1.13	0

Table 3.7: Unclustered items

As it can be seen from the Unclustered items table, all the elements from the collection U have very high similarity values with respect to each cluster and this implies membership degrees equal to zero. So these items are correctly left apart of any cluster.

Consequently, it can be argued that only item 130 is truly a misclassification with respect to the considered metric. On the other hand, it is clear that the approach has 15 classification errors if the results from [Win91] were to be taken ad litteram. This may suggest that another metric should be considered for computing the similarity between items. From a classification point of view, even in the most pessimistic result interpretation (15 classification errors), the accuracy would be 91%.

3.4 Conclusions and future work

We have introduced in this chapter new agent-based unsupervised learning approaches based on our original papers [GP10, GP11a, GP11b, Găc11, GP11c].

The algorithms presented in Section 3.2 are based on the adaptive ASM approach from [CXC04]. The major improvement is that, instead to moving the agents at a randomly selected site, we are letting the agents choose the best location. Agents can directly communicate with each other — similar to the approach from [CDG07]. In [SCCK04], the fuzzy IF-THEN rules are used for deciding if the agents are picking up or dropping an item. In our model we are using the fuzzy rules for deciding upon the direction and length of the movement. Moreover, in the approach from Section 3.2.2 the agents are able to adapt their movements if changes in the environment would occur. Case studies for these approaches have been performed in Section 3.2.3. In order to test the algorithm in a real-world scenario, the Iris and Wine datasets have been considered [Iri88, Win91]. Experiments outline the ability of our approaches to discover hybrid data. In Section 3.3 an incremental clustering algorithm is introduced. Incremental clustering is used to process sequential, continuous data flows or data streams and in situations in which cluster shapes change in time. Such algorithms are well fitted in real-time systems, wireless sensor networks or data streams because in such systems it is very hard or even impossible to store the entire datasets in memory. The algorithm considers one instance at a time and it basically tries to assign it to one of the existing clusters. Only cluster representatives have to be maintained in memory so computation is both fast and memory friendly. We have seen in the tests from the incremental approach (Section 3.3.3) that most of the apparently classification errors were actually items that have high membership degrees to more than one cluster. Nevertheless, in our opinion, it is again clear that we are dealing with hybrid data. Actually the hybrid nature of the data is suggested in [Iri88] and in [Win91] and this is the main reason for choosing these datasets for our analysis. By using fuzzy methods such features of the data are easy to be observed. The fact that there are hybrid items could be an indication of the quality of data.

For each approach proposed in this chapter we have outlined the advantages and drawbacks and emphasised improvement possibilities and directions for further extension.

As future research directions we intend to improve the approaches presented in this chapter, to extend the evaluation of the proposed techniques and to investigate the use of various metaheuristics in unsupervised learning.

Chapter 4

New supervised learning approaches to software development

This chapter is entirely original and it focuses on the problem of dynamically selecting, using supervised learning approaches, the most suitable representation for an abstract data type according to the software system's current execution context. In this direction, a neural network approach and a support vector machine approach are proposed.

The supervised learning approaches for the problem of automatic selection of data representations presented in this chapter are original works published in [CCGa] and under review in [CCGb].

Selecting and creating the appropriate data structure for implementing an abstract data type (ADT) can greatly impact the performance of a software system. It is not a trivial problem for a software developer, as it is hard to anticipate all the usage scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Due to this fact, the software system may choose the appropriate data representation, at runtime, based on the effective data usage pattern. This dynamic selection can be achieved using machine learning techniques, which can assure complex and adaptive systems development.

In this chapter we approach the problem of dynamically selecting, using supervised learning approaches, the most suitable representation for an abstract data type according to the software system's current execution context. In this direction, a neural network model and a support vector machine model are proposed. The considered problem arises from practical needs, it has a major importance for software developers. Improper use of data structures in software applications leads to performance degradation and high memory consumption. These problems can be avoided by properly selecting data structures for implementing ADTs, according to the nature of the manipulated data.

To our knowledge, so far, there are no existing machine learning approaches for the problem of automatic selection of data representations.

The chapter is structured as follows. In Section 4.1 the problem of dynamic data structure selection is presented. It is explained that this is a complex problem because each particular data structure is usually more efficient for some operations and less efficient for others and that is why a static analysis for choosing the best representation can be inappropriate, as the performed operations can not be statically predicted. A practical example is presented and an experiment is performed in order to motivate our approach. In Section 4.2 we present our first proposal of using supervised learning for dynamically selecting the implementation of an abstract data type from the software system, based on its current execution context. For this purpose, a neural network model will be used. In fact, selecting the most appropriate implementation of an abstract data type is equivalent to predicting, based on the current execution context, the type and the number of operations performed on the ADT, on a certain execution scenario. In Section 4.3 we evaluate the accuracy of the technique proposed in Section 4.2, i.e. the ANN model's prediction accuracy. Starting from a data set given at [For10], we have simulated an experiment for selecting the most appropriate data structure for implementing the *List* ADT. Experimental results suggest that our approach provides optimized data structure selection and reduces the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario. Section 4.4 presents a comparison to related work. In Section 4.5 the problem of data representation selection problem (DRSP) is approached using support vector machines. Computational experiments from Section 4.6 confirm a good performance of the proposed model and indicates the potential of our proposal. The advantages of our approach in comparison with similar approaches are also emphasized in Section 4.7.

The original contributions of this chapter are:

- To introduce a supervised learning approach for the dynamic selection of abstract data types implementations during the execution of a software system, in order to increase the system's efficiency (Section 4.2) [CCGa, CCGb] .
- To approach the considered problem using neural networks (Section 4.2.2) [CCGa].
- To evaluate the accuracy of the proposed neural network based technique on a case study (Section 4.3) [CCGa].
- To approach the considered problem using support vector machines (Section 4.5.3) [CCGb].
- To evaluate the accuracy of the proposed support vector machine based technique on a case study (Section 4.6) [CCGb].
- To emphasize the advantages of the proposed supervised learning approaches to DRSP in comparison with existing similar approaches (Section 4.4 and Section 4.7) [CCGa, CCGb] .

4.1 The problem of dynamic data structure selection

Abstract data types (ADTs) [WB01] are used in software applications to model real world entities from the application domain. An ADT can be implemented using different data structures. The study of data structures and the algorithms that manipulate them is among the most fundamental topics in computer science [Mou01]. Most of what computer systems spend their time doing is storing, accessing, and manipulating data in one form or another. There are numerous examples from all areas of computer science where a relatively simple application of good data structure techniques resulted in massive savings in computation time and, hence, money.

Selecting and creating the appropriate data structure for implementing an abstract data type can greatly impact the performance of a software system. It is not a trivial problem for a software developer, as it is hard to anticipate all the use scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Due to this fact, the software system may choose the appropriate data representation, at runtime, based on the effective data usage pattern. This dynamic selection can be achieved using machine learning techniques [Mit97], which can assure complex and adaptive systems development.

In this chapter we approach the problem of dynamically selecting, using a supervised learning approach, the most suitable representation for an abstract data type according to the software system's current execution context. In this direction, a neural network model is proposed. The considered problem arises from practical needs, it has a major importance for software developers. Improper use of data structures in software applications leads to performance degradation and improper memory consumption. These problems can be avoided by properly selecting data structures for implementing ADTs, according to the nature of the manipulated data.

To our knowledge, so far, there are no existing machine learning approaches for the problem of automatic selection of data representations.

We have chosen neural networks as a machine learning method, as we consider they are good non-linear classification models and they are useful in prediction processes. Neural networks are adaptive systems that change their structure based on the information flow. They have good accuracy and are tolerant to noisy data. Neural networks are well-suited for modelling complex relationships and for situations in which there is little knowledge between the attributes and the classes of a dataset. So, in our opinion neural networks are very good for predicting the most suitable implementation of an abstract data type.

The rest of the paper is structured as follows. Section 4.1 gives a motivation for our work. Our supervised leaning approach for a dynamic identification of the most suitable implementation of an ADT is introduced in Section 4.2. Section 4.3 provides an evaluation of the proposed model on a case study. Existing approaches in the direction of automatic selection of data representations are presented in Section 4.4. We also provide a comparison of our approach with other similar existing approaches. Section 4.8 contains some conclusions of our paper and future research directions.

The primary justification for a data structure selection approach is that finding good representations has proven to be difficult. This is mainly due to a lack of awareness of the importance of the proper choices for data structures, issue that has a major impact on software systems' performance.

Each software system use abstract data types [WB01] to model real world entities from the application domain. Because abstract data types represent the core for any software application, a proper use of them is an essential requirement for developing a robust and efficient system. We are focusing in this paper on container abstract data types, such as: collections, sets, lists and maps [CLRS09].

The design and implementation of efficient abstract data types are important issues for software developers. An ADT provides a set of operations that can be performed on it (the ADT's *interface/contract*). There are several possible implementations for an ADT, each implementation has to satisfy the ADT's contract. Each possible implementation of an abstract data type uses certain representations (*data structures*) for the elements within the ADT's domain. Thus, selecting and creating the appropriate data structure for implementing an abstract data type can greatly impact the performance of the system. It is not a trivial problem for a software developer, as it is hard to anticipate all the use scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Previous approaches rely on compile-time analyses or programmer annotations [CH96].

A common situation is where there are several data structures for implementing an ADT, with one data structure more efficient than the others for certain operations from the ADT's interface but worst for the remaining operations, and vice versa [CH96]. The problem is how to choose the most appropriate data structure for implementing the ADT, given that there is no *a priori* knowledge of what operations, and how often, the ADT will be mostly used for? This is known as the *data representation selection problem* [SSS81], or *data structure selection problem* [SSS79]. The problem can be considered for built-in data types (such as List, Collection), as well for user-defined abstract data types which can be implemented with several data structures.

The problem of automatic data structure selection is a complex one because each particular data structure is usually more efficient for some operations and less efficient for others and that is why a static analysis for choosing the best representation can be inappropriate, as the performed operations can not be statically predicted.

A "static" approach in solving the above presented problem can be, for example, to design a data structure for implementing the abstract data type based on an asymptotic analysis of the computational complexities [CLRS09] of the ADT's operations. Even if the data structure selected this way is not the best possible, its performance can be good enough. It is obvious that these "static" methods can be unreliable because they try to predict a program's behaviour before it is executed.

Another problem with the "static" approaches is the following. The considered ADT may be instantiated in the same software system in different contexts. A "static" selection method will indicate the same data structure for implementing all ADT's instantiations. Even if the behaviour of the ADT's operations are the same for all its possible implementations, it is very likely that different implementations may perform differently (in term of computational complexity). In different execution scenarios, different operations from the ADT's interface are executed. Consequently, in order to achieve optimal performance (in term of reduced time complexity), the most appropriate implementation for the abstract data type has to be dependent on the execution scenarios.

That is why we propose a "dynamic" approach, i.e. the selection of the most suitable data structure for implementing an abstract data type to be made at run-time, during the execution of the software

	Vector	**Linked List**	**Balanced Search Tree**
Insertion	$O(n)/O(1)$	$O(1)/O(1)$	$O(log_2(n))/O(log_2(n))$
Deletion	$O(n)/O(n)$	$O(n)/O(n)$	$O(log_2(n))/O(log_2(n))$
Searching	$O(n)/O(n)$	$O(n)/O(n)$	$O(log_2(n))/O(log_2(n))$

Table 4.1: Worst case/amortized time complexity.

system.

In the following subsections we give two examples in order to illustrate the importance of the analysed problem, and, moreover, the need for a dynamic data structure selection, instead of a static one.

4.1.1 Example

Let us consider that in a software application a *Collection* ADT (also known as *Bag*) is used. The main operations supported by a *collection* of elements are: **insertion** of an element into the collection, **deletion** of an element from the collection and **searching** an element in the collection.

The *Collection* ADT can be implemented with several data structures: a **vector** (dynamic array), a **linked list** or a **balanced search tree** if the type of the bag's elements is *ordinal*. Denoting by n the current size of the collection, we give in Table 4.1 the worst case as well as the amortized [CLRS09] time complexities of the collection's operations for different implementations.

From Table 4.1, we can conclude the following:

- If the collection will be used mostly for **insertion** operations, then *linked list* or *vector* implementation is preferred. As a bag is a container of elements which lacks order, we mention the following:
  - the insertion in a *linked list* will be made in the front of list in $O(1)$ time.
  - the insertion in a *vector* will be made at the end of it and it may require reallocation of the vector body. Thus, this operation requires $O(n)$ in the worst case, but it's amortized time complexity is still $O(1)$.
- If we have a large number of **deletions** and/or membership queries (**searching**) operations, then *balanced search tree* implementation is preferred.

Using the assumption that the three operations in the collection will be used with the same probability (i.e. $\frac{1}{3}$), we can conclude that the most appropriate data structure for implementing the collection is the *balanced search tree.*

But, in certain situations, this decision may be incorrect, as the number of operations performed on the collection during the execution of the system can not be predicted before its execution. More exactly, there is no *a priori* knowledge of how the collection looks like, including the total number of operations in the collection.

For example, let us consider an execution scenario in which the collection has, at a given moment 10 elements. On this collection, we perform 2 searches and 6 insertions. As we cannot predict the actual values of the elements from the bag, we will consider the amortized case time complexity for each possible operation. Using the time complexities from Table 4.1, we obtain the following:

- If the collection would be implemented using a *vector* or a *linked list*, the needed time would be approximately $2 \cdot 10 + 6 \cdot 1 = 26$.
- If the collection would be implemented using a *balanced search tree*, the needed time would be approximately $2 \cdot log_2 10 + \sum_{i=10}^{15} log_2 i \approx 28.42$.

	Vector	**Linked List**	**Balanced Search Tree**
Access	$O(1)$	$O(n)$	$O(log_2(n))$
Insertion at the beginning	$O(n)$	$O(1)$	$O(log_2(n))$
Insertion at the end	$O(n)$	$O(1)$	$O(log_2(n))$
Insertion at a given position	$O(n)$	$O(n)$	$O(log_2(n))$
Deletion at a given position	$O(n)$	$O(n)$	$O(log_2(n))$
Searching	$O(n)$	$O(n)$	$O(log_2(n))$

Table 4.2: Worst case time complexity.

So, based on the dynamic analysis, the most appropriate data structure for implementing the collection is the *vector* or *linked list*.

Similarly to the above presented example, if we know the number of operations performed on the ADT, we can select the ADT implementation that offers the best possible performance for the given usage scenario. We can conclude that the problem of predicting the most appropriate data structure for implementing the abstract data type during the execution has to be done dynamically and is equivalent to predicting, based on the current execution context, the type and the number of operations performed on the ADT.

4.1.2 Experiment

In order to better motivate our approach, we performed an experiment considering the *List* ADT and three data structures for implementing a *List*: **vector** (dynamic array), **linked list** and **balanced search tree**. The main operations supported by a *list* of elements are: **insertion** of an element into the list (at the beginning, at the end, at a certain position), **deletion** of an element from the list (a given element or from a given position), **searching** an element in the list, **iterating** through the list, **accessing** an element from the list at a certain position and **updating** an element from a certain position.

We have chosen *List* ADT in our experiment, as it is one the most used ADTs in real software systems. For comparing the performance of the considered data structures, we have defined several usage scenarios of a *List*, where a usage scenario is a set of pairs (*operation*, *usage probability*).

In order to outline the practical nature of the considered problem, we have chosen a popular programming language (Java) and three existing implementations of *List* ADT:

1. *java.util.ArrayList* which implements *List* functionalities using the **vector** data structure.
2. *java.util.LinkedList* which implements *List* functionalities using the **linked list** data structure.
3. *java.util.TreeList* which implements *List* functionalities using the **balanced search tree** data structure.

Denoting by n the current size of the list, we give in Table 4.2 the worst case time complexities for the main operations of the list for different implementations.

From Table 4.2, we can conclude the following:

- If the list will be used mostly for **accessing** elements from it, then *vector* implementation is preferred.
- If the list will be used mostly for **adding** elements at the beginning and at the end of it, then *linked list* implementation is preferred.

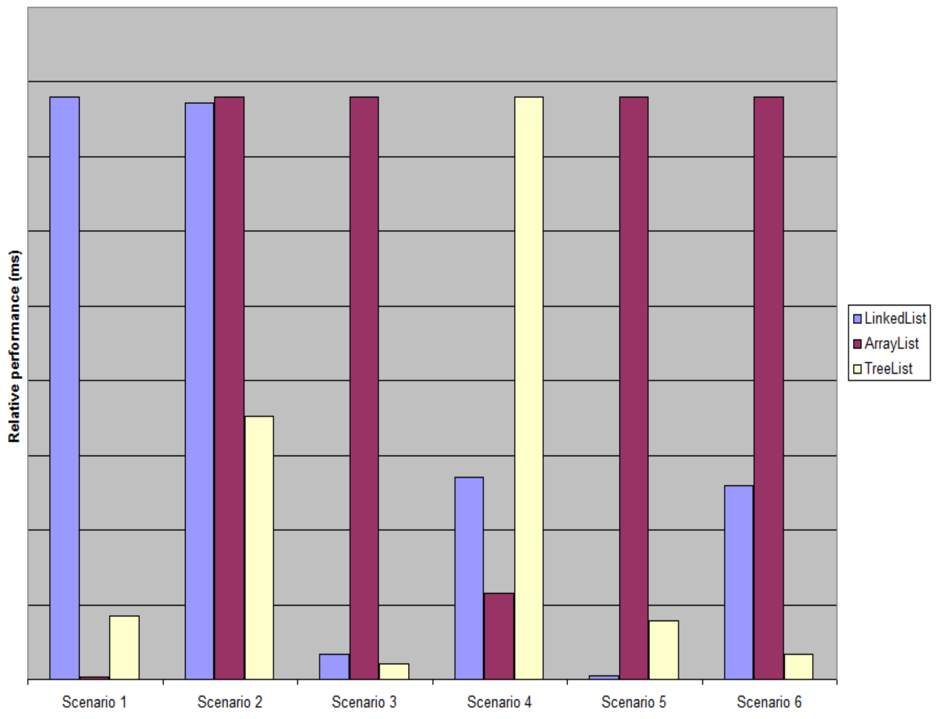


Figure 4.1: Execution scenarios

- If we have a large number of **deletions** at random positions and/or membership queries (**searching**) operations, then *balanced search tree* implementation is preferred.

The experiment was performed as follows. For each *List* implementation, the defined scenarios were executed on lists with 5000 elements. The total number of performed operations was 10000 at each execution. The type of the executed operations were chosen according to their probabilities within the usage scenario. In order to capture the performance of a certain implementing data structure, the execution time for each scenario was measured.

In Figure 4.1 we illustrate, in each usage scenario, the relative time performance of the data structures used for implementing the *List*. Lower bars indicate lower execution time and, consequently, better performance. From Figure 4.1, it can be seen that we can not decide what is the most appropriate implementation of the *List*, because a particular implementation leads to different performances in different execution scenarios. Consequently, we conclude the following:

- In Scenario 1, the implementation of the *List* that leads to a better performance (lower time) is *ArrayList*.
- In Scenario 2, the implementation of the *List* that leads to a better performance (lower time) is *TreeList*.
- In Scenario 3, the implementation of the *List* that leads to a better performance (lower time) is *TreeList*.
- In Scenario 4, the implementation of the *List* that leads to a better performance (lower time) is *ArrayList*.
- In Scenario 5, the implementation of the *List* that leads to a better performance (lower time) is *LinkedList*.
- In Scenario 6, the implementation of the *List* that leads to a better performance (lower time) is *TreeList*.

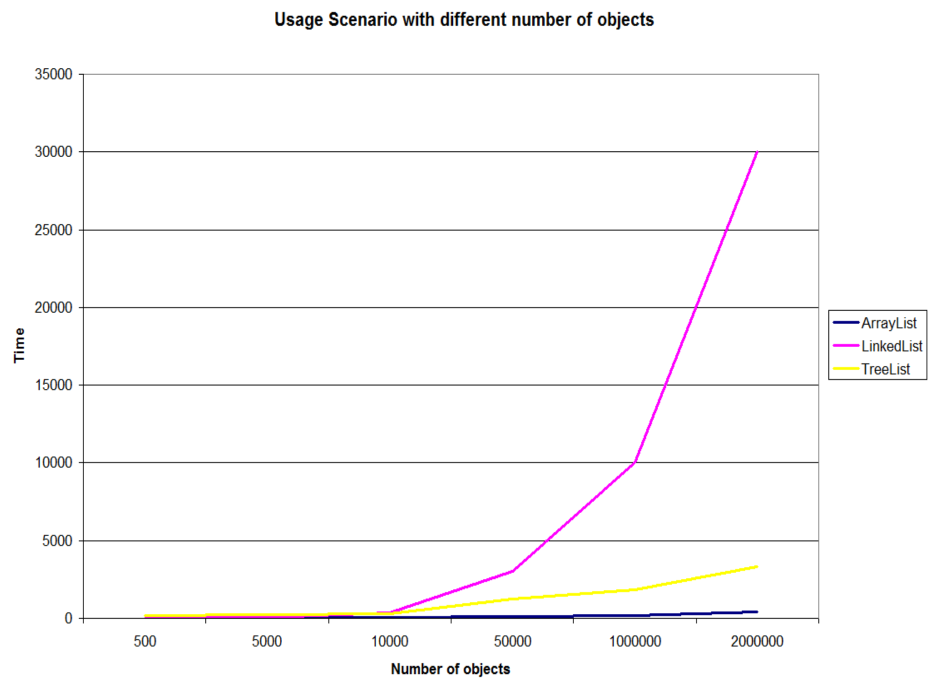


Figure 4.2: Performance

As the number of the elements from the list grows, a proper data structure selection becomes more important. We depict in Figure 4.2, for Scenario 1 (indicated in Figure 4.1), how the time (in milliseconds) needed for executing the scenario grows with the size of the list, for each possible implementation of the *List* (*ArrayList*, *LinkedList*, or *TreeList*). From Figure 4.2 it obvious that the best implementation for *List* in the usage scenario 1 is *ArrayList*, as it was indicated in Figure 4.1.

The experiments described in this section confirms the importance of proper data structure selection and reveal the need for a dynamic decision according to the software system's actual usage scenario.

4.2 Automatic selection of data representations using ANN

Data structures [WB01] provide means to customize an abstract data type according to a given usage scenario. The volume of the processed data and the data access flow in the software application influence the selection of the most appropriate data structure for implementing a certain abstract data type. During the execution of the software application, the data flow and volume is fluctuating due to external factors (such as user interaction), that is why the data structure selection has to be dynamically adapted to the software system's execution context. This adaptation has to be made during the execution of the software application and it is hard or even impossible to predict by the software developer. Consequently, in our opinion, machine learning techniques would provide a better selection at runtime of the appropriate data structure for implementing a certain abstract data type.

In this section we will present our proposal of using supervised learning for dynamically selecting the implementation of an abstract data type from the software system, based on its current execution context. For this purpose, a neural network model will be used. In fact, selecting the most appropriate implementation of an abstract data type is equivalent to predicting, based on the current execution context, the type and the number of operations performed on the ADT, on a certain execution scenario.

First, a brief background about artificial neural networks will be provided.

4.2.1 Formal aspects

Artificial neural networks are emerging as the technology of choice for many applications, such as pattern recognition, speech recognition [SH07], prediction [LB05], system identification and control.

An *artificial neural network* (ANN) [Roj96] is an adaptive system which learns a mapping from input output data. The system parameters are changed during operation, called the *training phase*, and in that sense the system is adaptive. With the training phase being completed, the ANN may be employed in problem solving (the testing phase). The ANN has a built in stepwise procedure to optimize a performance measure or to follow some implicit internal constraint, which is in general referred to as the learning rule.

In a supervised learning scenario, an input is presented to the neural network and a corresponding desired or target response set at the output [KKvdSS96]. These input-output pairs are often provided by an external teacher (supervisor). An error is composed from the difference between the desired response and the system output. This error information is fed back to the system and adjusts the system parameters in a systematic fashion (the learning rule). The process is repeated until the performance is acceptable.

Software systems are composed of several software components which are usually required to be easily extensible, allowing modifications to occur with minimal impact on the user. In order to achieve this requirement software components should first of all be accessible though interfaces such that the user is immune to any implementation changes. The object-orient programming paradigm offers solid grounds for achieving the separation between interface and implementation. In an object-oriented language, objects are instances of abstract types (classes) which consist of operations (methods) and components (attributes). Type relationships like inheritance, composition, aggregation and delegation are central in an object-oriented software system; for example inheritance enables the binding between interface and implementation types.

However, for our problem, we only need a subset of all these elements. Moreover we are interested in aspects of the software system's state at run-time rather than various static relationships. The entire reasoning from our approach uses the notion of execution context which will be introduced in this section together with some other elements that we need in order to address the data structure selection problem.

Let $S = \{s_1, s_2, ..., s_n\}$ be an object oriented software system, where $s_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class.

We will consider that:

- $Class(S) = \{C_1, C_2, \ldots, C_l\}$, $Class(S) \subset S$, is the set of applications classes of the software system S.
- Each application class C_i $(1 \leq i \leq l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, .$ $p_i \leq n,\ 1 \leq r_i \leq n$, 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 C_i.
- $Meth(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system S.
- $Attr(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system S.

Based on the above notations, the software system S can be defined as in Equation (4.3):

$$S = Class(S) \bigcup Meth(S) \bigcup Attr(S). \tag{4.1}$$

Let us consider that $\mathcal{T}$ is an abstract data type having in its interface a set of operations $\mathcal{O}$ that can be performed on an instance of $\mathcal{T}$. We also consider a set $\mathcal{D}_{\mathcal{T}} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n\}$ of data structures that can be used in the software system S for implementing $\mathcal{T}$. If a given instance c of a class $C_i \in Class(S)$ uses $\mathcal{T}$, the selection of the appropriate data structure from $\mathcal{D}$ that has to be used for implementing (and instantiating) $\mathcal{T}$ depends on the current *execution context* (the current state of c).

The **execution context** of $\mathcal{T}$ represents the state of $C_i \in Class(S)$ at runtime, when $\mathcal{T}$ is initialized. So the execution context of C_i can be seen as the set of all attribute values from C_i in the current execution step:

$$\mathcal{EC}_{\mathcal{T}} = \{v_{i1}, v_{i2}, \ldots, v_{ir_i}\}, \tag{4.2}$$

where r_i represents the number of attributes from class C_i and $\forall 1 \leq j \leq r_i$, v_{ij} denotes the value of attribute a_{ij} in the current execution step.

As it can be seen the size r_i of the *execution context* $\mathcal{EC}_{\mathcal{T}}$ does not depend on the size n of the software system S and hence our approach is scalable.

As we have shown in Section 4.5.1.1, it is very likely that in certain situations a proper selection of the most suitable data structure from $\mathcal{D}_{\mathcal{T}}$ that must be used for an efficient implementation of $\mathcal{T}$ has to be done at runtime by analyzing the current *execution context*, as the volume of data manipulated by the implementation of $\mathcal{T}$ is unpredictable at the development stage.

The most suitable implementation of an abstract data type $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$ is the data structure $\mathcal{D}_i \in \mathcal{D}$ that provides an efficient implementation of $\mathcal{T}$ in the execution scenario, i.e., the time needed for performing the operations on $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$ is minimized.

In fact, the most proper implementation of $\mathcal{T}$ in a given execution scenario depends on the type and the number of operations from $\mathcal{O}$ that are performed on $\mathcal{T}$ during the execution. Thus, we can conclude that the problem of dynamically selecting the *suitable implementation* of $\mathcal{T}$ is, in fact, the problem of predicting, based on the current execution context, the type and the number of operations performed on $\mathcal{T}$ on a certain execution scenario.

Considering the example given in Section 4.5.1.1, the most suitable implementation of ADT *Collection* in the execution context in which the collection has 10 elements, and 2 **searches** and 6 **insertions** are performed on it is *vector* or *linked list.*

4.2.2 Methodology

In this subsection we introduce our approach for predicting, at runtime, the most suitable implementation of an abstract data type $\mathcal{T}$, based on the system's current *execution context.*

Let us consider, in the following, the theoretical model from Subsection 4.5.2. Considering the issues described in the previous sections, for providing the most appropriate data structure for implementing an abstract data type $\mathcal{T}$ at runtime based on the system's current *execution context* we propose a *supervised learning* [Mit97] approach.

In a software system $\mathcal{S}$, multiple occurrences of $\mathcal{T}$ can be found. We mention that not all these r are subject for optimization, but only those that are considered to have a great impact on the system's performance. The software developer is responsible for selecting the locations in S where optimization is needed. So, let us focus, in the following, on a single occurrence of an abstract data type $\mathcal{T}$, for which n possible implementations exist, i.e. n data structures $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n$.

In fact, the problem of identifying the most suitable implementation of $\mathcal{T}$ can be viewed as a *classification* [Mit97] problem where each input is represented by an *execution context.* The output for a given *execution context* $\mathcal{EC}_{\mathcal{T}}$ is the index $i, 1 \leq i \leq n$ where $\mathcal{D}_i$ is the *most suitable implementation of $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$* .

The process takes place in two phases that reflect the principles of a supervised learning algorithm: *training* and *testing.* During the training a classification model will be built, and during testing, the model built during the training will be applied for classifying an unseen *execution context* (input), i.e to predict the *most suitable implementation* of $\mathcal{T}$ given the input *execution context.* We propose an *artificial neural network* classifier for solving this problem.

We summarize the main stages of our approach:

1. **During software development**: Several scenarios are executed (usually this is accomplished by running automated tests written for testing the software), we are interested in scenarios that use $\mathcal{T}$. During the execution of this scenarios data regarding the use of $\mathcal{T}$ and the execution context at creation of $\mathcal{D}$ is collected, this data will be used as training data. At this stage, the features selected for classification are identified based on the execution context of $\mathcal{T}$, training data is collected (this step will be detailed in Subsection 4.5.3.1) and the ANN is built/trained using the already collected data.

2. **Execution stage**. When the software system is deployed, the *classification* step takes place. The ANN classifier built during the previous stage will be applied in order to identify the most suitable implementation for the ADT $\mathcal{T}$, considering the current execution context. We mention that we are not focusing on switching the representation of $\mathcal{T}$. Consequently, our approach for a dynamic selection of an ADT's representation needs additional run-time overhead only for the classification using the ANN when $\mathcal{D}$ is instantiated.

Based on the considerations above, for predicting the most *suitable implementation* of a certain abstract data type $\mathcal{T}$, the following steps will be performed:

1. Data Collection, Conversion and Pre-processing.

2. Design of Neural Network.

3. Training.

4. Testing.

We mention that an inappropriate prediction does not impact the execution scenario of the software system. Even if the data structure selected for the implementation of the abstract data type $\mathcal{T}$ is not the most suitable one, the system execution is still successful, even if not the most efficient. We can imagine a classification model in which, if such an inaccurate prediction is made, a feed-back is sent and the model is correspondingly adapted. The idea would be to send a *reward* (*reinforcement*) to the model, as in a *reinforcement learning* [SB98] based model.

In the following we will briefly describe the first two steps. Details about **Training** and **Testing** step will be given in the experimental part (Section 4.3).

4.2.2.1 Data collection, Conversion and Pre-processing

First, the software system S is monitored during the execution of a set of scenarios that include the instantiation of the abstract data type $\mathcal{T}$. The result of this supervision performed by a software developer is a set of *execution contexts*, as well as the type and the number of operations from $\mathcal{O}$ performed on $\mathcal{T}$ saved in a log file. The software developer will analyze the resulted log file and will decide, for each *execution context* (input) , the *most suitable implementation* for $\mathcal{T}$ given the *execution context* (output). This decision will be based on computing the global computational complexity of the operations performed on $\mathcal{T}$ during the scenario given by the *execution context* for each possible implementation of $\mathcal{D}_i$ of $\mathcal{T}$, and then selecting the implementation that minimizes the overall complexity. The creation of training sets is semi-automatic, as the software developer has only to execute several usage execution scenarios, the actual execution context (training sample) being automatically collected.

This way, a data set consisting of (input, output) pairs is built. An input represents an *execution context* $\mathcal{EC}_\mathcal{T}$ of the abstract data type $\mathcal{T}$ and the associated output represents the the index $i, 1 \leq i \leq n$ where $\mathcal{D}_i$ is the *most suitable implementation of* $\mathcal{T}$ *given the execution context* $\mathcal{EC}_\mathcal{T}$.

The input data from the resulting data set (the *execution contexts*) is now converted and pre-processed.

First, the values of the attributes within an *execution context* are converted to numerical values. This conversion is made as follows:

- An ordinal type attribute (integer, character, boolean, enumerated type) is converted to the ordinal position of the attribute value in its type domain.
- For a numerical non-ordinal attribute (float, double), the actual value of the attribute remains unchanged.
- For a string attribute or other user defined types, the corresponding numerical value is determined using a hash function [CLRS09]. Such a hash function is available in most object-oriented programming languages. Even if two different attributes values can have the same hash value, the combination of all attributes values from the execution context may assure a uniqueness for the input data.

After the training set was collected, the input data is scaled to [0,1], and a statistical analysis is carried out in order to find a subset of attributes (from the *execution contexts*) that are correlated with the target output. The statistical analysis on the attributes from the *execution contexts* is performed in order to reduce the dimensionality of the input data (the execution contexts), by eliminating attributes from the *execution context* which do not influence the output value.

To determine the dependencies between attributes and the target output, the Spearman's rank correlation coefficient [Spe87] is used. A Spearman correlation of 0 between two variables X and Y indicates that there is no tendency for Y to either increase or decrease when X increases. A Spearman correlation of 1 or -1 results when the two variables being compared are monotonically related, even if their relationship is not linear. At the statistical analysis step we remove from the *execution context* those attributes that have no significant influence on the target output, i.e are slightly correlated with it. A slight correlation is indicated by a value that is very close to 0.

The data set preprocessed this way can now be used for building the classification model.

Example

In the following we will give an example of an *execution context*, as well as how the data is preprocessed.

We will consider a real software system, a DICOM (*Digital Imaging and Communications in Medicine*) [DICiM11] and HL7 (*Health Level 7*) [HL0] compliant PACS (*Picture Archiving and Communications System*) system, facilitating medical images management, offering quick access to radiological images, and making the diagnosing process easier. The analyzed application is a large distributed system, consisting of several subsystems in form of stand-alone and web-based applications.

In our example, we have considered one of the subsystems from this application, a stand-alone Java application used by physicians in order to interpret radiological images. The application fetches clinical images from an image server (using DICOM protocol) or from the local file system (using DICOM files), displays them, and offers various tools to manage radiological images.

Radiological images are produced by medical devices called modalities. The modalities generate multiple related images, organized in series.

Let us consider the source code presented in Figure 4.3, extracted from the analyzed software system, where **ImageSeries** class represents an image serie acquired by a medical device for a given human body part. There are multiple type of imaging devices (Computer tomgraph,Angiograph, etc), each device will produce different kind of medical image and the resulted image can be used in diagnosis process in different way (3Dreconstructing,view as a slide show, view side by side, etc). Different scenarios will manipulate the list of images in a different manner. That is why we can apply our approach for dynamically selecting the best implementation for the image list from an image serie, according to the current execution context.

In the source code from Figure 4.3, **ImageSeries** class has an attribute **images** of type **List** (line 10). The **List** interface from Java corresponds to the *List* ADT, i.e. $\mathcal{T} = List$. On line 20, the **images** attribute has to be initialized. It can be initialized with an **ArrayList** instance, or an **LinkedList** instance, or a **TreeList** instance. In fact it is about selecting the most appropriate implementation for the *List* ADT, i.e *dynamic array*, *linked list* or *balanced search tree* data structures.

```java
public class ImageSeries {

   private String modality;
   private String code;
   private String bodyPart;
   private String UID;
   private String manuf;
   private String model;
   private String description;
   private String localizer;
   private List<Image> images;

  public Series(DicomObject d) {
    UID =
     d.getString(Tag.SeriesInstanceUID);
    bodyPart =
     d.getString(Tag.BodyPartExamined);
    modality =
     d.getString(Tag.Modality);
    localizer = d.getString(Tag.ImageType);
    manuf = d.getString(Tag.Manufacturer);
    model =
     d.getString(Tag.ManufacturerModelName);
    description =
     d.getString(Tag.SeriesDescription);
    code =
     d.getString(Tag.RequestedProcedureID);
    // images = new ArrayList<Image>();
    // images = new LinkedList<Image>();
    // images = new TreeList<Image>();
  }

  public void setCode(String code) {
    this.code = code;
  }

  public void addImages(Image img) {
    images.add(img);
  }

  public boolean isLocalizer() {
    if (imgType == null) {
      return false;
    }
    return imgType.contains("LOCALIZER");
  }

  public Modality getModality() {
    return Modality.valueOf(modality);
  }

  public String getBodyPart() {
    return bodyPart;
  }

  public String getManuf() {
    return manuf;
  }

  public String getModel() {
    return model;
  }

  public String getProcT() {
    return code;
  }

  public String getDescr() {
    return description;
  }

  public String getUID() {
    return UID;
  }

  public int getNoOfImages() {
    return images.size();
  }

  public boolean equals(Object o) {
    return ((Series) o).UID.equals(UID);
  }
}
```

Figure 4.3: Java code example.

Modality	**Body Part**	**Localizer**	**Manufacturer**	**Model**	**Code**	**Description**	**UID**
8.0	0.0	1237.0	-736.0	0.0	0.0	0.0	-194.0
0.0	0.0	1237.0	0.0	0.0	0.0	990.0	939.0
8.0	0.0	1237.0	-130.0	0.0	0.0	0.0	995.0
0.0	0.0	1237.0	0.0	0.0	0.0	-574.0	970.0
5.0	0.0	1237.0	-116.0	680.0	0.0	0.0	-582.0
4.0	0.0	1237.0	-512.0	-413.0	0.0	0.0	-58.0
2.0	0.0	1237.0	-514.0	-841.0	0.0	0.0	-380.0
2.0	0.0	1237.0	-514.0	-841.0	0.0	0.0	-679.0
0.0	0.0	1237.0	0.0	0.0	0.0	-930.0	877.0
8.0	0.0	1237.0	104.0	282.0	0.0	0.0	-111.0
3.0	825.0	1237.0	-203.0	-396.0	0.0	0.0	-885.0
8.0	-782.0	1237.0	-116.0	-128.0	12.0	-777.0	-622.0
8.0	-782.0	1231.0	-116.0	-128.0	12.0	696.0	-784.0
8.0	0.0	1237.0	-500.0	-500.0	0.0	432.0	646.0
5.0	0.0	1237.0	-116.0	364.0	903.0	-95.0	855.0

Table 4.3: Input data.

Assuming that we want to apply our approach for dynamically selecting the most appropriate implementation for the *List* ADT from an image serie, according to the current execution context. As we have indicated in Subsection 4.5.2, the features characterizing the *execution context* of the *List* ADT are the attributes of the **ImageSeries** class. Consequently, there are 8 initial features considered for building our classification model: *modality*, *code*, *bodyPart*, *UID*, *manuf*, *model*, *description*, *localizer*.

In order to obtain different usage execution contexts, we have executed 15 execution scenarios with image series samples which were obtained from publicly available DICOM image files [Osi10] [RiplsDis10] [cir10] [oPDsf10] [hp10]. The images are real images from real patients, but anonymized for confidentiality reasons. For managing the DICOM image files, an open source implementation of the DICOM standard was used [sciom11].

For each image serie sample, the *execution contexts* were collected, and the appropriate implementation of the *images* list was obtained by inspecting the log files. In Figure 4.3 we give the 15 samples used for our experiment and the values of the classification features that were collected during the performed execution scenarios. As we have mentioned above, the string values are converted to a numerical value using a hash function, in our experiment this hash function is based on the hash function provided by the Java programming language.

4.2.2.2 The design of the ANN

We will use a *feedforward* neural network that will be trained using the *backpropagation-momentum* learning technique [RN02].

The design of the neural network involves designing the input neurons, the hidden processing elements, and the output neurons. We make the following remarks regarding the network architecture:

- The number n_i of input neurons is given by the dimensionality of the input data (*execution context*), collected at the previous step (Subsection 4.5.3.1), as determined after the statistical analysis. An input neuron is used for each attribute resulted after the analysis. A bias neuron is also used.

- The number n_o of output neurons is equal to the number of data structures that can be used for implementing the abstract data type $\mathcal{T}$.

- There is a single hidden layer. The number n_h of neurons from the hidden layer is computed as $n_h = \sqrt{n_i \cdot n_o}$.
- In order to speed up the convergence of the *backpropagation* algorithm and to avoid local minima, we have used the *momentum technique* [RHW86].

The connections between the network's neurons are for the feed forward activity. There are connections from every neuron in the input layer to every one in the hidden layer, and in turn, from every neuron in the hidden layer to every neuron in the output layer. The connections in the network are weighted. Using back propagation algorithm, in the training set, the weights are modified so as to reduce the mean squared error between the network's prediction and the actual target value. These modifications are made in the reverse direction, from the output layer, through the hidden layer, till the terminating condition is reached.

The classification process takes place in two phases that reflect the principles of a supervised learning algorithm: *training* and *testing*. As in any classification process, during the training the ANN classification *model* will be built, and during testing, the model built during the training will be applied for classifying an unseen *execution context* (input).

Therefore, the data set collected at step 1 has to be divided in two parts: a part for *training* and a part for *testing*. The ANN model, which is learnt in this manner, classifies the unseen *execution contexts* from the test set, which is disjoint to the training one.

The ANN system presented above is fully implemented in JDK 1.6.1. It was developed (designed, implemented) by us without using any third party libraries, using our neural network implementation.

For the monitored abstract data type $\mathcal{T}$ from the software application, a modified data structure implementation is used. This data structure implementation delegates all its methods to the actual data structure implementation, collects the state of the objects from the *execution context* and writes the collected data into a log file. The modified data structure implementation has a reference to its container object (received as a constructor parameter) and uses it in order to determine the current state of the object.

4.3 Experimental evaluation

In this section we aim at evaluating the accuracy of the technique proposed in Section 4.2, i.e. the ANN model's prediction accuracy.

As there is no publicly available case study for the problem of automatic selection of data representations, nor a case study in the related literature that can be reproduced, we consider our own case study. We describe in this section simulation results of applying our learning based approach to a selection problem that will be described below.

4.3.1 Case study

Starting from the data set given at [For10], we have simulated an experiment for selecting the most appropriate data structure for implementing the *List* ADT. The considered data set consists of the results of a chemical analysis of wines grown in the same region in Italy but derived from different cultivars. The analysis determined the quantities of 13 constituents found in each types of wines [Win91].

In the proposed case study, we consider a *WineShop* application that stores information about different wines and allows the user to locate wine shops according to different search criteria. In our simulation, we are focusing on the main class of the application, the class *Wine*, which models a type of wine by 13 constituents [Win91]. This class has a method *getShops* that returns a *List* of shops where the current wine is available. For the *List* ADT returned by *getShops* method we aim at identifying the most suitable data structure for implementation. As presented in Subsection 4.5.1.1, three data structures for implementing a *List* are considered: **vector** (dynamic array), **linked list** and **balanced search tree**.

The *List* of shops is used in three different usage scenarios:

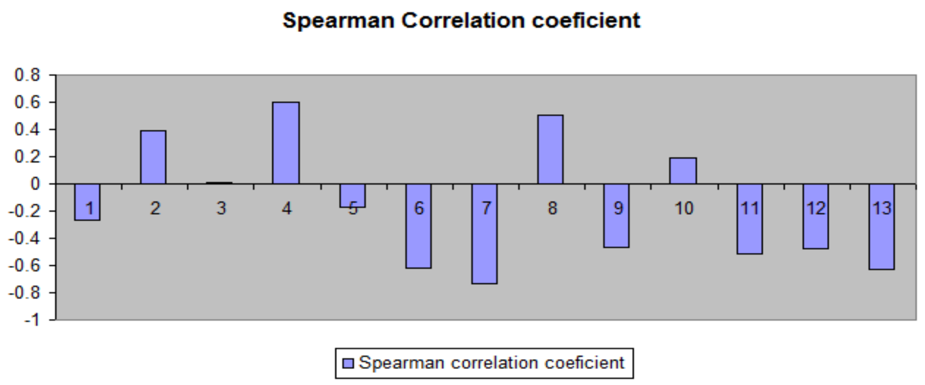


Figure 4.4: Spearman correlation

S1. The *List* of shops is presented to the user sorted according to different user selected criteria. In this scenario, the main operations performed on the *List* are: **accessing** an element from a certain position and **updating** an element from a certain position. Consequently, the **dynamic array** data structure would be the most efficient implementation for the *List* ADT.

S2. The *List* of shops is presented to the user filtered according to different user selected criteria. In this scenario, the main operation performed on the *List* is **removing** an element from a certain position. Consequently, the **balanced search tree** data structure would be the most efficient implementation for the *List* ADT.

S3. The *List* of shops is presented to the user by concatenating the shops lists from different wine types. In this scenario, the main operation is **adding** an element at the end of a list. Consequently, the **linked list** data structure would be the most efficient implementation for the *List* ADT.

In practice, the selection of most appropriate implementation of the *List* is difficult, even impossible for a software developer, as it is hard to anticipate the data flow during the execution of the application. That is why we apply our supervised learning approach for a dynamic selection of the suitable implementing data structure of the *List*, according to the system's current execution context.

4.3.2 Data collection and pre-processing

The data set for evaluating the ANN classification model presented in Section 4.2 consists of (input, output) samples collected and pre-processed as we have described in Subsection 4.2.2. An input represents an *execution context* and the target output is the *most suitable implementation* for the *List* ADT (1, 2 or 3 according to the selected implementation). In our case study, as the instantiation of the *List* ADT occurs in the *Wine* class, an *execution context* will contain the values of the attributes of this class (13 attributes corresponding to the wine constituents described at [Win91]). The collected data set consists of 178 input-output samples and will be denoted by $\mathcal{D}$.

As we have mentioned above, there are 13 initial features used for the classification. As we have shown in Subsection 4.5.3.1, all these features are measurable at run-time. After the collected data was scaled, a statistical analysis is carried out in order to find a subset of features that are correlated with the target output. To determine the dependencies between features and the target output, the Spearman's rank correlation coefficient [Spe87] is used.

Figure 4.11 shows the correlations between the considered features (13 attributes from the *execution context*) and the target output computed using the Spearman's rank correlation coefficient [Spe87].

After an analysis of the correlation between the attributes from the *execution context* and the target output, we can conclude that *Attribute* 3 is slightly correlated with the target output. Consequently, we can consider that the above mentioned attribute has no significant influence on the target output.

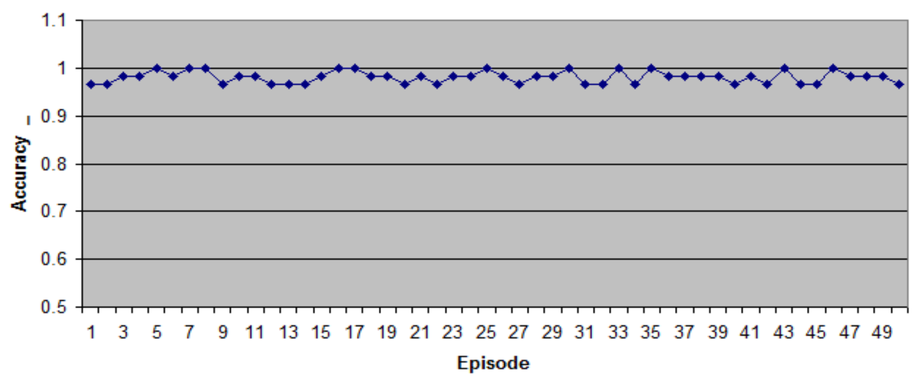


Figure 4.5: Learning accuracy/episode

Thus, after the performed statistical analysis, there are 12 attributes identified to be relevant for our learning task. Consequently, the neural network for this learning task has 13 input neurons (12 neurons corresponding to the selected attributes of the $WineShop$ class and a bias neuron) and 3 output neurons corresponding to the 3 possible implementations of the *List* ADT.

4.3.3 Testing

In order to obtain the accuracy of the network, considering the data set $\mathcal{D}$, the following cross-validation is used.

- For a given number of episodes (50 in our experiment) repeat the following:
  - $\frac{2}{3}$ randomly selected samples from $\mathcal{D}$ are used for training the network. Thus, for each training sample, the corresponding (pre-processed) *execution context* with the associated output is provided to the network.
  - The rest of $\frac{1}{3}$ samples from $\mathcal{D}$ are used for testing. After the training was completed, the network is tested as follows. The *execution context* within the testing sample is sent to the neural network. After receiving the *execution context*, the neural network will predict the most appropriate output (the most *suitable implementation* of the abstract data type $\mathcal{T}$ considered for optimization).
  - The *learning accuracy* on the current episode is computed as the number of accurate predictions divided by the number of testing samples.
- The overall *learning accuracy* of the network is computed as the average of the learning accuracies over all episodes.

The *learning accuracies* obtained for each episode are given in Figure 4.5. An average *learning accuracy* of 0.98133 was obtained.

As we can see in Figure 4.6, the results are stable, a standard deviation of 0.01231079 on the learning accuracies was obtained. The low value of the standard deviation indicates the precision of the proposed approach.

4.3.4 Discussion

Considering the experimental results presented above, we can conclude that our approach provides optimized data structure selection and reduces the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario.

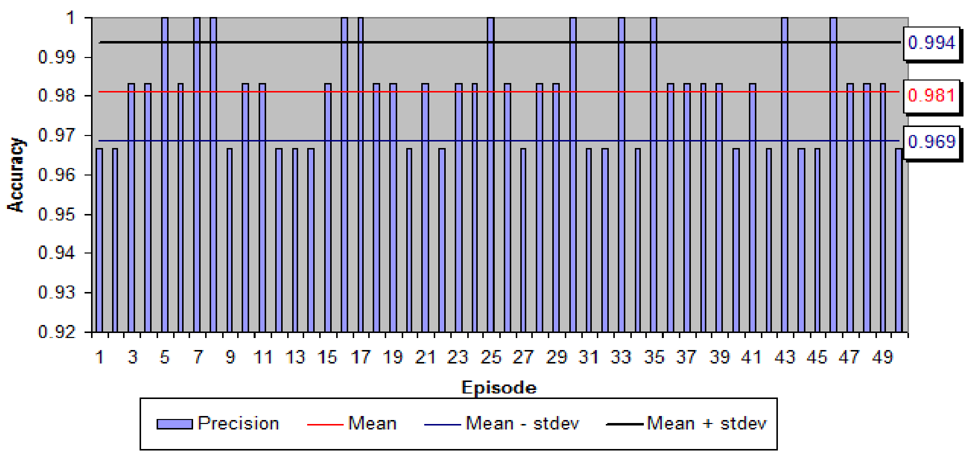


Figure 4.6: Results

The results obtained for automatic data selection using supervised learning are promising. But a problem that we have encountered is that, as in any supervised learning task, it is hard to supervise the training process. That is why, in the future, we will investigate other learning techniques for solving the considered problem, i.e. reinforcement learning [SB98] and unsupervised learning [Mit97].

4.4 Comparison to related work

In this section we aim at providing a brief comparison of our approach with several existing approaches for the problem of automatic selection of data representations. To our knowledge, so far, there are no existing machine learning approaches for the considered problem, and, moreover, there are no publicly available case studies for it.

Low [Low78] has introduced a method of choosing the most appropriate representation of a data structure based on data flow analysis, monitoring executions and user interrogations. The monitoring step uses a special compiler in order to acquire data about the number of executions of each statement in a program. Based on the above mentioned information, a selection algorithm will choose the most appropriate representation using a cost function. The criterion used for automatic selection is to minimize the expected space-time product for the execution of the resulting program. The selection process is static, i.e made at compile time, and uses an iterative technique similar to *hill climbing*.

Bik and Wijshoff [BW96, BW94] approach the problem of compiler optimization of sparse codes by automatic data structure selection at compile time - so the programmer does not have to deal with the matrix sparsity explicitly. Nevertheless the dense declared data structures which are actually sparse have to be annotated and the appropriate data representation is generated based on this information.

Schonberg et al [SSS79] present an algorithm for automatic selection of data representation at compile time in the SETL programming language. In SETL, the objects are (dynamically) assigned appropriate abstract data types, algorithms being implemented without specifying any concrete data structure at all. The authors note that the problem of automatic data structure selection is a complex one because each particular data structure is usually more efficient for some operations and less efficient for others and they are choosing the best representation based on a static analysis of the way in which specific objects are used.

Low et al [LR76] discuss about some techniques of automating the choice of data structure representation. They are conducting their experiments in an ALGOL-60 based programming language called SAIL and the paper is also focused on the automatic selection of associative data structures. In particular, the authors are looking for several alternate structures for storing triples in SAIL, using a static selection process. The compiler will choose the representation which minimizes a cost function, this cost function being influenced by both the amount of time and memory. The use of a data structure within the program is also influencing this choice, this information being collected by the compiler after performing a static analysis, from a monitoring phase (information like the frequency of each primitive operation) and also by user interaction (information about the size of certain data structures at a given time).

Rovner [Rov78] presents methods for automatically selecting data structures for programs which use associative data. A library of representations is described and model of associative data is defined based on the idea that classes of associations should be characterized in both terms of operations and properties of data. The process of automatic selection begins with a flow analysis in which the system determines the possible runtime values of item variables at their points of use in the program, and the possible associations that could exist at run-time. The representation selection is done based on a cost function which takes into consideration the memory consumption and execution time.

Compared to the approaches described above [BW96, BW94, SSS79, Low78, LR76, Rov78] which are based on a static analysis, the main advantage of our supervised learning based approach for data structures selection is that it is dynamic, i.e is made at runtime, not at compile time. As we have presented in Section 4.1, a dynamic selection is more accurate than a static one.

Yellin [Yel03] focuses on dynamically choosing the implementation of a component in order to provide optimized usage of resources. The proposed approach uses a mechanism to monitor the types of requests the component is receiving, and adaptively switches implementations for optimal application performance. An algorithm for choosing the most appropriate representation is given, but the algorithm is restricted to exactly two possible implementations of the component. The algorithm has been applied for solving the adaptive selection of data structure problem, but with the same restriction of having only two possible implementations.

Even if the approach introduced by Yellin [Yel03] is dynamic, our approach is more general than it, as it can be used for an arbitrary number of ADTs implementations.

A more detailed comparison with the previous described techniques can not be made, as the case studies used in experiments are not publicly available.

A probabilistic approach to the problem of automatic selection of data representations based on Markov processes is proposed by Chuang and Hwang [CH96]. The selection is done at runtime, but the request sequence (operations performed on the ADT) has to be provided in form of a probabilistic description. The algorithm has a preprocessing phase for constructing the Markov process using some local heuristics.

The main difference between our approach and the one from Chuang and Hwang [CH96] is that the request sequence does not to be a priori known, it will be dynamically predicted. This is a main advantage, because, as we have indicated in Section 4.1, the data access patterns are highly variant, or even unpredictable and can not be statically determined.

4.5 Automatic selection of data representations using SVM

The design and implementation of efficient abstract data types are important issues for software developers. Selecting and creating the appropriate data structure for implementing an abstract data type is not a trivial problem for a software developer, as it is hard to anticipate all the use scenarios of the deployed application. Moreover, it is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. The problem of automatic data structure selection is a complex one because each particular data structure is usually more efficient for some operations and less efficient for others, that is why a static analysis for choosing the best representation can be inappropriate, as the performed operations can not be statically predicted. Therefore, we propose a predictive model in which the software system learns to choose the appropriate data representation, at runtime, based on the effective data usage pattern. This paper describes a new attempt to use a Support Vector Machine model in order to dynamically select the most suitable representation for an aggregate according to the software system's execution context. Computational experiments confirm a good performance of the proposed model and indicates the potential of our proposal. The advantages of our approach in comparison with similar approaches are also emphasized.

4.5.1 Overview

The study of data structures and the algorithms that manipulate them is among the most fundamental topics in computer science [Mou01]. Most of what computer systems spend their time doing is storing, accessing, and manipulating data in one form or another. There are numerous examples from all areas of computer science where a relatively simple application of good data structure techniques resulted in massive savings in computation time and, hence, money. Software applications use abstract data types (ADTs) [WB01] to model real world entities from the application domain. An ADT can be implemented using different data structures.

Selecting and creating the appropriate data structure for implementing an abstract data type arises from practical needs, it has a major importance for software developers. Improper use of data structures in software applications leads to performance degradation and improper memory consumption. These problems can be avoided by properly selecting data structures for implementing ADTs, according to the nature of the manipulated data. This data structure selection problem is not a trivial problem for a software developer, as it is hard to anticipate all the use scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Due to this fact, the software system may choose the appropriate data representation, at runtime, based on the effective data usage pattern. The dynamic selection can be achieved using machine learning techniques [Mit97], which can assure complex and adaptive systems development.

This paper approaches the problem of dynamically selecting of the most suitable representation for an abstract data type according to the software system's execution context. Most of the previous approaches rely on compile-time analyses or programmer annotations. These approaches can be inaccurate since they try to predict a program's behaviour before it is executed.

This paper takes a different view of the data representation selection problem, and presents a machine learning approach to solve the problem. A support vector machine classification model is proposed for selection of abstract data types implementations during the execution of a software system, in order to increase the system's efficiency. Computational experiments confirm a good performance of the proposed model.

The rest of the paper is structured as follows. Section 4.1 gives a motivation for our work, emphasizing the relevance of the approached problem. The overview of the related work in the direction of automatic selection of data representations is presented in Section 4.5.1.3 and Section 4.5.1.4 presents the fundamentals of Suport Vector Machine (SVM) models. Our approach for dynamically identifying the most suitable data structure for implementing an ADT is introduced in Section 4.2. Section 4.6 provides an evaluation of the proposed SVM model on two case studies. A comparison of our approach with other similar existing approaches is given in Section 4.4. An analysis of our approach is given in Section 4.6.4 and Section 4.8 contains some conclusions of the paper and future development of our work.

The primary justification for a data structure selection approach is that finding good representations has proven to be difficult. This is mainly due to a lack of awareness of the importance of the proper choices for data structures issue that has a major impact on software systems' performance.

Each software system uses abstract data types [WB01] to model real world entities from the application domain. The most commonly used abstract data types are: collections, sets, lists and maps [CLRS09]. Because abstract data types represent the core for any software application, a proper use of them is an essential requirement for developing a robust and efficient system.

The design and implementation of efficient abstract data types are important issues for software developers. Selecting and creating the appropriate data structure for implementing an abstract data type can greatly impact the performance of the system. It is not a trivial problem for a software developer, as it is hard to anticipate all the use scenarios of the deployed application. It is not clear how to select a good implementation for an abstract data type when access patterns to it are highly variant, or even unpredictable. Most of the previous approaches rely on compiletime analyses or programmer annotations [CH96]. These approaches can be inaccurate since they try to predict a program's behavior before it is executed.

A common situation is where there are several data structures for implementing an ADT, with

	Vector	**Linked List**	**Balanced Search Tree**
Insertion	$O(1)$	$O(1)$	$O(log_2(n))$
Deletion	$O(n)$	$O(n)$	$O(log_2(n))$
Searching	$O(n)$	$O(n)$	$O(log_2(n))$

Table 4.4: Worst case time complexity.

one data structure more efficient than the others for certain operations but worst for the remaining operations, and vice versa [CH96]. The problem is how to choose the most appropriate data structure for implementing the ADT, given that there is no *a priori* knowledge of what operations, and how often, the ADT will be mostly used for? This is known as the *data representation selection problem* (*DRSP*) [SSS81], or *data structure selection problem* [SSS79]. The problem can be considered for buit-in data types (such as List, Collection), as well for user-defined abstract data types which can be implemented with several data structures.

The problem of automatic data structure selection is a complex one because each particular data structure is usually more efficient for some operations and less efficient for others, that is why a static analysis for choosing the best representation can be innapropriate, as the performed operations can not be statically predicted.

A "static" approach in solving the above presented problem can be, for example, to design a data structure for implementing the abstract data type based on an asymptotic analysis of the computational complexities [CLRS09] of the ADT's operations. Even if the data structure selected this way is not the best possible in every situation, its performance is not too bad for all situations. It is obvious that these "static" methods can be unreliable because they try to predict a program's behavior before it is executed.

Another problem with the "static" approaches is the following. The considered ADT may be instantiated in the same software system in different contexts. A "static" selection method will indicate the same data structure for implementing all ADT's instantiations. But, it is very likely (as we will indicate in the example from Subsection 4.5.1.1) that in different execution contexts the behavior of the ADT is different. So, the selection of the most appropriate data structure is very probable dependent on the execution scenarios.

That is why we propose a "dynamic" approach, i.e. the selection of the most suitable data structure for implementing an abstract data type to be made at run-time, during the execution of the software system.

In the following subsections we give two examples in order to illustrate the importance of the anayzed problem, and, moreover, the need for a dynamic data structure selection, instead of a static one.

4.5.1.1 Example

Let us consider that in a software application a *Collection* ADT (also known as *Bag*) is used. The main operations supported by a *collection* of elements are: **insertion** of an element into the collection, **deletion** of an element from the collection and **searching** an element in the collection.

The *Collection* ADT can be implemented with several data structures: a **vector** (dynamic array), a **linked list** or a **balanced search tree**. Denoting by n the current size of the collection, we give in Table 4.4 the worst case time complexities for the operations of the *Collection* ADT for different implementations.

From Table 4.4, we can conclude the following:

- If the collection will be used mostly for **insertion** operations, then *linked list* or *vector* implementation is preferred.
- If we have a large number of **deletions** and/or membership queries (**searching**) operations, then *balanced search tree* implementation is preferred.

Using a “static” analysis and assuming that the three operations in the collection can be used with the same probability (i.e. $\frac{1}{3}$), we can —conclude that the most appropriate data structure for implementing the collection is the *balanced search tree*.

But, this “static” decision can be incorrect, as the number of operations performed on the collection during the execution of the system can not be predicted before its execution. More exactly, there is no *a priori* knowledge of how the collection looks like, including the total number of operations in the collection.

For example, let us consider an execution scenario in which the collection has, at a given moment 10 elements. On this collection, we perform 2 searches and 6 insertions. Using the time complexities from Table 4.4, we obtain the following:

- If the collection would be implemented using a *vector* or a *linked list*, the needed time would be approximately $2 \cdot 10 + 6 \cdot 1 = 26$.
- If the collection would be implemented using a *balanced search tree*, the needed time would be approximately $2 \cdot log_2 10 + \sum\limits_{i=10}^{15} log_2 i \approx 28.42$.

So, based on the dynamic analysis, the most appropriate data structure for implementing the collection is the *vector* or *linked list*.

Similarly to the above presented example, if we know the number of operations performed on the ADT, we can select the ADT implementation that offers the best possible performance for the given usage scenario. We can conclude that the problem of predicting the most appropriate data structure for implementing the abstract data type during the execution has to be done dynamically and is equivalent to predicting the type and the number of operations performed on the ADT , based on the system's execution context.

4.5.1.2 Experiment

In order to better motivate our approach, we performed an experiment considering the *List* ADT and three data structures for implementing a *List*: **vector** (dynamic array), **linked list** and **balanced search tree**. The main operations supported by a *list* of elements are: **insertion** of an element into the list (at the beginning, at the end, at a certain position), **deletion** of an element from the list (a given element or from a given position), **searching** an element in the list, **iterating** through the list, **accessing** an element from the list at a certain position and **updating** an element from a certain position.

We have chosen *List* ADT in our experiment, as it is one the most used ADTs in real software systems. For comparing the performance of the considered data structures, we have defined several usage scenarios of a *List*, where a usage scenario is a set of pairs (*operation*, *usage probability*).

In order to outline the practical nature of the considered problem, we have chosen a popular programming language (Java) and three existing implementations of *List* ADT:

1. *java.util.ArrayList* which implements *List* functionalities using the **vector** data structure.
2. *java.util.LinkedList* which implements *List* functionalities using the **linked list** data structure.
3. *java.util.TreeList* which implements *List* functionalities using the **balanced search tree** data structure.

Denoting by n the current size of the list, we give in Table 4.5 the worst case time complexities for the main operations of the *List* ADT for different implementations.

From Table 4.5, we can conclude the following:

- If the list will be used mostly for **accessing** elements from it, then *vector* implementation is preferred.

	Vector	**Linked List**	**Balanced Search Tree**
Access	$O(1)$	$O(n)$	$O(log_2(n))$
Insertion at the beginning	$O(n)$	$O(1)$	$O(log_2(n))$
Insertion at the end	$O(n)$	$O(1)$	$O(log_2(n))$
Insertion at a given position	$O(n)$	$O(n)$	$O(log_2(n))$
Deletion at a given position	$O(n)$	$O(n)$	$O(log_2(n))$
Searching	$O(n)$	$O(n)$	$O(log_2(n))$

Table 4.5: Worst case time complexity.

- If the list will be used mostly for **adding** elements at the beginning and at the end of it, then *linked list* implementation is preferred.

- If we have a large number of **deletions** at random positions and/or membership queries (**searching**) operations, then *balanced search tree* implementation is preferred.

The experiment was performed as follows. For each *List* implementation, we have defined six possible usage scenarios which were executed on lists with 5000 elements. The total number of performed operations was 10000 at each execution. The type of the executed operations were chosen according to their probabilities within the usage scenario. In order to capture the performance of a certain implementing data structure, the execution time for each scenario was measured.

In Figure 4.7 we illustrate, in each usage senario, the performance of the data structures used for implementing the *List*. Higher bars indicate lower execution time and, consequently, better performance.

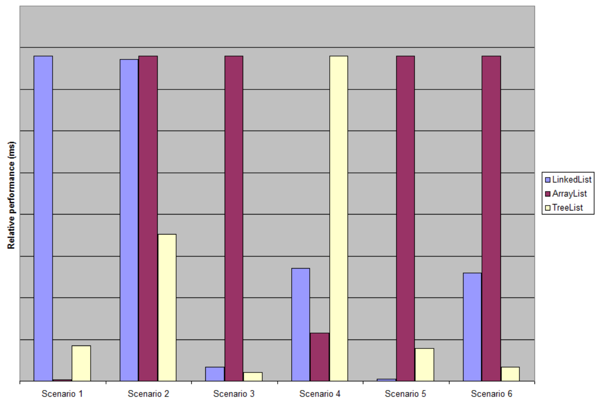


Figure 4.7: Execution scenarios

As the number of the elements from the list grows, a proper data structure selection becomes more important. We depict in Figure 4.8, for Scenario 1, how the time needed for executing a usage scenario grows with the size of the list.

The experiments described in this section confirms the importance of proper data structure selection and reveal the need for a dynamic decision according to the software system's actual usage scenario.

4.5.1.3 Related work for *DRSP*

In this section we present several existing approaches for the problem of automatic selection of data repesentations. To our knowledge, so far, there are no existing machine learning approaches for the

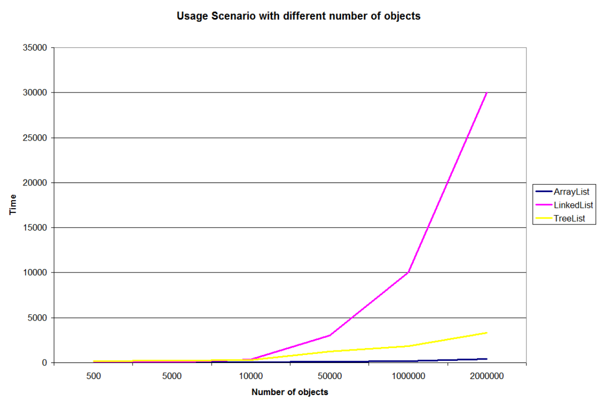


Figure 4.8: Performance

considered problem, and, moreover, there are no publicly available case studies for it.

Low has introduced in [Low78] a method of choosing the most appropriate representation of a data structure based on data flow analysis, monitoring executions and user interrogations. The monitoring step uses a special compiler in order to acquire data about the number of executions of each statement in a program. Based on the above mentioned information, a selection algorithm will choose the most appropriate representation using a cost function. The criterion used for automatic selection is to minimize the expected space-time product for the execution of the resulting program. The selection process is static, i.e made at compile time, and uses an iterative technique similar to *hill climbing*.

In [BW96] and [BW94], Bik and Wijshoff approach the problem of compiler optimization of sparse codes by automatic data structure selection at compile time - so the programmer does not have to deal with the matrix sparsity explicitly. Nevertheless the dense declared data structures which are actually sparse have to be annotated and the appropriate data representation is generated based on this information.

Schonberg et al present in [SSS79] an algorithm for automatic selection of data representation at compile time in the SETL programming language. In SETL, the objects are (dynamically) assigned appropriate abstract data types, algorithms being implemented without specifying any concrete data structure at all. The authors note that the problem of automatic data structure selection is a complex one because each particular data structure is usually more efficient for some operations and less efficient for others and they are choosing the best representation based on a static analysis of the way in which specific objects are used.

In [LR76], Low et al discuss about some techniques of automating the choice of data structure representation. They are conducting their experiments in an ALGOL-60 based programming language called SAIL and the paper is also focused on the automatic selection of associative data structures. In particular, the authors are looking for several alternate structures for storing triples in SAIL, using a static selection process. The compiler will choose the representation which minimizes a cost function, this cost function being influenced by both the amount of time and memory. The use of a data structure within the program is also influencing this choice, this information being collected by the compiler after performing a static analysis, from a monitoring phase (information like the frequency of each primitive operation) and also by user interaction (information about the size of certain data structures at a given time).

Rovner presents methods for automatically selecting data structures for programs which use associative data in [Rov78]. A library of representations is described and model of associative data is defined based on the idea that classes of associations should be characterized in both terms of operations and properties of data. The process of automatic selection begins with a flow analysis in which the system determines the possible runtime values of item variables at their points of use in the program, and the possible associations that could exist at run-time. The representation selection is done based on a cost function which takes into consideration the memory consumption and execution time.

Yellin focuses in [Yel03] on dynamically choosing the implementation of a component in order to provide optimized usage of resources. The proposed approach uses a mechanism to monitor the types of requests the component is receiving, and adaptively switches implementations for optimal application performance. An algorithm for choosing the most appropriate representation is given, but the algorithm is restricted to exactly two possible implementations of the component. The algorithm has been applied for solving the adaptive selection of data structure problem, but with the same restriction of having only two possible implementations.

A probabilistic approach to the problem of automatic selection of data representations based on Markov processes is proposed in [CH96] by Chuang and Hwang. The selection is done at runtime, but the request sequence (operations performed on the ADT) has to be a priori known in form of a probabilistic description. The algorithm has a preprocessing phase for constructing the Markov process using some local heuristics.

4.5.1.4 Support Vector Machines

Predictive modelling [Gei93] is the process by which a model is created or chosen to try to best predict the probability of a certain outcome. From a machine learning perspective, predictive models are generated using supervised learning techniques [RN02].

Support Vector Machines (SVMs) [SC08] are a set of related supervised learning methods used for classification and regression. SVM is a classification technique based on statistical learning theory [CST00, SS01] that was applied with great success in many challenging non-linear classification problems and on large data sets. The basic idea is that a SVM constructs a hyperplane or a set of hyperplanes in a high dimensional space, which can be further used for classification, regression, or other tasks. A decision hyperplane built by a SVM optimally splits the training set, being a boundary between a set of objects having different class memberships [KS08]. The optimal hyperplane can be distinguished by the maximum margin of separation between all training points and the hyperplane.

A SVM uses a nonlinear mapping to transform the original training data into a higher dimension [CST00]. This is a promising new method for the classification of both linear and non linear data. Within this new dimension, it searches for the linear optimal separating hyperplane that is, a “decision boundary” separating the tuples of one class from another.

The advantages of the SVM method are as follows [KS08]:

- They are highly accurate.
- SVM have the ability to model complex nonlinear decision boundaries.
- SVM are less prone to over fitting than other models.
- They provide a compact description of the learned model.

SVM classifiers have a wide range of applications, being applied to a number of areas, including handwritten digit recognition, object recognition, speech recognition [GKP02], predictions [KS08].

4.5.2 Formal aspects

Data structures [WB01] provide means to customize an abstract data type according to a given usage scenario. The volume of the processed data and the data access flow in the software application influence the selection of the most appropriate data structure for implementing a certain abstract data type. During the execution of the software application, the data flow and volume is fluctuating due to external factors (such as user interaction), that is why the data structure selection has to be dynamically adapted to the software system’s execution context. This adaptation has to be made during the execution of the software application and it is hard or even impossible to predict by the software developer. Consequently, in our opinion, machine learning techniques would provide a better selection at runtime of the appropriate data structure for implementing a certain abstract data type.

In this section we will present our proposal of using supervised learning for dynamically selecting the implementation of an abstract data type from the software system, based on its execution context. For this purpose, a SVM model is proposed. In fact, selecting the most appropriate implementation of an abstract data type is equivalent to predicting, based on the execution context, the type and the number of operations performed on the ADT, on a certain execution scenario.

Let $S = \{s_1, s_2, ..., s_n\}$ be an object oriented software system, where $s_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class.

We will consider that:

- $Class(S) = \{C_1, C_2, \ldots, C_l\}$, $Class(S) \subset S$, is the set of applications classes of the software system S.
- Each application class C_i $(1 \leq i \leq l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, .$ $p_i \leq n,\ 1 \leq r_i \leq n$, 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 C_i.
- $Meth(S) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system S.
- $Attr(S) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system S.

Based on the above notations, the software system S can be defined as in Equation (4.3):

$$S = Class(S) \bigcup Meth(S) \bigcup Attr(S). \tag{4.3}$$

Let us consider that $\mathcal{T}$ is an abstract data type having in its interface a set of operations $\mathcal{O}$ that can be performed on an instance of $\mathcal{T}$. We also consider a set $\mathcal{D}_{\mathcal{T}} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n\}$ of data structures that can be used in the software system S for implementing $\mathcal{T}$. If a given instance c of a class $C \in Class(S)$ uses $\mathcal{T}$, the selection of the appropriate data structure from $\mathcal{D}$ that has to be used for implementing (and instantiating) $\mathcal{T}$ depends on the states of the objects that are in a "neighbourhood" of object c. Intuitively, the classes that use directly or indirectly the $\mathcal{T}$ ADT influence the selection.

In order to express the "neighbourhood" of an object, we define the *distance* between any two classes from the software system. In our view, the *distance* between two classes C_i and C_j expresses the degree to which class C_i influences the behaviour of class C_j.

In the following we will introduce several auxiliary definitions needed for a formal description of the notion of "neighbourhood".

Definition 4.5.1 *Let c and c' be two objects which are instances of classes $C \in Class(S)$ and $C' \in Class(S)$, respectively. We say that c* **uses** *c' if c invokes a method from c' or c' invokes a method from c.*

Definition 4.5.2 *Let c and c' be two objects which are instances of classes $C \in Class(S)$ and $C' \in Class(S)$, respectively. The list $UC(c, c') = (c_1, c_2 \ldots c_k)$ is called the* **usage chain** *between c and c' if $k \geq 2$, c_i are instances of classes from S, c_i* **uses** *c_{i+1} $\forall 1 \leq i \leq k-1$, $c = c_1$ and $c' = c_k$.*

Intuitively, the *usage chain* is similar with the idea of *stack trace*. We denote by $\mathcal{UC}(c, c')$ the set of all existing *usage chains* between objects c and c'. We mention that $\mathcal{UC}(c, c')$ can be empty if there is no *usage chain* between the two objects.

Considering the definitions above, we express the *distance* between objects c and c' as given in Equation (4.4). By $|u|$ we denote the cardinality of set u.

$$d(c, c') = \begin{cases} 0 & if\, c = c' \\ \infty & if \mathcal{UC}(c, c') = \emptyset \\ min_{u \in \mathcal{UC}(c,c')} |u| & otherwise \end{cases}, \tag{4.4}$$

In defining the *distance* d between two objects c and c', as given in Equation (4.4), we have started from the intuition that, as smaller the minimum length of an *usage chain* between the objects is, as it is likely that the state of c' has a larger influence on c, so c' is "closer" to c (the distance between c and c' is smaller). It can be simply proved that d is a semi-metric function.

Now, the *neighbourhood* of radius $\mathcal{R}$ of an object o, denoted by $\mathcal{N}_{\mathcal{R}}(c)$ can be defined as expressed in Equation (4.5).

$$\mathcal{N}_{\mathcal{R}}(c) = \{c' | c' \in InstClass(S), d(c, c') \leq \mathcal{R}\} \tag{4.5}$$

where $\mathcal{R} \in N$ and $InstClass(S) = \{c | c \textit{ is an instance of } C \in Class(S)\}$ is the set of instances of the running software system S.

It can be simply observed the following:

- The neighbourhood of radius 0 of any object c consists only of itself, i.e, $\mathcal{N}_0(c) = \{c\}$.
- If $\mathcal{R}$ is large enough, then the neighbourhood of radius $\mathcal{R}$ of any object c consists of the entire system, i.e, $\mathcal{N}_{\mathcal{R}}(c) = InstClass(S)$.

Finally, the notion of *execution context* of radius $\mathcal{R}$ for the abstract data type $\mathcal{T}$ will be introduced in Definition 4.5.3.

Definition 4.5.3 Execution context *of $\mathcal{T}$.*
Let c be an instance of a class $C_i \in Class(S)$ that uses $\mathcal{T}$. The **execution context** *of radius $\mathcal{R}$ of $\mathcal{T}$ represents the state of the objects from the neighborhood of C_i at runtime, when $\mathcal{T}$ is initialized. Consequently, it is defined as a set $\mathcal{EC}_{\mathcal{R}} = \bigcup\limits_{c_j \in \mathcal{N}_{\mathcal{R}}(c)} \{v_{j1}, v_{j2}, \ldots, v_{jr_j}\}$, where $v_{j1}, v_{j2}, \ldots, v_{jr_j}$, $\forall j$, represent the values of the attributes from object c_j.*

As we have shown in Section 4.5.1.1, a "static" selection of the most suitable data structure from $\mathcal{D}_{\mathcal{T}}$ that has to be used for an efficient implementation of $\mathcal{D}$ is inappropriate and does not assure an efficient use of $\mathcal{T}$, as the volume of data manipulated by the implementation of $\mathcal{T}$ is unpredictable at the development stage. That is why a proper selection has to be done at runtime by analyzing the *execution context.*

In Definition 4.5.4 we will define the notion of *suitable implementation* of $\mathcal{T}$ given a certain *execution context.*

Definition 4.5.4 ***Most suitable implementation of $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$.***
Let $\mathcal{T}$ be an abstract data type and $\mathcal{EC}_{\mathcal{T}}$ the execution context of $\mathcal{T}$ in a certain execution scenario. The most suitable implementation of $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$ is the data structure $\mathcal{D}_i \in \mathcal{D}$ that provides an efficient implementation of $\mathcal{T}$ in the execution scenario, i.e. the time needed for performing the operations on $\mathcal{T}$ given the execution context $\mathcal{EC}_{\mathcal{T}}$ is minimized.

In fact, the most proper implementation of $\mathcal{T}$ in a given execution scenario depends on the type and the number of operations from $\mathcal{O}$ that are performed on $\mathcal{T}$ during the execution. Thus, we can conclude that the problem of dynamically selecting the *suitable implementation* of $\mathcal{T}$ is, in fact, the problem of predicting, based on the execution context, the type and the number of operations performed on $\mathcal{T}$ on a certain execution scenario.

Considering the example given in Section 4.5.1.1, the most suitable implementation of ADT *Collection* in the execution context in which the collection has 10 elements, and 2 **searches** and 6 **insertions** are performed on it is the *list.*

4.5.3 Methodology

Let us consider, in the following, the theoretical model from Subsection 4.5.2. Considering the issues described in the previous sections, for providing the most appropriate data structure for implementing an abstract data type $\mathcal{T}$ at runtime based on the system's *execution context* we propose a *supervised learning* [Mit97] approach.

In a software system $\mathcal{S}$, multiple occurences of $\mathcal{T}$ can be found. We mention that not all these occurences are subject for optimization, but only those that are considered to have a great impact on the system's performance. The software developer is responsible for selecting the locations in S where optimization is needed. So, let us focus, in the following, on a single occurence of an abstract data type $\mathcal{T}$, for which n possible implementations exist, i.e. n data structures $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n$.

In fact, the problem of identifying the most suitable implementation of $\mathcal{T}$, as illustrated in Definition 4.5.4, can be viewed as a *classification* [Mit97] problem where each input is represented by an *execution context*. The output for a given *execution context* $\mathcal{EC}_{\mathcal{T}}$ is the index $i, 1 \leq i \leq n$ where $\mathcal{D}_i$ is the *most suitable implementation of* $\mathcal{T}$ *given the execution context* $\mathcal{EC}_{\mathcal{T}}$ (Definition 4.5.4).

The classification process takes place in two phases that reflect the principles of a supervised learning algorithm: *training* and *testing*. As in any classification process, during the training a classification *model* will be built, and during testing, the model built during the training will be applied for classifiying an unseen *execution context* (input), i.e to predict the *most suitable implementation* of $\mathcal{T}$ given the input *execution context*. We propose a SVM classification model for the problem of identifying the most appropriate data reprezentation.

For predicting the most *suitable implementation* of a certain abstract data type $\mathcal{T}$, the following steps will be used:

1. Data Collection, Conversion and Pre-processing.
2. Design of SVM Model.
3. Training.
4. Validation.
5. Testing.

We mention that an innapropriate prediction does not impact the execution scenario of the software system. Even if the data structure selected for the implementation of the abstract data type $\mathcal{T}$ is not the most suitable one, the system execution is still successfull, even if not the most efficient. We can imagine a classification model in which, if such an innacurate prediction is made, a feed-back is sent and the model is correspondingly adapted. The idea would be to send a *reward* (*reinforcement*) to the model, as in a *reinforcement learning* [SB98] based model.

In the following we will briefly describe the above proposed steps.

4.5.3.1 Data collection, conversion and pre-processing

First, the software system S is monitored during the execution of a set of scenarios that include the instantiation of the abstract data type $\mathcal{T}$. The result of this supervision performed by a software developer is a set of *execution contexts*, as well as the type and the number of operations from $\mathcal{O}$ performed on $\mathcal{T}$ saved in a log file. The software developer will analyze the resulted log file and will decide, for each *execution context* (input) , the *most suitable implementation* for $\mathcal{T}$ given the *execution context* (output). This decision will be based on computing the global computational complexity of the operations performed on $\mathcal{T}$ during the scenario given by the *execution context* for each possible implementation of $\mathcal{D}_i$ of $\mathcal{T}$ and then selecting the implementation that minimizes the overall complexity.

This way, a data set consisting of (input, output) pairs is built. An input represents an *execution context* $\mathcal{EC}_{\mathcal{T}}$ of the abstract data type $\mathcal{T}$ and the associated output represents the the index $i, 1 \leq i \leq n$ where $\mathcal{D}_i$ is the *most suitable implementation of* $\mathcal{T}$ *given the execution context* $\mathcal{EC}_{\mathcal{T}}$ (Definition 4.5.4).

The input data from the resulting data set (the *execution contexts*) is now converted and pre-processed.

First, the values of the attributes within an *execution context* are converted to numerical values. This conversion is made as follows:

- An ordinal type attribute (integer, character, boolean, enumerated type) is converted to the ordinal position of the attribute value in its type domain.
- A numerical non-ordinal attribute (float, double), the actual value of the attribute remains unchanged.
- For a string attribute or other user defined types, the corresponding numerical value is determined using a hash function [CLRS09]. Such a hash function is available in most object-oriented programming languages. Even if two different attributes values can have the same hash value, the combination of all attributes values from the execution context may assure a uniqueness for the input data.

Then the input data is scaled to [0,1], and a statistical analysis is carried out in order to find a subset of attributes (from the *execution contexts*) that are correlated with the target output. The statistical analysis on the attributes from the *execution contexts* is performed in order to reduce the dimensionality of the input data (the execution contexts), by eliminating attributes from the *execution context* which do not influence the output value.

To determine the dependencies between attributes and the target output, the Spearman's rank correlation coefficient [Spe87] is used. A Spearman correlation of 0 between two variables X and Y indicates that there is no tendency for Y to either increase or decrease when X increases. A Spearman correlation of 1 or -1 results when the two variables being compared are monotonically related, even if their relationship is not linear. At the statistical analysis step we remove from the *execution context* those attributes that have no significant influence on the target output, i.e are slightly correlated with it. A slight correlation is indicated by a value that is very close to 0. Namely, an attribute $\mathcal{A}$ is removed from the *execution context* if the correlation coefficient between $\mathcal{A}$ and the target output is less than a given threshold ϵ close to 0. In our experiments, we have considered the threshold $\epsilon = 10^{-1}$.

The data set preprocessed this way, denoted by $\mathcal{D}$, can now be used for building the classification model.

4.5.3.2 The design of the SVM classification model

SVMs use a technique known as the "kernel trick" to apply linear classification techniques to non-linear classification problems. Using a Kernel function [Vap00], the data points from the input space are mapped into a higher dimensional space. Constructing (via the Kernel function) a separating hyperplane with maximum margin in the higher dimensional space yields a non-linear decision boundary in the input space separating the tuples of one class from another.

The classification process takes place in two phases that reflect the principles of a supervised learning algorithm: *training* and *testing*. As in any classification process, during the training a classification *model* will be built, and during testing, the model built during the training will be applied for clasifiying an unseen *execution context* (input).

Therefore, the data set collected at step 1 has to be divided in two parts: a part for *training* and a part for *testing*. The training part is divided again in: a *learning subset* used by the SVM algorithm in order to learn the model that performs the class separation, and a *validation subset* used in order to optimise the values of the hyper parameters. The SVM model, which is learnt in this manner, classifies the unseen *execution contexts* from the test set, which is disjoint to the training one.

In order to classify the *execution contexts*, the SVM algorithm uses a kernel function. The parameters of the SVM model (the penalty for miss-classification C and the *kernel parameters*) are optimized on the validation set. A cross-validation framework is used in order to avoid the overfitting problems.

4.5.3.3 Technical details

In our current implementation, we have considered *execution contexts* of radius 0 (i.e. $\mathcal{R} = 0$). This means that the execution context contains only the state of the object that uses the abstract data type $\mathcal{T}$ considered for optimisation.

For the monitored abstract data type $\mathcal{T}$ from the software application, a modified data structure implementation is used. This data structure implementation delegates all its methods to the actual data structure implementation, collects the state of the objects from the *execution context* and writes the collected data into a log file. The modified data structure implementation has a reference to its container object (received as a constructor parameter) and uses it in order to determine the current state of the object.

In the future we plan to extend the implementation in order to consider *execution contexts* of arbitrary radius. For this, we need a more elaborate implementation in order to consider the *usage chains* between objects. The *usage chains* can be obtained by analysing the stack trace of the current execution. For obtaining the states of all objects from an *usage chain*, references to the objects are needed. Using *aspect oriented programming* [KH01], or instrumenting the software system classes (by using bytecode manipulation frameworks [OSM]) the needed references can be obtained.

4.6 Computational experiments

In this section we aim at evaluating the accuracy of the technique proposed in Section 4.2, i.e. the SVM classification model's prediction accuracy.

As there is no publicly available case study for the problem of automatic selection of data representations, nor a case study in the related literature that can be reproduced, we consider our own case studies. We describe in this section simulation results of applying our classification approach to two selection problems that will be described in the following.

4.6.1 SVM model

From all the samples from $\mathcal{D}$ (obtained as indicated in Subsection 4.5.3.1), $\frac{2}{3}$ of them are considered for training the SVM algorithm and $\frac{1}{3}$ of them for testing the classifier. The C-SVM algorithm, provided by LIBSVM [CL11], with a RBF kernel is actually used in this experiment. It is not known beforehand which are the best values for the parameters of the SVM model (C and γ), so some kind of model selection must be done. The optimisation of the hyper-parameters is performed by a grid search method. A grid search makes repeated trials for each parameter across a specified interval using geometric steps. For each combination of these parameters, a 10-fold cross-validation is performed during the training phase, the quality of a combination being computed as the average of the accuracy rates estimated for each of the 10 divisions of the data set. We are using the following sequences for C and γ: $C = (2^{-5}, 2^{-3}, ..., 2^{15})$ and $\gamma = (2^{3}, 2^{1}, ..., 2^{-15})$.

Cross-validation is a popular technique for estimating the generalization error and there are several interpretations [WLZ00]. In k-fold cross-validation, the training data is randomly split into k mutually exclusive subsets (or folds) of approximately equal size. The SVM decision rule is obtained by using k-1 subsets on training data and then tested on the subset left out. This procedure is repeated k times and in this manner each subset is used for testing once. Averaging the test error over the k trials gives a better estimate of the expected generalization error.

Therefore, the best combination of parameters is indicated by the best average accuracy rate.

In order to obtain the accuracy of the model, considering the data set $\mathcal{D}$, the following cross-validation is used.

- For a given number of episodes (50 in our experiment) repeat the following:
  - $\frac{2}{3}$ randomly selected samples from $\mathcal{D}$ are used for training. Thus, for each training sample, the corresponding (pre-processed) *execution context* with the associated output is provided to the SVM.
  - The rest of $\frac{1}{3}$ samples from $\mathcal{D}$ are used for testing. After the training was completed, the SVM model is tested as follows. The *execution context* within the testing sample is sent to the SVM. After receiving the *execution context*, the SVM will predict the most appropriate output (the most *suitable implementation* of the abstract data type $\mathcal{T}$ considered for optimization).

  - The *learning accuracy* on the current episode is computed as the number of accurate predictions divided by the number of tesing samples.

- The overall *learning accuracy* of the SVM classification model is computed as the average of the learning accuracies over all episodes.

4.6.2 First case study

Starting from the data set given at [For10], we have simulated an experiment for selecting the most appropriate data structure for implementing the *List* ADT. The considered data set consists of the results of a chemical analysis of wines grown in the same region in Italy but derived from different cultivars. The analysis determined the quantities of 13 constituents found in each types of wines. More details about this data set can be found at [Win91].

In the proposed case study, we consider a *WineShop* application that stores information about different wines and allows the user to locate wine shops according to different search criteria. In our simulation, we are focusing on the main class of the application, the class *Wine*, which models a type of wine by 13 constituents given at [Win91]. This class has a method *getShops* that returns a *List* of shops where the current wine is available. For the *List* ADT returned by *getShops* method we aim at identifying the most suitable data structure for implementation. As presented in Subsection 4.5.1.2, three data structures for implementing a *List* are considered: **vector** (dynamic array), **linked list** and **balanced search tree**.

The *List* of shops is used in three different usage scenarios:

S1. The *List* of shops is presented to the user sorted according to different user selected criteria. In this scenario, the main operations performed on the *List* are: **accessing** an element from a certain position and **updating** an element from a certain position. Consequently, the **dynamic array** data structure would be the most efficient implementation for the *List* ADT.

S2. The *List* of shops is presented to the user filtered according to different user selected criteria. In this scenario, the main operation performed on the *List* is **removing** an element from a certain position. Consequently, the **balanced search tree** data structure would be the most efficient implementation for the *List* ADT.

S3. The *List* of shops is presented to the user by concatenating the shops lists from different wine types. In this scenario, the main operation is **adding** an element at the end of a list. Consequently, the **linked list** data structure would be the most efficient implementation for the *List* ADT.

In practice, the selection of most appropriate implementation of the *List* is difficult, even impossible for a software developer, as it is hard to anticipate the data flow during the execution of the application. That is why we apply our supervised learning approach for a dynamic selection of the suitable implementing data structure of the *List*, according to the system's execution context.

4.6.2.1 Data collection and pre-processing

The data set for evaluating the SVM classification model presented in Section 4.2 consists of (input, output) samples collected and pre-processed as we have described in Subsection 4.5.3.1. An input represents an *execution context* and the target output is the most suitable implementation for the *List* ADT (**vector**, **linked list** or **balanced search tree**) within the input *execution context.* The data set consists of 178 samples.

After data was scaled, a statistical analysis is carried out in order to find a subset of attributes that are correlated with the target output. To determine the dependencies between attributes and the target output, the Spearman's rank correlation coefficient [Spe87] is used.

Figure 4.9 shows the correlations between the attributes from the *Wine* class (the current *execution context* for the *List* of shops) and the target output (the most suitable data structure for implementing the *List* of shops) computed using the Spearman's rank correlation coefficient [Spe87].

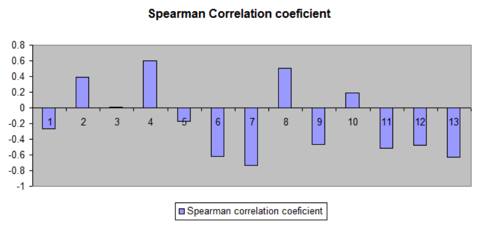


Figure 4.9: Spearman correlation

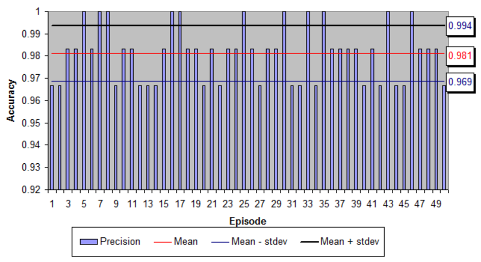


Figure 4.10: Results for the first case study

After an analysis of the correlation between the attributes from the $Wine$ class (current *execution context* for the $List$ of shops) and the target output, considering the value 10^{-1} for the threshold ϵ (Subsection 4.5.3.1), we can conclude that attribute **3** is slightly correlated with the target output (its correlation with the output is 0.01403). Consequently, we can consider that the above mentioned attribute has no significant influence on the target output. Thus, after the performed statistical analysis, there are 12 attributes identified to be relevant for our classification task (the attributes from the $Wine$ class excepting attribute **3**).

4.6.2.2 Results

The *learning accuracies* obtained for each episode are given in Figure 4.10. An overall *learning acuracy* of 0.9625 was obtained. As we can see in Figure 4.10, the results are stable, a standard deviation of 0.033071891 on the classification accuracies was obtained. The low value of the standard deviation indicates a good precision of our approach.

4.6.3 Second case study

We will consider a real software system as a case study for evaluating the learning accuracy of the SVM. It is a DICOM (*Digital Imaging and Communications in Medicine*) [DICiM11] and HL7 (*Health Level 7*) [HL0] compliant PACS (*Picture Archiving and Communications System*) system, facilitating medical images management, offering quick access to radiological images, and making the diagnosing process easier. The analyzed application is a large distributed system, consisting of several subsystems in form of stand-alone and web-based applications. We have considered as our case study one of the subsystems from this application. The analyzed subsystem is a stand-alone Java application used by physicians in order to interpret radiological images. The application fetches clinical images from an image server (using DICOM protocol) or from the local file system (using DICOM files), displays them, and offers various tools to manage radiological images.

Radiological images are produced by medical devices called modalities. The modalities generate multiple related images, organized in series.

Let us consider the Java source code fragment presented below, extracted from the analyzed software system, where **ImageSeries** class represents an image serie acquired by a medical device for a given human body part. The medical device sends additional information which characterize: the performed medical procedure (*code*, *description*), technical aspects related to the device (*thickness*, *frecquency*, *repetition time*), aspects related to the patient (*bodyPart*, *localizer*), etc. Some relevant aspects are captured in the **ImageSeries** attributes. In practice, the size of the list of images contained in an image serie significantly vary. That is why we apply our approach for dynamically configuring the image list from an image serie, according to the execution context.

```
public class ImageSeries {

   private String modality;
   private String code;
   private String bodyPart;
   private String manuf;
   private String model;
   private String description;
   private String localizer;
   private float thickness;
   private float frecquency;
   private int repetitionTime;
13.private List<Image> images;
   private String UID;

  public Series(DicomObject d) {
    UID =
     d.getString(Tag.SeriesInstanceUID);
    bodyPart =
     d.getString(Tag.BodyPartExamined);
    modality =
     d.getString(Tag.Modality);
    localizer = d.getString(Tag.ImageType);
    manuf = d.getString(Tag.Manufacturer);
    model =
     d.getString(Tag.ManufacturerModelName);
20. images = new ArrayList<Image>();
    ....
  }

  public void setCode(String code) {
    this.code = code;
  }

  public void addImages(Image img) {
    images.add(img);
  }

  public boolean isLocalizer() {
    if (imgType == null) {
    return false;
    }
    return imgType.contains("LOCALIZER");
  }

  public Modality getModality() {
    return Modality.valueOf(modality);
  }

  public String getBodyPart() {
    return bodyPart;
  }

  public String getManuf() {
    return manuf;
  }

  public String getModel() {
    return model;
  }
  public String getProcT() {
    return code;
  }

  public String getDescr() {
    return description;
  }

  public String getUID() {
    return UID;
  }

  public int getNoOfImages() {
    return images.size();
  }

  public boolean equals(Object o) {
    return ((Series) o).UID.equals(UID);
  }
  .....
}
```

For the *images* attribute in line 13 from the source code below, the most suitable data structure implementation has to be chosen at runtime (line 20). As presented in Subsection 4.5.1.2, three data structures for implementing a *List* are considered: **vector** (dynamic array) (*ArrayList* class from Java SDK), **linked list** (*LinkedList* class from Java SDK) and **balanced search tree** (*TreeList* class from Apache Collection API [sf]).

The *List* of images is used in three different usage scenarios:

S1. The *List* of images are sorted according to different criteria (position of the image relative to the patient, image acquisition time, etc). This operation is required in order to provide the physician a consistent presentation of the images list. In this scenario, the main operations performed on the *List* are: **accessing** an element from a certain position and **updating** an element from a certain position. Consequently, the **dynamic array** data structure would be the most efficient implementation for the *List* ADT.

S2. The *List* of images is filtered according to different criteria (echo time, constrast agent). This operation is required in particular medical procedure which require the acquisition of multiple images from the same body part, for example head CT performed twice: once without any contrast substance and second with a contrast agent. In this scenario, the main operation performed on the *List* is **removing** an element from a certain position. Consequently, the **balanced search tree** data structure would be the most efficient implementation for the *List* ADT.

S3. The *List* of images is created by the physician through an iterative step by step process. For example, the physician selects relevant images from source series and creates a new image serie with the relevant images. In this scenario, the main operation is **adding** an element at the begining/end of a list. Consequently, the **linked list** data structure would be the most efficient implementation for the *List* ADT.

In practice, the selection of the most appropriate implementation of the *List* of images is difficult, as it is hard to anticipate the usage scenario of the image serie object at its instantiation time. That is why we apply our supervised learning approach for a dynamic selection of the suitable implementing data structure of the *List*, according to the system's execution context.

4.6.3.1 Data collection and pre-processing

We have used for evaluation a set of 96 image series samples which were obtained from publicly available DICOM image files [Osi10, RiplsDis10, cir10, oPDsf10, hp10]. The images are real images from real patients, but anonymized for confidentiality reasons. For managing the DICOM image files, an open source implementation of the DICOM standard was used [sciom11].

The data set for evaluating the SVM classification model presented in Section 4.2 consists of (input, output) samples collected and pre-processed as we have described in Subsection 4.5.3.1. An input represents an *execution context* and the target output is the most suitable implementation for the *List* ADT (**vector**, **linked list** or **balanced search tree**) within the input *execution context*.

After data was scaled, a statistical analysis is carried out in order to find a subset of attributes that are correlated with the target output. To determine the dependencies between attributes and the target output, the Spearman's rank correlation coefficient [Spe87] is used.

Figure 4.11 shows the correlations between the attributes from the *ImageSeries* class (the current *execution context* of the *images* List) and the target output (the most suitable data structure for the *images* List implementation) computed using the Spearman's rank correlation coefficient [Spe87].

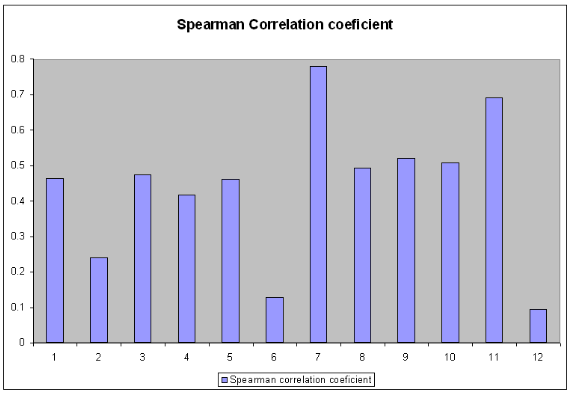


Figure 4.11: Spearman correlation

After an analysis of the correlation between the attributes from the *ImageSeries* class (current *execution context* for the *images* list) and the target output, considering the value 10^{-1} for the threshold ϵ (Subsection 4.5.3.1), we can conclude that attribute **12** (**UID**) is slightly correlated with the target output (its correlation with the output is 0.09527). Consequently, we can consider that the above mentioned attribute has no significant influence on the target output. Thus, after the performed statistical analysis, there are 11 attributes identified to be relevant for our classification task (the attributes from the *ImageSeries* class excepting *UID* attribute).

4.6.3.2 Results

The *learning accuracies* obtained for each episode are given in Figure 4.12. An overall *learning acuracy* of 0.941875 was obtained. As we can see in Figure 4.12, the results are stable, a standard deviation of 0.040024407 on the classification accuracies was obtained. The low value of the standard deviation indicates a good precision of the proposed approach.

4.6.4 Discussion

Considering the experimental results presented in Section 4.6, we can conclude that our approach provides optimized data structure selection and reduces the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario.

We mention that the accuracy of the prediction process may depend on the radius $\mathcal{R}$ of the *execution context*. In our opinion, choosing a larger value for $\mathcal{R}$ may increase the accuracy of the predicted customization parameters values. However, increasing the value for $\mathcal{R}$ leads to larger *execution contexts*, and, consequently, to larger computational complexity. The most appropriate value for $\mathcal{R}$ may depend on the domain and complexity of the considered software system.

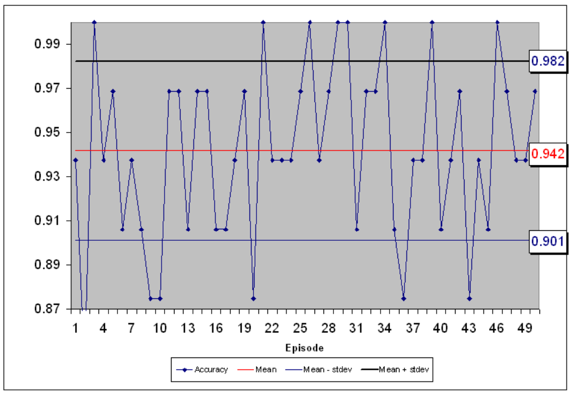


Figure 4.12: Results for the second case study

The results obtained for automatic data selection using supervised learning are promising. Considering the results presented in Section 4.6, we can conclude that the approach introduced in this paper for a dynamic selection of data representations has the following advantages:

- It is general, as it can be used for determining the appropriate *implementation* for any abstract data type, and with arbitrary number of data structures that can be chosen for implementing the ADT.
- It reduces the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario. Consequently, it increases the efficiency of the software system during its evolution.
- It is scalable, as even if the considered software system is large, the abstract data types optimisation depends on the radius $\mathcal{R}$ of the *execution context*. The size of the execution context does not depend on the size of the software system.

However, the main drawback of our approach is that it is hard to supervise the learning process, as the supervision of an expert software developer is required for inspecting the collected execution contexts. That is why, in the future, we will investigate other learning techniques for solving the cosidered problem, i.e. reinforcement learning [SB98] and unsupervised learning [Mit97].

4.7 Comparison to related work

In this section we aim at providing a brief comparison of our approach with the existing approaches for the problem of automatic selection of data representations presented in Section 4.5.1.3.

Compared to the approaches from [BW96, BW94, SSS79, Low78, LR76, Rov78] which are based on a static analysis, the main advantage of our supervised learning based approach for data structures selection is that it is dynamic, i.e is made at runtime, not at compile time. As we have presented in Section 4.1, a dynamic selection is more accurate than a static one.

Even if the approach from [Yel03] is dynamic, our approach is more general than it, as it can be used for an arbitrary number of ADTs implementations.

A more detailed comparison with the techniques from [BW96, BW94, SSS79, Low78, LR76, Rov78, Yel03] can not be made, as the case studies used in experiments are not publicly available.

The main difference between our approach and to the one from [CH96] is that the request sequence does not have to be a priori known, it will be dynamically predicted. This is a main advantage, because, as we have indicated in Section 4.1, the data access patterns are highly variant, or even unpredictable and can not be statically determined.

4.8 Conclusions and future work

In this chapter we have presented our model for dynamically selecting the most suitable implementation of an abstract data type from a software application based on the system's execution context. For predicting, at runtime, the most appropriate data representation, a neural network and a support vector machine classification model were used. We have also illustrated the accuracy of both proposed approaches on case studies.

Considering the results presented in Section 4.3 and in Section 4.6, we can conclude that the approaches introduced in this paper for a dynamic selection of data representations have the following advantages:

- They are general, as they can be used for determining the appropriate *implementation* for any abstract data type, and with arbitrary number of data structures that can be chosen for implementing the ADT.
- They reduce the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario. Consequently the efficiency of the software system during its evolution is increased.
- They are is scalable, as even if the considered software system is large, the abstract data types are locally optimized, considering only the current execution context. The size of the execution context does not depend on the size of the software system (as shown in Section 4.5.2).

However, the main drawback of both approaches is that it is hard to supervise the learning process, as the supervision of an expert software developer is required for inspecting the collected execution contexts.

Further work will be focused on:

- Improving the proposed classification model by adding to it the capability to adapt itself using a feed-back received when inappropriate data representations are selected.
- Applying other machine learning techniques [KTL11, ZDYZ11] , self-organizing feature maps [SK99], or other modelling techniques [RKY10, Ngu10, TD10] for solving the problem of automatic selection of data representations during the execution of a software system.
- Studying the applicability of other learning techniques, like semi-supervised learning [ZL10] or reinforcement learning [SB98] in order to avoid as much as possible the supervision during the training process.
- Evaluating our approach on other case studies and real software systems.

Chapter 5

Conclusions

It has been that seen pattern recognition is central in data analysis tasks. As data is often characterised by a great deal of imprecision and uncertainty, intelligent, autonomous systems need to be developed that can handle such complex problems.

In Chapter 4 we have presented our model for dynamically selecting the most suitable implementation of an abstract data type from a software application based on the system's execution context. For predicting, at runtime, the most appropriate data representation, a neural network and a support vector machine classification model were used. We have also illustrated the accuracy of both proposed approaches on case studies.

Considering the results presented in Section 4.3 and in Section 4.6, we can conclude that the approaches introduced in this paper for a dynamic selection of data representations have the following advantages:

- They are general, as they can be used for determining the appropriate *implementation* for any abstract data type, and with arbitrary number of data structures that can be chosen for implementing the ADT.
- They reduce the computational time by selecting the data structure implementation which provides a minimum overall complexity for the operations performed on a certain abstract data type on a given execution scenario. Consequently the efficiency of the software system during its evolution is increased.
- They are is scalable, as even if the considered software system is large, the abstract data types are locally optimized, considering only the current execution context. The size of the execution context does not depend on the size of the software system (as shown in Section 4.5.2).

In Chapter 3 we have presented our contribution to agent-based clustering, particularly in two main directions: ASM-based batch clustering and incremental clustering. We have focused on developing clustering algorithms that allow the discovery and analysis of hybrid data. The algorithms presented in Section 3.2 are based on the adaptive ASM approach from [CXC04]. The major improvement is that, instead to moving the agents at a randomly selected site, we are letting the agents choose the best location. Agents can directly communicate with each other — similar to the approach from [CDG07]. In [SCCK04], the fuzzy IF-THEN rules are used for deciding if the agents are picking up or dropping an item. In our model we are using the fuzzy rules for deciding upon the direction and length of the movement. Moreover, in the approach from Section 3.2.2 the agents are able to adapt their movements if changes in the environment would occur. Case studies for these approaches have been performed in Section 3.2.3. In order to test the algorithm in a real-world scenario, the Iris and Wine datasets have been considered [Iri88, Win91]. Experiments outline the ability of our approaches to discover hybrid data. In Section 3.3 an incremental clustering algorithm is introduced. Incremental clustering is used to process sequential, continuous data flows or data streams and in situations in which cluster shapes change over time. Such algorithms are well fitted in real-time systems, wireless sensor networks or data streams because in such systems it is difficult to store the datasets in memory. The algorithm considers one instance at a time and it basically tries to assign it to one of the existing clusters. Only cluster representations need to be kept in memory so computation is both fast and memory friendly. We have seen in the tests from the incremental approach (Section 3.3.3) that most of the apparently classification errors were actually items that have high membership degrees to more than one cluster. Nevertheless, in our opinion, it is again clear that we are dealing with hybrid data. Actually the hybrid nature of the data is suggested in [Iri88] and in [Win91] and this is the main reason for choosing these datasets for our analysis. By using fuzzy methods such features of the data are easy to be observed. The fact that there are hybrid items could be an indication of the quality of data.

In Chapter 2, we have presented our contribution to NP optimization problems, focusing on two well-known NP-hard problems: Travelling Salesman Problem (TSP) and Set Covering Problem (SCP). In Section 2.1 a short overview of NP completeness is made. In Section 2.2 the travelling salesman problem is approached using the stigmergic agent model. The Stigmergic Agent System (SAS) combines the strengths of Multi-agent Systems (MAS) and Ant Colony Systems (ACS). Stigmergy provides a general mechanism that relates individual and colony level behaviours: individual behaviour modifies the environment, which in turn modifies the behaviour of other individuals. The stigmergic agent mechanism employs several agents able to interoperate in order to solve problems by using both direct communication and indirect (stigmergic) communication. The algorithm was evaluated on several standard datasets outlining the potential of the method. In Section 2.3 the soft agent model is introduced. A soft agent is an intelligent agent that has to deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both. This new agent model is used in Section 2.4 where a new incremental clustering approach to the Set Covering Problem is presented. Experiments on standard datasets suggest that the approach is promising.

As future research directions, we intend to improve the proposed approaches, to extend the evaluation of the techniques that were proposed in this thesis and to investigate the use and to develop other computational models in pattern recognition.

Future work will be conducted in the following directions:

- investigating other metaheuristics with the aim of identifying additional potentially beneficial hybrid models
- using our models for solving other NP-optimization problems
- extending our methods in order to handle categorical data
- applying the incremental clustering approach in Intrusion Detection Systems
- improving the proposed classification model for DRSP by adding to it the capability to adapt itself using a feedback received when inappropriate data representations are selected
- applying other machine learning techniques like self-organizing feature maps or other modelling techniques for solving the problem of automatic selection of data representations during the execution of a software system
- studying the applicability of other learning techniques like semi-supervised learning or reinforcement learning in order to avoid as much as possible the supervision during the training process
- evaluating our techniques on other case studies and real software systems.

Bibliography

[AAA03] Noga Alon, Baruch Awerbuch, and Yossi Azar. The online set cover problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 100–105, New York, NY, USA, 2003. ACM.

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60, 1999.

[AM10] Laurent Alfandari and Jérôme Monnot. Approximation of the clustered set covering problem. *Electronic Notes in Discrete Mathematics*, 36:479–485, 2010.

[AY01] Charu C. Aggarwal and Philip S. Yu. Outlier detection for high dimensional data. In *SIGMOD Conference*, pages 37–46, 2001.

[AZAY10] Moh'd Belal Al-Zoubi, Al-Dahoud Ali, and Abdelfatah A. Yahya. Fuzzy clustering-based approach for outlier detection. In *Proceedings of the 9th WSEAS international conference on Applications of computer engineering*, ACE'10, pages 192–197, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[BC96] J.E Beasley and P.C Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94(2):392 – 404, 1996.

[Bea] J E Beasley. OR-Library. http://people.brunel.ac.uk/∼mastjjb/jeb/orlib /scpinfo.html.

[BG08] Irad Ben-Gal. *Bayesian Networks*. John Wiley & Sons, Ltd, 2008.

[Bla94a] Betty Blair. Interview with zadeh, creator of fuzzy logic. *Azerbaijan International*, 2(4):46–47, Winter 1994.

[Bla94b] Betty Blair. Short biographical sketch. *Azerbaijan International*, 2(4):4, Winter 1994.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[BW94] Aart J. C. Bik and Harry A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 57–75, London, UK, 1994. Springer-Verlag.

[BW96] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Trans. Parallel Distrib. Syst.*, 7:109–126, February 1996.

[CCFM97] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 626–635, New York, NY, USA, 1997. ACM.

[CCGa] Gabriela Czibula, Istvan Czibula, and Radu Găceanu. Intelligent data structures selection using neural networks. *Knowledge and Information Systems*, pages 1–22. 10.1007/s10115-011-0468-3.

[CCGb] Gabriela Czibula, Istvan Czibula, and Radu Găceanu. A support vector machine model for intelligent selection of data representations. *Applied Soft Computing*. under review.

[CDG07] C. Chira, D. Dumitrescu, and R. D. Găceanu. Stigmergic agent systems for solving NP-hard problems. *Studia Informatica*, Special Issue KEPT-2007: Knowledge Engineering: Principles and Techniques (June 2007):177–184, June 2007.

[CH96] Tyng-Ruey Chuang and Wen L. Hwang. A probabilistic approach to the problem of automatic selection of data representations. *SIGPLAN Not.*, 31:190–200, June 1996.

[cir10] Patient contributed image repository. http://www.pcir.org/, 2010.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[CM01] Maria Stella Fiorenzo Catalano and Federico Malucelli. Practical parallel computing. chapter Parallel randomized heuristics for the set covering problem, pages 113–132. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[CMP06] C. Chira C. M. Pintea, P. Pop. Reinforcing ant colony system for the generalized traveling salesman problem. In *Volume of Evolutionary Computing, International Conference Bio-Inspired Computing - Theory and Applications (BIC-TA)*, pages 245–252, New York, NY, USA, September 18–22, 2006. Wuhan, China.

[CMR$^+$07] Oscar Castillo, Patricia Melin, Oscar Montiel Ross, Roberto Seplveda Cruz, Witold Pedrycz, and Janusz Kacprzyk. *Theoretical Advances and Applications of Fuzzy Logic and Soft Computing*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[coNop] A compendium of NP optimization problems. http://www.nada.kth.se/viggo/problemlist/compendium.html.

[CSM$^+$11] Broderick Crawford, Ricardo Soto, Eric Monfroy, Fernando Paredes, and Wenceslao Palma. A hybrid ant algorithm for the set covering problem. *International Journal of the Physical Sciences*, 6(19):4667–4673, September 16 2011.

[CST00] Nello Cristianini and John Shawe-Taylor. *An introduction to support Vector Machines: and other kernel-based learning methods*. Cambridge University Press, New York, NY, USA, 2000.

[CXC04] L. Chen, X. H. Xu, and Y. X Chen. An adaptive ant colony clustering algorithm. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on, Vol. 3*, pages 1387–1392, 2004.

[DB05] Marco Dorigo and Christian Blum. Ant colony optimization theory: A survey. *Theor. Comput. Sci.*, 344(2-3):243–278, 2005.

[DDC99] Marco Dorigo and Gianni Di Caro. *The ant colony optimization meta-heuristic*, pages 11–32. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[DGF$^+$91] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien. The dynamic of collective sorting robot-like ants and ant-like robots. In *SAB90 - 1st Conf. On Simulation of Adaptive Behavior: From Animals to Animats*, pages 356–365. MIT Press, 1991.

[DICiM11] Digital Imaging and Communications in Medicine. http://medical.nema.org/, 2011.

[DL11] Steven Simske Dalong Li. Training set compression by incremental clustering. *Journal of Pattern Recognition Research*, 6:56–64, 2011.

[Dor07] M. Dorigo. Ant colony optimization. *Scholarpedia*, 2(3):1461, 2007.

[DP95] Dan Dumitrescu and Horia Florin Pop. Degenerate and non-degenerate convex decomposition of finite fuzzy partitions — I. *Fuzzy Sets and Systems*, 73:365–376, 1995.

[DP98] Dan Dumitrescu and Horia Florin Pop. Degenerate and non-degenerate convex decomposition of finite fuzzy partitions — II. *Fuzzy Sets and Systems*, 96:111–118, 1998.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.

[EKS$^+$98] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 323–333, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[EpKSX96] Martin Ester, Hans peter Kriegel, Jrg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[Est09] Martin Ester. Density-based clustering. In *Encyclopedia of Database Systems*, pages 795–799. 2009.

[Etz96] O. Etzioni. Moving up the information food chain: deploying softbots on the world wide web. In *Proceedings of the 13rd national Conference on Artificial Intelligence (AAAI- 96)*, pages 4–8. Portland, OR, 1996.

[fIPA] Foundation for Intelligent Physical Agents. http://www.fipa.org/.

[FLM97] T. Finin, Y. Labrou, and J. Mayfield. *Kqml as an Agent Communication Language.* Software Agents, B.M. Jeffrey, MIT Press, 1997.

[FMPS00] Kilian Foth, Wolfgang Menzel, Horia Florin Pop, and Ingo Schröder. An experiment on incremental analysis using robust parsing techniques. In *The 18th International Conference on Computational Linguistics*, pages 1026–1030. Universität des Saarlandes, Saarbrücken, Germany, July-August 2000.

[For10] M Forina. http://archive.ics.uci.edu/ml, 2010.

[Găc11] Radu D. Găceanu. A bio-inspired fuzzy agent clustering algorithm for search engines. *Procedia Computer Science*, 7(0):305 – 307, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[GC10] Serge Guillaume and Brigitte Charnomordic. Interpretable fuzzy inference systems for cooperation of expert knowledge and data in agricultural applications using fispro. In *FUZZ-IEEE*, pages 1–8, 2010.

[Gei93] S. Geisser. *Predictive inference: an introduction.* Monographs on statistics and applied probability. Chapman & Hall, 1993.

[GKP02] Mihaela Gordan, Constantine Kotropoulos, and Ioannis Pitas. A support vector machine-based dynamic network for visual speech recognition applications. *EURASIP J. Appl. Signal Process.*, 2002:1248–1259, January 2002.

[GKS09] Betsy George, James M. Kang, and Shashi Shekhar. Spatio-temporal sensor graphs (stsg): A data model for the discovery of spatio-temporal patterns. *Intell. Data Anal.*, 13(3):457–475, 2009.

[Glo89] Fred Glover. Tabu search - part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[Glo90] Fred Glover. Tabu search - part II. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[GO11] R. D. Găceanu and G. Orbán. Using rsl to describe the stock exchange domain. In *microCAD International Scientific Conference*. University of Miskolc, Hungary, 31 March – 1 April 2011.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[GP08] Darwin Gouwanda and S. G. Ponnambalam. Evolutionary search techniques to solve set covering problems. In *World Academy of Science, Engineering and Technology*, pages 20–26. WASET, March 2008.

[GP10] R. D. Găceanu and H. F. Pop. An adaptive fuzzy agent clustering algorithm for search engines. In *MACS2010: Proceedings of the 8th Joint Conference on Mathematics and Computer Science*, pages 185–196. Komarno, Slovakia, 2010.

[GP11a] R. D. Găceanu and H. F. Pop. A context-aware ASM-based clustering algorithm. *Studia Universitatis Babes-Bolyai Series Informatica*, LVI(2):55–61, 2011.

[GP11b] R. D. Găceanu and H. F. Pop. A fuzzy clustering algorithm for dynamic environments. In *KEPT2011: Knowledge Engineering Principles and Techniques, Selected Papers, Eds: M. Frentiu, H.F. Pop, S. Motogna*, pages 119–130. Babes-Bolyai University, Cluj-Napoca, Romania, July 4–6 2011.

[GP11c] R. D. Găceanu and H. F. Pop. An incremental ASM-based fuzzy clustering algorithm. In *Informatics'2011, Slovakia, i'11:Proceedings of the Eleventh International Conference on Informatics, Informatics 2011, Eds: V. Novitzká, Štefan Hudák*, pages 198–204. Slovak Society for Applied Cybernetics and Informatics, Rožňava, Slovakia, November 16–18 2011.

[GP12] R. D. Găceanu and H. F. Pop. An incremental approach to the set covering problem. *Studia Universitatis Babes-Bolyai Series Informatica*, LVIII(2), 2012.

[Har75] J.A. Hartigan. *Clustering algorithms*. Wiley series in probability and mathematical statistics. Applied probability and statistics. Wiley, 1975.

[HCL00] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification, 2000.

[HFH$^+$09] 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):10–18, 2009.

[HK06] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques, 2nd ed.* Morgan Kaufmann, 2006.

[HKT01] J. Han, M. Kamber, and A. K. H. Tung. *Spatial Clustering Methods in Data Mining: A Survey*. Taylor and Francis, 2001.

[HL0] Health Level 7. www.hl7.org/, 0.

[hp10] DICOM home page. ftp://medical.nema.org/medical/dicom/datasets/, 2010.

[HSP08] Samer Hassan, Mauricio Salgado, and Juan Pavón. Friends forever: Social relationships with a fuzzy agent-based model. In *HAIS*, pages 523–532, 2008.

[HZK+09] Pari Delir Haghighi, Arkady B. Zaslavsky, Shonali Krishnaswamy, Mohamed Medhat Gaber, and Seng Wai Loke. Context-aware adaptive data stream mining. *Intell. Data Anal.*, 13(3):423–434, 2009.

[Ins] MP-TESTDATA The TSPLIB Symmetric Traveling Salesman Problem Instances. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html.

[Iri88] Machine Learning Repository Iris. http://archive.ics.uci.edu/ml/datasets/iris, 1988.

[Kam10] A. Kamble. Incremental clustering in data mining using genetic algorithm. *International Journal of Computer Theory and Engineering*, 2(3):1793–8201, 2010.

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26:313–, September 2001.

[KHK99] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.

[KJV83] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[KKvdSS96] Ben Krse, Ben Krose, Patrick van der Smagt, and Patrick Smagt. An introduction to neural networks, 1996.

[Kot07] Sotiris B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[KS05] Vikas Kumar and Marta Schuhmacher. Fuzzy uncertainty analysis in system modelling. In *European Symposium on Computer Aided Process Engineering 15 L. Puigjaner and A. Espua (Editors)*, 2005.

[KS08] P. R. Kumar K. Sreelakshmi. Performance evaluation of short term wind speed prediction techniques. *IJCSNS International Journal of Computer Science and Network Security*, 8(8):162–169, 2008.

[KTL11] Suzan Köknar-Tezel and Longin Jan Latecki. Improving svm classification on imbalanced time series data sets with ghost points. *Knowl. Inf. Syst.*, 28:1–23, July 2011.

[KY95] GEORGE J. Klir and BO Yuan. *FUZZY SETS AND FUZZY LOGIC Theory and Applications*. Prentice Hall, 1995.

[LB05] Kristopher R. Linstrom and A. John Boye. A neural network prediction model for a psychiatric application. In *Proceedings of the Sixth International Conference on Computational Intelligence and Multimedia Applications*, pages 36–40, Washington, DC, USA, 2005. IEEE Computer Society.

[LF94] E. Lumer and B. Faieta. Diversity and adaptation in populations of clustering ants. In *J.-A.Meyer, S.W.Wilson(Eds.), Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animats, Vol.3*, pages 501–508. MIT Press/Bradford Books,Cambridge, MA, 1994.

[LKC02] Kyung-Soon Lee, Kyo Kageura, and Key-Sun Choi. Implicit ambiguity resolution using incremental clustering in korean-to-english cross-language information retrieval. In *Proceedings of the 19th international conference on Computational linguistics - Volume 1*, COLING '02, pages 1–7, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[LLLH10] Zhenhui Li, Jae-Gil Lee, Xiaolei Li, and Jiawei Han. Incremental clustering for trajectories. In Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe, editors, *Database Systems for Advanced Applications*, volume 5982 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 2010.

[Low78] James R. Low. Automatic data structure selection: an example and overview. *Commun. ACM*, 21:376–385, May 1978.

[LR76] James Low and Paul Rovner. Techniques for the automatic selection of data structures. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 58–67, New York, NY, USA, 1976. ACM.

[MA75] E. H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies*, 7(1):1–13, 1975.

[Mar09] Stephen Marsland. *Machine Learning: An Algorithmic Perspective.* Chapman & Hall/CRC, 1st edition, 2009.

[Mat] Fuzzy Logic Toolbox MathWorks. http://www.mathworks.com/help/toolbox /fuzzy/fp351dup8.html.

[Mit97] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MK91] S. Miyake and F. Kanaya. A neural network approach to a bayesian statistical decision problem. *IEEE Trans. Neural Networks*, 2:538–540, 1991.

[Mou01] D.M. Mount. Lecture notes CMSC 420, Data Structures, 2001.

[NGS11] K Venkatramaiah Deepak P C Navneet Goyal, Poonam Goyal and Sanoop P S. An efficient density based incremental clustering algorithm in data warehousing environment. In *2009 International Conference on Computer Engineering and Applications IPCSIT vol.2 (2011), IACSIT Press, Singapore*, pages 482 – 486, 2011.

[Ngu10] Nam Nguyen. A new svm approach to multi-instance multi-label learning. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 384–392, Washington, DC, USA, 2010. IEEE Computer Society.

[NMM06] Giuseppe Narzisi, Venkatesh Mysore, and Bud Mishra. Multi-objective evolutionary optimization of agent-based models: An application to emergency response planning. In *Computational Intelligence*, pages 228–232, 2006.

[oPDsf10] Washington State University College of Pharmacy DICOM sample files. http://info.betaustur.org/, 2010.

[Osi10] Advanced Imaging in 3D/4D/5D Sample DICOM Image Sets Osirix. http://pubimage.hcuge.ch:8080/, 2010.

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

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[PS96] Horia Florin Pop and Costel Sârbu. A new fuzzy regression algorithm. *Anal. Chem.*, 68:771–778, 1996.

[PSHD96] Horia Florin Pop, Costel Sârbu, Ossi Horowitz, and Dan Dumitrescu. A fuzzy classification of the chemical elements. *J. Chem. Inf. Comput. Sci.*, 36:465–482, 1996.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[RiplsDis10] J.-P.: Signal Roux and image processing lab sample DICOM image sets. http://www.creatis.insa-lyon.fr/ jpr/public/gdcm/gdcmsampledata/, 2010.

[RKY10] Vikas C. Raykar, Balaji Krishnapuram, and Shipeng Yu. Designing efficient cascaded classifiers: tradeoff between accuracy and cost. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 853–860, New York, NY, USA, 2010. ACM.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[Roj96] Raúl Rojas. *Neural networks: a systematic introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[Rov78] P. Rovner. *Automatic representation selection for associative data.* Managing Requirements Knowledge, International Workshop, 1978.

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[SC08] Ingo Steinwart and Andreas Christmann. *Support Vector Machines.* Springer Publishing Company, Incorporated, 1st edition, 2008.

[SCCK04] S. Schockaert, M. De Cock, C. Cornelis, and E. E. Kerre. Fuzzy ant based clustering. In *Ant Colony Optimization and Swarm Intelligence, 4th International Workshop (ANTS 2004), LNCS 3172*, pages 342–349, 2004.

[sciom11] Open source clinical image and object management. http://www.dcm4che.org/, 2011.

[SdLFdCG09] Eduardo J. Spinosa, André Carlos Ponce de Leon Ferreira de Carvalho, and João Gama. Novelty detection with application to data streams. *Intell. Data Anal.*, 13(3):405–422, 2009.

[Ser06] Gabriela Serban. *Sisteme Mutiagent in inteligenta artificiala distribuita. Arhitecturi si aplicatii.* Ed. Risoprint, Cluj-Napoca, 2006.

[sf] Apache software foundation. http://www.apache.org/.

[SGT$^+$02] Ingo Schröder, K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, T. Fogarty (eds, Horia F. Pop, A Fachbereich Informatik, Wolfgang Menzel, and Kilian A. Foth. Learning weights for a natural language grammar using genetic algorithms, 2002.

[SH07] Mark D. Skowronski and John G. Harris. Automatic speech recognition using a predictive echo state network classifier. *Neural Networks*, 20(3):414 – 423, 2007. Echo State Networks and Liquid State Machines.

[SK99] Panu Somervuo and Teuvo Kohonen. Self-organizing maps and learning vector quantization forfeature sequences. *Neural Process. Lett.*, 10:151–159, October 1999.

[SP00] Costel Sârbu and Horia Florin Pop. Fuzzy clustering analysis of the first 10 meic chemicals. *Chemosphere*, 40:513–520, 2000.

[SP04] Gabriela Serban and Horia Florin Pop. *Tehnici de Inteligenta Artificiala. Abordari bazate pe Agenti Inteligenti.* Ed. Mediamira, Cluj-Napoca, 2004.

[Spe87] C. Spearman. The proof and measurement of association between two things. By C. Spearman, 1904. *The American journal of psychology*, 100(3-4):441–471, 1987.

[SS01] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, USA, 2001.

[SSS79] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in setl. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 197–210, New York, NY, USA, 1979. ACM.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3:126–143, April 1981.

[Ste90] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, 1990.

[TD10] Balint Takacs and Yiannis Demiris. Spectral clustering in multi-agent systems. *Knowl. Inf. Syst.*, 25:607–622, December 2010.

[Vap00] V.N. Vapnik. *The nature of statistical learning theory.* Statistics for engineering and information science. Springer, 2000.

[VSP09] K. R. Venugopal, K. G. Srinivasa, and L. M. Patnaik. *Soft Computing for Data Mining Applications.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[Wat89] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[WB01] D.A. Watt and D.F. Brown. *Java collections: an introduction to abstract data types, data structures, and algorithms.* John Wiley, 2001.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.

[Win91] Machine Learning Repository Wine. http://archive.ics.uci.edu/ml/datasets/wine, 1991.

[WLZ00] G Wahba, Y Lin, and H Zhang. Generalized approximate cross validation for support vector machines, or, another way to look at margin-like quantities. *Advances in large margin classifiers*, (1006):297309, 2000.

[Woo99] Michael Wooldridge. *Intelligent Agents, An Introduction to Multiagent Systems.* Ed. G. Weiss, 1999.

[Woo09] Michael J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.).* Wiley, 2009.

[WYM97] Wei Wang, Jiong Yang, and Richard R. Muntz. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, pages 186–195, 1997.

[Yel03] D. M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Syst. J.*, 42:85–97, January 2003.

[Zad65] Lotfi Askar Zadeh. Fuzzy sets. *Inf. Control*, 8:338–353, 1965.

[Zad94] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37:77–84, March 1994.

[Zad97] Lotfi A. Zadeh. The roles of fuzzy logic and soft computing in the conception, design and deployment of intelligent systems. In *Software Agents and Soft Computing*, pages 183–190, 1997.

[Zad02] Lotfi A. Zadeh. Toward a perception-based theory of probabilistic reasoning with imprecise probabilities. *Journal of Statistical Planning and Inference*, 105(1):233 – 264, 2002. Imprecise Probability Models and their Applications.

[Zad08] Lotfi Askar Zadeh. Is there a need for fuzzy logic. *Information Sciences*, 178(13):2751–2779, July 2008.

[ZDYZ11] Xingquan Zhu, Wei Ding, Philip Yu, and Chengqi Zhang. One-class learning and concept summarization for data streams. *Knowledge and Information Systems*, 28:523–553, 2011. 10.1007/s10115-010-0331-y.

[Zha00] G. P. Zhang. Neural networks for classification: a survey. *IEEE Trans. Systems, Man and Cybernetics*, 30(4):451–462, November 2000.

[ZL10] Zhi-Hua Zhou and Ming Li. Semi-supervised learning by disagreement. *Knowl. Inf. Syst.*, 24:415–439, September 2010.

[ZRL97] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Min. Knowl. Discov.*, 1(2):141–182, 1997.