

Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
User Guide – Draft

The Mining Mart User Guide

Timm Euler, Detlef Geppert, Olaf Rem, Martin Scholz

Dortmund, July 19, 2004

Contents

1	The Philosophy of MiningMart	5
1.1	The MiningMart approach	6
1.2	Basic notions in MiningMart	7
2	Installing the MiningMart system	13
2.1	General issues and downloading	13
2.2	Preparing the database	14
2.3	Configuring the system	15
2.4	Plugins	15
2.5	Upgrading from older versions	16
2.6	Remarks	17
2.7	Appendix: List of operators that use external algorithms	17
3	The Human Computer Interface	19
3.1	Introduction	19
3.2	Main Application	20
3.2.1	Getting started	21
3.2.2	Main functionality	23
3.2.3	Closing the application	32
3.3	The Concept Editor	32
3.3.1	Using the Concept Editor	33
3.4	The Chain Editor	38
3.4.1	Overview of Functionality	38
3.4.2	Inserting a chain	39
3.4.3	Inserting a step	39
3.4.4	Changing properties	41
3.4.5	Editing the step parameters	41
3.4.6	Changing Positions	44
3.4.7	Selecting objects in the graph view	44
3.4.8	Deleting objects	44
3.4.9	Connecting steps	45
3.4.10	Merge steps to a chain	45
3.4.11	Unmerge sub chains	46
3.4.12	Cut, Copy, Paste	46

4	Compiler Constraints and Operator Parameters	47
4.1	What this chapter is about	47
4.2	Compiler constraints on metadata	47
4.2.1	Naming conventions	47
4.2.2	Relations	48
4.3	Operators and their parameters	48
4.3.1	General issues	49
4.3.2	Concept operators	50
4.3.3	Feature selection operators	60
4.3.4	Feature construction operators	65
4.3.5	Other Operators	80
5	The Case Repository	83
5.1	The Internet Presentation of Cases	83
5.2	How to download a case	84
5.3	How to document a case	85
5.4	How to upload a case	85

Chapter 1

The Philosophy of MiningMart

In this chapter you will learn about the basic ideas behind MiningMart. Its different components and the way they interact will be explained. Basic notions that will be needed for any MiningMart session are presented. This will also help you to understand this document and any other documents related to MiningMart.

MiningMart is a system that supports the development, documentation and re-use of results in knowledge discovery. It is assumed that you are familiar with general concepts in Knowledge Discovery (Data Mining). However, we give a few informal definitions here to provide a common understanding. More information about Data Mining can be found on the MiningMart webpages:

<http://mmart.cs.uni-dortmund.de>

- The *Knowledge Discovery Process* refers to the technical steps of data acquisition, data cleaning, data preparation as well as data mining and model testing.
- *Data Mining* is the step in the knowledge discovery process where a Machine Learning algorithm is applied to learn a model which is used to make predictions on new data.
- *Preprocessing* comprises all steps that are undertaken in order to bring the data into a format that is accessible for data mining. The result of preprocessing is the input for data mining without any further modifications. The input for preprocessing is the data as it is stored in a data warehouse or even the operational database of an institution.

Section 1.1 gives an overview of the MiningMart approach to the knowledge discovery process. In section 1.2, basic terms that are used in MiningMart are defined and explained. Those terms will be used everywhere in the MiningMart system and documentation, so it is a good idea to familiarize yourself with them.

1.1 The MiningMart approach

MiningMart provides support for knowledge discovery applications. Thus the system is aimed at those people in an institution who actually work with the institution's data and process it in various ways in order to gather statistics or other higher-level information. While the system provides an intuitive access to data and easy handling of processing steps, users should have a certain knowledge about how their data is stored *before* the application of MiningMart.

MiningMart works with relational databases. It assumes that all input data is given in tables in a relational database and its output are new tables in this database. It also stores its own data in relational tables. Thus, there are no limitations to the amount of data that MiningMart can handle.

Referring to the definitions at the beginning of this chapter, MiningMart supports the whole knowledge discovery process but focusses clearly on pre-processing. That is, the system provides a few common data mining algorithms which can be applied directly from the system, but its main value is the support for the technical steps that are needed to bring the data into a format which can be used for data mining. Like the input, the output of the system is a number of relational database tables, but in the output tables the data is stored in a representation suitable for data mining. Thus, you can use your favourite data mining algorithm easily because the input data for it is stored in a table in your database in exactly the right format after the application of MiningMart.

MiningMart supports preprocessing by applying a number of data processing steps to its input. Each step is graphically represented in the MiningMart workspace. The complete sequence of steps is stored in the database and can also be exported to other sites where MiningMart is in use. In this way, a documentation of the whole knowledge discovery process is achieved. All the details of a discovery process can be easily saved for later usage, can be modified using a graphical user interface, and can be transferred from one discovery process to another.

MiningMart uses a layer of abstraction of the actual data to model the knowledge discovery process. This abstraction allows to publish successful discovery applications for the benefit of other users, while sensitive details are hidden. This means that you can benefit easily from the work done by other MiningMart users. The MiningMart web pages provide a central platform for the exchange of successful discovery processes, called *cases* (see section 1.2). On this platform, such cases are described both in terms of their relevance to a business and in technical terms, which allows you to find cases which are similar to the application you have in mind. You can then download such cases into your MiningMart system and make the necessary modifications towards your own data.

The following section describes these central ideas in more detail by explaining the basic MiningMart terminology. Once you have become familiar with those basic notions, you can start your own MiningMart application easily.

1.2 Basic notions in MiningMart

This section explains several terms that are used throughout the MiningMart system and its documentation. You can use this section for general reference. Where words are printed in italics, they have their own entry in this section.

Business data This is the data in which knowledge is to be discovered. It must be stored in a relational database. It can consist of any number of tables, views and relations between them. The MiningMart system assumes that all data is stored in one database schema; if this is not the case, a single schema with database links to the needed tables should be set up (please refer to the documentation of your DBMS).

Metadata This is “administrative” data which MiningMart uses to store information **about** the business data as well as **about** the knowledge discovery process. Metadata can be stored in a separate database schema (which can live in a separate database) from the business data, or in the same schema. MiningMart uses a fixed data model for its metadata, which is called *M4* (MiningMart MetaModel).

M4 (MiningMart MetaModel) This is the fixed data model in which MiningMart stores its own information, called *Metadata*. M4 consists of several parts, but it is not important for users of MiningMart to know much about it.

Conceptual level As explained in section 1.1, MiningMart uses a layer of abstraction of the business data in order to hide sensitive details from other MiningMart users. This layer is the conceptual level. Its name stems from the fact that on this level, the data is described in everyday *concepts* rather than in terms of its technical representation. For example, many institutions have got data about their customers. So it could make sense to introduce the common *concept* “Customer” on the conceptual level, where it represents the data about customers. Information about this level forms part of the *Metadata* described above.

The conceptual level is the most important one for MiningMart users, because all the data processing is described in terms of the conceptual level. That is, whenever the customer data in the above example is accessed, this is done via the *concept* “Customer”. In contrast to this level, there is the *relational level* which also forms part of the *Metadata*, but which contains less abstract information about the business data. Both levels must be *connected* (see below).

Relational level On this level, the business data is described in terms of its technical representation. This means that the relational level (being part of the *Metadata*) stores exact information about the tables and columns that contain the business data. While a *concept* such as “Customer” may be rather common in several institutions, the way the data about customers is organised

will be different in each institution. Therefore, sharing MiningMart applications (as explained in section 1.1) makes use only of the conceptual level.

Connections (of the conceptual and relational level) Information about a concept like “Customer” and about the specific business data table containing customer data must be linked. Thus, there exist *connections* in MiningMart between the conceptual and the relational level. *Concepts* are connected to *columnsets*, *features* are connected to *columns* (see the definitions of these terms).

There are two ways to create a connection: the user can create one, or the MiningMart *compiler* can do that. The central idea is that there are some *concepts*, called DB concepts, that represent the input *business data* for the *case*. For these, their connection to the right *ColumnSets* is defined by the user (with the help of the *concept editor*). They must be set up by a user who is familiar with the information needed for the relational level, that is, the exact information about the tables and columns in the business data.

Other concepts, called MINING concepts, represent business data that was created during the execution of a MiningMart *step*. This execution is done by the *compiler*; thus, the *compiler* creates not only the data but also the connections to the *concepts* and *features*.

Case A case is a knowledge discovery process, or data preprocessing application, as modelled in MiningMart. Users work on one case at a time. A case contains the processing *steps* which may be organised in *chains*. Cases can be exported and imported. They are the unit of knowledge sharing: the web platform mentioned in section 1.1 lists successful cases (knowledge discovery or data preprocessing applications) which were exported by other MiningMart users and can be downloaded and imported. (Only the conceptual level is ex- or imported; after import, you need to *connect* that information to the relational level.)

Step A step represents a single processing task in a case. In each step, exactly one *operator* is applied. Steps are represented by icons in the MiningMart workspace (the case editor). Steps are applied to the data in a certain user-defined order, where the input of one step depends on the output of the previous one. These dependencies are represented in the MiningMart workspace by arrows. They form a Directed Acyclic Graph (DAG), that is, there must not be any cyclic dependencies. You can give explanatory names to the steps of a *case*.

Chain Any number of *steps* can be organised into chains. This provides a means to organise large *cases* with many steps so that the functions performed in that case become clearer. Comprising several steps which together perform some definable task (like data cleaning, for example) gives a better overview of the case. You can give explanatory names to the chains of a case.

Operator An operator performs a single, precisely defined task on the *business data*. Each operator is applied in exactly one *step*. Each operator has *parameters* which define its input and output in terms of the data on the *conceptual level*. There are two basic kinds of operators: those whose output is a *concept* and those that add an extra *feature* to their input concept. A few operators do not belong to either of these categories. Examples for tasks that operators perform are the replacement of missing values in the data that belongs to the input concept, or the creation of a new view on the data from the input concept, or the selection of important *features* from the input concept, etc.

A list of all operators with their technical description and details can be found in chapter 4.

Parameter Parameters are related to *operators*; they define their input and output on the *conceptual level*. Some parameters that many operators have are: `TheInputConcept`, which defines the *concept* whose data a certain operator uses as input; `TheOutputConcept` or `TheOutputAttribute`, which define the output of an operator; etc. For every operator, its parameters are listed in detail in chapter 4.

Concept A concept in MiningMart represents an everyday notion for which there exists data in the database. For example, as mentioned earlier, a concept “Customer” may exist in MiningMart and refer to one or more tables in the database that contain data about customers. Concepts have *features* which define them. The MiningMart system provides a concept editor to create, edit and delete concepts and their features. Concepts belong to the *conceptual level* and define the input for every *step* (or its *operator*, more precisely). Concepts are *connected* to *ColumnSets* which represent the database contents on the *relational level*.

There are two types of concepts: DB and MINING. The first type are concepts whose data exists before any MiningMart *step* is executed. That is, these concepts represent the input data for the *case*. All MINING concepts, in contrast, are not *connected* to any data before the execution (called *compilation*) of a MiningMart *step*. The MiningMart *compiler* creates the data that belongs to the MINING concepts and *connects* it to them. See also under *compiler* and *connection*.

Feature A feature is an attribute of a *concept*. For example, a concept “Customer” may have the features “Age”, “Income”, “Address”, etc. A concept “Product” may have the features “Price”, “Number of Sales” and others. There exist two kinds of features in MiningMart: *BaseAttributes* and *MultiColumnFeatures*. Like concepts, features can be *parameters*.

BaseAttribute A BaseAttribute is a *feature*. It represents a single attribute of the MiningMart *concept* it belongs to. BaseAttributes are *connected* to *Columns* which represent a database column on the *relational level*. For example, the

concept “Customer” may have a BaseAttribute “Age” which is *connected* to a column of a table in the database called “cust_age”.

MultiColumnFeature A MultiColumnFeature is a *feature*. It represents a conceptual bundle of attributes of a *concept*. Thus, it consists of at least two *BaseAttributes*. For example, a MultiColumnFeature “Address” may be used to bundle the BaseAttributes “Street”, “City” and “TelephoneNumber”. MultiColumnFeatures are a conceptual device in MiningMart which may be used to structure the concepts in order to give a more intuitive view on the business data.

Relation A relation represents a database link between two tables. It can either be a 1:n-relation or an n:n-relation. Relations in MiningMart store the information about foreign keys and primary keys as well as (optional) cross tables so that the *operators* can use this information. Thus, relations can be *parameters* like concepts and features. As such, they should belong to the *conceptual level*; however, since they also store database-related information, they might also be said to belong to both levels (conceptual and relational).

ColumnSet ColumnSets are MiningMart objects that directly represent a database table or view. As such, they belong to the *relational level*. Each ColumnSet is *connected* to exactly one *concept* (but a concept may have more than one ColumnSet). Each ColumnSet contains one or more *Columns*.

Column A Column is a MiningMart object that directly represents a column in a database table or view. Columns belong to the *relational level*. Each Column belongs to exactly one *ColumnSet*, but a ColumnSet can contain any positive number of Columns.

Compiler, compilation The MiningMart compiler performs the central task in MiningMart: it executes *operators*. That is, it reads the input *parameters* of an operator, applies the operator-specific processing to the data that corresponds to (is *connected* to) the input, and creates the output data and *connects* it to the *concepts* or *features* that are specified by the operator’s output parameters. The compilation of any *step* depends on the compilation of previous steps if a step uses input that is the output of a previous step.

The compiler can be executed in two modes: *lazy* and *eager*. This only makes a difference if there are concepts in the case that have more than one ColumnSet, which can happen as the result of a segmentation operator (see sections 4.3.2, 4.3.2 and 4.3.2 in chapter 4). In *lazy* mode, the compiler executes the operator-specific task only on the first of the ColumnSets that belong to the input concept of that operator, which saves time for testing. For full compilation, the *eager* mode is needed.

MiningMart workspace This is what you see when MiningMart is started: the graphical user interface which contains the *concept editor* and the *chain editor*. See chapter 3.

Concept editor In this window you can create, view, or delete *concepts* and their *relations* on both the *conceptual* and *relational level*. This editor is described in detail in chapter 3.3.

Chain editor In this window you can create, view, or delete *steps*; you can arrange them into *chains* and define the input and output *parameters* of their *operators*. The chain editor shows the currently defined sequence of steps, with their dependencies represented by arrows. More details can be found in chapter 3.

Export *Cases* can be exported with the export function. This will store all the *Metadata* that defines the case into a single file. This file can then be used for *importing* the case into another database (by another user, for example). See also chapter 5.

Import After *exporting*, a *case* can be imported into a new database. After import, all the *Metadata* of the case is available; however, the *connections* between the *conceptual* and *relational level* must still be made (see under *connections*). See also chapter 5.

InfoLayer InfoLayer is the name of the software that is used to run the web platform for the exchange of *cases*. This platform is mentioned in section 1.1. The InfoLayer software allows to browse through the MiningMart objects that define a case. At the same time, it allows to link descriptions to these objects which explain the case to a general audience. These descriptions form the so-called business layer. In the instance of the InfoLayer running on the MiningMart web pages, the business layer objects and the MiningMart objects linked. This instance also has a section called “Downloadable case” where *exported* case files can be put for the benefit of other MiningMart users.

More on the InfoLayer-based web platform for MiningMart can be found in the chapter 5.

Chapter 2

Installing the MiningMart system

2.1 General issues and downloading

This chapter contains all installation procedures for the different parts of the MiningMart system. If you are already running an older version of MiningMart please skip to section 2.5 to learn about upgrading.

This chapter assumes that there is already an Oracle database set up with one schema (user) for storing your business data to be processed, and with another user for storing the metadata. Please do not try to choose the same database user for these two tasks with version 0.21 of the system!

The first step is to download

- the platform independent base system archive
- the Java archive `GraphView.jar`
- the Java archive `hotdraw.jar`

from <http://mmart.cs.uni-dortmund.de/downloads/>

The platform dependent archives contain binaries from other authors and are not publically available. If you want to use these tools then you should download them directly from the author's pages (see 2.4). The two Java archives have to be downloaded separately because of their license.

Please unzip the platform independent base system to the directory where you want to have the system installed. On Unix or Linux you may use the command `unzip <filename>`. To run the system no write permission to the target directory is necessary. Especially on multi-user systems it is probably a good idea to install the base system just once for read-only accesses.

Unpacking the archive will create the sub-directory `MiningMart` containing the system files. In the remainder of this section we will assume that you

have already downloaded and unpacked the system. The directory MiningMart will be referred to as the `<base_directory>`. The next step is to put the files `GraphView.jar` and `hotdraw.jar`, to the directory `<base_directory>/lib`

Then the database needs to be prepared by installing a metadata schema and executable code like stored procedures. This is described in section 2.2.

Finally you should refer to section 2.3 on how to configure the system.

2.2 Preparing the database

The MiningMart system generally handles two database schemas (users). The first one is called the *business data schema*. It holds the data you want to analyse and preprocess with the MiningMart system. The second schema, the so called *M4 schema* holds metadata information about your business data and your preprocessing chains. You should not only reserve sufficient space on disk for your source business data, but account some extra space for materializing some of the views. For the M4 schema, on the other hand, 100 MByte should be sufficient for normal usage. In principle it should be no problem to split the schemas to two different Oracle databases or to use just one schema, referenced for both purposes. Please note, that this has never been tested! The standard installation foresees a separate schema for M4 and business data in the same database.

Preparing the database for MiningMart means to create the M4 schema in the database, in the first place. This schema consists of relational tables and a sequence. M4 is exclusively stored as the M4 user.

Additionally some parts of the M4 compiler make use of executable code stored directly in the business data schema. For reasons of convenience this code is installed by the same script as the M4 schema.

To run the installation script please change to the directory `<base_directory>/M4` and edit the file `install.sh` on Unix/Linux machines, or `install.bat` on Windows platforms:

1. The database connection information must be entered for the M4 schema. Please adjust the variables `M4USER` (database user of M4 schema), `M4PASS` (password), and `M4SID` (database server). Then you should change the according variables for the business schema, namely `BDUSER`, `BDPASS`, and `BDSID`.
2. If your system does not recognize the commands `sqlplus` and `loadjava`, then please set the variable to the absolute path to these ORACLE tools. You should find them in a subdirectory of your ORACLE software.
3. If you have never installed the metamodel before, you can now type `./install.sh` on a Unix or Linux machine, or `./install.bat` on Windows to have it installed. Otherwise, before running the installation, make sure that no data you might need is still in your previous metamodel, because such data will be lost during installation. If there are compilation errors during installation, please try to run the script for a second time.

If the script exits without errors, then you should have your database prepared for running the MiningMart system. The next section will teach you how to configure the system before the first usage.

2.3 Configuring the system

After you have your database prepared you finally just need to set up the start script to run the system. The file for Unix/Linux users is `<mmart.sh>`, on Windows platforms please use the file `<mmart.bat>`. Edit the file relevant for your platform with a text editor and change the following variable to the values that apply to your system:

- `MM_HOME` should be set to the absolute path to the MiningMart base directory (referred to as `<base_directory>` throughout this section).
- `MM_LOCAL_DIR` is a variable that points to the user's local directory. If the directory does not exist, then the system tries to create it. The default value is `<home_directory>/mmart`, which should be fine for Unix/Linux machines. On Windows you can leave the variable unset to have a comparable default behaviour, which should work fine, as long as your system supports home directories at Java level. Alternatively you can set the variable relatively to a user dependent environment variable like `HOME`, which is set properly on your machine, or choose an absolute path to a directory with write privileges for all users.
- The MiningMart system is a Java application with a recommended version of at least 1.4.2. If your system does not recognize the command `java` (i.e. in shells), then you should set the variable `JAVA` or `JAVA_COMMAND` to the executable `java` binary.
- Finally the system needs access to a JDBC driver. The default driver, which is part of Oracle 8.1.6 is `classes12.zip`. You should find this or a similar driver in your Oracle directory. If you use the same driver, then you may simply copy this file to `<base_directory>/lib` (on Unix/Linux symbolic links will also do). Alternatively you may set the variable `JDBC` in the start script to the absolute path to your Oracle JDBC driver.

If you have completed these steps, then you can start the system by running the start script. Please note, that the database settings are not yet known to the system, so the first thing you should do after the main screen comes up is to select "Tools" → "Edit DB settings" from the menu and enter your connection data. Afterwards you can directly use the system!

2.4 Plugins

The MiningMart system V0.21 supports some specific implementations of Machine Learning algorithms by implementing a wrapper approach. Currently the

following tools are supported if put to the directory
`<base_directory>/compiler/runtime/bin/<platform>/` :

- *mySVM*, a Support Vector Machine by Stefan Rueping
 For details please refer to
<http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/>
 The wrapper expects the target files `mysvm` and `predict`.
- *Apriori*, implementation of Bart Goethals
 For details please refer to
<http://www.cs.helsinki.fi/u/goethals/software/index.html>
 The wrapper expects the binary `apriori`.
- *C4.5* by Quinlan (SunOS only):
 The software may be downloaded at
<http://www.cse.unsw.edu.au/quinlan/>
 Expected binaries are `c4.5` and `c4.5rules`.

For Unix additional operators are available. Please download the file
`MMart_SunOS_plugins.tar.gz` from the MiningMart download page.

To enable a smooth integration between preprocessing, learning, and applying models a new operator called *YaleModelApplier* has been implemented to bridge the gap between MiningMart and the *Yale learning toolbox*. Yale has a Weka wrapper, offers automatic parameter setting, and powerful feature selection and construction algorithms on top of common classifiers as provided by Weka. After preprocessing raw data within your database you might want to draw a sample fitting in main memory, which can be read directly by a Yale operator. Then you are able to train your classifiers (or induce some other kind of model) based on the samples. Finally you may want to apply the model to unseen data. In this case MiningMart's *ModelApplier* can be used to create a new database view, holding the predictions for the new data.

Yale can be downloaded at
<http://yale.cs.uni-dortmund.de/>

All you have to make sure to be able to apply a model is that the two files `yale.jar` and `weka.jar`, both to be found in Yale's `lib/` directory are in the `CLASSPATH` when starting MiningMart.

2.5 Upgrading from older versions

This section is relevant only, if you have old MiningMart cases (before version 0.2) that you still want to use after upgrading the system. If you do not care about your old cases, then you can simply install the latest system from scratch. If you have at least M4 version 0.2, then you can backup your cases to XML files and run the script(s) for upgrading M4 to the current version.

Otherwise you should make sure first, that all of your cases are stored in the database. The old export files are no longer supported, because they are based on the Java serialization mechanism, which is highly sensitive to changes

at the implementation level of the M4 interface. The new import/export format is based on XML.

After you have all your cases in the M4 schema you can delete the old system's components outside the database (if not also used by other applications). Download the new system as described in section 2.1, just do *not* run the database installer script! Please download the scripts for upgrading to the current version instead, which are available at <http://mmart.cs.uni-dortmund.de/downloads/>.

Please run the scripts in this fashion:

```
sqlplus -user <dbuser>@database_sid @upgrade_to_v0.2.sql>
```

If you had a version before 0.2 then you should configure the `mmart.sh` or `mmart.bat` script as described in 2.3, start the system and export (backup) the cases with the new export format. Afterwards you can upgrade M4 to the current version.

2.6 Remarks

- If you should ever need to change the base directory of the MiningMart system in version 0.2 or 0.21, then please delete the automatically generated file properties in the users' local directories (i.e. `<user_home>/mmart`) before starting the system anew.
- If you want to increase the verbosity of log messages of the M4 interface on the console there is a variable `VERBOSITY` in the file properties in the users' local directories. Valid values are integers between 0 and 20. 0 means most verbose, 20 is least verbose. Changing the compiler verbosity level is not possible in versions 0.2 and 0.21.
- Although the compiler was implemented in Java some operators using external algorithms are only available on Unix systems. Please refer to section 2.4 on how to download these external binaries. The compiler itself was tested for Unix, Linux and Windows2000. A list of operators using external algorithms can be found in the appendix (section 2.7).

2.7 Appendix: List of operators that use external algorithms

- Apriori
- FeatureSelectionWithSVM
- GeneticFeatureSelection
- StatisticalFeatureSelection
- SGFeatureSelection

- MissingValuesWithRegressionSVM
- MissingValueWithDecisionTree
- MissingValueWithDecisionRules
- PredictionWithDecisionTree
- PredictionWithDecisionRules
- DecisionTreeForRegression
- SupportVectorMachineForRegression
- SegmentationWithKMean
- YaleModelApplier

Chapter 3

The Human Computer Interface

The Human Computer Interface (HCI) provides an easy way to use the Mining Mart System. It supports you in doing the work described in chapter 1.1 and integrates all components.

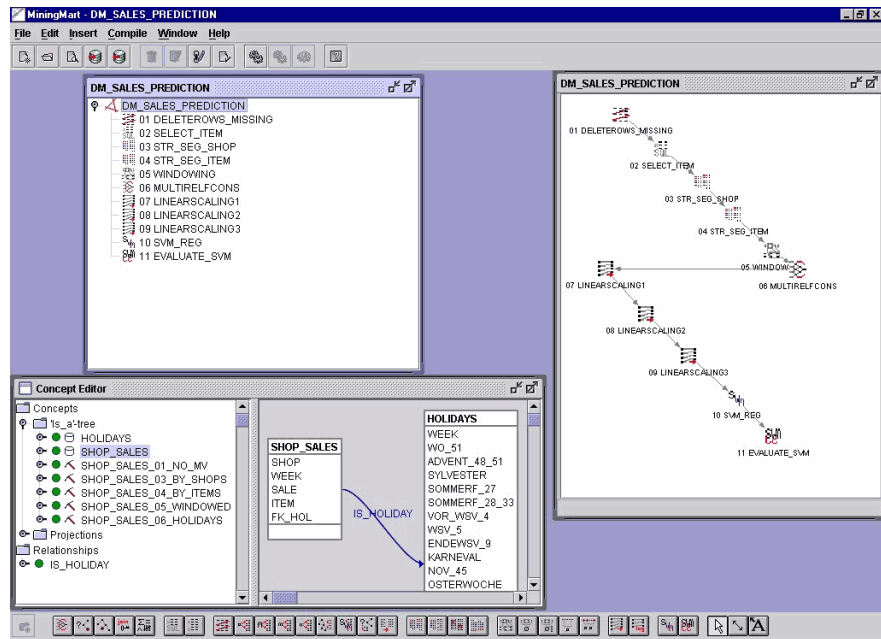
The HCI consists of two main components, the Concept Editor and the Chain Editor. The Concept Editor allows to create and manipulate concepts and connect them to the business data. These concepts are inputs for preprocessing operators that can be specified using the Chain Editor. The Chain Editor provides support in building preprocessing chains which consist of preprocessing steps.

This chapter first describes the main application which builds the framework for the Chain Editor and the Concept Editor, its main functionality and how it connects the Chain Editor with the Compiler. Then it focuses on the two components Chain Editor and Concept Editor.

3.1 Introduction

The main objective for the Mining Mart system (see Figure 3.1) is to provide a user-friendly interface for enhanced preprocessing of data for a knowledge discovery task. The system architecture (see Figure 3.2) consists of several components of which the Concept Editor is one. The other major components are: Chain Editor, Compiler, Mining Mart Meta Model (M4) Schema, M4 Interface, and Business Data Schema.

The heart of the Mining Mart system is the M4. It stores meta information about preprocessing steps and data. The M4 Interface provides a Java object interface to access the M4. The Concept Editor and Chain Editor act closely together and are both part of the Mining Mart system HCI. They both use the M4 Interface to manipulate the M4. They provide a user-friendly way to work with the meta data. The Concept Editor allows you to work with meta data



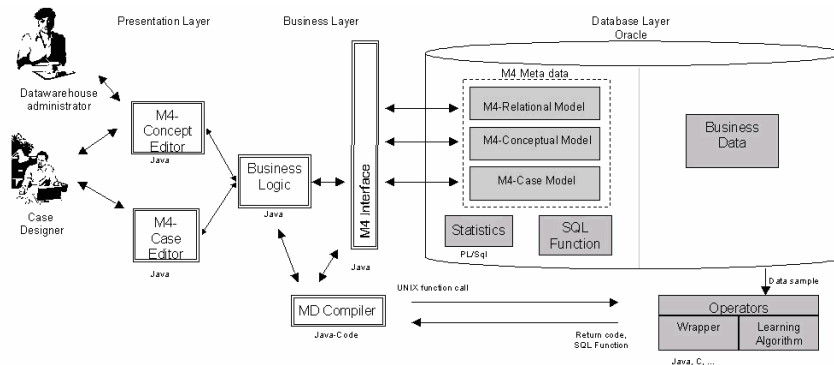
This screen shot of the Mining Mart HCI (human computer interface) depicts three internal windows. The upper and right windows form the Chain Editor; the lower left window shows the Concept Editor.

Figure 3.1: Screen shot of the Mining Mart HCI.

about business data. You need this information when working with the Chain Editor for defining preprocessing steps. The Compiler manages the execution of preprocessing steps. It triggers operators and writes the results back in the M4.

There are various other sources available that provide more information about the Mining Mart project and the Mining Mart system. A good place to start is the Mining Mart website ¹ which offers a good overview of the available documentation. Here also many documents can be downloaded directly. The Mining Mart approach is described in [MS03], [MS02], [KVZ01] and [KVZ00]. Further information about the Mining Mart system can be found in [LR02] (M4Interface), [VKZD01] (the Mining Mart Meta Model), the MiningMart final report (deliverable 20.4) and the technical reports which can be found on the website.

¹ <http://mmart.cs.uni-dortmund.de/>



Schematic view of the Mining Mart components. The Concept Editor and Case Editor are part of the HCI. The M4 Interface provides a Java object interface to the M4. The Compiler executes operators and creates resulting tables and views. The database (Oracle) contains the M4 and the business data.

Figure 3.2: Mining Mart components.

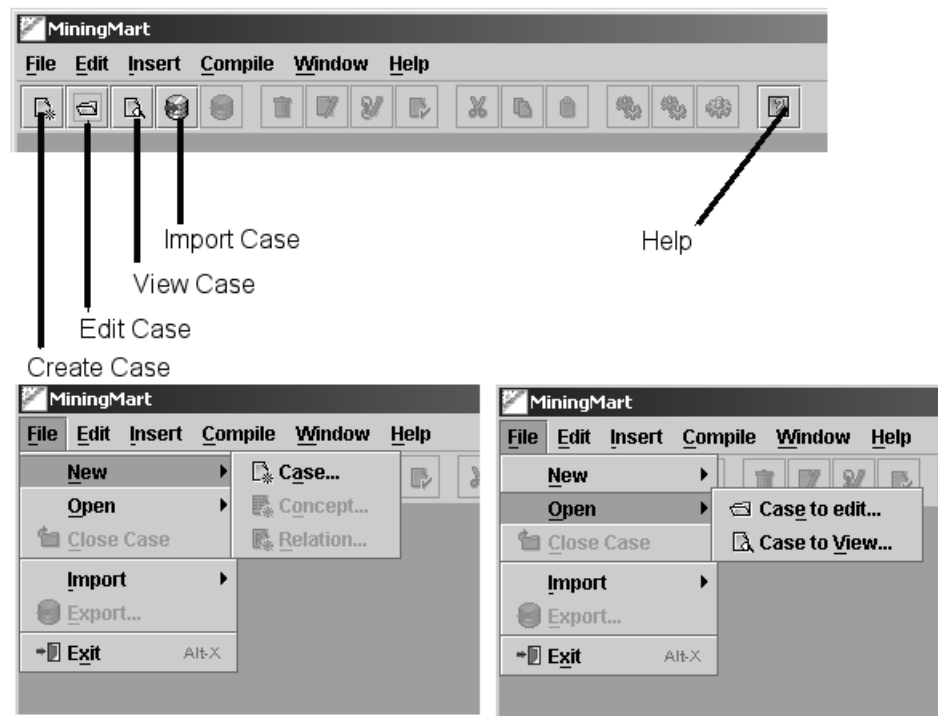
3.2 Main Application

To give a first impression the next subchapter briefly describes the first steps for starting the HCI and beginning to work with the application. Then the main functionality of the framework, which contains the both editors, is described in more detail. This chapter only focuses on the application frame which contains the two editors, the functionality of the Concept Editor is described in chapter 3.3 and the Chain Editor's functionality is shown in chapter 3.4.

3.2.1 Getting started

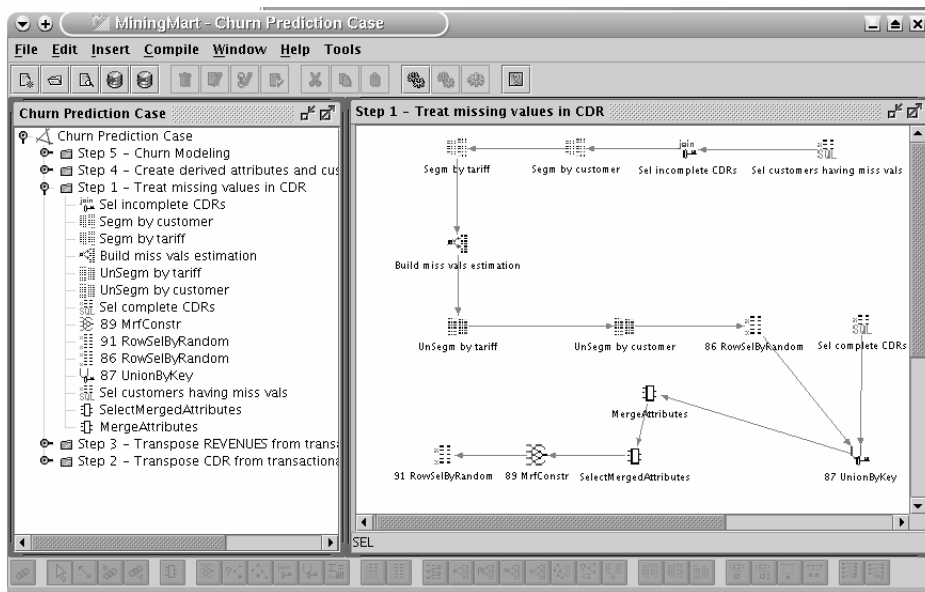
Since version 0.2 of the MiningMart system the HCI is no longer a separate application, but it is part of a single Java archive, containing the M4Interface, the M4Compiler, the Case Editor and the Chain Editor. Please refer to chapter 2 on how to set up MiningMart. After configuring the system you should have a working `mmart.sh` or `mmart.bat` file for starting the system. First of all the main frame of the application can be seen. Five Buttons are enabled: you can create a new case, open a case from the database to edit or to view only, import a case from the file system or call the help system. This functionality is also provided via the menus `File → New`, `File → Open` and `File → Import`. The menus and tool buttons are illustrated in figure 3.3.

Opening a case is only possible if you have already installed a case in the database or if anyone has worked with the system previously and created a case. To import a case you need to have a file which is exported with a MiningMart system. For more details see the subchapter 3.2.2.



The figure shows the first active toolbuttons and how to open or create a case with the file menu.

Figure 3.3: Getting started



Both windows of the Chain Editor. The left window is the tree view, the right one is the graph view.

Figure 3.4: Chain Editor

After you have opened or created a case, three windows are shown. The one with the title “Concept Editor” belongs to the Concept Editor (see figure 3.13), the two others to the Chain Editor. For a new case you have to describe the conceptual model and build a connection to the business database with the Concept Editor first. Chapter 3.3 explains how to work with the Concept Editor.

One of the two windows which belong to the Chain Editor has as its title the name of the case and shows all steps in a tree structure (*Tree View*); the other has the name of the chain if a node with a chain name is selected or if a node which is part of the chain is selected. It shows all steps belonging to the selected chain (*Graph View*). Figure 3.4 shows the windows of the Chain Editor. If a new case is created, the first chain has the same name as the case. How to change a chain name or the name of a node (step) is described in the next chapter.

3.2.2 Main functionality

The HCI enables you to create and manipulate cases, to export them into a file or to import such files. It provides menus for using the integrated components, which are the Chain Editor, Concept Editor and Compiler. Some menus and menu entries belong to one component only, others call the corresponding function for the active editor.

The following lists the possible global actions and menu entries. After that every use case is discussed in detail. The results are described, too.

- Create a Case
- Open a Case
- Re-use of Cases
- Manipulating objects
- Inserting an object
- Compile
- Window list
- Help

Create a Case

If you are working as a Case Designer, the main object you have to deal with is a *case*. To start working from scratch, you have to create a case first. Figure 3.3 shows how the menu looks like. After clicking the button or selecting the corresponding menu item a window is shown to enter a name for the new case and to select, if the case is in test mode or if it is final. The window is shown in figure 3.5.

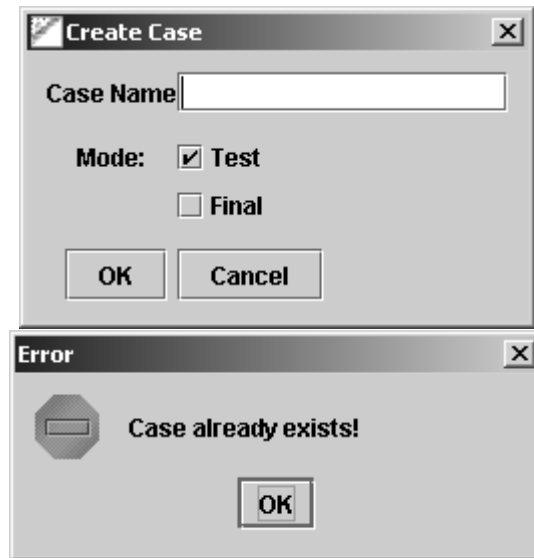
After pressing the ok- button the new case is created in the database and is opened in the editors. If the name already exists, a message is shown and you have to enter a different name. Now you are able to build the conceptual data model (see chapter 3.3) and to create preprocessing chains (see chapter 3.4).

Open a Case

As mentioned before there are two ways to open a Case. A Case can be opened for editing or for viewing only. After clicking the button for one of this actions or after selecting the menu item, you get a window for choosing an existing case (see figure 3.6). The same kind of window is always used if you need to select an existing M4Object in the Chain Editor, for example a Concept as an input for a step or the existing operators to insert a step with this operator into a Chain. If the case is already opened by another user, it is locked for this action and a message is shown.

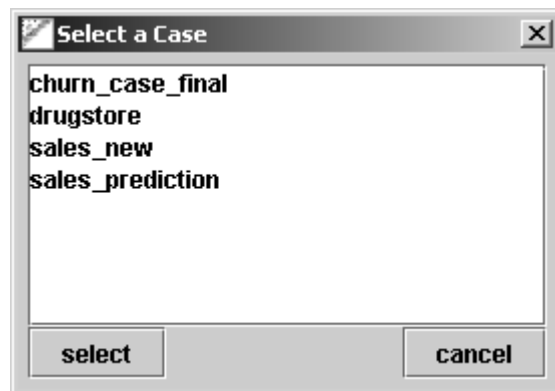
Editing means that you as the case designer can change the case. You can work with the Concept Editor to manipulate the conceptual data and you can work with the Chain Editor to change chains, steps, any parameter of a step etc. In the database a write lock is inserted and nobody else is able to open this case.

Viewing means that you can only view the case. Every action for changing something is disabled. Unfortunately this functionality is not supported by the



The figure shows the window to create a new Case (first window). The case designer has to insert a new name and select, if the case is in test mode or final. The figure also shows the message if the name already exists.

Figure 3.5: Create new Case



The figure shows a window called data chooser. This kind of window is always used if the user has to select an existing M₄Object. This example shows it for selecting a case.

Figure 3.6: Select a Case

Concept Editor in this version; in other words, the conceptual level can be manipulated even if the case is opened for viewing only. Opening a Case for viewing will insert a read lock in the data base for this Case. Everybody should be able to view the case, too, but nobody is able to edit the case. But it is not possible for one user to open one and the same case twice. If you try to open a case a second time, you always get a case locked exception.

Re-use of Cases

An important functional possibility is to reuse a Case. You are able to use a Case from any other user and to make a Case available to other users. The HCI supports this functionality with two actions, import a Case and export a Case. There exist menu items in the file menu and tool buttons for both.

After choosing one of the actions you are asked if you want to import (respectively export) the columns and column sets, too. This only makes sense if the user who has exported (respectively who is going to import) the Case uses exactly the same tables for his business data. This may only hold if two case designers are working in the same company with the same business data and want to exchange cases. After answering this question the standard java file-chooser is shown to select or to enter a file. This is shown in figure 3.7. After selecting a file/ entering a file name the import/ the export starts. During the import all m4- objects are stored as metadata in the database; afterwards you are able to open the imported Case as described in chapter 3.2.2.

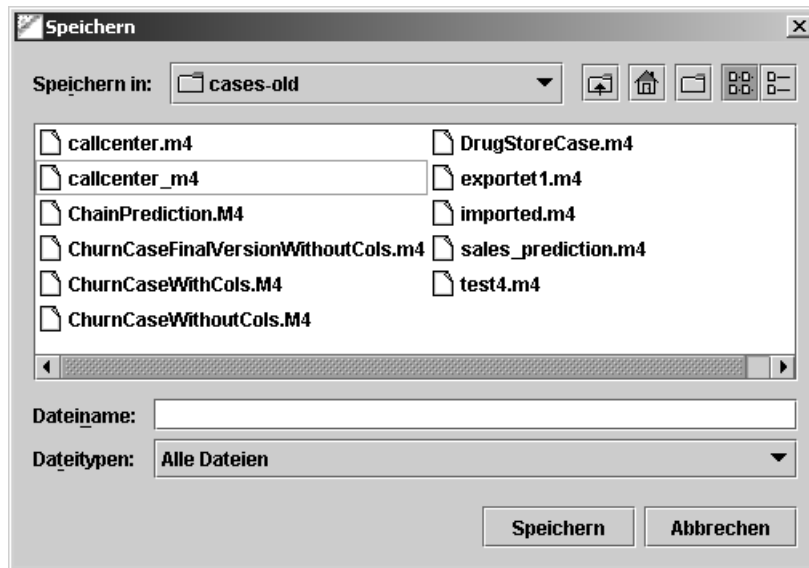
Another sort of reusability is the import of concepts from other cases in the database into the currently opened case. Figure 3.8 shows how to do this. After selecting this menu item you have to select a Case from which you want to import the concept, and then you have to select a concept. Finally you have to connect the concept with your business data as described in chapter 3.3.1.

Manipulating objects

The menu items for manipulating an object are collected in the menu "Edit". Which items are selectable depends on the active editor and sometimes on the objects which are selected in the active editor. In the first group the menu items "Delete" and "Properties" are active for both editors, the items "Open" and "Connections" only for the Chain Editor. The second block of menu items belongs to the Chain Editor and the third block to the Concept Editor. The functions are described in the chapters 3.4 for the Chain Editor and 3.3 for the Concept Editor. Figure 3.9 shows an example. In this picture the Concept Editor is active and a Concept is selected.

Inserting an object

The menu "Insert" is divided in two parts. The first two items belong to the Concept Editor and are only enabled if the Concept Editor is active and a corresponding object is selected. If a concept (a relation) is selected, you can



The standard file chooser of java. The user has to select a file for import or save a file for export via this window.

Figure 3.7: Java file chooser

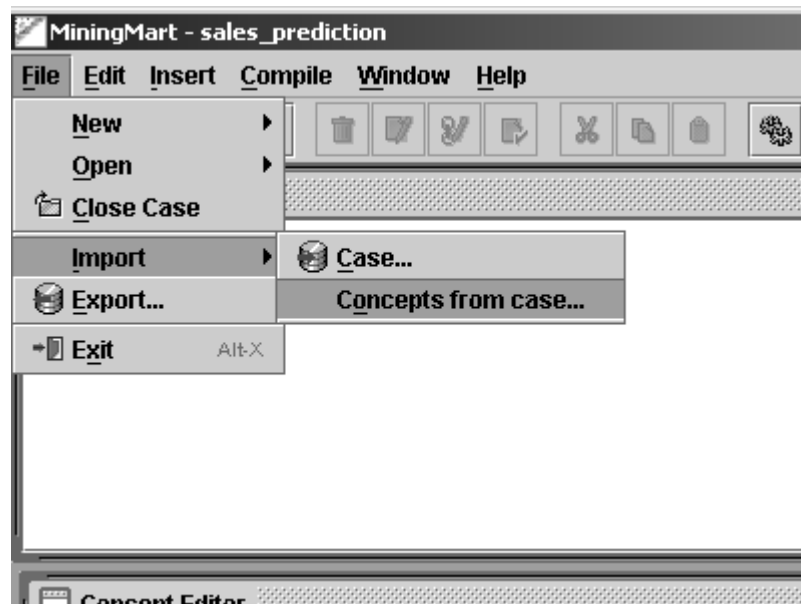
insert a Sub Concept (Sub Relation) by selecting the item. Then you get the property window for concepts (relations), described in chapter 3.3.

The second part contains the menu items “Chain” and “MiningMart Operator”. Choosing one of these items inserts a sub chain or an operator step in the selected chain. This functionality is described in more detail in chapter 3.4.

Compile

One important component of the Mining Mart system is the Compiler. The task of the Compiler is described in chapter 1.2. The menu “Compile” provides various calls to the Compiler, parameter settings and some additional functionality. The menu is shown in 3.10. The following explains the menu items.

- **Validate step, Validate all steps:** To be sure that the Compiler can compile a step without errors, you can test if a step or all steps are valid. The method for validating a step first checks if all parameters are specified in the property editor for steps and second if the parameters violate their constraints. If some parameters are missing, the step cannot be compiled without an error and no compilation will be started. If a constraint is violated, the compilation of this step may cause an error. But it is also possible that the compilation runs without an error. You are asked if you want to start the compilation despite the violated constraint.



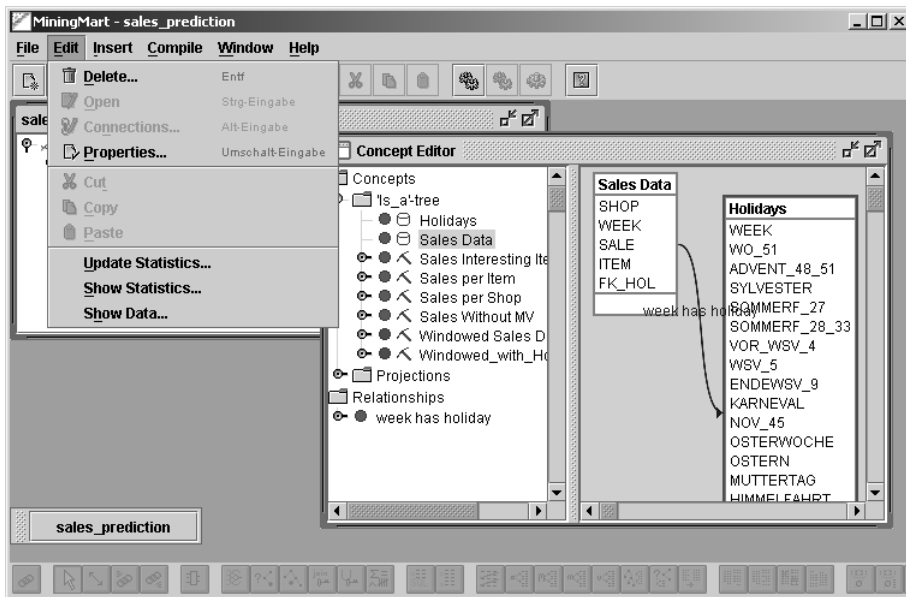
Selection of the menu item for importing a concept from another Case

Figure 3.8: Import concept

The test for validity uses operator-specific information. To learn more about the requirements of an operator, you can open the step which uses the operator, and click on the “Help” button (see also section 3.2.2). Then you are shown a description of what the operator does, and what parameters and conceptual input it expects. You can also refer to chapter 4 of this document. Most of the constraints that apply for an operator follow easily from these explanations. Please check that all parameters are in the right range and the input is correct.

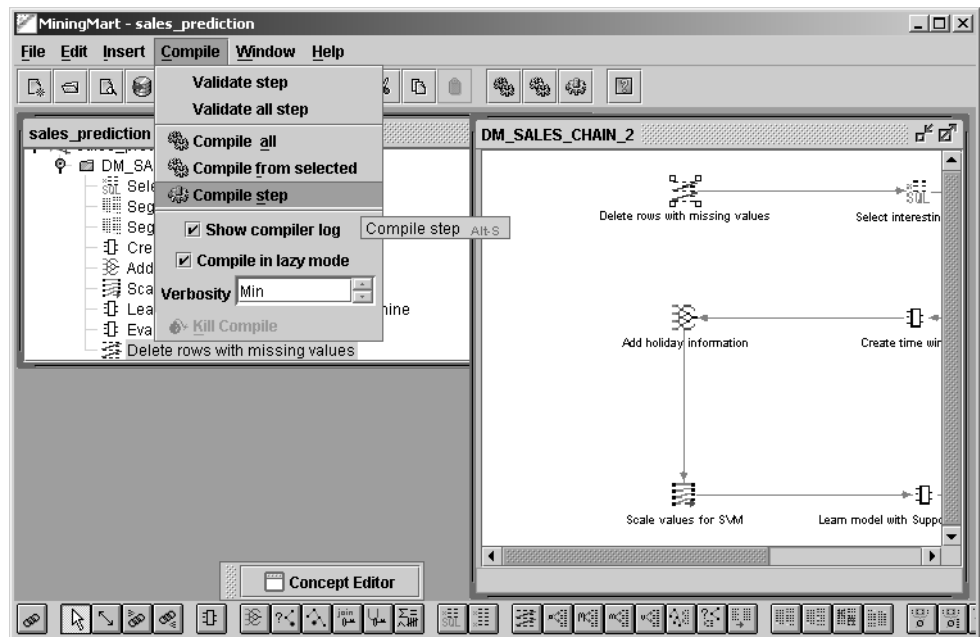
- **Compile all, Compile from step, Compile step:** These menu items call the Compiler. “Compile all” starts the compilation of all steps. The Compiler sorts all steps according to the dependencies between the steps (see chapter 3.4.9). Then it compiles one step after the other. “Compile from step” does the same but only for the selected step and all successors of this step. “Compile step” only compiles the selected step. The latter two items are only enabled if a step is selected. If these two methods are called, the Compiler assumes that all predecessors of the selected step are compiled. The HCI checks if the predecessors are compiled and if not, it will give an error message. In this case a compilation is not started.

These three method calls can be called via the three tool buttons (the buttons with the toothed wheels), too.



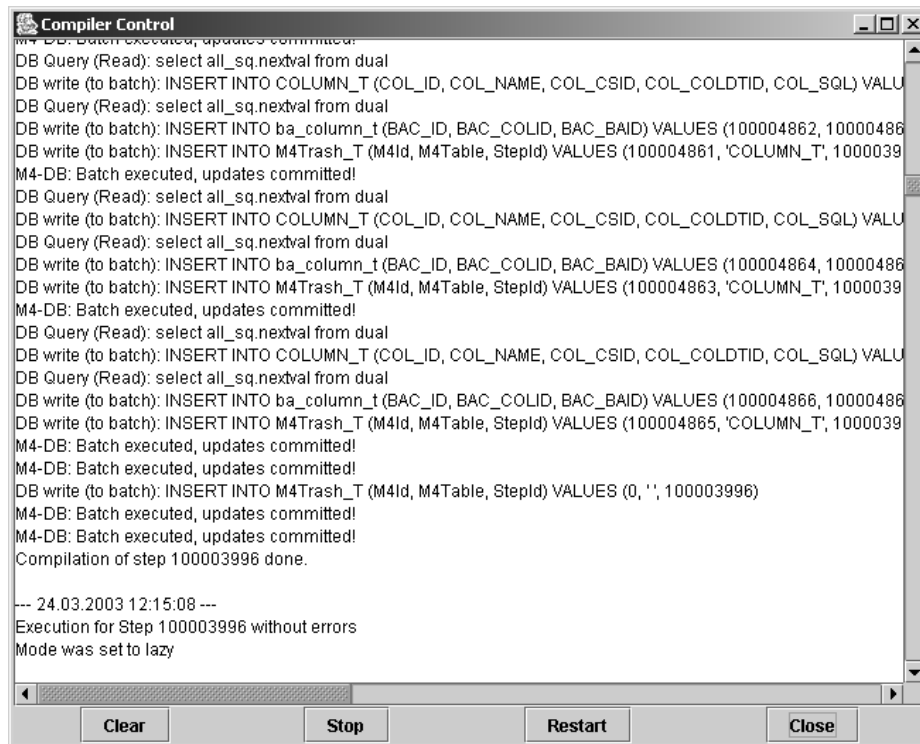
Only the menu items which belongs to the active editor (in this picture the Concept Editor) are selectable.

Figure 3.9: Menu “Edit”



The menu provides the method calls for the Compiler, parameter settings and some additional functionality

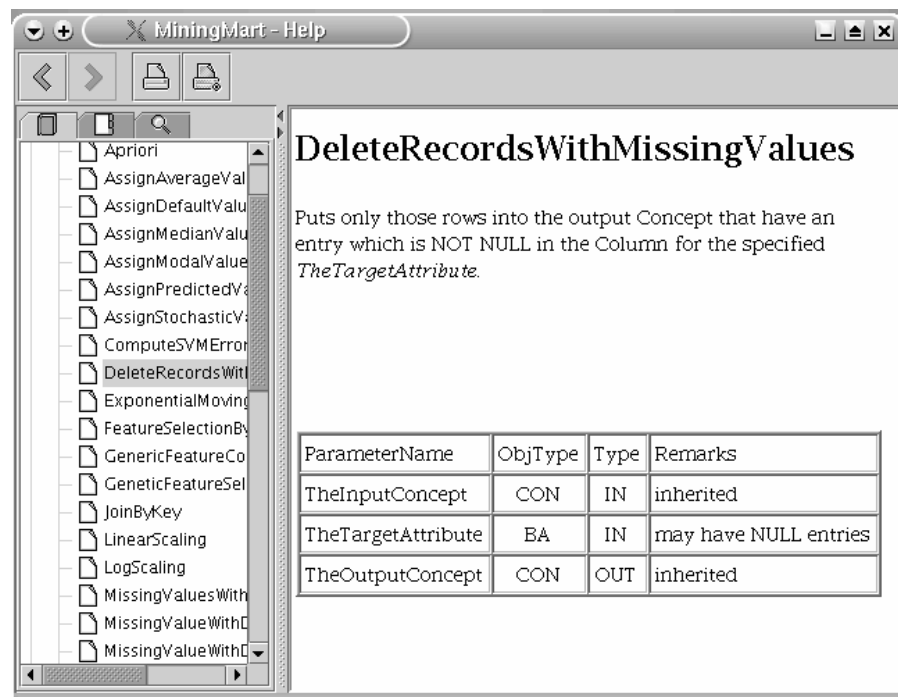
Figure 3.10: Menu "Compile"



A separate window to show the compiler messages. It provides buttons to clear the window, to stop the output of messages, to restart the output of messages and to close the window.

Figure 3.11: Compiler log Window

- **Show compiler log:** With this check box you can decide if you want to see the compiler messages. If selected, a compiler window is opened and the compiler log messages are displayed. Selecting this check box enables the spinner verbosity. Here you can specify how specific the displayed messages should be. The window is shown in figure 3.11.
- **Compile in lazy mode:** You can decide if the Compiler should compile the steps in lazy mode or in eager mode. For an explanation of “lazy mode” see chapter 1.2.
- **Kill Compiler:** If a compilation is running, this menu item is enabled. It allows you to stop a compilation thread on the compiler server. The compiler server stops the compilation at the current point, meaning that the current step compilation is not finished but the steps that were already compiled remain so.



This window with a description of the operator is shown if the step editor is active and the user presses the F1- button.

Figure 3.12: Help system

Window list

The window list shows a list of all windows and indicates which is the active one. A button for refreshing the active window is provided, too. The list can be used to switch from one window to another.

Help

The Mining Mart system also provides some help functionality. The help can be started with the help- button or with the help menu. For some windows of the chain editor a context sensitive help is provided. You can use this help by pressing the F1- button. The information which is shown in the help window depends on the active window. For example, if a step editor is active, a description of the operator is shown in the help window. Figure 3.12 shows an example.

3.2.3 Closing the application

The application has to be closed with the menu item “Exit” in the file menu or using the X- button of the frame. Only this way will ensure that the opened case is unlocked and openable again. If the application is closed externally (for example on a windows platform by closing the cmd window), the case remains locked. The only way to unlock it again is to delete the lock entry in the database.

3.3 The Concept Editor

The Concept Editor is part of the HCI. It allows to create and manipulate concepts and connect them to the business data. These concepts are inputs for preprocessing operators that can be specified using the Chain Editor.

3.3.1 Using the Concept Editor

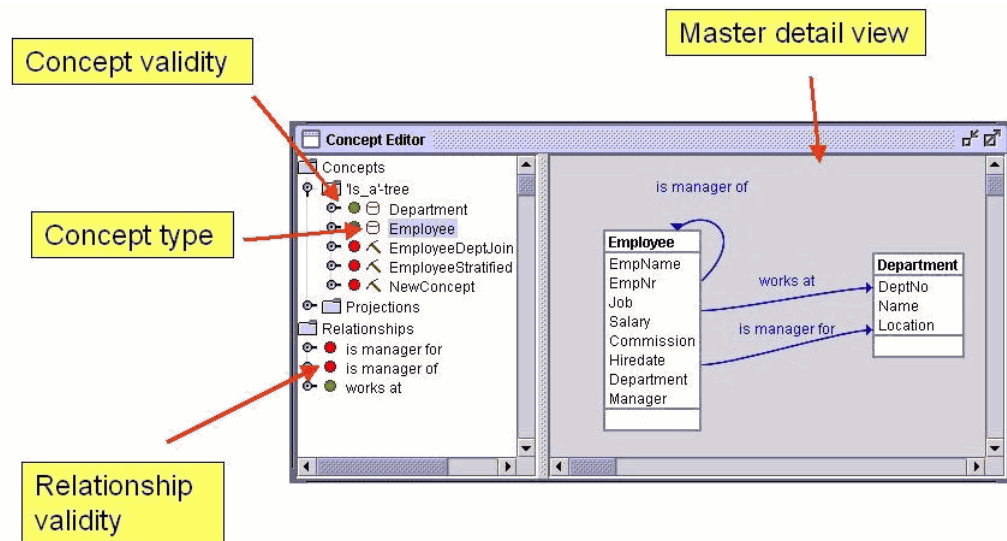
In this chapter an overview is given about the functionality of the Concept Editor and it is explained how to use it. The focus will be on the use cases for the concept editor, starting at a high level and then specifying these use cases further.

Overview of Functionality

The primary functions of the Concept Editor are to build a Conceptual Data Model (Concepts, FeatureAttributes and Relationships) and map this to the Relational Data Model. The editor provides an interface for doing this. It is also responsible for validation of Conceptual Data Model elements. The editor does not provide an interface for M4 objects that are not involved in the realization of the primary goals of the editor (e.g.: Case, Step, Operator).

The following lists the use cases:

- Build Conceptual Data Model
- Map Conceptual Data Model to Relational Data Model
- Validate the Conceptual Data Model
- Viewing Concept Data
- Create and View Statistics
- Reuse of Concepts



Overview of a Conceptual model in the Concept Editor.

Figure 3.13: Screenshot of the Concept Editor.

Building a Conceptual Data Model

An important part in the work of the case designer is to build a conceptual data model. The concepts can have relationships to each other, may be ordered in a hierarchy and will be, together with the operators, the building stones for preprocessing chains in a case.

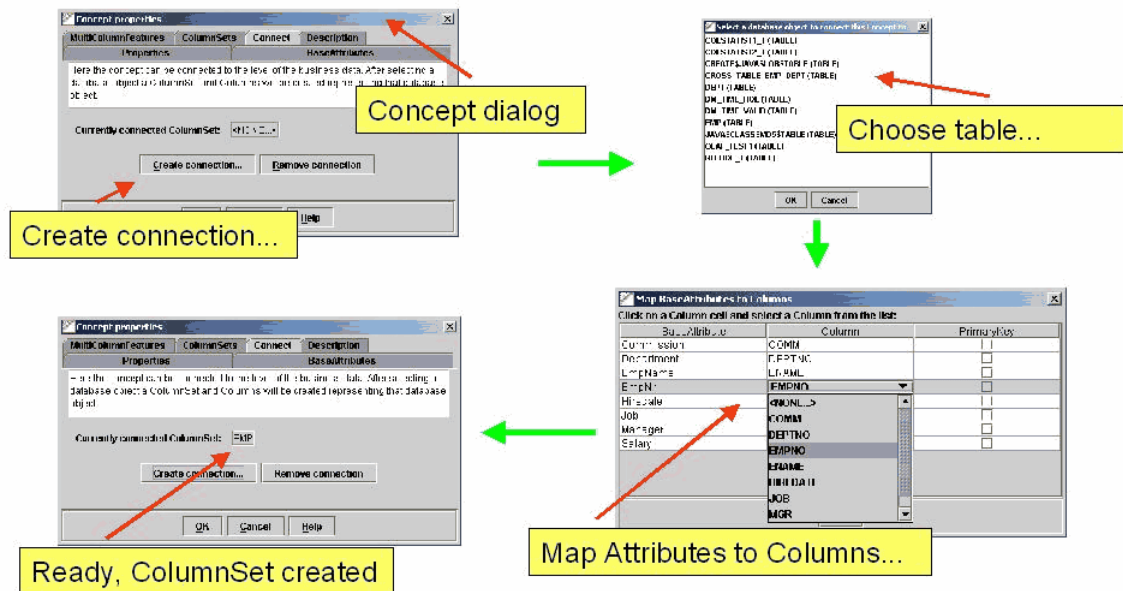
Concepts and Relationships can be created by choosing “New Concept” or “New Relationship” from the menu and filling in the properties for the Concept or Relationship. Editing and deleting existing concepts is done in a straightforward way by using the respective menu items in the Mining Mart HCI.

For an example of a Conceptual model see figure 3.13.

Mapping of Conceptual Data Model to the Relational Data Model

The Conceptual Data Model defined by the case designer has to be mapped to the Relational Data Model (the database) in order to be able to execute a case. This is only relevant for Concepts that are indeed based on existing tables in the database (Concept type DB). For Concepts that are created in one of the Steps of a Case (Concept type MINING) the corresponding ColumnSets are created by the Compiler. For Concepts of type BASE no mapping is allowed. See also chapter 1.2.

For an example of mapping a concept see figure 3.14. Double-clicking on a concept will present the concept dialogue. Choose the “Connect” option and click on “Create Connection”. This will present you with a list of possible



Schematic view of connecting a Concept with the Business data using the Concept Editor.

Figure 3.14: Connecting a concept.

database objects (tables or views) to connect the current concept to. After choosing an object, the relational-level metadata for it is automatically created and you need to link it to the conceptual level; this means to link every column of the database object to the corresponding BaseAttribute of the concept.

If you have Relationships in your conceptual model, you need to link them to the relational level, too. Again, double-click on the Relationship and go to “connect”. After choosing your type of Relationship (1:n or n:m), you must identify the primary and foreign keys of the columnsets involved and the cross table in the database (for n:m). Each Relationship holds between two concepts and these concepts are called “FromConcept” and “ToConcept” respectively. For the cross table, there is no concept.

Validity of Objects

The case designer needs to know if the current conceptual data model is valid or not. The validity of a conceptual data model can be summarized in the following way:

The Conceptual Data Model is valid if:

1. All Concepts are valid.
2. All FeatureAttributes are valid.
3. All Relationships are valid.

A Concept is valid if:

1. It is generated by an operator or based on a ColumnSet,
2. at least one included FeatureAttribute exists, and

A FeatureAttribute is valid if:

1. It is connected to a Concept.
2. It is generated by an operator or based on a Column.
3. (for MultiColumnFeature) at least two BaseAttributes exist, which belong to the same Concept as this MultiColumnFeature.

A Relationship is valid if:

1. both related Concepts exist.
2. it is based on Keys or a ColumnSet.

The HCI implements validation checking when creating, editing or deleting Concepts, FeatureAttributes or Relationships. The GUI shows the validity using a red icon (invalid) or green icon (valid) (see figure 3.13). Note that concepts of type MINING remain invalid until the compiler has created the relational metadata for them.

Viewing Data

You might want to see the data that is associated with a concept. This can be of importance in making decisions for preprocessing. Therefore the concept editor provides an option for viewing the data that is associated with a concept. The Mining Mart HCI provides a method for showing the data for a concept. Figure 3.15 shows an example of the dialog that is presented after choosing this option.

Creating and Viewing Statistics

Concept data statistics concerning cardinality, missing values, minimum, maximum, average and distribution blocks are helpful in making preprocessing decisions. These statistics can be generated by choosing the “update statistics” menu item in the HCI. They can be viewed by choosing the “view statistics” menu item. Figure 3.16 shows an example the statistics dialog.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-1...	800		20
7499	ALLEN	SALESMAN	7698	1981-02-2...	1600	300	30
7521	WARD	SALESMAN	7698	1981-02-2...	1250	500	30
7566	JONES	MANAGER	7839	1981-04-0...	2975		20
7654	MARTIN	SALESMAN	7698	1981-09-2...	1250	1400	30
7698	BLAKE	MANAGER	7839	1981-05-0...	2850		30
7782	CLARK	MANAGER	7839	1981-06-0...	2450		10
7788	SCOTT	ANALYST	7566	2087-04-1...	3000		20
7839	KING	PRESIDENT		1981-11-1...	5000		10
7844	TURNER	SALESMAN	7698	1981-09-0...	1500	0	30
7876	ADAMS	CLERK	7788	2087-05-2...	1100		20
7900	JAMES	CLERK	7698	1981-12-0...	950		30
7902	FORD	ANALYST	7566	1981-12-0...	3000		20
7934	MILLER	CLERK	7782	1982-01-2...	1300		10

Viewing the data that is associated with a concept in the Concept Editor.

Figure 3.15: Screenshot of viewing data for a concept.

View Statistics
ColumnSet busdat.EMP

	all	ord	nom	time
14	5	2	1	

Column Statistics 1

column na...	unique	missing	min	max	avg	stddev	variance	median	modal
EMPNO	14	0							
ENAME	14	0	ADAMS	WARD				KING	ADAMS
JOB	5	0	ANALYST	SALESMAN				MANAGER	CLERK
MGR	6	1	7566	7902	7739.307...	103.714660	10756.73...	7782.00000	7698.00000
HIREDATE	13	0	17-12-80	23-05-87				1981	1981
SAL	12	0	800	5000	2073.214...	1182.503...	1398313...	1600.00000	1250.00000

Column Statistics 2

column name	distvalue	distcount	distmin	distmax
EMPNO				
ENAME	ADAMS	1		
ENAME	ALLEN	1		
ENAME	BLAKE	1		
ENAME	CLARK	1		
ENAME	KING	1		

Viewing the statistics from data that is associated with a concept in the Concept Editor.

Figure 3.16: Screenshot of viewing statistics for data from a concept.

Re-using Concepts

You can also reuse an existing Conceptual Data Model from another case. You can select a Conceptual Data Model from another case, import it into the Concept Editor and adapt it to your wishes. For adapting the imported Concepts, FeatureAttributes and Relationships you can use the functionality which has been mentioned in “build Conceptual Data Model” (see Section 3.3.1).

Cases can be exported by the HCI to a file using the export option in the file menu. Via the import menu you can import a case from a file (from another database) or import concepts from another case (in the same database). See section 3.2.2.

3.4 The Chain Editor

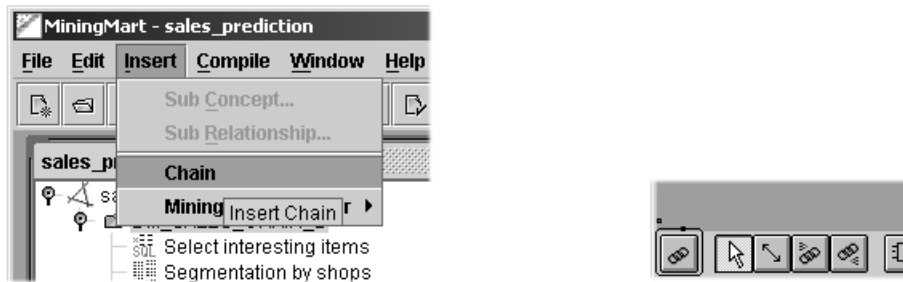
In this chapter the editor for pre-processing chains (Chain Editor) is described. Other parts of the HCI are the editor for concepts and relationships (Concept Editor - see chapter 3.3), which is integrated in a common environment with the Chain Editor, the M4 Interface, which is used by both editors and the M4 Compiler which can be called by the HCI. For a short explanation of these components see chapter 1.2. We discuss the usage of the Chain Editor, starting with a list of the functionality and then giving a more detailed view of how this functionality is provided.

3.4.1 Overview of Functionality

The primary goal for the Chain Editor is to support the creation of valid pre-processing chains. The preprocessing chain is made visible in two windows, the *tree view*, where all elements of the chain are shown in a tree structure and in a *graph view*, where only one (sub)chain is visualised. Some methods can be used with both windows and some are only usable with one of the window. Some methods can be called via tool buttons, some via menu items and some with both.

The following lists all actions you can perform with the Chain Editor. Here only the use cases for building and changing chains are listed, the other functionality is described in other chapters (for example how to start the Compiler).

- Creating or inserting a (sub)chain into the Mining Mart workspace or into a chain.
- Inserting a step (with an Mining Mart operator) into a chain.
- Changing properties for a step or a chain
- Editing the step parameter
- Changing position of a step or *folder* (subchain)
- Connecting steps



Inserting a chain can be done by using the menu item or by clicking the toolbar.

Figure 3.17: Inserting a chain

- Deleting steps, chains or connections
- Merging steps to a subchain (folder)
- Unmerge a subchain
- Cut, copy and paste parts of a chain

3.4.2 Inserting a chain

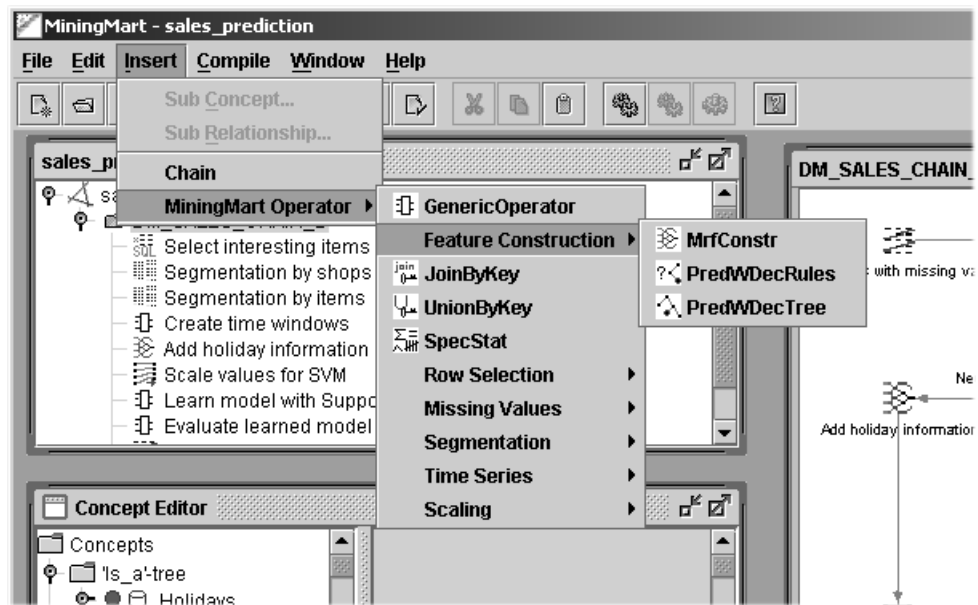
A chain can be inserted into a workspace or into another chain. If the tree view is the active view, then a chain can be inserted by selecting the workspace or the chain where the subchain should be inserted and then pressing the button “Insert chain” in the tool bar at the bottom of the HCI or using the menu “Insert”. Both is shown in the figure 3.17. If the selected node is not the workspace or a chain, the button and the menu entry are disabled. After that a folder-symbol is inserted in both views.

If the graph view is active, then a chain can be inserted via clicking the tool button mentioned above. After clicking the button a cross is shown and you can click anywhere in the graph view. After that a folder icon is shown at this position and the folder gets this position.

The new chain receives an automatically generated name and all information is stored in the M4- Schema immediately.

3.4.3 Inserting a step

Inserting a step can be done analogous to “Inserting a chain” described in chapter 3.4.2. A step can be inserted only into a chain, so in the tree view a chain has to be selected. Figure 3.18 shows how the menu structure for inserting



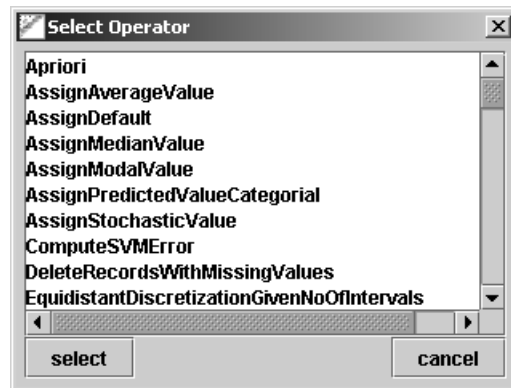
The menu structure to insert a step with a specific operator

Figure 3.18: Select a operator



The tool buttons to insert a step with a specific operator

Figure 3.19: Insert step- tool buttons



Selecting an operator by using the generic operator

Figure 3.20: Generic Operator

a step looks like. You select an operator and a step with this operator will be inserted. Figure 3.19 shows the corresponding tool buttons for inserting a step.

Some operators have their own menu item and toolbutton to insert them. Every operator can be inserted with the menu entry and the toolbutton *Generic Operator*. If you click on the generic operator button, the window shown in figure 3.20 is opened. Here you see a list of all specified operators and you can select one operator. Perhaps the term “Generic Operator” is a bit misleading. It means the parameter editor is generated automatically, while the other operators have their fixed editor window. This mechanism provides an easy way to expand the list of operators in the Mining Mart system.

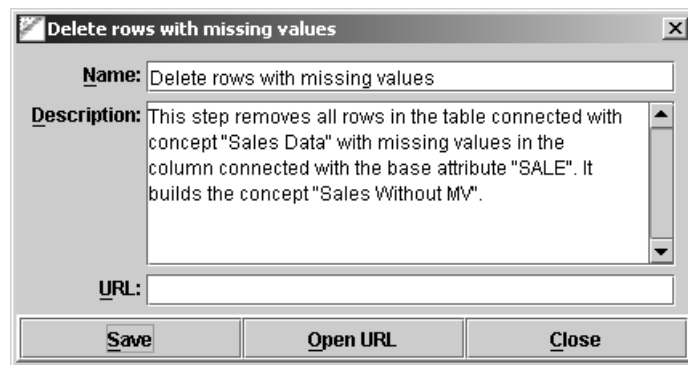
3.4.4 Changing properties

With the window shown in figure 3.21 you can change the name of a step or a chain and can enter or change the description. In the current version the field “URL” isn’t stored and the button “Open URL” isn’t used.

You get this window by selecting the object you want to change and then using the menu item “Properties” in the edit menu or by clicking the corresponding button.

3.4.5 Editing the step parameters

One important issue in the Chain Editor is the possibility to enter the parameter for a step. In general, a step and the included operator has some input parameters and one or more output parameters. There is an editor for every operator, in which you can specify these parameters. Figure 3.22 shows the editor for one of the most complex operators, the Support Vector Machine for Regression. In the following the main aspects of the parameters and the editor are listed. If an



Window to enter or change a description or change the name of a selected object

Figure 3.21: Change Properties

example is mentioned, this refers to this editor.

Loopable Operator

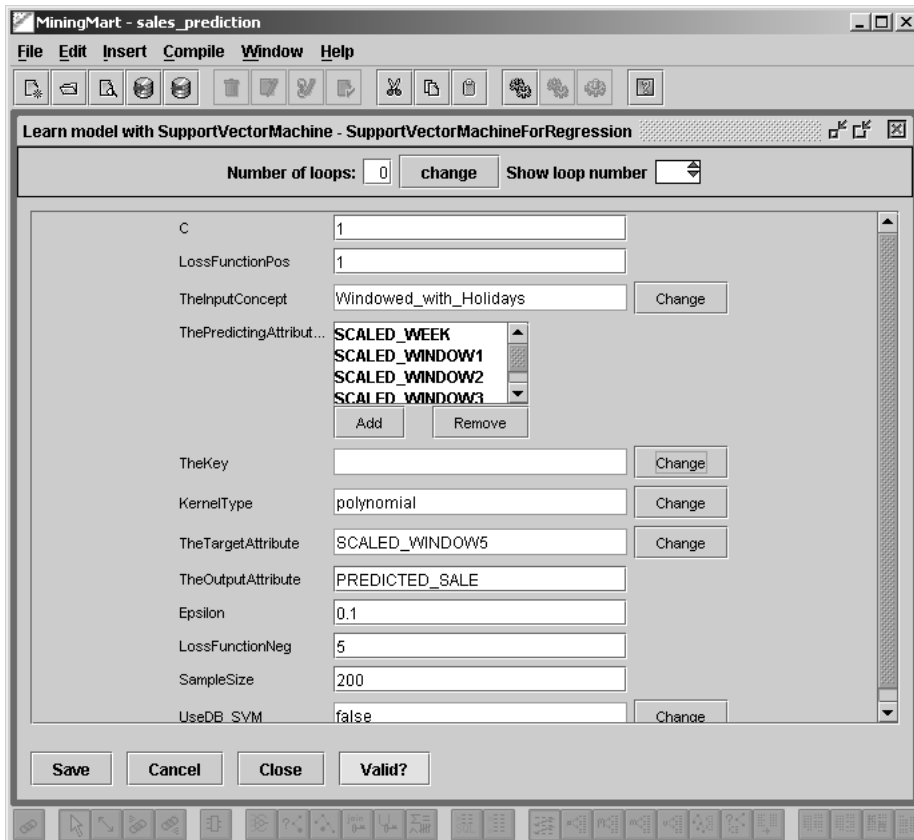
Some operators are loopable (see chapter 4.3.1 for an explanation of this term). In this case the upper box in the step editor is shown. You can enter a number of loops (0 means the steps isn't applied in loops) by typing the number and pressing the change- button and you can select the shown loop via the spinner on the right side. In the generic editor only the loopable parameters are shown if you select a loop number greater than 1. For example, in the step shown in figure 3.22 the *Input Concept* isn't shown for loop numbers greater than 1.

Choosing M4Objects

For input parameters which contain an M4Object (for example *Input Concept* or *Target Attribute*) you can change this object by pressing the change button next to the parameter. Then you get a data chooser with the possible objects (for the parameter "Input Concept" all concepts which are created so far and all concepts of Type "DB" or all base attributes from the input concept for the parameter "Target Attribute"). Changes are stored immediately.

Output Parameters and Values

For output parameters and values the editor provides fields to enter a string as a name for the new object. In the shown example such a parameter is the OutputAttribute and the parameters for values like "C", "Epsilon" etc. These objects are created after you have pressed the "Save" button at the bottom of the step editor.



The editor to enter the parameters for the operator Support Vector Machine For Regression. The window shows the different parts of the editor and parameters of different kinds.

Figure 3.22: An example editor for the step parameter

List Parameters

For parameters with a list of objects you can see a list of corresponding objects (in the example window: ThePredictingAttributes). You can change the list with the buttons “Add” and “Remove”. If the parameter is an input parameter using the “Add”- button will provide a data chooser (see chapter 3.4.5); for output parameters you get a box to enter a name for the new object. Other things mentioned in chapter 3.4.5 hold for list parameters, too.

Buttons

The step editor has four buttons at the bottom of the window.

- **Save** - All new objects (objects for the output parameters or values for value parameters) are created and every parameter is stored.
- **Cancel** - The output- and value parameters are set to the values after the last “save”. Every parameter is read from the database again.
- **Close** - Closes the step editor without any changes.
- **Validate** - The validity of the step is tested (see chapter 3.2.2).

3.4.6 Changing Positions

Every object in the Chain Editor has a position which is stored and retrieved from the database during opening a case. The position depends on the chain the object (step or subchain) belongs to. If an object belongs to a subchain, the position is stored as a position within this subchain. After unmerging the subchain or putting the object into another subchain, the step has a different position. To change the position of an object (step or subchain) in the graph view, you can press the left mouse button over this object and drag it to the new position. The position is stored automatically.

3.4.7 Selecting objects in the graph view

The graph view provides a method for selecting more than one object. Please click on the button for the Selection Tool (second button from left at the bottom of the HCI). Pressing the left mouse button and moving it in the graph view will show a rectangle and after releasing the mouse button all elements in the rectangle are selected. This also includes connections between steps. To select a single object, just click on it.

3.4.8 Deleting objects

Deleting objects can be done in different ways. An object (subchain, step) can be deleted by selecting the object in the tree view or graph view and pressing the delete- button or using the menu item “Delete” in the menu “Edit”. If the

graph view is active a delete can be enforced by pressing the “Del”- button on the keyboard, too.

In the graph view it is possible to delete more than one object at once. Selecting one or more objects is described in chapter 3.4.7. Deleting the selected objects is done like deleting a single object.

For deleting a connection you can click on the connection in the graph view and then use one of the methods mentioned above or you can use the connection window described in chapter 3.4.9.

3.4.9 Connecting steps

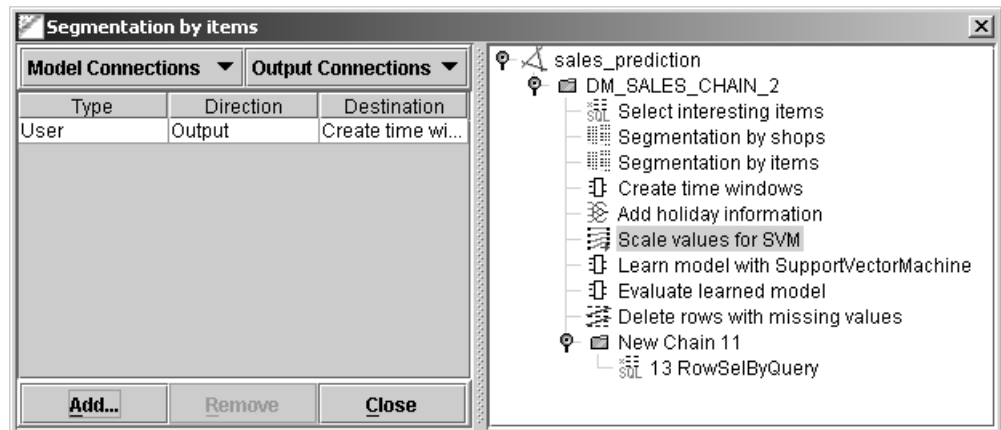
Connecting steps means to insert a connection from step A to step B. In Mining Mart this means to make step B dependent on step A, or step B is a successor of step A. This is necessary if step B uses an output (base attribute or concept) from step A as input and if step A isn't a predecessor of step B yet. Here predecessor means any predecessor, not only direct ones.

The easiest way to build a connection is to use the connection tool from the graph view, which is called with the tool button from the tool bar at the bottom of the HCI. The button can be identified by the double arrow on it. To do so the graph view has to be active. Then you can click the button, press the left mouse button over step A, then move the mouse cursor to step B and release the button. While doing you can see if step B can be a successor of step A. If step B can't be a direct successor of step A, the arrow remains grey and isn't inserted after releasing the mouse button, otherwise it gets black and is inserted.

The second way is using the “Edit Connections”- menu item in the edit menu. First you need to select step A and then use this menu item or the corresponding tool button. Then select step B from the tree in the window shown in figure 3.23 and a connection is inserted after clicking “add”. The button “Add” only gets enabled if step B is allowed to be a successor of step A. **Important:** Steps in different folders can be connected only via the Connection Window shown in figure 3.23. If they depend on each other in the above sense, these connections between different folders must not be omitted, otherwise the compilation of the chain will run into problems.

3.4.10 Merge steps to a chain

As described in chapter 1.1, steps can be organised in chains. There are two ways to achieve this. First, you can create a new chain and insert new steps in it; second, you can merge existing steps of a chain to a subchain. To do so, please use the graph view. First, press the button “Merge to chain” (fourth button from left at the bottom of the HCI), then select all steps and other subchains that you want to put in the new sub chain (see chapter 3.4.7). After releasing the mouse button a box is shown where you have to enter a unique name for the chain. As a result a folder object is shown instead of the selected objects in the graph view and a new node with this chain is inserted into the tree view.



Window for adding or removing connections between two steps.

Figure 3.23: Edit Connections

3.4.11 Unmerge sub chains

In a chain a subchain can be replaced with the objects it contains. You need to click on the fifth button (“Unmerge”) of the tool bar at the bottom of the HCI and then click on the folder symbol which represents the subchain you want to expand. After that the subchain is removed and all objects it contains are inserted instead of it.

3.4.12 Cut, Copy, Paste

This functionality is only provided in the graph view. You select one or more objects you want to copy (cut) and press the corresponding tool button in the top tool bar or use the menu item “Copy” (“Cut”) in the menu “Edit”. After that the menu item “Paste” and the corresponding tool button are enabled and you can open the chain where you want to paste the objects to in the graph view. Pressing the “Paste” button will insert the previously marked objects.

Chapter 4

Compiler Constraints and Operator Parameters

4.1 What this chapter is about

This chapter explains two things in detail: Firstly, section 4.2 describes some details about how the MiningMart compiler expects the metadata for a case description to be set up. Secondly, section 4.3 describes the current operators and their parameters.

4.2 Compiler constraints on metadata

This section explains in detail some issues in describing a case in such a way that it is operational for the MiningMart compiler.

4.2.1 Naming conventions

Operator names

The name of an operator (entry `op_name` in M4 table `Operator_T`) corresponds exactly (respecting case!) to the Java class that implements this operator in the compiler. This is only important to know if you want to implement additional operators. What is more generally important is that the names of the parameters of an operator are also fixed, because the compiler recognizes the type of a parameter by its name. This is described in more detail in section 4.3.1.

BaseAttribute names

Some operators have as their output on the conceptual level a `Concept` rather than a `BaseAttribute` (see section 4.3.1). This output `Concept` will generally be similar to the input `Concept`, in the sense that it copies some of the input

BaseAttributes without changing them. To find out which BaseAttribute in the output Concept corresponds to which BaseAttribute in the input concept, their names are used. They must match exactly, ignoring case. This also means that it is necessary to give the output BaseAttribute in a feature construction operator (see section 4.3.1) a name which is different from all BaseAttribute names in the input Concept, so that no names are mixed up. If the output of the operator is a Concept, and a BaseAttribute in this output concept has no corresponding BaseAttribute in the input concept, it will be ignored by the compiler, because it may be needed for later steps. Ignoring means that no Column is created for it.

A similar mechanism is applied when Relations are used (see following section 4.2.2).

4.2.2 Relations

Relations are defined by the user between the initial Concepts of a case. In a case, the Concepts may then be modified. If later in the chain an operator is applied that makes use of relations, it must be able to find the Columns that realize the keys. To this end, again the names of the BaseAttributes are used. Currently only `MultiRelationalFeatureConstruction` (MRFC) uses relations. This means that in the Concepts used by MRFC, the BaseAttributes that correspond to the key BaseAttributes in the initial Concepts must have the same name (ignoring case).

Example: Suppose there are initial Concepts *Customer* and *Product* linked by a relation *buys* which is realized by a foreign link from the *Customer* to the *Product* table. The foreign key Column in the *Customer* table is named `fk_prod` and its BaseAttribute is named *CustomerBuys*. The Concept *Customer* may be the input to a chain which results in a new Concept *PrivateCustomer*. This new Concept must still have a BaseAttribute named *CustomerBuys*, which must not be the result of a feature construction, but must be copied from Concept to Concept in the chain¹. Then the compiler can find the Column `fk_prod` by comparing the BaseAttributes of the current input concept *PrivateCustomer* and of the Concept which is linked to the relation *buys* (this relation is an input to the MRFC operator). The Column can be used to join the two Concepts *PrivateCustomer* and *Product*, although the first is a subconcept of *Customer*.

4.3 Operators and their parameters

This section explains the current MiningMart operators and the exact way of setting their parameters.

¹Copying is done by simply having a BaseAttribute of this name in every output Concept in the chain.

4.3.1 General issues

There are two kinds of operators, distinguished by their output on the conceptual level: those that have an output Concept (*Concept Operators*, listed in section 4.3.2), and those that have an output BaseAttribute (*Feature Construction Operators*, listed in section 4.3.4).

All operators have parameters, such as input Concept or output BaseAttribute. The name of such a parameter is fixed, for instance *TheInputConcept* is used for the input Concept for all operators. This means that the entry for this parameter in `par_name` in the M4 table `Parameter_T` must be *TheInputConcept*, respecting case. The parameter specification for each operator is stored in the M4 table `OP_PARAMS_T` (see MiningMart deliverable D18).

Some operators have an unspecified number of parameters of the same type. For example, the learning operators take as input a number of BaseAttributes of the same concept and use them to construct their training examples. All these BaseAttributes use the same prefix for their parameter name (here *ThePredictingAttributes*) in `Parameter_T`. Since all parameters for one step are expected to have different names (for HCI use), number suffixes are added to these prefixes (*ThePredictingAttributes1*, *ThePredictingAttributes2*, etc). The compiler uses `ORDER BY par_nr` when reading them. Such parameters, which may contain a list, are marked with the word *List* in the operator descriptions in sections 4.3.2 and 4.3.4.

Special attention is needed if an operator is applied in a loop. All feature construction operators are loopable; further, the concept operator `RowSelectionByQuery` is loopable. Feature construction operators are applied to one target attribute of an input concept and produce an output attribute. Looping means that the operator is applied to several target attributes (one after the other) and produces the respective number of output attributes, but the input concept is the same in all loops.

To decide whether an operator must be applied in a loop, the compiler checks the field `st_loopnr` in the M4 table `Step_T`, which gives the number of loops to be executed. If 0 or NULL is entered here, the operator is still executed once! If a number x (greater than 0) is entered here, the compiler looks for x sets of parameters for this operator in `Parameter_T`, excluding the parameters that are the same for all loops, which need to be entered only once. Thus, the parameter *TheInputConcept* must be declared only once, with the field `par_stloopnr` in the table `Parameter_T` set to 0, while the other parameters are given for every loop, with the respective loop number set in the field `par_stloopnr`, starting with 1. If no looping is intended, this field must be left NULL or 0. **Note:** Again, all parameters that are given for more than one loop must have a number suffix to their name, like the *List* parameters, to ensure that parameter names are unique within one step.

For the concept operator `RowSelectionByQuery`, looping means that several query conditions are formulated using the parameters of this operator (one set of parameters for each condition), and that they are connected with AND. See the description of this operator.

In the following sections, all current operators are listed with their exact name (see section 4.2.1), a short description and the names of their parameters. In general, all input BaseAttributes belong to the input Concept, and all output BaseAttributes belong to the output Concept.

4.3.2 Concept operators

All Concept operators take an input Concept and create at least one new ColumnSet which they attach to the output Concept. The output Concept must have all its Features attached to it before the operator is compiled. All Concept operators have the two parameters *TheInputConcept* and *TheOutputConcept*, which are marked as *inherited* in the following parameter descriptions.

MultiRelationalFeatureConstruction

Takes a list of concepts which are linked by relations, and selects specified Features from them which are collected in the output Concept, via a join on the concepts of the chain. To be more precise: Recall (section 4.2.2) that Relations are only defined by the user between initial Concepts of a Case. Suppose there is a chain of initial Concepts C_1, \dots, C_n such that between all C_i and C_{i+1} , $1 \leq i < n$, C_i is the *FromConcept* of the i -th Relation and C_{i+1} is its *ToConcept*. These Concepts may be modified in the Case being modelled, to result in new Concepts C'_1, \dots, C'_n , where some C'_i may be equal to C_i . However, as explained in section 4.2.2, the BaseAttributes that correspond to the Relation keys are still present in the new Concepts C'_i . By using their names, this operator can find the key Columns and join the new Concepts C'_i .

The parameter table below refers to this explanation. Note that all input Concepts are the new Concepts C'_i , but all input Relations link the original Concepts C_i .

ParameterName	ObjectType	Type	Remarks
TheInputConcept	CON	IN	Concept C'_1 (inherited)
TheConcepts	CON List	IN	Concepts C'_2, \dots, C'_n
TheRelations	REL List	IN	they link C_1, \dots, C_n
TheChainedFeatures	BA or MCF List	IN	from C'_1, \dots, C'_n
TheOutputConcept	CON	OUT	inherited

JoinByKey

Takes a list of concepts, plus attributes indicating their primary keys, and joins the concepts. In *TheOutputConcept*, only one of the keys must be present. Each BaseAttribute specified in *TheKeys* must be a primary key of one of *TheConcepts*; thus, the number of entries in *TheConcepts* and *TheKeys* must be equal.

If several of the input concepts contain a BaseAttribute (or a MultiColumn-Feature) with the same name, a special mapping mechanism is needed to relate them to different features in *TheOutputConcept*. For this, the parameters *MapInput* and *MapOutput* exist. Use *MapInput* to specify any feature in one

of *TheConcepts*, and use *MapOutput* to specify the **corresponding** feature in *TheOutputConcept*. To make sure that for each *MapInput* the right *MapOutput* is found by this operator, it uses the looping mechanism. Although the parameter is not looped, the loop numbers in the parameter table in M4 are used to ensure the correspondence between *MapInput* and *MapOutput*. However, these two parameters only need to be specified for every pair of equally-named features in *TheConcepts*. So there are not necessarily as many “loops” as there are features in *TheOutputConcept*.

The field `par_stloopnr` in the M4 parameter table must be set to the number of pairs of *MapInput/MapOutput* parameters (may be 0). Each of these pairs gets a different loop number while all the other parameters get loop number 0.

ParameterName	ObjectType	Type	Remarks
TheConcepts	CON <i>List</i>	IN	no <i>TheInputConcept</i> !
TheKeys	BA <i>List</i>	IN	
MapInput	BA or MCF	IN	“looped”!
MapOutput	BA or MCF	OUT	“looped”!
TheOutputConcept	CON	OUT	inherited

UnionByKey

Takes a list of concepts, plus attributes indicating their primary keys, and unifies the concepts. In contrast to the operator *JoinByKey* (section 4.3.2), the output columnset is a union of the input columnsets rather than a join. For each value occurring in one of the key attributes of an input columnset a tuple in the output columnset is created. If a value is not present in all key attributes of the input columnsets, the corresponding (non-key) attributes of the output columnset are filled by *NULL* values.

In *TheOutputConcept*, only one of the keys must be present. Each *BaseAttribute* specified in *TheKeys* must be a primary key of one of *TheConcepts*; thus, the number of entries in *TheConcepts* and *TheKeys* must be equal.

If several of the input concepts contain a *BaseAttribute* (or a *MultiColumnFeature*) with the same name, a special mapping mechanism is needed to relate them to different features in *TheOutputConcept*. For this, the parameters *MapInput* and *MapOutput* exist. Use *MapInput* to specify any feature in one of *TheConcepts*, and use *MapOutput* to specify the **corresponding** feature in *TheOutputConcept*. To make sure that for each *MapInput* the right *MapOutput* is found by this operator, it uses the looping mechanism. Although the parameter is not looped, the loop numbers in the parameter table in M4 are used to ensure the correspondence between *MapInput* and *MapOutput*. However, these two parameters only need to be specified for every pair of equally-named features in *TheConcepts*. So there are not necessarily as many “loops” as there are features in *TheOutputConcept*.

The field `par_stloopnr` in the M4 parameter table must be set to the number of pairs of *MapInput/MapOutput* parameters (may be 0). Each of these pairs gets a different loop number while all the other parameters get loop number 0.

ParameterName	ObjectType	Type	Remarks
TheConcepts	CON <i>List</i>	IN	no <i>TheInputConcept!</i>
TheKeys	BA <i>List</i>	IN	
MapInput	BA or MCF	IN	“looped”!
MapOutput	BA or MCF	OUT	“looped”!
TheOutputConcept	CON	OUT	inherited

PivotizeWithAggregation

Compare this operator to the tool *PivotizeWithoutAggregation* which can be called via the Tools menu from the MiningMart system. Pivotisation means to take the values that occur in an index attribute and create a new attribute for each of these values. The new attributes contain the values of a pivot attribute in those rows that contain the corresponding index value. Thus the pivot values are distributed over the new attributes which correspond to the index values. For clarification a simple example is given here. Assume that this is the input table/concept:

PrimaryKey	IndexAttr	PivotAttr
1	M	5
2	M	4
3	F	7

Then pivotization **without** aggregation produces the following output:

PrimaryKey	IndexAttr	PivotAttr	PivotAttr_M	PivotAttr_F
1	M	5	5	NULL
2	M	4	4	NULL
3	F	7	NULL	7

The tool *PivotizeWithoutAggregation* provided in the system realises this by creating a MiningMart Step that uses the operator *GenericFeatureConstruction* (see section 4.3.4) to create the new features.

However, this operator (*PivotizeWithAggregation*) creates a new output concept that does not contain the original index nor pivot attribute. Instead, the rows in the output concept are grouped by additional attributes and aggregated using an aggregation function. In the example above, if no *GroupBy-Attributes* are selected and *SUM* is the aggregation function, the output will be

PivotAttr_M	PivotAttr_F
9	7

However, if the primary key attribute is selected as a *GroupBy-Attribute*, the output will look like this (because each value of the primary key is its own group):

PrimaryKey	PivotAttr_M	PivotAttr_F
1	5	NULL
2	4	NULL
3	NULL	7

The values in *TheIndexAttribute* must be mapped by hand to names for the new attributes. For this, the parameters *IndexValue* and *MappedAttribute* are provided in loops, so that each loop specifies one mapping. Every value used in *IndexValue* should of course occur in *TheIndexAttribute*. *ThePivotAttribute* will not appear in *TheOutputConcept*, only in *TheInputConcept*.

The *AggregationOperator* is one of SUM, MIN, MAX, AVG (average) and COUNT.

The parameter *NullOrZero* specifies whether NULL or 0 should be used wherever empty fields are created, such as in the last output table in the example above.

ParameterName	ObjectType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheIndexAttribute	BA	IN	see text
ThePivotAttribute	BA	IN	see text
TheGroupByAttributes	BA or MCF <i>List</i>	IN	optional
IndexValue	V	IN	“looped”!
MappedAttribute	BA	OUT	“looped”!
AggregationOperator	V	IN	SUM or MIN etc.
NullOrZero	V	IN	one of <i>Null</i> , <i>Zero</i>
TheOutputConcept	CON	OUT	inherited

SpecifiedStatistics

An operator which computes certain statistical values for the *TheInputConcept*. The computed values appear in a *ColumnSet* which contains exactly one row with the statistical values per group of tuples, and which belongs to *TheOutputConcept*. Groups of tuples are built by listing attributes with the *GroupBy* parameter. Each combination of values of the underlying *BaseAttributes* forms one group. If no attributes are listed with the parameter list *GroupBy*, then the operator will output a single tuple with the statistics of all the *ColumnSet*.

The sum of all values of a numerical attribute can be computed by specifying a *BaseAttribute* with the parameter *AttributesComputeSum*. There can be more such attributes; the sum is computed for each. *TheOutputConcept* must contain a *BaseAttribute* for each sum which is computed; their names must be those of the input attributes, followed by the suffix “_SUM”. The total number of entries in an attribute can be computed by specifying a *BaseAttribute* with the parameter *AttributesComputeCount*. There can be more such attributes; the number of entries is computed for each. *TheOutputConcept* must contain a *BaseAttribute* for each count which is computed; their names must be those of the input attributes, followed by the suffix “_COUNT”.

The number of unique values in an attribute can be computed by specifying a *BaseAttribute* with the parameter *AttributesComputeUnique*. There can be more such attributes; the number of unique values is computed for each. *TheOutputConcept* must contain a *BaseAttribute* for each number of unique values which is computed; their names must be those of the input attributes, followed by the suffix “_UNIQUE”.

For ordinal attributes the parameter lists `AttributesComputeMin` and `AttributesComputeMax` exists. The operator writes the minimum and maximum values of the corresponding attributes to the output `BaseAttributes` with the suffixes “_MIN” and “_MAX”.

Further, for a `BaseAttribute` specified with `AttributesComputeDistrib`, the distribution of its values is computed. For example, if a `BaseAttribute` contains the values 2, 4 and 6, three output `BaseAttributes` will contain the number of entries in the input where the value was 2, 4 and 6, respectively. For each `BaseAttribute` whose value distribution is to be computed, the possible values must be given with the parameter `DistribValues`. One entry in this parameter is a comma-separated string containing the different values; in the example, the string would be “2,4,6”. Thus, the number of entries in `AttributesComputeDistrib` and `DistribValues` must be equal. `TheOutputConcept` must contain the corresponding number of `BaseAttributes` (three in the example); their names must be those of the input attributes, followed by the suffix “_<value>”. In the example, `TheOutputConcept` would contain the `BaseAttributes` “inputBaName_2”, “inputBaName_4” and “inputBaName_6”.

ParameterName	ObjectType	Type	Remarks
TheInputConcept	CON	IN	inherited
AttributesComputeSum	BA List	IN	numeric
AttributesComputeCount	BA List	IN	(see
AttributesComputeUnique	BA List	IN	
AttributesComputeMin	BA List	IN	
AttributesComputeMax	BA List	IN	
AttributesComputeDistrib	BA List	IN	text)
GroupBy	BA List	IN	as GROUP BY in SQL
DistribValues	V List	IN	
TheOutputConcept	CON	OUT	inherited

UnSegment

This operator is the inverse to any segmentation operator (see 4.3.2, 4.3.2, 4.3.2). While a segmentation operator segments its input concept’s `ColumnSet` into several `ColumnSets`, `UnSegment` joins several `ColumnSets` into one. This operator makes sense only if a segmentation operator was applied previously in the chain, because it exactly reverses the function of that operator. To do so, the parameter `UnsegmentAttribute` specifies indirectly which of the three segmentation operators is reversed:

If a `SegmentationStratified` operator is reversed (section 4.3.2), this parameter gives the name of the `BaseAttribute` that was used for stratified segmentation. Note that this `BaseAttribute` must belong to `TheOutputConcept` of this operator, because the re-unified `ColumnSet` contains different values for this attribute (whereas before the execution of this operator, the different `ColumnSets` did not contain this attribute, but each represented one of its values).

If a `SegmentationByPartitioning` operator is reversed (section 4.3.2), this

parameter must have the value “(Random)”.

If a `SegmentationWithKMean` operator is reversed (section 4.3.2), this parameter must have the value “(KMeans)”.

Note that the segmentation to be reversed by this operator can be any segmentation in the chain before this operator.

ParameterName	ObjectType	Type	Remarks
TheInputConcept	CON	IN	inherited
UnsegmentAttribute	BA	OUT	see text
TheOutputConcept	CON	OUT	inherited

RowSelectionByQuery

The output Concept contains only records that fulfill the SQL condition formulated by the parameters of this operator. This operator is **loopable**! If applied in a loop, the conditions from the different loops are connected by AND. Every condition consists of a left-hand side, an SQL operator and a right-hand side. Together, these three must form a valid SQL condition. For example, to specify that only records (rows) whose value of attribute `sale` is either 50 or 60 should be selected, the left condition is the BaseAttribute for `sale`, the operator is `IN`, and the right condition is (50, 60).

If this operator is applied in a loop, only the three parameters modelling the condition change from loop to loop, while input and output Concept remain the same.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited (same in all loops)
TheLeftCondition	BA	IN	any BA of input concept
TheConditionOperator	V	IN	an SQL operator: <, =, ...
TheRightCondition	V	IN	
TheOutputConcept	CON	OUT	inherited (same in all loops)

RowSelectionByRandomSampling

Puts atmost as many rows into the output Concept as are specified in the parameter `HowMany`. Selects the rows randomly.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
HowMany	V	IN	max. no. of rows
TheOutputConcept	CON	OUT	inherited

DeleteRecordsWithMissingValues

Puts only those rows into the output Concept that have an entry which is NOT NULL in the Column for the specified `TheTargetAttribute`.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	may have NULL entries
TheOutputConcept	CON	OUT	inherited

SegmentationStratified

A MultiStep operator (creates several ColumnSets for the output Concept). The input Concept is segmented according to the values of the specified attribute, so that each resulting Columnset corresponds to one value of the attribute. For numeric attributes, intervals are built automatically (this makes use of the statistics tables and the functions that compute the statistics).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttribute	BA	IN	
TheOutputConcept	CON	OUT	inherited

SegmentationByPartitioning

A MultiStep operator (creates several ColumnSets for the output Concept). The input Concept is segmented randomly into as many Columnsets as are specified by the parameter *HowManyPartitions*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
HowManyPartitions	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

SegmentationWithKMean

A MultiStep operator (creates several ColumnSets for the output Concept). The input Concept is segmented according to the clustering method KMeans (an external learning algorithm). The number of ColumnSets in the output concept is therefore not known before the application of this operator. However, the parameter *HowManyPartitions* specifies a maximum for this number. The parameter *OptimizePartitionNum* is a boolean that specifies if this number should be optimized by the learning algorithm (but it will not exceed the maximum). The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm. The algorithm (KMeans) uses *ThePredictingAttributes* for clustering; these attributes must belong to *TheInputConcept*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
HowManyPartitions	V	IN	positive integer
OptimizePartitionNum	V	IN	<i>true</i> or <i>false</i>
ThePredictingAttributes	BA List	IN	
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

Windowing

Windowing is applicable to time series data. It takes two `BaseAttributes` from the input `Concept`; one of contains time stamps, the other values. In the output `Concept` each row gives a time window; there will be two time stamp `BaseAttributes` which give the beginning and the end of each time window. Further, there will be as many value attributes as specified by the *WindowSize*; they contain the values for each window. *Distance* gives the distance between windows in terms of number of time stamps.

While *TimeBaseAttrib* and *ValueBaseAttrib* are `BaseAttributes` that belong to *TheInputConcept*, *OutputTimeStartBA*, *OutputTimeEndBA* and the *WindowedValuesBAs* belong to *TheOutputConcept*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TimeBaseAttrib	BA	IN	time stamps
ValueBaseAttrib	BA	IN	values
WindowSize	V	IN	positive integer
Distance	V	IN	positive integer
OutputTimeStartBA	BA	OUT	start time of window
OutputTimeEndBA	BA	OUT	end time of window
WindowedValuesBA	BA List	OUT	as many as <i>WindowSize</i>
TheOutputConcept	CON	OUT	inherited

SimpleMovingFunction

This operator combines windowing with the computation of the average value in each window. There is only one *OutputValueBA* which contains the average of the values in a window of the given *WindowSize*; windows are computed with the given *Distance* between each window. See also the description of the Windowing operator in section 4.3.2.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
InputTimeBA	BA	IN	
InputValueBA	BA	IN	
WindowSize	V	IN	
Distance	V	IN	
OutputTimeStartBA	BA	OUT	
OutputTimeEndBA	BA	OUT	
OutputValueBA	BA	OUT	
TheOutputConcept	CON	OUT	inherited

WeightedMovingFunction

This operator works like `SimpleMovingFunction` (section 4.3.2), but the weighted average is computed. The window size is not given explicitly, but is determined from the number of *Weights* given. The sum of all *Weights* must be 1.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
InputTimeBA	BA	IN	
InputValueBA	BA	IN	
Weights	V List	IN	sum must be 1
Distance	V	IN	positive integer
OutputTimeStartBA	BA	OUT	
OutputTimeEndBA	BA	OUT	
OutputValueBA	BA	OUT	
TheOutputConcept	CON	OUT	inherited

ExponentialMovingFunction

A time series smoothing operator. For two values with the given *Distance*, the first one is multiplied with *TailWeight* and the second one with *HeadWeight*. The resulting average is written into *OutputValueBA* and becomes the new tail value. *HeadWeight* and *TailWeight* must sum to 1.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
InputTimeBA	BA	IN	
InputValueBA	BA	IN	
HeadWeight	V	IN	
TailWeight	V	IN	
Distance	V	IN	positive integer
OutputTimeBA	BA	OUT	
OutputValueBA	BA	OUT	
TheOutputConcept	CON	OUT	inherited

SignalToSymbolProcessing

A time series abstraction operator. Creates intervals, their bounds are given in *OutputTimeStartBA* and *OutputTimeEndBA*. The average value of every interval will be in *AverageValueBA*. The average increase in that interval is in *IncreaseValueBA*. *Tolerance* determines when an interval is closed and a new one is opened: if the average increase, interpolated from the last interval, deviates from a value by more than *Tolerance*, a new interval begins.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
InputTimeBA	BA	IN	
InputValueBA	BA	IN	
Tolerance	V	IN	non-negative real number
AverageValueBA	BA	OUT	
IncreaseValueBA	BA	OUT	
OutputTimeStartBA	BA	OUT	
OutputTimeEndBA	BA	OUT	
TheOutputConcept	CON	OUT	inherited

Apriori

An implementation of the well known Apriori algorithm for the data mining step. It works on a sample read from the database. The sample size is given by the parameter *SampleSize*.

The input format is fixed. There is one input concept (*TheInputConcept*) having a *BaseAttribute* for the customer ID (parameter: *CustID*), one for the transaction ID (*TransID*), and one for an item part of this customer/transaction's itemset (*Item*). The algorithm expects all entries of these *BaseAttributes* to be integers. No null values are allowed.

It then finds all frequent (parameter: *MinSupport*) rules with at least the specified confidence (parameter: *MinConfidence*). Please keep in mind that these settings (especially the minimal support) are applied to a sample!

The output is specified by three parameters. *TheOutputConcept* is the concept the output table is attached to. It has two *BaseAttributes*, *PremiseBA* for the premises of rules and *ConclusionBA* for the conclusions. Each entry for one of these attributes contains a set of whitespace-separated item IDs (integers).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
CustID	BA	IN	customer id (integer, not NULL)
TransID	BA	IN	transaction id (integer, not NULL)
Item	BA	IN	item id (integer, not NULL)
MinSupport	V	IN	minimal support (integer)
MinConfidence	V	IN	minimal confidence (in [0, 1])
SampleSize	V	IN	the size of the sample to be used
PremiseBA	BA	OUT	premises of rules
ConclusionBA	BA	OUT	conclusions of rules
TheOutputConcept	CON	OUT	inherited

Feature Construction with TF/IDF

This operator calculates term frequencies / inverse document frequencies, a measure known from information retrieval. In this setting the operator is applied for time series with binary attributes, instead.

The parameter *TheSelectedAttributes* contains a list of attributes, for which the TF/IDF values should be calculated. *TheKey* is the primary key attribute of this time series, while *TheTimeStamp* is the attribute holding the time information of the tuple.

Unlike other Feature Construction operators this one yields a concept, not a single feature.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheSelectedAttributes	BA <i>List</i>	IN	attrib. to cal. TF/IDF for
TheTimeStamp	BA	IN	type TIME
TheKey	BA	IN	key attribute
TheOutputConcept	CON	OUT	inherited

Union

This operator implements the normal UNION functionality known from SQL, thus the different `Concepts` specified as input need to be union-compatible. There is one “main” `InputConcept`, which specifies the `BaseAttributes` of the `OutputConcept`. If features of this `InputConcept` are deselected, then the features will also be deselected in the output. The features of all attributes in `TheInputConcept`. All further attributes in these `Concepts` will be ignored, all missing attributes will be replaced by artificially added “named NULL values”, which does not work for all datatypes!

Please note, that if you have no primary keys defined and you have multiple occurrences of the same tuples, then SQL will usually remove all duplicates when applying a UNION-operation. In some cases you may prefer a “bag” or “multi-set” semantics. For this reason the parameter `DataMode` allows to switch between `set` and `multi-set`.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
FurtherConcepts	CON <i>List</i>	IN	Union compatible Concepts
DataMode	V	IN	set or multi-set mode?
TheOutputConcept	CON	OUT	inherited

4.3.3 Feature selection operators

Feature selection operators are also concept operators in that their output is a `Concept`, but they are listed in their own section since they have some common special properties. All of them (except *FeatureSelectionByAttributes*, see 4.3.3) use external algorithms to determine which features are taken over to the output concept. This means that at the time of designing an operating chain, it is not known which features will be selected. How can a complete, valid chain be designed then, since the input of later operators may depend on the output of a feature selection operator, which is only determined at compile time?

The answer is that conceptually, **all** possible features are present in the output concept of a feature selection operator, while the compiler creates `Columns` for only some of them (the selected ones). This means that in later steps, some of the features that are used for the input of an operator may not have a `Column`. If the operator depends on a certain feature, the compiler checks whether a `Column` is present, and shows an error message if no `Column` is found. If the operator is executable without that `Column`, no error occurs.

All feature selection operators have a parameter *TheAttributes* which specifies the set of features from which some are to be selected. (Again this is not true for *FeatureSelectionByAttributes*, see 4.3.3.) The parameter is needed because not all of the features of *TheInputConcept* can be used, as they may include a key attribute or the target attribute for a data mining step, which should not be deselected. This means that all attributes from *TheInputConcept* that are *not* listed as one of *TheAttributes* will be present in *TheOutputConcept* both on the conceptual and on the relational level.

FeatureSelectionByAttributes

This operator can be used for manual feature selection, which means that the user specifies all features to be selected. This is done by providing all and only the features that are to be selected in *TheOutputConcept*. The operator then simply copies those features from *TheInputConcept* to *TheOutputConcept* which are present in *TheOutputConcept*. It can be used to get rid of features that are not needed in later parts of the operator chain. All features in *TheOutputConcept* must have a corresponding feature (with the same name) in *TheInputConcept*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheOutputConcept	CON	OUT	inherited

StatisticalFeatureSelection

A Feature Selection operator. This operator uses the stochastic correlation measure to select a subset of *TheAttributes*. All of *TheAttributes* must be present in *TheOutputConcept*. The parameter *Threshold* is a real number between 0 and 1 (default is 0.7). *SampleSize* specifies a maximum number of examples that are fed into the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA list	IN	see section 4.3.3
SampleSize	V	IN	positive integer
Threshold	V	IN	real between 0 and 1
TheOutputConcept	CON	OUT	inherited

GeneticFeatureSelection

A Feature Selection operator. This operator uses a genetic algorithm to select a subset of *TheAttributes*. It calls C4.5 to evaluate the individuals of the genetic population. *TheTargetAttribute* specifies which attribute is the target attribute for the learning algorithm whose performance is used to select the best feature subset. *PopDim* gives the size of the population for the genetic algorithm. *StepNum* gives the number of generations. The probabilities of mutation and crossover are specified with *ProbMut* and *ProbCross*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
SampleSize	V	IN	positive integer
PopDim	V	IN	positive integer; try 30
StepNum	V	IN	positive integer; try 20
ProbMut	V	IN	real between 0 and 1; try 0.001
ProbCross	V	IN	real between 0 and 1; try 0.9
TheOutputConcept	CON	OUT	inherited

SGFeatureSelection

A Feature Selection operator. This operator is a combination of *StochasticFeatureSelection* (see 4.3.3), which is applied first, and *GeneticFeatureSelection* (see 4.3.3), applied afterwards. The parameter descriptions can be found in the sections about these operators (4.3.3 and 4.3.3).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
SampleSize	V	IN	
PopDim	V	IN	
StepNum	V	IN	
ProbMut	V	IN	
ProbCross	V	IN	
Threshold	V	IN	real, between 0 and 1
TheOutputConcept	CON	OUT	inherited

FeatureSelectionWithSVM

A Feature Selection operator. This operator uses the $\xi\alpha$ -estimator as computed by a Support Vector Machine training run to compare the classification performance of different feature subsets. Searching either forward or backward, it finds the best feature subset according to this criterion. Thus it performs a simple beam search of width 1.

TheTargetAttribute must be binary as Support Vector Machines can only solve binary classification problems. (The $\xi\alpha$ -estimator can only be computed for classification problems.) The parameter *PositiveTargetValue* specifies the class label of the positive class. There are some SVM-specific parameters; the table gives reasonable values to choose if nothing is known about the data or SVMs. For the *KernelType*, only the following values (Strings) are possible: *dot*, *polynomial*, *neural*, *radial*, *anova*. *Dot* is the linear kernel and can be taken as default.

This operator can use two different versions of the Support Vector Machine algorithm. One runs in main memory; it needs the parameter *SampleSize* to determine a maximum number of training examples. The other runs in the database; it is used if the optional parameter *UseDB_SVM* is set to the String

true. When this version is used, an additional parameter *TheKey* is needed which gives the *BaseAttribute* whose column is the primary key of *TheInputConcept*. (*TheKey* can be left out only if the *ColumnSet* that belongs to *TheInputConcept* represents a table rather than a view.) The database algorithm restricts the possible kernel types to *dot* and *radial*. It can also use the parameter *SampleSize*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
TheTargetAttribute	BA	IN	must be binary
PositiveTargetValue	V	IN	the positive class label
KernelType	V	IN	see explanation above
SampleSize	V	IN	see explanation above
C	V	IN	positive real; try 1.0
Epsilon	V	IN	positive real; try 0.1
UseDB_SVM	V	IN	optional; one of <i>true</i> , <i>false</i>
TheKey	BA	IN	optional
SearchDirection	V	IN	one of <i>forward</i> , <i>backward</i>
TheOutputConcept	CON	OUT	inherited

SimpleForwardFeatureSelectionGivenNoOfAttributes

A Feature Selection operator. This operator adds one feature a time starting from the empty set until the required number of features *NoOfAttributes* is reached. The attributes are selected with respect to *TheClassAttribute*, the group optimises the information dependence criterion. Use this operator if only a small number of original attributes is to be selected. The selection is done from the set of *TheAttributes*, attributes not specified in this set are selected automatically.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
TheClassAttribute	BA	IN	must be categorical
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

SimpleBackwardFeatureSelectionGivenNoOfAttributes

A Feature Selection operator. This operator removes one feature a time starting from all attributes until the required number of features *NoOfAttributes* is reached. The attributes are selected with respect to *TheClassAttribute*, the group optimises the information dependence criterion. Use this operator if a large number of original attributes is to be selected. The selection is done from the set of *TheAttributes*, attributes not specified in this set are selected automatically.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

FloatForwardFeatureSelectionGivenNoOfAtt

A Feature Selection operator. This operator adds one feature a time starting from empty set until the required number of features *NoOfAttributes* is reached. The attributes are selected with respect to *TheClassAttribute*, the group optimises the information dependence criterion. Unlike the simple operator, after adding a feature a check is performed if another feature should be removed. Use this operator if only a small number of original attributes is to be selected. The selection is done from the set of *TheAttributes*, attributes not specified in this set are selected automatically.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

FloatBackwardFeatureSelectionGivenNoOfAtt

A Feature Selection operator. This operator removes one feature a time starting from all attributes until the required number of features *NoOfAttributes* is reached. The attributes are selected with respect to *TheClassAttribute*, the group optimises the information dependence criterion. Unlike the simple operator, after removing a feature a check is performed if another feature should be added. Use this operator if a large number of original attributes is to be selected. The selection is done from the set of *TheAttributes*, attributes not specified in this set are selected automatically.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA <i>list</i>	IN	see section 4.3.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

UserDefinedFeatureSelection

A Feature Selection operator. This operator copies exactly those features from *TheInputConcept* to *TheOutputConcept* that are specified in *TheSelectedAttributes*. It can be used for the same task as the operator *FeatureSelectionByAttributes*, see 4.3.3, namely when the user knows which features to select. The difference is that *FeatureSelectionByAttributes* copies all features that are present in *TheOutputConcept*, while this operator copies those that are specified in the extra parameter *TheSelectedAttributes*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheSelectedAttributes	BA list	IN	the user's selection
TheOutputConcept	CON	OUT	inherited

4.3.4 Feature construction operators

All operators in this section are loopable. For loops, *TheInputConcept* remains the same (`par_stloopnr = 0`) while *TheTargetAttribute*, *TheOutputAttribute* and further operator-specific parameters change from loop to loop (loop numbers start with 1).

AssignAverageValue

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by the average value of that Column. The operator computes the column statistics if they are not computed yet, which may take some time.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheOutputAttribute	BA	OUT	inherited

AssignModalValue

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by the modal value of that Column. The operator computes the column statistics if they are not computed yet, which may take some time.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	
TheOutputAttribute	BA	OUT	inherited

AssignMedianValue

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by the median of that Column. The operator computes the column statistics if they are not computed yet, which may take some time.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	
TheOutputAttribute	BA	OUT	inherited

AssignDefault Value

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by the *DefaultValue*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
DefaultValue	V	IN	
TheOutputAttribute	BA	OUT	inherited

AssignStochastic Value

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by a value which is randomly selected according to the distribution of present values in this attribute. For example, if half of the entries in *TheTargetAttribute* have a specific value, this value is chosen with a probability of 0.5. The operator computes the column statistics if they are not computed yet, which may take some time.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
TheOutputAttribute	BA	OUT	inherited

MissingValuesWithRegressionSVM

A MissingValue operator. Each missing value in *TheTargetAttribute* is replaced by a predicted value. For prediction, a Support Vector Machine (SVM) is trained in regression mode from *ThePredictingAttributes* (taking *TheTargetAttribute* values that are not missing as target function values). All *ThePredictingAttributes* must belong to *TheInputConcept*. *TheOutputAttribute* contains the original values, plus the predicted values where the original ones were missing.

There are some SVM-specific parameters; the table gives reasonable values to choose if nothing is known about the data or SVMs. For the *KernelType*, only the following values (Strings) are possible: *dot*, *polynomial*, *neural*, *radial*, *anova*. *Dot* is the linear kernel and can be taken as default.

This operator can use two different versions of the Support Vector Machine algorithm. One runs in main memory; it needs the parameter *SampleSize* to determine a maximum number of training examples. The other runs in the database; it is used if the optional parameter *UseDB_SVM* is set to the String *true*. When this version is used, an additional parameter *TheKey* is needed

which gives the `BaseAttribute` whose column is the primary key of *TheInputConcept*. (*TheKey* can be left out only if the `ColumnSet` that belongs to *TheInputConcept* represents a table rather than a view.) The database algorithm restricts the possible kernel types to *dot* and *radial*. It can also use the parameter *SampleSize*.

With the parameters *LossFunctionPos* and *LossFunctionNeg*, the loss function that is used for the regression can be biased such that predicting too high is more expensive ($LossFunctionPos > LossFunctionNeg$) or less expensive ($LossFunctionNeg > LossFunctionPos$) than predicting too low. If both values are equal, no bias is used. The parameter *C* balances training error against generalisation quality; positive values between 0.01 and 1000 have been used successfully in the literature. *Epsilon* limits the allowed error an example may produce; small values under 0.5 should be used.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA <i>List</i>	IN	
KernelType	V	IN	see explanation above
SampleSize	V	IN	see explanation above
LossFunctionPos	V	IN	positive real; try 1.0
LossFunctionNeg	V	IN	positive real; try 1.0
C	V	IN	positive real; try 1.0
Epsilon	V	IN	positive real; try 0.1
UseDB_SVM	V	IN	optional; one of <i>true</i> , <i>false</i>
TheKey	BA	IN	optional
TheOutputAttribute	BA	OUT	inherited

LinearScaling

A scaling operator. Values in *TheTargetAttribute* are scaled to lie between *NewRangeMin* and *NewRangeMax*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
NewRangeMin	V	IN	new min value
NewRangeMax	V	IN	new max value
TheOutputAttribute	BA	OUT	inherited

LogScaling

A scaling operator. Values in *TheTargetAttribute* are scaled to their logarithm to the given *LogBase*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
LogBase	V	IN	
TheOutputAttribute	BA	OUT	inherited

Support Vector Machine For Regression

A data mining operator. Values in *TheTargetAttribute* are used as target function values to train the SVM on examples that are formed with *ThePredictingAttributes*. All *ThePredictingAttributes* must belong to *TheInputConcept*. *TheOutputAttribute* contains the predicted values.

There are some SVM-specific parameters; the table gives reasonable values to choose if nothing is known about the data or SVMs. For the *KernelType*, only the following values (Strings) are possible: *dot*, *polynomial*, *neural*, *radial*, *anova*. *Dot* is the linear kernel and can be taken as default.

This operator can use two different versions of the Support Vector Machine algorithm. One runs in main memory; it needs the parameter *SampleSize* to determine a maximum number of training examples. The other runs in the database; it is used if the optional parameter *UseDB_SVM* is set to the String *true*. When this version is used, an additional parameter *TheKey* is needed which gives the *BaseAttribute* whose column is the primary key of *TheInputConcept*. (*TheKey* can be left out only if the *ColumnSet* that belongs to *TheInputConcept* represents a table rather than a view.) The database algorithm restricts the possible kernel types to *dot* and *radial*. It can also use the parameter *SampleSize*.

With the parameters *LossFunctionPos* and *LossFunctionNeg*, the loss function that is used for the regression can be biased such that predicting too high is more expensive ($LossFunctionPos > LossFunctionNeg$) or less expensive ($LossFunctionNeg > LossFunctionPos$) than predicting too low. If both values are equal, no bias is used. The parameter *C* balances training error against generalisation quality; positive values between 0.01 and 1000 have been used successfully in the literature. *Epsilon* limits the allowed error an example may produce; small values under 0.5 should be used.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA <i>List</i>	IN	
KernelType	V	IN	see explanation above
SampleSize	V	IN	see explanation above
LossFunctionPos	V	IN	positive real; try 1.0
LossFunctionNeg	V	IN	positive real; try 1.0
C	V	IN	positive real; try 1.0
Epsilon	V	IN	positive real; try 0.1
UseDB_SVM	V	IN	optional; one of <i>true</i> , <i>false</i>
TheKey	BA	IN	optional
TheOutputAttribute	BA	OUT	inherited

Support VectorMachineForClassification

A data mining operator. Values in *TheTargetAttribute* are used as target function values to train the SVM on examples that are formed with *ThePredictingAttributes*. *TheTargetAttribute* must be binary as Support Vector Machines can only solve binary classification problems. The parameter *PositiveTargetValue* specifies the class label of the positive class. All *ThePredictingAttributes* must belong to *TheInputConcept*. *TheOutputAttribute* contains the predicted values.

There are some SVM-specific parameters; the table gives reasonable values to choose if nothing is known about the data or SVMs. For the *KernelType*, only the following values (Strings) are possible: *dot*, *polynomial*, *neural*, *radial*, *anova*. *Dot* is the linear kernel and can be taken as default.

This operator can use two different versions of the Support Vector Machine algorithm. One runs in main memory; it needs the parameter *SampleSize* to determine a maximum number of training examples. The other runs in the database; it is used if the optional parameter *UseDB_SVM* is set to the String *true*. When this version is used, an additional parameter *TheKey* is needed which gives the *BaseAttribute* whose column is the primary key of *TheInputConcept*. (*TheKey* can be left out only if the *ColumnSet* that belongs to *TheInputConcept* represents a table rather than a view.) The database algorithm restricts the possible kernel types to *dot* and *radial*. It can also use the parameter *SampleSize*.

The parameter *C* balances training error against generalisation quality; positive values between 0.01 and 1000 have been used successfully in the literature. *Epsilon* limits the allowed error an example may produce; small values under 0.5 should be used.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited; must be binary
ThePredictingAttributes	BA List	IN	
KernelType	V	IN	see explanation above
SampleSize	V	IN	see explanation above
C	V	IN	positive real; try 1.0
Epsilon	V	IN	positive real; try 0.1
UseDB_SVM	V	IN	optional; one of <i>true</i> , <i>false</i>
TheKey	BA	IN	optional
PositiveTargetValue	V	IN	the positive class label
TheOutputAttribute	BA	OUT	inherited

MissingValueWithDecisionRules

A Missing value operator. Each missing value (NULL value) in *TheTargetAttribute* is replaced by a predicted value. For prediction, a set of Decision Rules is learned from *ThePredictingAttributes*, which must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA List	IN	
SampleSize	V	IN	positive integer
PruningConf	V	IN	between 0 and 100
TheOutputAttribute	BA	OUT	inherited

MissingValueWithDecisionTree

A Missing value operator. Each missing value (NULL value) in *TheTargetAttribute* is replaced by a predicted value. For prediction, a Decision Tree is learned from *ThePredictingAttributes*, which must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA List	IN	
SampleSize	V	IN	positive integer
PruningConf	V	IN	between 0 and 100
TheOutputAttribute	BA	OUT	inherited

PredictionWithDecisionRules

A Feature Construction operator. Decision rules are learned using *ThePredictingAttributes* as learning attributes and *TheTargetAttribute* as label. *TheOutputAttribute* contains the labels that the decision rules predict. The operator may

be used to compare predicted and actual values, or in combination with the operator `AssignPredictedValueCategorical` (see section 4.3.4). All *ThePredictingAttributes* must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA List	IN	
SampleSize	V	IN	positive integer
PruningConf	V	IN	between 0 and 100
TheOutputAttribute	BA	OUT	inherited

PredictionWithDecisionTree

A Feature Construction operator. A Decision Tree is learned using *ThePredictingAttributes* as learning attributes and *TheTargetAttribute* as label. *TheOutputAttribute* contains the labels that the decision tree predicts. The operator may be used to compare predicted and actual values, or in combination with the operator `AssignPredictedValueCategorical` (see section 4.3.4). All *ThePredictingAttributes* must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA List	IN	
SampleSize	V	IN	positive integer
PruningConf	V	IN	between 0 and 100
TheOutputAttribute	BA	OUT	inherited

AssignPredictedValueCategorical

A Missing Value operator. Any missing value of *TheTargetAttribute* is replaced by the value of the same row from *ThePredictedAttribute*. The latter may have been filled by the operator `PredictionWithDecisionRules` (4.3.4) or `PredictionWithDecisionTree` (4.3.4). It must belong to *TheInputConcept*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictedAttribute	BA	IN	
TheOutputAttribute	BA	OUT	inherited

GenericFeatureConstruction

This operator creates an output attribute on the basis of a given SQL definition (Parameter *SQL_String*). The definition must be well-formed SQL defining how

values for the output attribute are computed based on one of the attributes in *TheInputConcept*. To refer to the attributes in *TheInputConcept*, the names of the *BaseAttributes* are used—and not the names of any *Columns*. For example, if there are two *BaseAttributes* named “INCOME” and “TAX” in *TheInputConcept*, this operator can compute their sum if *SQL_String* is defined as “(INCOME + TAX)”. Since the operator must resolve names of *BaseAttributes*, it cannot be used if there are two or more *BaseAttributes* in *TheInputConcept* with the same name.

TheTargetAttribute is needed to have a blueprint for *TheOutputAttribute*. The operator ignores *TheTargetAttribute*, except that it uses the relational datatype of its column to specify the relational datatype for the column of *TheOutputAttribute*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited; specifies datatype
SQL_String	V	IN	see text
TheOutputAttribute	BA	OUT	inherited

DateToNumeric

This operator extracts numerical parts of database fields in DATE format. This is useful if you need to perform arithmetic operations on time stamps, for example when you need to represent the time as days since a given start date. Parameters are simply an *TheInputConcept*, *TheTargetAttribute* of type DATE, and *OutputFormat*, currently one of Year_YYYY, Year_YY, Month_of_Year, Day_of_Month, Hour_of_Day, Minute_of_Hour, and Second_of_Minute. The result is stored in *TheOutputAttribute*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited; type: DATE
OutputFormat	V	IN	see text
TheOutputAttribute	BA	OUT	inherited

TimeIntervalManualDiscretization

This operator can be used to discretize a time attribute manually. The looped parameters specify a mapping to be performed from *TheTargetAttribute*, a *BaseAttribute* of type *TIME* to a set of user specified categories. As for all *FeatureConstruction* operators a *BaseAttribute* *TheOutputAttribute* is added to the *TheInputConcept*.

The mapping is defined by looped parameters. An interval is specified by its lower bound *IntervalStart*, its upper bound *IntervalEnd* and two additional parameters *StartIncExc* and *EndIncExc*, stating if the interval bounds are included (value: “I”) or excluded (value: “E”). The value an interval is mapped to is given by the looped parameter *MapTo*. If an input value does not belong to any interval, it is mapped to the value *DefaultValue*.

To be able to cope with various time formats (e.g. 'HH-MI-SS') the operator reads the given format from the parameter *TimeFormat* (ORACLE-specific).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited, type: TIME
IntervalStart	V	IN	“looped”, lower bound of interval
IntervalEnd	V	IN	“looped”, upper bound of interval
MapTo	V	IN	value to map time interval to
StartIncExc	V	IN	one of “I” and “E”
EndIncExc	V	IN	one of “I” and “E”
DefaultValue	V	IN	value if no mapping applies
TimeFormat	V	IN	ORACLE specific time format
TheOutputAttribute	BA	OUT	inherited

NumericIntervalManualDiscretization

This operator can be used to discretize a numeric attribute manually. It is very similar to the operator *TimeIntervalManualDiscretization* described in 4.3.4. The looped parameters *IntervalStart*, *IntervalEnd*, *StartIncExc*, *EndIncExc*, and *MapTo*. again specify a mapping to be performed. If an input value does not belong to any interval, it is mapped to the value *DefaultValue*. *TheTargetAttribute* needs to be of type ordinal.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited, type: ORDINAL
IntervalStart	V	IN	“looped”, lower bound of interval
IntervalEnd	V	IN	“looped”, upper bound of interval
MapTo	V	IN	value to map time interval to
StartIncExc	V	IN	one of “I” and “E”
EndIncExc	V	IN	one of “I” and “E”
DefaultValue	V	IN	value if no mapping applies
TimeFormat	V	IN	ORACLE specific time format
TheOutputAttribute	BA	OUT	inherited

EquidistantDiscretizationGivenWidth

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into intervals with given width *IntervalWidth* starting at *StartPoint*. The first and the last interval cover also the values out of range.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
StartPoint	V	IN	optional
IntervalWidth	V	IN	a positive real number
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
TheOutputAttribute	BA	OUT	should be categorial

EquidistantDiscretizationGivenNoOfIntervals

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into the given number of intervals *NoOfIntervals* with the same width. The first and the last interval cover also the values out of range. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
NoOfIntervals	V	IN	integer
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
Label	V List	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

EquiprequentDiscretizationGivenCardinality

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into intervals with given *Cardinality* (number of examples whose values are in the interval). The first and the last interval cover also the values out of range. *CardinalityType* decides how the parameter *Cardinality* is to be interpreted. Values of *TheOutputAttribute* can be specified in the parameter *Label* (this makes sense only if *CardinalityType* is *RELATIVE*).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
CardinalityType	V	IN	<i>ABSOLUTE</i> or <i>RELATIVE</i>
Cardinality	V	IN	positive
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
Label	V List	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

EquiprequentDiscretizationGivenNoOfIntervals

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into the given number of intervals *NoOfIntervals*. The intervals have the same

cardinality (number of examples with values within the interval). The first and the last interval cover also the values out of range. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
NoOfIntervals	V	IN	positive integer > 1
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
Label	V <i>List</i>	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

UserDefinedDiscretization

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into intervals according to user given cutpoints *TheCutpoints*, which is a list of values which each give a cutpoint for the intervals to be created. The cutpoints must be given in ascending order. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheCutpoints	V	IN	see text
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
Label	V <i>List</i>	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

ImplicitErrorBasedDiscretization

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into intervals by merging subsequent values with the same majority class (or classes) given in *TheClassAttribute*. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. The resulting intervals minimize the classification error. If *FullMerge* is set to *YES*, then an interval with two or more majority classes is merged with its neighbour, if both intervals share the same majority class. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
FullMerge	V	IN	one of <i>YES</i> or <i>NO</i>
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

ErrorBasedDiscretizationGivenMinCardinality

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into intervals with cardinality greater or equal to *MinCardinality*. *MinCardinalityType* decides if *MinCardinality* values are read as absolute values (integers) or relative values (real, between 0 and 1). *TheTargetAttribute* is divided into intervals with respect to *TheClassAttribute*, but unlike the implicit discretization, intervals with single majority class are further merged if they do not have the required cardinality. This will increase the classification error. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
MinCardinalityType	V	IN	<i>ABSOLUTE</i> or <i>RELATIVE</i>
MinCardinality	V	IN	positive
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

ErrorBasedDiscretizationGivenNoOfInt

A discretization operator. Numeric attributes are discretized and the output is a categorial attribute. This operator divides the range of *TheTargetAttribute* into at most *NoOfIntervals* intervals. *TheTargetAttribute* is divided into intervals with respect to *TheClassAttribute*, but unlike the implicit discretization, if the number of interval exceeds *NoOfIntervals*, intervals are further merged. This will increase the classification error. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. Values of *TheOutputAttribute* can be specified in the parameter *Label*. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
NoOfIntervals	V	IN	positive integer > 1
ClosedTo	V	IN	one of <i>LEFT</i> or <i>RIGHT</i>
Label	V <i>List</i>	IN	optional
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

GroupingGivenMinCardinality

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator groups values of *TheTargetAttribute* by iteratively merging in each step two groups with the lowest frequencies until all groups have the cardinality (number of examples with values within the interval) at least *MinCardinality*. The algorithm has been inspired by hierarchical clustering. *MinCardinalityType* decides if *MinCardinality* values are read as absolute values (integers) or relative values (real, between 0 and 1).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
MinCardinalityType	V	IN	<i>ABSOLUTE</i> or <i>RELATIVE</i>
MinCardinality	V	IN	positive
TheOutputAttribute	BA	OUT	should be categorial

GroupingGivenNoOfGroups

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator groups values of *TheTargetAttribute* by iteratively merging in each step two groups with the lowest frequencies until the number of groups *NoOfGroups* is reached. The algorithm has been inspired by hierarchical clustering. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
NoOfGroups	V	IN	positive integer
Label	V <i>List</i>	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

UserDefinedGrouping

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This

operator creates groups of *TheTargetAttribute* according to specifications given by the user in *TheGroupings*, which is a list of values. Each of the values in the list in turn is a String that lists values of *TheTargetAttribute* which should be grouped together, separating them with a comma. Values not specified for grouping retain their original values. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheGroupings	V List	IN	see text
Label	V List	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

UserDefinedGroupingWithDefault Value

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator creates groups of *TheTargetAttribute* values according to specifications given by the user in *TheGroupings*, which is a list of values. Each of the values in the list in turn is a String that lists values of *TheTargetAttribute* which should be grouped together, separating them with a comma. Values not specified for grouping are grouped into default group *Default*. Values of *TheOutputAttribute* can be specified in the parameter *Label*.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
Default	V	IN	default group
Label	V List	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

ImplicitErrorBasedGrouping

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator merges the values of *TheTargetAttribute* into groups with the same majority class (or classes) given in *TheClassAttribute*. If *FullMerge* is set to yes, then a group with two or more majority classes is merged with a group that has the same majority class. The resulting grouping minimizes the classification error. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
FullMerge	V	IN	one of <i>YES</i> or <i>NO</i>
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

ErrorBasedGroupingGivenMinCardinality

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator merges the values of *TheTargetAttribute* into groups with the cardinality above the given threshold *MinCardinality*. *MinCardinalityType* decides if *MinCardinality* values are read as absolute values (integers) or relative values (real, between 0 and 1). The grouping is performed with respect to *TheClassAttribute*, but unlike implicit grouping, groups with a single majority class are further merged if they do not have the required cardinality. This will increase the classification error. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
SampleSize	V	IN	optional; positive integer
MinCardinalityType	V	IN	<i>ABSOLUTE</i> or <i>RELATIVE</i>
MinCardinality	V	IN	positive
TheOutputAttribute	BA	OUT	should be categorial

ErrorBasedGroupingGivenNoOfGroups

A grouping operator. Values of *TheTargetAttribute* are grouped under a certain label which is stored in *TheOutputAttribute*, which must be categorial. This operator merges the values of *TheTargetAttribute* into at most *NoOfGroups* groups. The grouping is performed with respect to *TheClassAttribute*, but unlike the implicit discretization, if the number of groups exceeds *NoOfGroups*, groups are further merged. This will increase the classification error. Values of *TheOutputAttribute* can be specified in the parameter *Label*. *TheClassAttribute* contains the labels of an example as in a Machine Learning setting. The parameter *SampleSize* gives a maximum number of learning examples for the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	must be numeric
TheClassAttribute	BA	IN	must be categorial
NoOfGroups	V	IN	integer > 1
Label	V <i>List</i>	IN	optional
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.3.5 Other Operators

Materialize

This operator is a normal `ConceptOperator`, but it is a pure technical construction to enforce materialization of `ColumnSets`. The table name of the output can (optionally) be specified as a parameter, which is useful if you want to access the preprocessed data afterwards. If multiple `ColumnSets` exist, then each of the corresponding table names will be extended by a numerical suffix like “_1”.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TableName	V	IN	name of output table
TheOutputConcept	BA	IN	inherited

ComputeSVMError

A special evaluation operator used for obtaining some results for the regression SVM. Values in *TheTargetValueAttribute* are compared to those in *ThePredictedValueAttribute*. The average loss is determined taking the asymmetric loss function into account. That is why the SVM parameters are needed here as well. **Note** that they must have the same value as for the operator `SupportVectorMachineForRegression`, which must have preceded this evaluation operator in the chain.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetValueAttribute	BA	IN	actual values
ThePredictedValueAttribute	BA	IN	predicted values
LossFunctionPos	V	IN	(same values
LossFunctionNeg	V	IN	as in SVM-
Epsilon	V	IN	ForRegression)

SubgroupMining

A special operator without output on the conceptual level. The output of the algorithm is a textual description of discovered subgroups which will be printed to the compiler output (log file). The operator is only applicable to a table which is suitable for spatial subgroup discovery. Thus, *ThePredictingAttributes* must

only contain categorical data. Therefore only features with a finite (and small) number of distinct values should be selected.

TheTargetAttribute and *TheKey* must belong to *TheInputConcept*; *TheKey* must refer to the primary key column. *ThePredictingAttributes* are used to learn from. *TargetValue* is one value from *TheTargetAttribute*. *SearchDepth* limits the search for generating hypotheses. *MinSupport* and *MinConfidence* give minimum values between 0 and 1 for support and confidence of the generated subgroups. *NumHypotheses* specifies the number of hypotheses to be generated. *RuleClusters* is a boolean parameter specifying whether or not clustering should be performed on the generated rules.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	
TheKey	BA	IN	
ThePredictingAttributes	BA List	IN	
TargetValue	V	IN	from TheTargetAttribute
SearchDepth	V	IN	positive integer
MinSupport	V	IN	real between 0 and 1
MinConfidence	V	IN	real between 0 and 1
NumHypotheses	V	IN	positive integer
RuleClusters	V	IN	one of YES, NO

YaleModelApplier

This operator is able to apply a model written by the learning toolbox *Yale* to an example set as given by a database table. The result is available as a database view. Please refer to section 2.4 on how to prepare integration of Yale models technically. The first thing you need to have at the operator's parameter level is a **PrimaryKey** feature in your example set view, represented by **TheInputConcept**. Unlike other operators this operator will not work correctly if the specified primary key attribute is not unique. Usually not all of the available attributes will be available for prediction, so an array of **PredictingAttributes** has to be specified. Please note that the primary key must not be part of this list. The model is referenced by an absolute path in your file system to the model file written by Yale. Finally the base attribute to be predicted (**PredictedAttribute**) and **TheOutputConcept** need to be specified.

Up to version 0.21 the order of predicting attributes needs to be the same as during learning. If you want to induce a model with Yale using a database view, then please give an explicit list of attributes in the SELECT part of the **DatabaseExampleSource** operator of Yale. The order of attributes in MiningMart is given by the order in the array **PredictingAttributes**.

It is possible to apply this operator in loops. The input and output concept, and the predicting attributes will be the same for all loops, while the model file and the output base attribute should change for each loop.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
PrimaryKey	BA	IN	a unique attribute
PredictingAttributes	BA <i>List</i>	IN	attribute list as during learning
ModelFile	V	IN	absolute path to model file
PredictedAttribute	BA	OUT	new attribute to be predicted
TheOutputConcept	CON	OUT	inherited

CreatePrimaryKey

Simple concept operator for creating a view representing the same concept with an additional primary key. If the original concept has duplicates, then the ability of this operator to remove (`SELECT DISTINCT ...`) or keep these duplicates might be interesting. This operator is also applicable to views that already have a primary key. In the database the newly created attribute will be used for indexing in the output view.

Parameters are `TheInputConcept` and `TheOutputConcept`, the `PrimaryKey` to be added, and a flag `AllowDuplicates`, indicating whether the created view should explicitly make sure that the same tuple will not appear multiple times in the output concept.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
PrimaryKey	BA	IN	the new primary key
AllowDuplicates	V	IN	one of “true” or “false”
TheOutputConcept	CON	OUT	inherited

Repeat

The repeat operator will create the same view of `TheInputConcept` as often as specified by `HowOften` in the `OutputConcept`, which leads to repeated applications of the following steps. This is e.g. useful to apply the same chain on different samples, drawn in one of the succeeding steps.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
HowOften	V	IN	number of views to create
TheOutputConcept	CON	OUT	inherited

Chapter 5

The Case Repository

One of the basic ideas behind MiningMart is the aspect of sharing knowledge about successful cases. The MiningMart project has set up a central web platform which allows the public exchange and documentation of exported cases. The platform makes use of a special software called `InfoLayer`. This chapter describes how the platform can be used to benefit from other users' work and to let others benefit from one's own work.

The web address for the case base is:

<http://kiss.cs.uni-dortmund.de:8080/mmart/>

5.1 The Internet Presentation of Cases

As soon as an efficient chain of preprocessing has been found, it can easily be exported and added to an Internet repository of best-practice MiningMart cases. Only the conceptual meta-data is submitted, so even if a case handles sensitive information, as is true for most medical or business applications, it is still possible to distribute the valuable meta-data for re-use, while hiding all the sensitive data and even the local database schema.

To support users in finding the most relevant cases, their inherent structure is exploited. An according Internet interface is accessible that visualizes the conceptual meta-data. It will be possible to navigate through the case-base and to investigate single steps, to see which operators were used on which kind of concepts. The Internet interface reads the data directly from the M4 tables in the database, avoiding additional efforts and redundancies.

Additionally to the data explicitly represented in M4, a business level has been added. This level aims at relating the case to business goals and to give several kinds of additional descriptions, like which success criteria were important for the case. This allows other users to easily relate the work done in one case to their own goals, rather than getting too much involved in technical details at an early stage. Figure 5.1 shows the ontology of the business level.

To use the internet case repository, please use an ordinary web browser

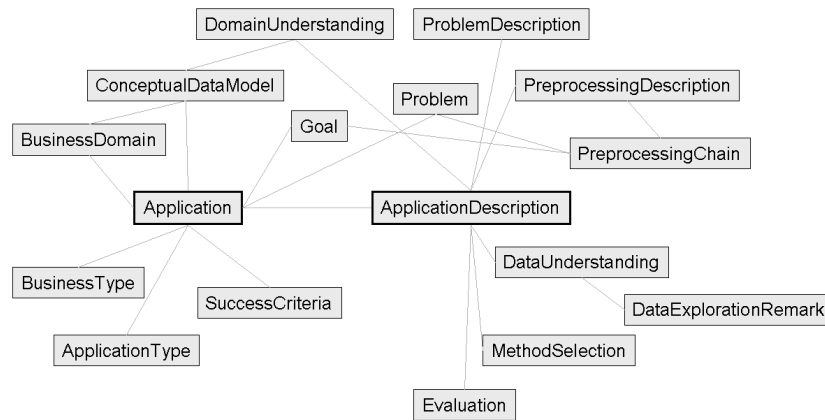


Figure 5.1: The ontology of the business layer, used to describe M4 cases in business terms.

and go to the address given at the beginning of this chapter. You can click through the metadata of the cases which are already there. The business level descriptions can be found on the same page and reached via their links to the cases present.

The following sections describe what to do if you have found a case that you would like to download and modify in your own MiningMart system, and what to do if you want to contribute a case to the internet repository.

5.2 How to download a case

In the InfoLayer there is a section called “Downloadable Case”. Here all cases in the repository are listed. If you click on the one you chose, you get a short overview description of the case together with a file. You need to download this file (usually the extension `.m4` is used to mark it as a MiningMart file).

Using your MiningMart system, you can find the menu item “Import” in your “File”-menu. You are then asked whether you would like to import only the conceptual level, or the relational level, too. Usually you will only want to import the conceptual level, especially when you have downloaded a case from the internet repository because they include only the conceptual level. After this, you are prompted with a file browsing dialogue. Choose the downloaded file. Then you can give a name to the case you are about to import. Please wait until all M4 objects are imported.

At this moment, you have access to the conceptual level of the case. If you want to execute the case or a modified version of it, you now have to link the concepts of type DB to your own database tables or views. This may mean that you have to adjust the exact form of concepts to the structure of

your database objects, or that you have to insert additional steps to the case which bring your data into a suitable format. For every concept, use the concept editor and its “connect”-function as explained in section 3.3. Then continue with the relationships between the concepts, if there are any. Once these items are connected to your database objects, you can continue by compiling the steps or making adjustments to the case.

5.3 How to document a case

For the documentation of your case, which is especially important if you want to publish its conceptual level in the internet case repository, you have two basic possibilities. First there exists a documentation or description field for every step, chain, concept, baseattribute etc. which can be edited directly in the HCI, that is, in the concept editor and the case editor. Entries made here are stored together with the metadata in M4 which means that they will be available in the InfoLayer software should the case be published. However, these documentations allow only to describe the M4 objects that make up the case. If the more general aspects of a case (its goal, way of processing, success criteria etc.) are to be documented, this can be done using the InfoLayer software on the MiningMart webpages **if** the case has been uploaded.

The next subsection describes how to upload a case to the MiningMart repository. Let us assume that this has already been done. Then the M4 objects of your case are present in the InfoLayer. You would be given a user name and password which allows you to use the editing functionality of the InfoLayer software. Click on “Login” at the low end of the left-hand side navigation bar at the web address given above (under “Administration”). Enter your user name and password. Afterwards you can add instances to the business level by clicking on “create instance” in any category. It is a good idea to start with the Business-LayerObject “Application”. From here you find links to the most important M4 and business level objects for which you can add descriptions using the “edit” button. Any description you enter will be immediately available over the web to other users. You may want to refer to figure 5.1 in this document in order to understand how the different objects in the business level are linked.

The general idea of business level descriptions is that they should allow other users to understand what the particular purpose of your knowledge discovery application was. That is, you should abstract away from technical details and describe what benefits your institution had when applying your case, what the success criteria were and so on. Other users should be able to decide whether your type of case is suitable for their own processing needs.

5.4 How to upload a case

If you have developed a successful knowledge discovery case, you have the option to let other users benefit from your work by publishing its conceptual metadata

in the internet case repository. MiningMart allows you to export all conceptual metadata into a single file. After you have opened a case, choose “Export” from the “File” menu. You are then asked whether you would like to export only the conceptual level, or the relational level, too. Usually you will only want to export the conceptual level, especially when you want to upload a case to the internet repository. The relational level would give away the structure of your business data!

You are then shown a file browsing dialogue with which you can choose a name for the exported file. It is common to use the file extension `.m4` for exported MiningMart files. Please wait until all M4 objects are exported.

You can now send the exported file to the following email address:
`mmcoord@ls8.cs.uni-dortmund.de`

The MiningMart team will then import the case into the central repository database and do some technical tests to check its consistency and executability. As soon as the case is accepted, its metadata is available on the above web address via the InfoLayer software.

Then you will be sent a user name and password and are kindly asked to fill in some general descriptions of your case in the business level of the InfoLayer. This allows other users to judge the relevance of your case for their own needs. Please refer to the explanations in section 5.3.

Bibliography

- [KVZ00] Jörg-Uwe Kietz, Anca Vaduva, and Regina Zücker. Mining Mart: Combining Case-Based-Reasoning and Multi-Strategy Learning into a Framework to reuse KDD-Application. In R.S. Michalki and P. Brazdil, editors, *Proceedings of the fifth International Workshop on Multistrategy Learning (MSL2000)*, Guimares, Portugal, May 2000.
- [KVZ01] Jörg-Uwe Kietz, Anca Vaduva, and Regina Zücker. MiningMart: Metadata-driven preprocessing. In *Proceedings of the ECML/PKDD Workshop on Database Support for KDD*, September 2001.
- [LR02] Bert Laverman and Olaf Rem. Description of the M4 Interface used by the HCI of WP12. Deliverable D12.2, IST Project MiningMart, IST-11993, 2002.
- [MS02] Katharina Morik and Martin Scholz. The MiningMart Approach. In *Workshop Management des Wandels der 32. GI Jahrestagung*, 2002.
- [MS03] Katharina Morik and Martin Scholz. The MiningMart Approach to Knowledge Discovery in Databases. In Ning Zhong and Jiming Liu, editors, *Intelligent Technologies for Information Analysis*. Springer, 2003. to appear.
- [VKZD01] Anca Vaduva, Jörg-Uwe Kietz, Regina Zücker, and Klaus R. Dittrich. M4 – the MiningMart meta model. Technical Report ifi-2001.02, Institute for Computer Science, Univ. Zürich, 2001.