Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Klára Pešková

Ontological Description of Intelligent Agents

Ontologický popis inteligentních agentů

Department of Theoretical Computer Science and
Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc.

Study programme: Theoretical Computer Science, Functional
Programming and Artificial Inteligence

Acknowledgments
A very special thanks to a very special advisor. Another special thanks belong to my two colleagues, Martin Pilát and Ondřej Kazík, and to all people who have helped me and supported me — thank you all very much, your help and support is strongly appreciated.

I declare that I have written the thesis on my own and that I cited all used sources of information. I agree with public availability and lending of the thesis.

Prague, 4th August 2010

Klára Pešková

Contents

1 Introduction **7**

2 Developing Agents in JADE **9**
2.1 Agents 9
2.2 Ontology 11
2.3 The Foundation for Intelligent Physical Agents (FIPA) 12
2.3.1 Communicative Act (CA) 14
2.3.2 Interaction Protocols 14
2.3.3 FIPA-SL Content Language 15
2.3.4 Messages Format 15
2.4 JADE 16
2.4.1 JADE & FIPA 16
2.4.2 What Does JADE provide? 17

3 Related and Used Methods and Software **19**
3.1 Data mining 19
3.2 Weka (Waikato Environment for Knowledge Analysis) 20
3.3 ExpML 21

4 Pikater Multi-agent System **23**
4.1 Messages Ontology Language 24
4.2 Four Layer Architecture 26
4.3 Solving Problems 28
4.3.1 Scenario 1 29
4.3.2 Scenario 2 29
4.3.3 Scenario 3 29
4.3.4 Scenario 4 30
4.3.5 Meta-learning — Searching for the Best Agent 30

5 Implementation **33**
5.1 Agents 33

5.2 Solving a Problem 37
5.3 Agent Behavior and Communication 39
5.4 Ontologies 40

6 Experiments - Showing the Software 43
6.1 Experiment 1 — Setting the Parameters 43
6.2 Experiment 2 — Choosing the Best Options 47
6.3 Experiment 3 — Choosing the Best Agent 50

7 Conclusion 53

List of Figures 57

List of Tables 58

Název práce: Ontologický popis inteligentních agentů
Autor: Klára Pešková
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Roman Neruda, CSc.
e-mail vedoucího: roman@cs.cas.cz

Abstrakt:

V práci představujeme multi-agentní systém Pikater, který zastřešuje různé výpočetní metody používané v umělé inteligenci, a nabízí uživatelům prostředí pro pohodlné provádění jednoduchých i složitějších experimentů, jednak s metodami, které už jsou v systému implementovány, jednak s jejich vlastními metodami. Systém získává zkušenosti z prováděných experimentů a umí doporučit nejvhodnější výpočetní metodu i pro zcela nová data. Algoritmus výběru nejvhodnější metody je založen na hledání podobností v obecných vlastnostech dat a na výsledcích pedchozích experimentů. Systém využívá multi-agentní platformu JADE a výpočetní metody z Weky. Komunikace mezi agenty probíhá podle standardních komunikačních FIPA protokolů, obsah zpráv je definován pomocí ontologií.
Klíčová slova: agent, multi-agentní systém, dobývání znalostí, ontologie

Title: Ontological description of intelligent agents
Author: Klára Pešková
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Roman Neruda, CSc.
Supervisor's e-mail address: roman@cs.cas.cz

Abstract:

In this thesis we present the Pikater multi-agent system, which is designed to provide the researchers an environment for testing various data mining methods, that are the part of the system, as well as let them experiment with their own data mining methods. The system is also capable of recommending a data mining method for the never-before-seen datasets. The algorithm of choosing the best possible method is based on finding the similarities in the datasets and on the system's previous experience with solving similar tasks. Finding the nearest dataset is based on the proposed metric on metadata — the general information about the datasets. To implement our system, we used the JADE platform, agents embed the Weka data mining methods. The communication among individual agents follows the FIPA protocols and the content of the messages is specified by messages ontology.
Keywords: agent, multi-agent system, data mining software, ontology

Chapter 1

Introduction

The importance of gaining information from data have still been increasing over the last few decades. With the raise of fast computers, huge data storage space, and the Internet, we are literally flooded with large amounts of data.

To discover the patterns in data or generally to get some knowledge from them and to learn from them is one of the most important goals of todays research on the field of artificial intelligence. It can be used in wide range of areas of human interest, starting with analyzing the data from space, through diagnosing the patients with certain symptoms, to choosing the advertisement the user will most likely click on when they enter their Internet search-engine.

The artificial intelligence provides many different methods how to do this, but it usually requires deeper understanding of both the data and the methods, to be able to use them with satisfactory results, as the methods and also the datasets are different in many ways.

It should therefore be the next goal of artificial intelligence research to develop meta learning systems, which would learn from their previous experience, and which would be able to give advise on what methods to use in particular situations. Such systems would make the sophisticated data mining methods accessible to the wider public.

A software that would be capable of learning and giving advise should be autonomous to a great extinct, it should show an intelligent behavior and it should also be able to gain its experience from as many sources as possible. It is plain to see that such a system would have to implement quite complex behavior, but on the other hand solving such a complex problem comprises of several isolated subproblems, each of which can be solved separately, some of them even in parallel.

Over the last years the new approach to programming based on agents has expanded. It suggests to divide the problems addressed in software systems

in a more human-like way, so that the individual parts more or less copy the action a human would take if he was to solve the same problem. The word "agent" usually stands for a piece of software, which at least to some extent behaves in a human-like way. That is exactly what we want, because our ultimate long term goal is to develop a system that would substitute human expert on data mining.

The multi-agent approach naturally covers many of the traits that we expect such a system to have. This relatively new programming paradigm is nowadays frequently used both commercially and in academic sphere. Thanks to the large community around it, such a complex systems as described above are nowadays much more easier to develop, as there are the platforms that already cover some of the technical details and provide tools needed when the multi-agent system is developed. The international specifications that have been created in last few years bring advantage of combining even very different agents as long as they share the same input/output interface. The input and output of agents is in fact created by messages that the agents send among each other (or a human user), the common language defined by ontologies is used in the messages, to make sure that all the involved parties understand.

In this thesis we present the Pikater multi-agent system that embeds different data mining methods. The system not only provides an environment for experimenting with different data mining methods, but it also learns with every single task that it solves, so that it is eventually capable of giving advise on choosing the appropriate data mining method. Because we stick to the international standards, the system is easily extensible, so that the researchers' own data mining methods can be tested. We also want the individual parts of the system — the agents — to communicate in a structured ontology language understandable by humans, and possibly by every other agent they meet.

The thesis is divided into seven chapters. In Chapter 2 we define some of the most important conceptions and introduce the standards and the software platform we have used to develop our system. In Chapter 3, we mention other pieces of software and methods we have used. Chapters 4 a 5 focus on describing our multi-agent system — Chapter 4 is mostly theoretical, while in Chapter 5 we present the specific solution. After we describe our system, we present some experiments that have been performed to show the features of our system. In conclusion we outline the extensions of our system we would like to implement in the future.

Chapter 2

Developing Agents in JADE

2.1 Agents

What is an agent? There are many different definitions of the *agent* notion, as the authors try to describe the collection of agents they have on their minds. Some of them are very wide and cover nearly every piece of software — they need further defining of the notions used [4], some of them on the other hand are very specific.

The most mentioned one is probably that by [5]:

> "...a hardware or (more usually) software-based computer system that enjoys the following properties:
>
> - *autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
> - *social ability*: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
> - *reactivity*: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
> - *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative."

As the whole agent-oriented programming is rather a conceptual question than a matter of exact definitions, there's no need to hold on to the definitions

strictly. It should be used to get the right impression of what the agenthood is about.

Russell and Norvig, who wanted all the machine learning algorithm to satisfy their definition, put it this way: "The notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents." [4]

However, our perception of agenthood meets the above definition quite accurately. For us, an agent is an independent piece of software, that lives in its environment and communicates with the outside world and other agents by sending and receiving messages, and by perceiving and changing the environment (there is no other way how to change the internal state of agent or how to make it perform an action). It responds to the user and other agents' requests (note that it doesn't mean that it has to meet them, agents can agree or refuse), and it can also undertake actions on its own.

By environment we understand the user interface combined with the data and other agents. The communication with user can be either local or via Internet, the agents and data can be distributed over the network as well, but in our implementation they will all be located on one machine.

In addition to the above definition, we want our agents to satisfy some of the traits mentioned in [23]. Our agents are

- *truthful*, providing the certainty that they will not deliberately communicate false information,
- *benevolent*, trying to perform what they are asked for,
- *rational*, always acting in order to achieve its goals and never to prevent its goals being achieved,
- *mobile* - independent on their location, so that the multi-agent system can be spread over the network (Internet).
- They can *learn* to get better performance of their future action.

We typically need more than one agent, to be able to perform a requested goal. *Multi-agent system (MAS)* is a system composed of multiple intelligent autonomous agents that interact with each other - either directly by messaging or indirectly by changing the state of the environment.

Agent-based programming is a relatively new programming paradigm that brings alternative approach how to create large distributed software systems. While all the communication among the different parts of the system is obtained by messaging, it doesn't really depend on the location nor

the platform. It is a multi-agent system's nature to be spread over the large area networks. It makes multi-agent systems an excellent solution for distributed systems. MAS can be very heterogeneous. As long as they agree on basic communication principles the agents can be very different - they can run on different platforms, have their own internal representation of data or can be programmed in different types of programming languages.

The behavior of agents in a MAS can be either *cooperative*, i.e. all the agents in the system try to achieve the same goal, or *competitive* - agents have different goals, e.g. their owners can have different intentions (think of a system where people are selling and buying things).

The actions of agents can be planned centrally i.e. there is one (or more) agent(s) that tell the others what to do, or the planning of the actions can be distributed - the agents decide for themselves. If the agents are cooperative, usually the centralized planning is used, while the competitive behavior corresponds more likely with the distributed planning.

In our thesis, we consider multi-agents systems that are cooperative and centralized.

2.2 Ontology

Originally, *ontology* (from the Greek *ontos* meaning "of being" and *-logia* meaning "science, study, theory") is a philosophical discipline, a part of the major branch of philosophy known as metaphysics. Ontology was considered a significant part of philosophy as far back as ancient Greece, and was established by philosophers such as Aristotle or Platon.

In the philosophical sense, ontology is the study of what there is — it deals with questions concerning the existence of entities, their most general features and the relations among them.

In computer science, an ontology is a description of the knowledge domain in a machine interpretable way. The way we look at the conception is not that far from its original meaning, as it may seem at the first sight. To capture the knowledge means to describe the entities, their features and define their relations. An ontology defines a (often hierarchical) structure of information and a common vocabulary for researchers (or agents) who need to share information in a domain.

Nowadays, ontologies are widely used in computer science, especially on the web, because of the growing requirements on processing the information by machines, not only providing them to human. They are used when designing *semantic web*, to categorize web sites (very large ontologies, such as an ontology that Yahoo! provides users for searching the Internet[13]), or to

categorize the products (such as on Amazon.com[14]). Another application area is in multi-agent systems, where we use ontologies to specify the content of communication language (the meaning of terms in vocabularies and the relationships between those terms).

An ontology usually deals with

- *concepts* - the types of objects, classes in programming
- *instances* - the specific examples of concepts, the individuals,
- *attributes* - the features or the properties of the concepts (and instances)
- *relations* that are between the concepts and the individuals

Ontologies are encoded using ontology languages, that are commonly based on either first-order logic or on description logic. Different ontology languages provide different facilities. One of the recently most developed ontology language is *Web Ontology Language (OWL)* from the World Wide Web Consortium (W3C). OWL has three increasingly-expressive sublanguages: OWL-Lite, OWL-DL, and OWL-Full.

OWL-Lite is the syntactically simplest sub-language. It is intended to be used in situations where only a simple class hierarchy and simple constraints are needed.

OWL-DL is much more expressive than OWL-Lite and is based on description logics. Description logics are a family of formal knowledge representation languages, they are a decidable fragment of First Order Logic, so that the automated reasoning, such as checking for inconsistencies, is possible. When designing the ontology in our system, we used this OWL sublanguage.

OWL-Full is the most expressive OWL sub-language. It is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies.[24]

For an example of ontology see our *messages ontology* in Chapter 4.

2.3 The Foundation for Intelligent Physical Agents (FIPA)

One of the benefits of using the multi-agent approach is its distributed nature. As mentioned above the single parts of a multi-agent system can be internally very different, but on the outside they should communicate in a transparent

way. The need of standardization process in the agent community led to the establishment of the non-profit association FIPA in 1996.

While the agent-oriented programming approach was already well-known on academic fields at that time, the use in commercial sphere was still waiting for its upswing. Therefore the essential fact for the success was the initial membership of both academic and industrial organizations.

Nowadays FIPA is a well-established IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. It provides standards for agents and multi-agent systems, that are available on-line [15] and evolve continually. To validate the formal FIPA specifications, the JADE project was started a decade ago.

The FIPA specifications do not cover the agents' internal mechanics, but rather the general concepts concerning agents, such as *agent communication* (Agent Communication Language — types of communicative acts, message exchange interaction protocols and content language), *7 communication layers* (transport, encoding, messaging, ontology, content expression, communicative act and interaction protocol layer) and *agent management* (creation, registration, location, communication, migration and operation of agents).

One of the features of *FIPA Agent Communication Language (FIPA ACL)* is the possibility of using different content languages and the management of conversations through predefined interaction protocols. *FIPA ACL* also specifies the *communicative acts (CAs)*.

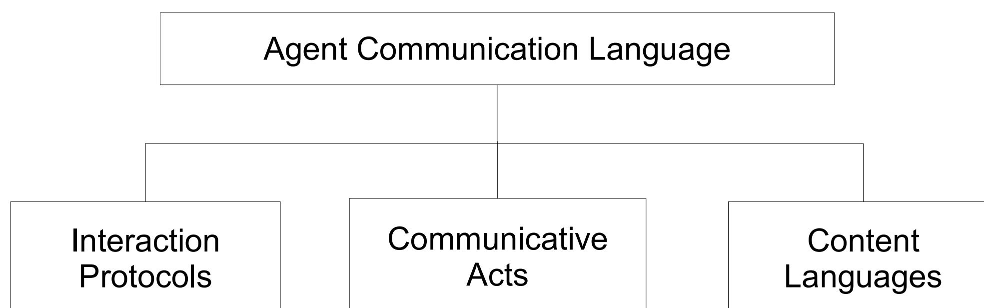

Figure 2.1: Agent Communication Language.

In the following subsections we will take a closer look at the three parts of the *FIPA ACL*:

2.3.1 Communicative Act (CA)

The basic concept of agent communication is a *communicative act (CA)*, specified in Communicative Act Library Specification [16], that represents the meaning (or the function) of the message in the conversation. *CAs* are based on speech act theory [17] which defines the functions of simply specified actions, such as *interrogatives* which query for information, *exercitives* which ask for an action to be performed, *phatics* which establish, prolong or interrupt communication, *paralinguistics* which relate a message to other messages, or *expressives* which express attitudes, intentions or beliefs [21].

As the following example shows, a message typically supports more than one function: The *FIPA CA Agree*, which is send when the agent agrees to perform an action in the future, is *phatic*, *paralinguistic* and *expressive* (all *FIPA CAs* support the expressive function as they are defined in a modal logic form that expresses attitudes, intentions and beliefs).

Other examples of *CAs* are:

Communicative Act	*Description*
Failure	The action was attempted but the attempt failed
Inform	The sender informs the receiver that a given proposition is true
Propose	The action of submitting a proposal to perform a certain action
Refuse	The action of refusing to perform a given action, and explaining the reason for the refusal
Query If	The action of asking another agent whether or not a given proposition is true
Request	The sender requests the receiver to perform some action

Table 2.1: Communicative acts.

2.3.2 Interaction Protocols

The *interaction protocols* stand for frequently used communication schemes, that are used to achieve a certain goal. They specify the course of the conversation, so that no important part of it is left out. FIPA specifications describe wide range of interaction protocols, from simple protocols such as Request Interaction Protocol [18] or Query Interaction Protocol [19] to more

complicated conversation schemes such as Contract Net Interaction Protocol [20].

For example the Request Interaction Protocol has two participants — an *initiator* and a *responder*. The *initiator* agent sends a message with a *request* performative, the *responder* agent replies either with a *response* message (with an *agree* or a *refuse* performative) or it sends a *result* right away (it can be a message with an *inform*, or a *failure* performative).

2.3.3 FIPA-SL Content Language

To define *FIPA CAs*, the *FIPA Semantic Language (FIPA SL)* is used. *FIPA SL* is formalized in a first-order modal language with identity. The content is defined in a string expression grammar, and can represent either *action expressions* or *propositions*. Content expressions are represented as well-formed formulas consisting of terms (constant, set, sequence, functional term, action expression) and constants (numerical constants, string, datetime). A well-formed formula is constructed from an atomic formula by applying construction operators or logical connective operators, e.g. negation, conjunction, disjunction, implication, equivalence or quantifiers [21].

In some cases, SL can also be used as a content of messages.

2.3.4 Messages Format

Now that we know all parts of *FIPA ACL*, we will take a closer look at the message format. The message is a set of parameters written in *FIPA ACL*. There is only one mandatory parameter (the *performative*), but usually the message also contains *sender*, *receiver* and *content* parameters.

In the following example of a FIPA-ACL message with a request performative, the "GUI" agent wants the "rbf" agent to send him his options; we use the fipa-request protocol; the ontology parameter specifies an ontology that gives meaning to symbols in the message content:

```
(request
  :sender  ( agent-identifier :name GUI@computer:1099/JADE )
  :receiver  (agent-identifier :name rbf@computer:1099/JADE )
  :content  "
    ((action
     (agent-identifier
       :name GUIagent@computer:1099/JADE
        (GET-OPTIONS)))"
  :language  fipa-sl
  :ontology  messages-ontology
  :protocol  fipa-request
  :conversation-id  C14418997_1279485578709
)
```

Figure 2.2: Example of an ALC message.

2.4 JADE

JADE is a software platform that provides basic middleware-layer functionalities which are independent of the specific application and which simplify the realization of distributed applications that exploit the software agent abstraction. [21][5]

The project was started in 1998 by Telecom Italia, motivated by the need to validate the early FIPA specifications, and it has eventually become a large software platform.

JADE is implemented in Java programing language, and thus provides the advantages of object oriented programing and makes it particularly easy to connect different pieces of software together.

2.4.1 JADE & FIPA

JADE is tightly connected to FIPA - as JADE is in fact an implementation of FIPA standards, it provides all the tools needed for creating a multi-agent system, such as protocols for messages communication, yellow pages service, which provides the list of available agents, etc. The JADE team even made an active contribution to the FIPA standardization process.

Because JADE has extended the FIPA model in several areas, the specifications do not provide complete coverage of JADE functions, however in all core FIPA aspects, JADE is fully compliant with the FIPA specifications.

2.4.2 What Does JADE provide?

JADE provides means to implement agents that have all (or at least some) of the above mentioned properties - the agents created by JADE are *autonomous*, *social*, they can be *reactive*, *pro-active*, *truthful*, *benevolent*, or *rational*.

Jade also provides an easy way to make agents *mobile*.

In JADE, the environment where agents live is represented by an *Agent Platform (AP)*, that consists of the machines, operating systems, FIPA agent management components and the agents themselves.

From JADE's point of view an agent is a computational process that inhabits an AP and typically offers one or more computational services that can be published as a service description.

Agents are labeled by a unique FIPA Agent Identifier (AID) so that they may be distinguished unambiguously.

JADE provides three basic services that help users to deal with agents within the AP. They are

- *Agent Management System (AMS)* - a mandatory component of an AP that is responsible for managing the operation of an AP, such as the creation and deletion of agents, and overseeing the migration of agents to and from the AP. Each agent must register with an AMS in order to obtain an AID which is then retained by the AMS as a directory of all agents present within the AP and their current state (e.g. active, suspended or waiting). The life of an agent with an AP terminates with its deregistration from the AMS.

- *Message Transport Service (MTS)* - a service provided by an AP to transport FIPA ACL messages between agents on a given AP and between agents on different APs. Messages are providing a transport envelope that comprises the set of parameters detailing, for example, to whom the message is to be sent.

- *Directory Facilitator (DF)* - an optional but very useful component providing yellow pages services to other agents. It maintains an accurate, complete and timely list of agents and must provide the most current information about agents.

JADE also supports use of ontologies in messages.

Chapter 3

Related and Used Methods and Software

3.1 Data mining

Data mining is a process of gaining information (or knowledge) from data. Some of the methods that try to discover patterns in large datasets are centuries old (Bayes' theorem, regression analysis). A data mining task can be very demanding when performed manually, but nowadays, in computer era, the pattern extraction is automated. Therefore the datasets can be much larger and the methods are much more sophisticated. In the second half of the 20th century many new methods such as neural networks, genetic algorithms, clustering or decision trees were developed.

Data mining deals with four basic types of tasks:

- *classification* — knowing the categories to which the data belong, we try to predict the categories for the new never-before-seen data.
- *regression* tries to find a function that models the data with the least error. Regression is similar to classification, but instead of predicting categories it predicts a continuous variable.
- *clustering* is a task of organizing the data into groups with regard to similarities in the data.
- *association rule learning*, also called the *market basket analysis* tries to discover the most common dependencies in the data. The example of such a task (that also gave a problem its name) is predicting which products are frequently bought together.

To solve the first two types of the above tasks, the *supervised learning* methods are used (for the *training* data we know what the result should be), while in the other two we have no previous knowledge about the data and we use *unsupervised learning* algorithms.

For various reasons, not all the patterns found by data mining algorithms are always correct. Thus the verifying of results is an important part of a data mining process. To test the algorithm, we use the *testing data* that were not part of the dataset used for learning. Usually the data are initially divided to two unevenly large parts — the larger one is used as a *training set* while the smaller is used to test our algorithm. We measure how good the method is by computing the *error* on the *testing data*. There are various methods how to divide the data to training and testing dataset, and also different ways how to compute the error.

There are many software systems that implement data mining tools. Among the most frequently used commercial pieces of software are for example SAS Enterprise Miner [6], SPSS Clementine [7] or STATISTICA Data Miner [8], the well-known freely available open-source software systems are Weka [10] or Orange [11].

3.2 Weka (Waikato Environment for Knowledge Analysis)

In the machine learning community, Weka is well-established and also very popular software for data mining tasks. It has been developed since 1993 at the University of Waikato, New Zealand, and is implemented in Java. It is a large collection of machine learning algorithms and other data mining tools, that can either be called from an outside code, or applied directly to a dataset, using the commandline or the Weka workbench, that provides a sophisticated graphical user interface, tools for visualization etc. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. [9]

Weka is open source software issued under the GNU General Public License, which, together with the fact, that it is implemented in Java, makes it possible (and easy) to re-use the code in other Java applications, and the programmers are encouraged to use the Weka tools in their own software.

3.3 ExpML

While the machine learning systems can be of a very different nature, there is a call for a unified way of describing and storing the learning experiments. The formalization has to be independent on how the experiments are implemented, so that the experiments could be easily exchanged between different machine learning systems. While the new learning approaches often bring new features to the learning algorithms description, one of the other requirements on how the experiments are stored is that it has to be easily extensible.

ExpML is a suggestion of a XML-based markup language to share not only the setting but also the result of an experiment in an easily extensible XML format [12].

Chapter 4

Pikater Multi-agent System

Pikater is a multi-agent system designed for researchers to run their experiments, and to experiment with their own learning methods. The system contains a set of data mining methods and provides an environment to set their parameters. But it is not only a tool that provides a neat environment, it also includes intelligent methods, which, if demanded, give advise on how the experiments should look like.

The system is designed to deal with all types of data mining tasks as described in the previous chapter. So far we have implemented methods for classification and regression.

Researchers usually have a dataset (or more different datasets), and they typically want to gain some knowledge from the data. They have decided to use a particular data mining method, or maybe they haven't yet, or they just want to test their own method and experiment with it.

When designing our system, we tried to analyze what do researchers usually want to do. According to our experience, there are several possible scenarios from the researcher's point of view:

- Scenario 1: *She knows what method she wants to use and she prefers to set the parameters manually.*
- Scenario 2: *She knows what method to use, but she doesn't know what its parameters should be.*
- Scenario 3: *She wants to try more possible methods and she wants to know which one is the best (or better) when running on her dataset.*
- Scenario 4: *She doesn't know what method to use at all, but she doesn't want to try all of them, and she'd appreciate some advise.*

The system dealing with these situations should be able to handle requests from several users at the same time. Therefore we need the computations to run in parallel. We also want the system to be distributed over the network, as the user is often physically elsewhere than the system is located, and we don't want the user to have to run the system on their own computer. Moreover, the more users run their experiments on the system, the more data is available for the system to learn from. We also want the system to be easily extended, so that it would be possible for researchers to run and test their own methods. The system should also be able to recover from errors, such as unavailable parts of the system, or handle the situations when some of the parts are busy.

It is obvious that a system that would meet all the above requirements would be quite a complex piece of software, which should contains means for communication between its different parts and enable adding the new parts to the system, such as new data mining methods, in an easy way. Natural solution of such a system is using the agent-based approach and implementing the whole system as a multi-agent system, as all these above mentioned issues are addressed in MAS.

The individual parts of MAS — the agents — are autonomous to some degree. It means that they can be easily added to the system, even when the system is already running. The agent-based approach also provides a solution to the communication issues, as the social ability is one of the agents' basic features. Such a system as described above also takes benefits from other agents' features, such as *pro-activeness* — when not busy with computing users requests, agents can run computations on the data provided earlier by the users and "educate" themselves. *Reactivity* is also a welcomed feature — agents perceive their environment (which is the Internet and the data provided by users) and they respond in a timely fashion to the user's requests.

In the following sections, we are going to describe single agents, their relations and the way they communicate with each other.

4.1 Messages Ontology Language

Agents communicate by sending messages, according to the FIPA protocols. The protocols and the communication will be described in detail in Implementation section. In messages, agents use their own language, defined by a *messages ontology.* In the following part, we will focus on the vocabulary this language uses and at the same time it will give us the definitions needed to describe our system.

The *messages ontology* defines *concepts* (basic notions the agents deal

with), the *actions*, that tell the agents what action they should perform, and the *predicates* that tell us something about the state of the world the agents live in. (*Predicates* are used when sending results of *actions*.)

Before we get to the key terms that describe the data mining tasks, we need to define a couple of terms that we will use in further definitions:

Within the scope of *messages ontology*, by an **agent** we understand an agent — a part of the MAS — that encapsulates a data mining method. It can be specified by *type* or by *name*. The parameters of the method are specified by a list of *options*. An **option** is specified by its *name*, *data type*, *value* (which can be set as mutable), *number of arguments*, as the *value* can in fact be a list of values, and a *range* or a *set* of possible values, *default value*, that is used if the user doesn't have any requirement on this option's value, and other optional parameters such as a *description*.

Due to the parameters that are or are not set, an *agent* can be either very specific, or it can be thought of as a pattern of an agent to which the specific agent is later matched.

By **data** we understand two dataset identifiers — one with the *training* dataset and the other with *testing* dataset. When the data are read from a file, they are stored as **data instances**. In this form they can be further processed by *agents*. The general features of a dataset are described by **metadata** — an extra information about the datasets. It contains information about the default type of the task that is connected with the data, number of attributes, their types etc. (*Metadata* will be described in detail in Section 4.3.5 at the end of this chapter).

When talking about the tasks specified by users we distinguish three basic ontology concepts:

- the most specific of them is a **task**, that stands for an *agent-data* couple. At the level of *tasks* the agent is a specific computational unit, that has all its parameters set.

- a **computation** is again a agent-data couple, but some of the agent's parameters don't have the values assigned. A *computation* is divided to *tasks* later on.

- a **problem** is the most general notion of these three, it represents all the information received from the user. It can contain more *agents*, more *data* and a method of choosing the *options*. A *problem* is divided to *computations* later on.

Results represent the output from the system. They contain the *tasks* that were computed together with their results, some general statistic infor-

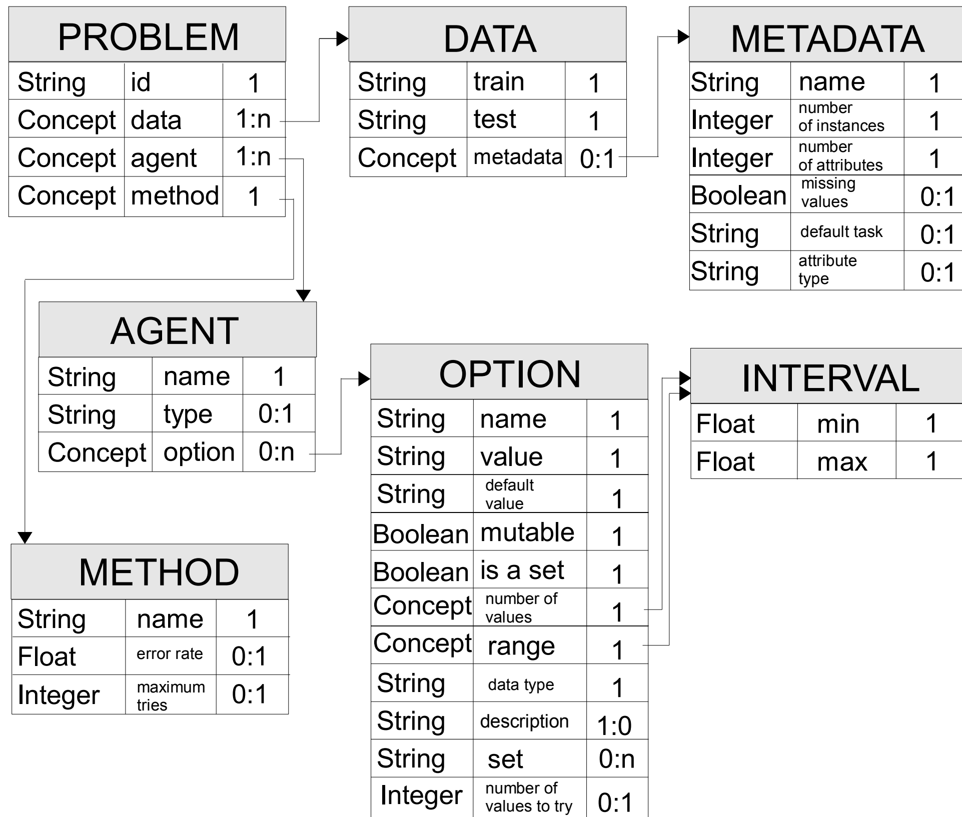

Figure 4.1: The figure shows the *problem* concept and its attributes (some of them are created by another concepts). There are three columns for each concept, telling the type, name and the number of values for each attribute. If the lower range of number of values is 0, the attribute is optional.

mation, and can contain the outputs for the single *data instances*. *Results* notion refers to a *computation*.

4.2 Four Layer Architecture

Now that we have defined the basic notions we can describe the individual parts of Pikater MAS. The whole process that our system goes through when solving a problem, splits into four separate layers, that each have its specific functionality. We distinguish the following layers:

- *user interface layer*
- *computational layer*

- *data layer*
- and *administrative layer*

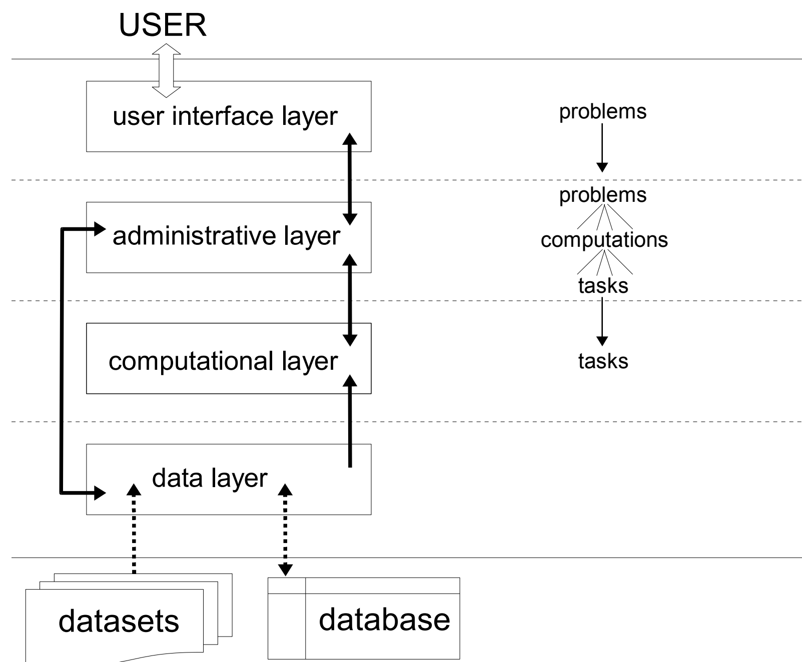

Figure 4.2: The four layer architecture. Communication by sending messages is represented by the solid line arrows.

In the following paragraphs, we are going to describe these four layers in detail:

At the *user interface layer*, the system handles all the communication with user, i.e. all the inputs and outputs. The user needs to define the *problem* in a human understandable language and subsequently communicate the *problem* to the system. As we have said before, the agents communicate with each other using the ontology language, described above. The same language is used for communication with the user, it is only translated to a form suitable for humans. This layer provides means to translate the ontology vocabulary to a human readable form, and vice versa. At this layer, we deal with *problems* (one or more *agents*, that at this level could be very abstract, and one or more datasets), and with *results* that are to be displayed for the user. The *user interface layer* is the only part of the system that is visible for the common user (unless she wants to experiment with her own methods and extend our system using her own code).

At the *computational layer*, we deal with *tasks* and with *data*. This layer contains the set of data mining methods, that all share the same interface so that it is possible for the researchers to add their own methods. They also return the same type of results, so that we can easily compare them.

Data layer contains means to read data from files and return them as *data instances*, and to write/read the results and *metadata* to/from a database.

Lastly, we describe the *administrative layer*, which is the most important and also the most sophisticated part of our system. It carries out several functions:

- It is in charge of the whole problem solving process. It connects the *user interface layer* and the *computational layer*.

- It is capable of giving advise on what *agent type* is the most appropriate according to the given *task*, when the user is not sure what *agent type* to choose.

- It decomposes the *problem* into *computations* (the *agent-data*) couples by choosing the particular agents.

- Consequently, it fills in the missing values to the mutable *options* to create the specific *tasks*. At this time, the system provides two possible ways how to do it. The mutable *options* can be either chosen randomly, or the required number of the values uniformly distributed over the specified interval is selected.

- It processes the results — it gathers the task results together and computes some statistic information.

In the following section, we will describe the process of solving the scenarios described at the beginning of the chapter in terms of the above described layers.

4.3 Solving Problems

All the scenarios introduced at the beginning of this chapter can be described by a *problem*. Whether the problem represents a simple scenario or the most complicated one depends on the number of *agents*, number of mutable *agent options* and number of *data*. As all the scenarios can be described by a problem, the process of solving it is pretty much the same in all the cases, it only differs at the *administrative layer*. We are going to start with the simplest scenario and describe what parts of the system are involved

in solving it. With the remaining three scenarios, we will describe only what happens at the *administrative layer*, because that is what makes the difference.

4.3.1 Scenario 1

The researcher has one or more data *and she knows what* agent *or* agent type *she wants to use.*

At the user interface layer, the user communicates the problem to a system, it is translated to the Agent Communication Language and passed to the *administrative layer*, where the problem is decomposed into *computations* (if there is only one *dataset*, only one *computation* is created). To each *computation* that represents an *agent-data* couple, the particular agent is assigned (if it has not already been chosen by the user by its *name*). In this scenario, there are no mutable options, therefore the *computation* is a *task* at the same time, and it can be passed directly to the *computational layer*. The *testing* and *training data* are read from files using the *data layer* and are sent back to the *computational layer* as *data instances*. The method is run on the *data* and the result is sent back to the *administrative layer*, where it is stored to the database (using the *data layer*) and passed to the *user interface layer* as *results*. At the *user interface layer* the *results* are translated back to the human readable form and displayed.

4.3.2 Scenario 2

The researcher knows what method she wants to use, but wants to try different agent options.

The process of solving the problem differs only at the *administrative layer*. For each *computation* that contains an *agent* with mutable options, a *task* is created by generating the missing *options* values. The *task* is sent to the *computational layer*, and the result is returned to the *administrative layer*. Until the result does not meet the final condition (or the time limit is not reached), the new *options* are generated and the system keeps iterating these two actions. After this process is finished, the results of all *tasks* are gathered, some statistic information are computed, and the *results* are sent to the *user interface layer* as described above.

4.3.3 Scenario 3

The researcher wants to try more agents *with different* options *and she wants to know which one is the best (or better) when running on her* data.

The process is nearly the same as in Scenario 1 or 2 (depending on whether there are some mutable options or not), except for the *administrative layer*, where the problem splits into more *computations*, as there are more *agents* involved. As we have said before, *computation* represents an *agent-data* couple. At the *administrative layer* every *agent* is assigned to all *data* specified in the *problem*, thus creating ($|agents| \cdot |data|$) *computations*. *Computations* are further processed as in the previous cases. Before the *results* are sent to the *user interface layer*, the statistic information are computed so that the user can easily compare the chosen methods.

4.3.4 Scenario 4

The researcher wants the system to give advise on what agent type *to use.*

Again, the only difference from Scenario 1 is at the *administrative layer*, namely in the way the particular *agent* is chosen. When a new *computation* is received, the *metadata* along with the results of the previously computed *tasks* stored in the database are searched (using the *data layer*) to find the similar datasets and to predict what agent would be the best choice to solve the new *computation*. This meta-learning algorithm is described in detail in the following section.

4.3.5 Meta-learning — Searching for the Best Agent

As stated before, Pikater is not only a computational environment, it also involves an intelligent behavior. It can give advise on what agent is good at solving the particular *computation*. Along with the *data* themselves, the system also stores *metadata* (see this section bellow). It also stores the results of *tasks* — every time the *administrative layer* receives all *task* results belonging to one *computation*, it stores them into the database, using the tools from the *data layer*. The system uses these stored information in a meta-learning algorithm to predict the best possible agent for new tasks.

Searching for the best agent to solve the *computation* is done in two steps: First, we need to find the most similar dataset to the dataset specified in the *computation*, and consequently to find the agent that had the best performance on these data in the past.

Along with the data themselves, there are *metadata*, i.e. some extra information about the training data, stored in the database. Some of those information can be obtained directly from the data, some of them need to be set by the user along with the data when defining the problem. *Metadata* contain the following *items*:

- *number of attributes* — number of columns in the data file; attributes specify data characteristics, the last attribute represents the class to be predicted.

- *number of instances* — number of the rows in the data file; each row represents one data record (i.e. comma separated values of attributes).

- *data type* — this attribute relates to all values of all attributes in the dataset. We consider four categories of data types — the values of attributes can be *categorical* (in most cases categories are represented by character strings, but it can also be integers or even real numbers), *integer*, *real* or *multivariate* (different attributes are of different types).

- *default task* — type of a task that is connected with the data, currently the system can solve classification and regression types of tasks. Value of this attribute has to be set by the user.

- *missing values* — indicates whether there are some values missing in the dataset.

To determine the most similar dataset, we compute the distance between the user specified training dataset and all the metadata stored in the database. The distance between the two metadata is defined by a *metadata metric*. If all the metadata *item* values are the same, the distance equals to zero; it is the largest when the two datasets differ the most. The distance is computed as follows:

$$d(m_1, m_2) = \sum_{i=1}^{n} w_i \cdot d_i(m_1[i], m_2[i]) \tag{4.1}$$

where m_1 and m_2 are the two compared metadata, n is the number of *items* in the metadata, i stands for the particular metadata *item*, w_i is the weight for the single metadata *items* and d_i is the distance of the two values of the i *item*. The distance of the two *item* values (d_i function) differs according to the type of the *item*.

In case of the boolean and categorical *item* type, where the value is one of the given set of values, we use the following formula (4.2); missing *item* value is treated as another category, v_1 and v_2 are the particular values of i metadata *item*.

$$d_i(v_1, v_2) = \begin{cases} 0, & \text{if } v_1 = v_2; \\ 1, & \text{otherwise.} \end{cases} \tag{4.2}$$

For numerical attributes, we use the (4.3) formula that maps the distance between two values (v_1 and v_2) of the metadata *item* i to the $\langle 0, 1 \rangle$ interval.

$$d_i(v_1, v_2) = \frac{|v_1 - v_2|}{max(v)_{v \in i} - min(v)_{v \in i}} \tag{4.3}$$

In the first step, we find the file that is the closest to the testing dataset provided by user according to the described metric. Afterwards we search the database for the method that showed the lowest error rate on the selected data. When recommending the best possible method we also provide the agent *options* that were used when the results stored in the database were achieved.

We end this chapter with describing the meta-learning algorithm in a pseudocode:

```
function findTheBestAgent(newMetadata)
  // step 1: choose the nearest file
  metadata = (SELECT * FROM metadata)

  bestDistance = MAXINT
  for m in metadata do
    if distance(newMetadata, m) < bestDistance then
       bestDistance = distance(newMetadata, m)
       nearestMetadata = m

  // step 2: choose the best agent
  nearestFileName = m.getFileName

  SELECT agent FROM results WHERE
      dataFile = nearestFileName
      AND errorRate = minErrorRate

  return agent
```

Figure 4.3: Meta-learning algorithm.

Chapter 5

Implementation

Pikater is implemented in Java programming language. Java is an object-oriented, cross-platform language, that offers a very simple way of joining different pieces of software together just by importing the appropriate packages.

Our system is based on JADE, which is a Java implemented platform that provides tools for handling most of the issues addressed when developing a multi-agent system, while covering the technical details. We connect JADE with Weka and we use its data mining methods.

5.1 Agents

In the previous chapter, we have described the four layer architecture of our system. Each of these layers' functionality is accomplished by one or more agents (see Figure 5.1). In the following paragraphs we are going to describe the agents that form our MAS, their roles in it and their relations.

The central point of our MAS is a *manager* agent, that, along with the *options manager*, represent the administrative layer. As we are aware of the fact that a central agent could be a bottle neck of our system, there is no limitation on the number of *managers* running in our system. *Manager's* main purpose is to be a negotiator between user and the agents that provide the specific computational methods. A *manager* receives a *problem*, which can contain more than one *agent* and more than one *data* and decomposes it to *computations*. For each *computation* it subsequently chooses the particular agent, that is determined to compute the *computation*. If the agent (or at least the *agent type*) has not been specified by the user before, it chooses the *agent* according to its previous experience with the similar *tasks*, as described in the previous chapter. After the computation results are received, they are

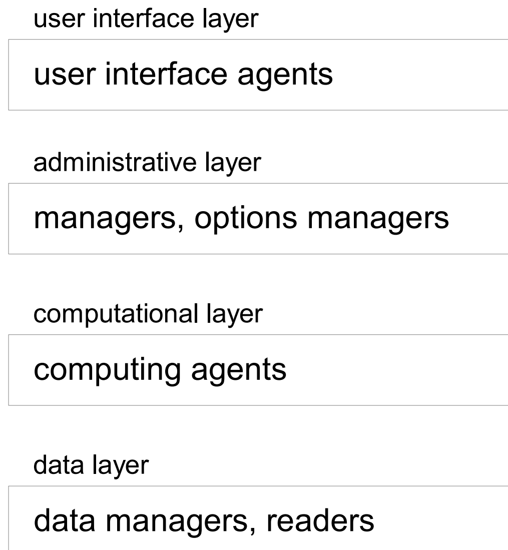

Figure 5.1: Four layer abstract architecture and the agents that create the individual layers.

stored into the database for further purposes. All the *managers* in the system use the same database, so that they all can share the stored results.

Options manager can be thought of as an extension of the *manager* agent. It receives the particular *agent* specified by its *name* and generates the values for each *option* that the user has chosen to be mutable. At this time, there are two types of *option managers* implemented - one of them chooses the values randomly (*Random agent*), the other one (*ChooseXValues agent*) chooses the user specified number of values uniformly distributed over the *interval*. When specifying a *problem* the user can choose which of these methods should be used and specify its parameters — for the random method, the parameters are the threshold error rate and maximum number of tries, when the threshold in not reached. Number of values to try can be specified individually for each *option* when the ChooseXValues method is chosen.

By generating all the missing option values, the *option manager* creates *tasks* and keeps sending them to the specified *computing agent*, until the final condition that indicates that the processing of the computation is over, is met. The final condition can be getting under the specified error rate, reaching the maximum number of tries, or testing out all the element of a *set*. If the agent is unreachable for some reason, or busy, *options manager* keeps trying until the timeout expires.

The **user interface layer** is represented by an abstract GUI agent type. There are three types of *user interface (UI) agents*, each of them can take all the required action. They are designed to receive *problems* from a user and to display the results. The system thus provides three different ways of entering user's requirements. Setting a *problem* can be done by specifying the problem in a form of a XML file, by using a graphical user interface, or by using a web interface, that connects the user with the system running on a remote machine. In case of the two agents that have a graphical interface, the problems are entered and results displayed in these interfaces, agent options are entered in Weka format (a character string of option name-value couples). In the third case results are output on the console. All the *results* are also saved in the ExpML format, co that they can be further processed.

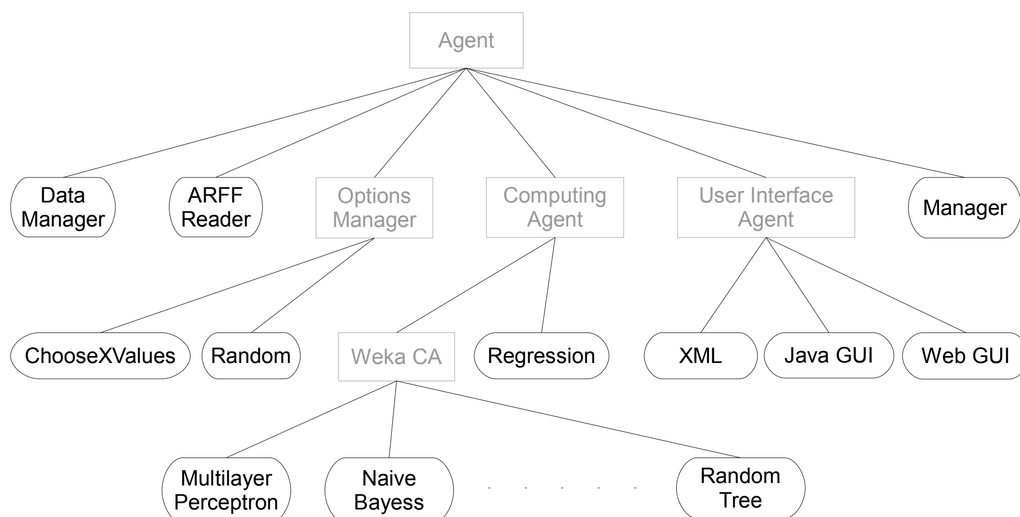

Figure 5.2: Agent hierarchy. Abstract agents are in gray.

The **computational layer** is created by many different types of *computing agents (CA)*, each encapsulating a data mining method. We adapted several agents from Weka, and present one simple example of our own agent. All these agents share the same interface — they can train and test themselves on the given *data*, and return the results in the same form, so that they can be compared to each other. At this layer, agents deal with *tasks*. A *computing agent* receives *options*, and *data*, i.e. the names of files with training and testing data. It sends a request to a *reader* agent, that reads the data from a specified file and sends it back to the *computing agent* in the form of *data instances*. *CA* than trains itself on the received *data instances*,

and returns the results, which can be the error rate (and other optional values like kappa statistic, or root mean squared error), or the output values for the given data. For each *CA* type there is a configuration file containing the options, their type, number of values and the interval from which the values should be chosen (or a set of values).

A functionality of the **data layer** is divided between two types of agents — *reader* agents and *data manager* agents. *Reader agents* read the data from files, and send them back as *data instances*. The system contains at least one, but usually more *reader* agents. Currently there is only one type of a *reader*, that is capable of reading *Attribute-Relation File Format (ARFF)* files — a data format used by Weka. ARFF is an ASCII text file that describes a list of instances sharing a set of attributes. At the beginning of the file, there is the header information, that can contain name of the relation, a list of the attributes (the columns in the data), and their types. The header is followed by the data information — a comma separated list of attribute values, each row representing one instance [10]. We provide the example of a header and several instances from probably the most frequently cited ARFF file distributed with Weka:

```
@RELATION iris

@ATTRIBUTE sepallength  REAL
@ATTRIBUTE sepalwidth   REAL
@ATTRIBUTE petallength  REAL
@ATTRIBUTE petalwidth   REAL
@ATTRIBUTE class  {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
6.5,2.8,4.6,1.5,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
6.0,3.4,4.5,1.6,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

Figure 5.3: Part of the iris.arff file.

Data manager agent manages the communication with the database, using Structured Query Language (SQL). It also takes charge of storing new data files to the system.

The agents create a hierarchical structure, that is depicted in Figure 5.2.

We developed Pikater as an open system and we expect researches to write their own agents, namely *option managers*, *computing agents*, *readers* and *user interface agents*, to extend its functionality.

5.2 Solving a Problem

To make things even more clear we give an example how a *problem* is being processed:

- When a *problem* is being specified, *UI agent* asks selected *computing agents* for their options, so that it can display their description and default value to the user.

- *UI agent* receives the problem from the user (either in XML format, or as information retrieved from the graphical user interface).

- *UI agent* passes the problem on to a *manager*, it then waits for the final results as well as for the partial results, that are sent whenever a *computation* is computed.

- *manager* receives the *problem*, decomposes it to *computations* and select an appropriate *computing agent* for each computation. Each time there is a *computation* to compute, a *manager* creates a special *option manager* just for this *computation*, and the *computations* are passed on to him.

- *options manager* generates the missing *option* values for the given agent. Now that a *task* is created, it is sent to the *computing agent.*

- The *computing agent* asks a *reader* for *data instances* and computes the given *task.* It sends the results back to the *options manager.*

- The *option manager* keeps generating new *options* and iterating with the *computing agent* until the final condition is accomplished.

- The *option manager* sends the results back to the *manager.*

- When the *manager* receives the results of a *computation* it sends it straight to the *UI* agent, so that the user gets the partial results before they get the final ones. After the *computation* is resolved, the *options manager* is killed by the *manager*. The manager also inserts the results to the database.

- After all the *computations* belonging to the initial *problem* have been received, the *manager* computes results statistics, stores the *results* in an ExpML file, and requests the *data manager* to write *tasks* results into the database. The *UI agent* is informed that there will be no more results coming.

The agents' communication and relations are shown in Figure 5.4.

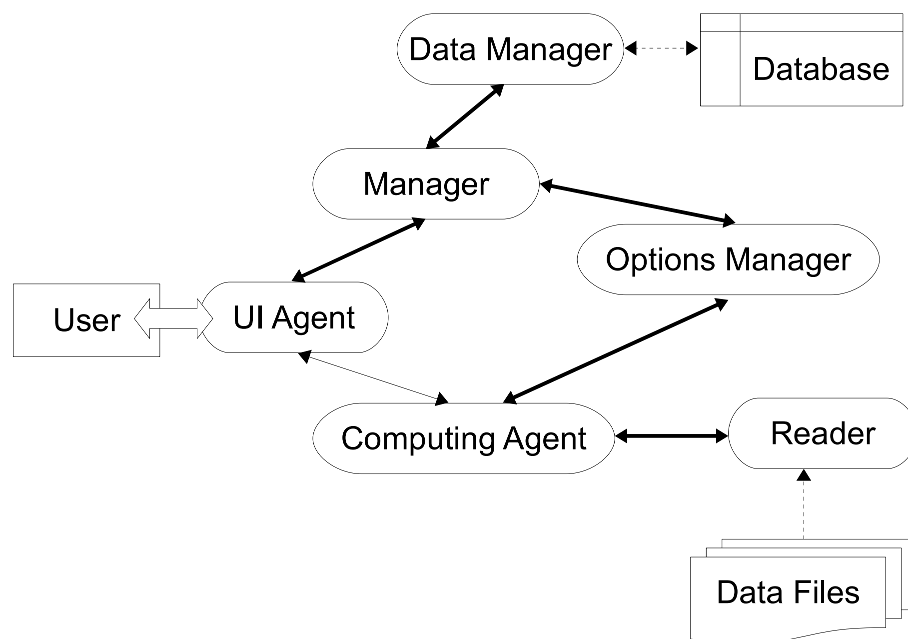

Figure 5.4: The picture shows the agent types and the data exchange among them. The solid line stands for the communication via messages, the dotted for reading data from file/communication with database. The main communication route is stressed by the bold line.

5.3 Agent Behavior and Communication

Now that we have described the agent types that are present in our system, we are going to have a closer look at what actually is an agent. In the previous Chapter, we have defined an *agent* as an ontology concept representing a data mining method. In the following section agent is defined more widely, according to the definition used by JADE. Agent is a piece of software, each represented by a its own Java class. Each agent has its type, which is named when the agent registers with (DF) (the yellow pages JADE component), so that it can be found by other agents.

As we have said before, agents communicate by sending messages. A message is a set of parameters written in FIPA ACL, (the FIPA message structure is described in chapter 3). The content of a message is formed by ontology *action* or *predicate*. In our system, we use the Messages Ontology Language.

While the agents are *autonomous*, sending an agent a message is the only way how to make it perform an action. The way the agent sends messages to initiate conversations and how it responds to the incoming messages representing different communicative acts, as well as the other actions agent takes, is described by *behaviors*. Agent can implement more than one behavior, in that case, behaviors can be processed in parallel. It means that the agents can for example respond to more than one message at the same time. The behaviors range from the simple ones like one-shot or cyclic behavior to the more complex ones, where the behavior is represented by a final state machine or is described by a FIPA Interaction Protocol. The behaviors can be combined and nested to create even more complex behaviors.

Typically, to achieve the requested effect, we need the agents to exchange more than one message in each direction. The most frequently used conversation schemes are described by FIPA interaction protocols, that define what message should be sent in the particular situation and how to handle the incoming messages. In the interaction protocols, there are always two parties involved — there is always one Initiator agent, who initiates the conversation by sending a message and one or more Responders. Each agent can play different roles in different conversation.

We took advantage of the fact that JADE implements all FIPA Interaction Protocols. In the next paragraphs, we will describe the behaviors, mostly represented by FIPA Interaction Protocols, that we use in our agents.

FIPA Request protocol is an example of a simple behavior. It describes the conversation scheme for request-reply communicative act. The protocol involves an Initiator and one or more Responders. Initiator sends a message (i.e. it performs a communicative act) in which it asks one or more Re-

sponders to provide some information. The Responder either sends a *result notification* straight away or it sends a *response* first, stating whether or not it is willing to continue in the conversation. The *response* contains one of the following communicative acts: agree, not-understood or refuse. *Result notification* is either *inform* message containing the requested information, or a *failure* message, if anything went wrong.

Another protocol we use is Iterated Request protocol. [1] It is a modification of the FIPA Request protocol, where the request-reply acts are iterated until the requested effect is achieved. After that the behavior is canceled by a Cancel communicative act.

In FIPA Subscribe Interaction Protocol the initiator sends a subscription message to the responder. The responder can reply by sending a *response* (not-understood, a refuse or an agree message) to communicate whether the subscription has been agreed. Each time the condition indicated in the subscription message is fulfilled, the responder sends a notification messages to the initiator.

Most of the conversations held between agents in our system matches the FIPA Request Interaction Protocol, except for the *options manager*-*computing agent options*-*result* exchange, that follows the Iterated Request protocol, the sending of *computation* results form *manager* to an UI agent which is due to the Subscription protocol. The behavior of *computing agents* is represented by a final state machine, however the single states are handled by FIPA Request protocol.

The scheme of agents' communication is depicted in the Figure 5.5.

5.4 Ontologies

In Chapter 3 we have defined the most important vocabulary of the Agent Language. Each notion, which can be a concept, an action or a predicate, is represented by a Java class. The attributes can be of different types, it can also be another concepts. To give an example of a complex concept, we show the *agent* concept described in the previous chapter again as it is sent in a message:

Note that the above example is simplified, there are in fact more than two *options*, and some of the less important *option* attributes have been removed to make the example more transparent.

JADE provides the tools for verifying the correctness of the ontologies sent as message content, checking it and comparing it with the specified ontology schemes.

[1] Iterated Request protocol is not a part of FIPA specification.

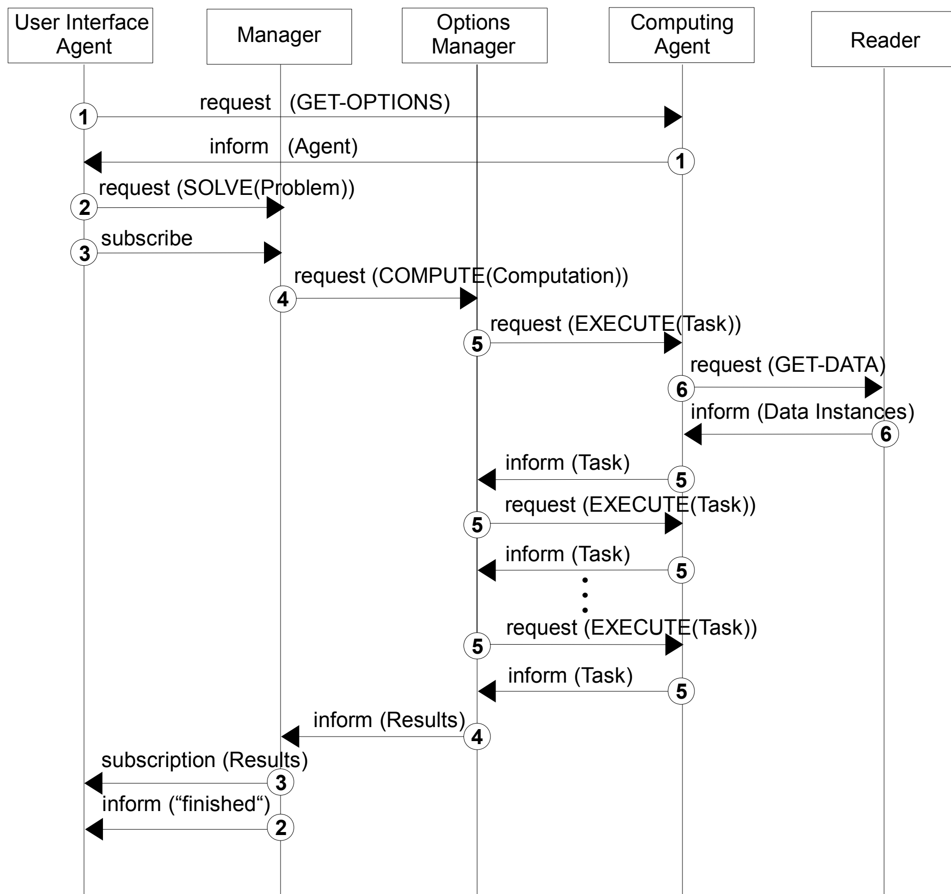

Figure 5.5: The most important messages flow in Pikater system. Conversations 1,2,4 and 6 match the FIPA Request Interaction Protocol, conversation 2 matches FIPA Subscribe Interaction Protocol and conversation 5 the Iterated Request protocol.

```
(AGENT
 :name MultilayerPerceptron0
 :type MultilayerPerceptron
 :options
   (sequence
     (OPTION
       :mutable false
       :range
         (INTERVAL
           :min 0.0
           :max 1.0)
       :is_a_set false
       :number_of_args
         (INTERVAL
           :min 1.0
           :max 1.0)
       :data_type FLOAT
       :description " Learning Rate for the
                      backpropagation algorithm."
       :name L
       :value "0.2"
       :default_value "0.3")
     (OPTION
       :mutable true
       :range
         (INTERVAL
           :min 0.0
           :max 0.0)
       :set (sequence "3" "4" "5" i o)
       :is_a_set true
       :number_of_args
         (INTERVAL
           :min 1.0
           :max 3.0)
       :data_type MIXED
       :description " The hidden layers to be
                      created for the network."
       :name H
       :value "?,3"
       :default_value a)))
```

Figure 5.6: Agent ontology as sent in a message.

Chapter 6

Experiments - Showing the Software

In this chapter we present several experiments to demonstrate the functionality of our system. We start with solving the simple problems, and we subsequently get to the more advanced features of our system.

6.1 Experiment 1 — Setting the Parameters

In the first experiment, we intend to show different user interfaces. We will show how the experiment is entered to the system and also show the output that the system provides.

We have chosen a simple problem of running the Weka multilayer perceptron network classifier on the well-known iris dataset(e.g. [1]), used as both training and testing dataset. We want to use a network with two hidden layers, first containing 5 perceptrons, the second one 3 perceptrons, and we want to set the learning rate to 0.4.

There are three different ways of entering the problem to our system — XML experiment configuration file, local Java graphical user interface and the remote web graphical user interface. The way of setting an experiment in both local and remote graphical user interfaces is basically the same, therefore we are going to describe just two ways of setting the problem. Using the graphical user interface presents the more convenient way of setting the experiment, it also enables to run more experiments in a row, as it is interactive. The XML format, on the other hand, provides more options and is suitable for advanced experiments.

Problem settings

First, we show the above described experiment in the XML format:

```
<experiment>

  <dataset train="iris.arff" test="iris.arff" >

  <agent type="MultilayerPerceptron">
    <parameter name="L" value="0.4" />
    <parameter name="H" value="5,3" />
  </agent>

</experiment>
```

When using the GUI, most of the experiment settings are chosen by filling a form (some parameters can be selected from the option lists), the agent's options are set as follows:

```
MultilayerPerceptron -L 0.4 -H 5,3
```

All the classifier options need not be set. When the option value is not specified, default values are used.

Results

Result of the experiment are the statistic information provided by Weka, that are displayed on the command line or in the GUI. Results are also stored in a ExpML-like format.

For each *task* we get the following statistic information:

method	MultilayerPerceptron
options	-L 0.4 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H 5, 3
error rate	0.020
kappa statistic	0.970
mean absolute error	0.027
root mean squared error	0.122
relative absolute error	6.109
root relative squared error	25.836

Table 6.1: Results for multilayer perceptron network with "-L 0.4 -H 5,3" options and iris dataset.

The experiment results in ExpXL-like format:


```
<result>
  <experiment>
    <setting>
      <algorithm name="MultilayerPerceptron1" libname="weka">
        <parameter name="L" value="0.4" />
        <parameter name="M" value="0.2" />
        <parameter name="N" value="500" />
        <parameter name="V" value="0" />
        <parameter name="S" value="0" />
        <parameter name="E" value="20" />
        <parameter name="D" value="False" />
        <parameter name="I" value="False" />
        <parameter name="G" value="False" />
        <parameter name="H" value="5,3" />
        <parameter name="B" value="False" />
        <parameter name="C" value="False" />
      </algorithm>
      <dataset train="data\files\iris.arff" test="data\files\iris.arff" />
    </setting>
    <evaluation>
      <metric error_rate="0.019999999552965164" />
      <metric kappa_statistic="0.9700000286102295" />
      <metric mean_absolute_error="0.027149345725774765" />
      <metric root_mean_squared_error="0.1217924952507019" />
      <metric relative_absolute_error="6.108603000640869" />
      <metric root_relative_squared_error="25.836090087890625" />
    </evaluation>
  </experiment>
  <statistics>
    <metric average_error_rate="0.019999999552965164" />
    <metric average_kappa_statistic="0.9700000286102295" />
    <metric average_mean_absolute_error="0.027149345725774765" />
    <metric average_root_mean_squared_error="0.121792495250719" />
    <metric average_relative_absolute_error="6.108603000640869" />
    <metric average_root_relative_squared_error="25.8360900875" />
  </statistics>
</result>
```

Note that all classifier options and their values are displayed and stored, also the average values are computed (computation can comprise of more tasks).

All results are also stored in the database and can be further processes using SQL queries.

6.2 Experiment 2 — Choosing the Best Options

In the next experiment we want to show how the system handles the mutable options set by user. We let the system choose some of the options for the Weka multilayer perceptron model. We want to choose the values for each mutable option and try all the possible combinations.

In this experiment we use again the Weka multilayer perceptron classifier and the iris dataset, we want the system to choose the values of the following options:

- *L — learning rate*
- *M — momentum rate for the backpropagation algorithm*
- *D — boolean option telling whether the learning rate decay will occur*
- *H — the hidden layers to be created for the network, value of this argument is a list of comma separated numbers or the letters 'a' = (attributes + classes) / 2, 'i' = attributes, 'o' = classes, 't' = attributes + classes) — this is a feature provided by Weka.*

In this example, we want to train and test a two layer perceptron network, that has mutable number of perceptrons in the first hidden layer, and 3 perceptrons in the second layer. The values for the first layer are chosen from the set — 2, 3, or i perceptrons, where i is number of attributes in the given dataset, which is 5 for the iris dataset. For each mutable option we want to try three different values.

To generate all the combinations of the values possible, we use the ChooseXValues method. The *number of values to try* parameter can be set individually for each option (if it is not set, the default value specified in the user interface agent is used). If we specify the values by a set, all the named values are tested, independent on the *number of values to try* parameter. If there are less than *number of values to try* values (i.e. the option is boolean, or integer and the range is not wide enough), there are less than *number of values to try* chosen.

Problem settings

The transcription from the verbal description to XML format is straightforward:

```
<experiment>

  <method name="ChooseXValues" />
  <dataset train="iris.arff" test="iris.arff" ></dataset>

  <agent type="MultilayerPerceptron">
    <parameter name="L" value="?" number_of_values_to_try="3"/>
    <parameter name="M" value="?" />
    <parameter name="D" value="?" />
    <parameter name="H" value="?,3" set="3,4,i"/>
  </agent>

</experiment>
```

Results

In this experiment, we had one dataset and one agent, which created one computation. The computation splits into 54 tasks (3 possible values for L, M, and H options and 2 for boolean D option). In the Table 6.2. we present some of the best and some of the worst results.

Interpretation

Several option settings provided comparable results, we can see that adding more perceptrons to the first hidden layer or changing the learning rate option does not significantly improve the performance in this particular case.

Note that XML format offers more possibilities when defining the experiment, we can specify a set or a range from which the values are chosen.

options	*error rate*	*kappa statistic*	*mean absolute error*	*root mean squared error*	*relative absolute error*	*root relative squared error*
-L 0.0 -M 0.45 -H 3,3	0.027	0.960	0.034	0.140	7.595	29.757
-L 0.5 -M 0.9 -D -H 3,3	0.027	0.960	0.064	0.135	14.506	28.661
-L 1.0 -M 0.9 -D -H 4,3	0.027	0.960	0.045	0.128	10.119	27.171
-L 1.0 -M 0.9 -D -H i,3	0.027	0.960	0.042	0.123	9.480	26.140
-L 0.0 -M 0.0 -H 4,3	0.033	0.950	0.035	0.144	7.948	30.465
-L 0.0 -M 0.45 -H 4,3	0.033	0.950	0.028	0.130	6.318	27.630
	...					
-L 1.0 -M 0.9 -H i,3	0.153	0.770	0.112	0.280	25.183	59.497
-L 1.0 -M 0.45 -D -H 4,3	0.160	0.760	0.277	0.334	62.312	70.870
-L 1.0 -M 0.9 -H 3,3	0.187	0.720	0.131	0.312	29.499	66.150
-L 1.0 -M 0.45 -D -H 3,3	0.233	0.650	0.290	0.346	65.311	73.484
-L 0.5 -M 0.45 -D -H 4,3	0.627	0.060	0.444	0.471	99.996	99.997
	...					
-L 0.5 -M 0.0 -D -H 3,3	0.667	0.000	0.444	0.471	100.000	100.000
-L 0.5 -M 0.0 -D -H 4,3	0.667	0.000	0.444	0.471	100.000	100.000
-L 0.5 -M 0.0 -D -H i,3	0.667	0.000	0.444	0.471	100.000	100.000
-L 0.5 -M 0.45 -D -H 3,3	0.667	0.000	0.444	0.471	100.000	100.000
-L 1.0 -M 0.0 -D -H 4,3	0.667	0.000	0.444	0.471	99.987	99.987

Table 6.2: Some of the best and some of the worst results for multilayer perceptron network with two hidden layers. Mutable options were: learning rate (L), momentum rate for the backpropagation algorithm (M), learning rate decay (D), and the number of perceptrons in the second hidden layer.

6.3 Experiment 3 — Choosing the Best Agent

In the last experiment we want to show how the choosing of the best computing agent works.

We used the contact-lenses dataset [22] (that is used for predicting what kind of the contact lenses the subject should wear) that we have never used before, so that there are no data related to this problem in the database. We let the system decide what agent and options to use. Afterwards we let all agent types solve the given task, to prove whether the best possible method have actually been chosen.

To generate the data for meta-learning algorithm we used 12 different datasets (see Table ...). Running different data mining methods on these datasets produced over 5000 different task results, that have been stored in the database.

Following datasets were used to generate the data to make the prediction:

file name	*task*	*data*	*instances*	*attributes*	*missing*
car.arff	C	Cat	1728	7	F
magic.arff	C	Real	19020	11	F
iris.arff	C	Real	150	5	F
letter-recog.arff	C	Int	20000	17	F
tic-tac-toe.arff	C	Cat	958	10	F
weather.arff	C	Mult	14	5	F
machine.arff	R	Mult	209	10	F
haberman.arff	C	Int	306	4	F
communities.arff	R	Real	1994	128	T
lung-cancer.arff	C	Int	32	57	T

Table 6.3: Training data for choosing the closest dataset. C - Classification, R - Regression; Cat - Categorical, Int - Integer, Mult - Multivariate; T - True, F - False

Problem settings

Because we haven't used the contact-lenses dataset before, the metadata for this dataset had not been stored in the database, so we had to provide them when setting the problem. The rest of the XML file is very simple:

```
<experiment>

  <dataset train="contact-lenses.arff" test="contact-lenses.arff" >
    <metadata
        missing_values="False"
        number_of_attributes="5"
        number_of_instances="24"
        attribute_type="Categorical"
        default_task="Classification"
        />
  </dataset>

  <agent type="?">
  </agent>

</experiment>
```

Results

Using the metric defined in Chapter 4, the tic-tac-toe dataset was chosen as the nearest.

file name	*task*	*data*	*instances*	*attribs*	*missing*	*distance*
car.arff	C	Cat	1728	7	F	0.101
magic.arff	C	Real	19020	11	F	1.999
iris.arff	C	Real	150	5	F	1.006
letter-recog.arff	C	Int	20000	17	F	2.096
tic-tac-toe.arff	**C**	**Cat**	**958**	**10**	**F**	**0.087**
weather.arff	C	Mult	14	5	F	1.001
machine.arff	R	Mult	209	10	F	2.050
haberman.arff	C	Int	306	4	F	1.022
communities.arff	R	Real	1994	128	T	4.091
lung-cancer.arff	C	Int	32	57	T	2.420
contact-lenses.arff	**C**	**Cat**	**24**	**5**	**F**	**0.000**

Table 6.4: Training data for choosing the closest dataset, including the distance from contact-lenses dataset. The tic-tac-toe dataset have been chosen as the nearest neighbor. C - Classification, R - Regression; Cat - Categorical, Int - Integer, Mult - Multivariate; T - True, F - False

In the next step the system searches the database for the best results on the tic-tac-toe dataset. The Weka PART method with "-M 1 -U" parameters have been chosen.

Method PART uses separate-and-conquer approach to create a decision list. It builds a partial C4.5 decision tree in each iteration and makes the "best" leaf into a rule. For more information, see [2] (U — the unpruned tree, M — the minimum number of instances per rule). The selected method returned zero error rate on tic-tac-toe dataset and it also happened to return zero error rate on the contact-lenses file.

Results of the PART method run on the contact-lenses dataset:

method	PART
options	-M 1 -U
error rate	0.0
kappa statistic	1.0
mean absolute error	0.0
root mean squared error	0.0
relative absolute error	0.0
root relative squared error	0.0

Table 6.5: Results for PART method with "-M 1 -U" options and contact-lenses dataset.

Afterwards we run all the methods implemented in our system on the contact-lenses dataset. There were several other methods, that returned zero error rate, e.g. Random Tree method, or NNge — Nearest-neighbor-like algorithm using non-nested generalized exemplars.

Interpretation:

In this example, we have demonstrated the meta-learning algorithm. The system chose the method that returned zero error rate on the never-before-tested dataset.

Chapter 7

Conclusion

In the thesis we have introduced the multi-agent system designed for conducting experiments as well as for experimenting with researchers' own data mining methods. The system shows intelligent behavior, as it is capable of recommending the best possible data mining method to process the never-before-seen dataset according to its previous experience.

We have analyzed the most frequently used scenarios that the researchers follow, when solving their tasks, and created an abstract *four layer architecture*. Later we have suggested the implementation of the system, where each layer's functionality is represented by particular agent. Finally we have presented several experiments to show the features of our system.

Agents communicate by sending messages, the conversation schemes match the standard FIPA protocols. Content of the messages is specified by the *messages ontology*. This opens our system to other FIPA compliant agents as well as the world of semantically annotated web services.

We have described the meta-learning algorithm for choosing the best agent possible. The algorithm uses the *metadata metric*, that determines the distance between two datasets according to the general information on these datasets.

In the future work we would like to focus on further development of the intelligent behavior of the system, namely on improving the meta-learning algorithm. Both steps of the meta-learning algorithm could be improved in several ways — in the current implementation, when storing the results, we don't consider the way the method was tested. We should involve the testing dataset as well, as the error rate can differ a lot in dependency on the dataset the model was tested on.

There are more parameters describing a result stored in the database, so far we have consider only the error rate parameter. We could consider the other stored information and define the results metric to tell more precisely

what method is better than the other. One of the parameters included in the metric would be the complexity of the chosen model — the time that was needed to train the given model would be stored to the database and taken into consideration.

Another improvement concerns choosing the most similar dataset. In the current version, we pick just one dataset. If we would train the system using larger dataset base, we expect the datasets to create clusters. Consequently, we would choose among all the methods from the selected cluster.

We would also like to improve the user-friendliness of the graphical user environment, and include more tools for conducting the experiments, such as tools for visualizing data and the results, displaying the partial results while the computations are still running or saving and loading trained agents.

Bibliography

[1] Fisher R.A. (1936) *The use of multiple measurements in taxonomic problems.* Annual Eugenics, 7, Part II, 179-188.

[2] Frank E., Witten I. H. (1998) *Generating Accurate Rule Sets Without Global Optimization.* In: Fifteenth International Conference on Machine Learning, 144-151.

[3] Brent M. (1995) *Instance-Based learning: Nearest Neighbor With Generalization.* Hamilton, New Zealand.

[4] Russell, Stuart J., Norvig P. (1995) *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

[5] Wooldridge M., Jennings N. (1995) *Intelligent agents: theory and practice.* Knowledge Engineering Review 10 (2), 115–152.

[6] SAS Enterprise Miner website.
http://www.sas.com/technologies/analytics/datamining/miner

[7] SPSS Clementine website. http://www.spss.com

[8] STATISTICA Data Miner. http://www.statsoft.com

[9] Hall M., Frank E., Holmes G., Pfahringer B., Reutemann P., Witten I.H. (2009) The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.

[10] http://www.cs.waikato.ac.nz/ml/weka

[11] Orange website. http://www.ailab.si/orange

[12] Vanschoren J. (2008) *ExpML Description.*
http://expdb.cs.kuleuven.be/expdb/expml.php

[13] Yahoo! website: http://www.yahoo.com

[14] Amazon website: http://www.amazon.com

[15] FIPA website: http://www.fipa.org

[16] *FIPA37. Specification SC00037, FIPA Communicative Act Library Specification.*

[17] Searle J. (1969) *Speech Acts.* Cambridge, MA, Cambridge University Press.

[18] *FIPA26. Specification SC00026, FIPA Request Interaction Protocol Specification.* http://www.fipa.org

[19] *FIPA27. Specification SC00027, FIPA Query Interaction Protocol Specification.* http://www.fipa.org

[20] *FIPA29. Specification SC00029, FIPA Contract Net Interaction Protocol Specification.* http://www.fipa.org

[21] Bellifemine F., Giovanni C., Greenwood D. (2007) *Developing Multi-Agent Systems with JADE.* John Wiley and Sons, Chichester, UK.

[22] Cendrowska, J. (1987) *PRISM: An algorithm for inducing modular rules.* International Journal of Man-Machine Studies, 27, 349-370.

[23] Wooldridge M. (2002) *An Introduction to Multiagent Systems.* John Wiley and Sons Ltd.

[24] M. K. Smith, C. Welty, D. McGuinness: *Web Ontology Language (OWL) Guide, World Wide Web Consortium.* http://www.w3.org/TR/2004/REC-owl-guide-20040210/

List of Figures

2.1 Agent Communication Language. 13
2.2 Example of an ALC message. 16

4.1 The figure shows the *problem* concept and its attributes (some of them are created by another concepts). There are three columns for each concept, telling the type, name and the number of values for each attribute. If the lower range of number of values is 0, the attribute is optional. 26
4.2 The four layer architecture. Communication by sending messages is represented by the solid line arrows. 27
4.3 Meta-learning algorithm. 32

5.1 Four layer abstract architecture and the agents that create the individual layers. 34
5.2 Agent hierarchy. Abstract agents are in gray. 35
5.3 Part of the iris.arff file. 36
5.4 The picture shows the agent types and the data exchange among them. The solid line stands for the communication via messages, the dotted for reading data from file/communication with database. The main communication route is stressed by the bold line. 38
5.5 The most important messages flow in Pikater system. Conversations 1,2,4 and 6 match the FIPA Request Interaction Protocol, conversation 2 matches FIPA Subscribe Interaction Protocol and conversation 5 the Iterated Request protocol. . . 41
5.6 Agent ontology as sent in a message. 42

List of Tables

2.1 Communicative acts. 14

6.1 Results for multilayer perceptron network with “-L 0.4 -H 5,3” options and iris dataset. 44
6.2 Some of the best and some of the worst results for multilayer perceptron network with two hidden layers. Mutable options were: learning rate (L), momentum rate for the backpropagation algorithm (M), learning rate decay (D), and the number of perceptrons in the second hidden layer. 49
6.3 Training data for choosing the closest dataset. C - Classification, R - Regression; Cat - Categorical, Int - Integer, Mult - Multivariate; T - True, F - False 50
6.4 Training data for choosing the closest dataset, including the distance from contact-lenses dataset. The tic-tac-toe dataset have been chosen as the nearest neighbor. C - Classification, R - Regression; Cat - Categorical, Int - Integer, Mult - Multivariate; T - True, F - False 51
6.5 Results for PART method with “-M 1 -U” options and contact-lenses dataset. 52