Enabling End-User Datawarehouse Mining
MiningMart Version 1.0
User Guide

The MiningMart User Guide

Timm Euler, Martin Scholz

Dortmund, Germany, May 17, 2006

Contents

Chapter 1

Basic Concepts in 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 preprocessing. 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 level* 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, the connection to the database object they represent is defined by the user (with the help of the *concept editor*). When selecting a `DB` concept in the *concept editor*, the menu item 'Create Connection' is available to do so.

Other concepts, called `MINING` concepts, represent business data that was created automatically 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 described in chapter 5 lists successful cases 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. You can also nest chains, i.e. have chains inside chains.

Operator An operator performs a single, precisely defined task on the *business data.* An operator is applied by creating a *step*, setting its *parameters*, and *compiling* the step. A step's *parameters* define the input and output for an operator 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 *steps*; they define their input and output on the *conceptual level.* Some parameters that many steps need are: `TheInputConcept`, which defines the *concept* whose data is processed; `TheOutputConcept` or `TheOutputAttribute`, which define the output, ie the result of processing; 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.* 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.*

Subconcept link A concept is a subconcept of another concept if it has the same list of *features*, but the corresponding *Columnset* contains only rows that are also present in the other concept's columnset. The link between a subconcept and its super concept is displayed in the *Concept Editor* as a dashed arrow.

Projection A *concept* is a projection of another concept if all of its *features* appear in the other concept, but the other concept also has at least one additional feature. Further, the corresponding *columnset* contains only rows that are also present in the other concept's columnset (except for the additional feature(s)). A projection link is displayed in the *Concept Editor* as a dotted arrow.

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:m-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). A relation is displayed in the *Concept Editor* as a solid arrow (for 1:n-relations) or a solid double-arrow (for n:m relations).

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) this 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 4.2.9, 4.2.10 and 4.2.11 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.

Concept editor In this view you can create, view, or delete *concepts* and their *relations* on both the *conceptual* and *relational level.*

Case editor In this view you can create, view, or delete *steps*; you can arrange them into *chains* and define the input and output *parameters* of their *operators.* The case 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.

Chapter 2

Installing the MiningMart system

2.1 Support

Support for installing and working with MiningMart is available in this document and some `README` files in the distribution. Further, support questions can be addressed by email to:

`miningmart@ls8.cs.uni-dortmund.de`

The MiningMart team, consisting of Timm Euler and Martin Scholz, will be happy to answer your specific questions.

2.2 General issues

Unpacking the zip file you have downloaded from the MiningMart website, or acquired elsewhere, will create a directory named `MiningMart-1.0`. The complete path up to and including this directory is abbreviated in this and all other MiningMart documentation by `MM_HOME`. Assume that somewhere in the documentation, the file `MM_HOME/lib/postgresql.jar` is mentioned. This indicates that you can find the file `postgresql.jar` in the directory

- `C:\Programs\MiningMart-1.0\lib\`, to use an example for Windows;
- `/home/username/analysis/MiningMart-1.0/lib/`, as an example for Linux or Solaris.

MiningMart works with relational databases. Two database management systems (DBMS) are supported: Oracle (version 8.1.6 or higher) and PostgreSQL (version 7.2 or higher). Although MiningMart uses standard SQL wherever possible, access to certain important database system tables is not standardised among DBMS vendors, so that adapting MiningMart to other DBMS is not difficult but has not been done yet.

MiningMart distinguishes between *business data* and its own *M4 data* (see also chapter 1). The business data is the collection of tables you want to preprocess or analyse. One specific database schema should be used for this data. This schema is called *business schema* everywhere in this documentation. For information on database schemas, and how to create and manipulate them, please refer to your Oracle or PostgreSQL documentation. It is useful to create a second database schema for the M4 data. This M4 schema should be empty to start with, and will be filled with MiningMart-specific tables (see section 2.5).

With these hints you should be able to install MiningMart easily by following the instructions in section 2.3, which are also to be found in the files

- `MM_HOME/README`
- `MM_HOME/INSTALL`

2.3 Installing and running MiningMart

Follow these steps:

1. You ought to read the file `MM_HOME/README`.
2. You must install JAVA on your system. MiningMart is tested for Java 1.4 and 1.5. For more information see `http://java.sun.com`.
3. Your Oracle or Postgres installation provides a JDBC driver. This is a file which a Java program can use to read and write from the database. Your Oracle or Postgres documentation will tell you where to find the file. The Oracle file is often called 'classes12.zip' or similar. The Postgres file is often called 'postgresql.jar'.

   As an Oracle user, please place a copy of that file into MM_HOME/lib/. If you are using Linux or Solaris, that's it. If you are using Windows, you may now have to edit one line in the file `MM_HOME\bin\mmart.bat`: you will find the line `set ORA=%MMART_HOME%\lib\classes12.zip` in this file. Change the line so that the file name is correct, ie matches the Oracle JDBC driver file name.

   If you are going to use postgres: A Postgres JDBC file is already provided in `MM_HOME/lib/` under the license described in `MM_HOME/licenses/LICENSE_POSTGRESQL_JDBC`. However, you may still have to find the JDBC file that belongs to your own Postgres installation if you are experiencing troubles (particularly for Postgres versions 8.x or higher). Just copy the file to `MM_HOME/lib/`. If you are using Linux or Solaris, that's it. If you are using Windows, you may now have to edit one line in the file MM_HOME\bin\mmart.bat: you will find the line `set POSTGR=%MMART_HOME%\lib\postgresql.jar` in this file. Change the line so that the file name is correct, ie matches the Postgres JDBC driver file name.

4. Open and edit the file `MM_HOME/bin/mmart.sh` for Linux or Solaris, or the file `MM_HOME\bin\mmart.bat` under Windows. Follow the instructions given in comments in that file: there is only one obligatory line to edit. The line looks like this before you have changed it:

   `MMART_HOME=..` or `set MMART_HOME=.\..`

   After you have changed it, it should look something like this:

   `MMART_HOME=/home/myname/analysis/MiningMart-1.0/` (Linux/Unix) or

   `set MMART_HOME=C:\Programs\Analysis\MiningMart-1.0\` (Windows)

   Save the edited version of the file under the same name.

5. Make sure that the file mentioned in step 4 is executable. Under Linux or Solaris the command `chmod u+x mmart.sh` will ensure this. Under Windows, right-click on the file and change its properties.

6. That's all. You can now run MiningMart by issuing the command `./mmart.sh &` from the `MM_HOME/bin` directory under Linux/Solaris, or by double-clicking on the file `MM_HOME\bin\mmart.bat` under Windows.

2.4 Licensing information

Please refer to the file `MM_HOME/LICENSES`.

2.5 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. You can use the same scheme for both purposes if you want. The standard installation foresees a separate schema for M4 and business data in the same database.

When you start MiningMart for the first time, the system will create a number of tables in the M4 schema which it uses to store the cases etc, unless these tables exist already (from an older MiningMart version). If any table in the M4 schema happens to have a name that is used by MiningMart, that table will be deleted! If you want to make sure that none of your tables are deleted, you can check the file `MM_HOME/m4install/oracle/CreateM4Tables.sql` for the table names that are going to be used.

You can also choose to prepare the M4 tables yourself, by using the scripts in the directories `MM_HOME/m4install/oracle/` or `MM_HOME/m4install/postgres/`. Note that when running the provided scripts that install M4 tables to be used by MiningMart into your M4 database schema, any old tables with identical names are deleted. If you have used MiningMart before, this will delete all your cases! Please refer to section 2.7 to learn about saving your old cases.

Please refer to the file `MM_HOME/m4install/README.txt` for further information on how to prepare the database manually, or simply start the system if you want the system to prepare the database automatically.

2.6 Integration with YALE

To enable a smooth integration between preprocessing, learning, and applying models, two new operators called *PrepareForYale* and *YaleModelApplier* have been implemented to bridge the gap between MiningMart and the very powerful open-source *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. This can easily be done with the help of the *PrepareForYale* 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 *YaleModelApplier* operator 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/

2.7 Upgrading from older versions

This section is relevant if you have old MiningMart cases 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. Otherwise you should *export* all your cases to XML files using the export facility of your old system. Then you may delete your M4 schema and install it anew (see section 2.5). Using the import facility of the new system, you can then re-create all your cases.

However, you do not need to delete your cases. Simply run the script

- `MM_HOME/m4install/oracle/replace_operators.sql`, or
- `MM_HOME/m4install/postgres/replace_operators.sql`,

depending on your database system. Start these scripts from an SQL command line in the M4 database schema. This will update the information about MiningMart's operators without changing the M4 schema.

2.8 External code

Although MiningMart was implemented in Java, some operators use external algorithms, not all of which are included in all distributions or for all operating systems. A list of operators using external algorithms can be found in the appendix (section 2.9).

The operators that use these external implementations all employ machine learning algorithms. The MiningMart team cannot give support for these external implementations. If these operators do not work on your system, it is suggested to use the very powerful YALE learning environment (see section 2.6) for all machine learning experiments. MiningMart provides two operators that ease the integration with YALE (see section 2.6).

If your installation does not include them, binary files for the external operators can be put into the directory `MM_HOME/ext/bin/<os-name>/`, where `<os-name>` is to be replaced with one of `Windows`, `Linux` or `SunOS`.

Currently the following tools are supported if put to the directory specified above:

- *mySVM*, a Support Vector Machine by Stefan Rueping
  For details please refer to
  http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/
  The wrapper expectes the target files mysvm and predict.

- *Apriori*, implemetation of Bart Goethals
  For details please refer to
  http://www.cs.helsinki.fi/u/goethals/software/index.html
  The wrapper expects the binary apriori.

2.9 Appendix: List of operators that use external algorithms

- Apriori
- FeatureSelectionWithSVM
- MissingValuesWithRegressionSVM
- SupportVectorMachineForRegression
- SegmentationWithKMean

Chapter 3

The Graphical User Interface

This chapter explains the main issues for handling MiningMart as a user. Users should read chapter 1 and section 4.1 before they start to work with the system. MiningMart provides two views on the data transformation process: one is centred on the steps and their dependencies, the other focuses on the data sets and semantic links between them. The former view is called the *Case editor* and is described in section 3.2. The latter view is called the *Concept editor* and is described in section 3.3. Before describing these editors, section 3.1 gives a few general hints.

3.1 General issues

The central notion in MiningMart is the *Case*; it can be compared to a file in text editors or other applications, except that only one case can be open at any time. The `Case` menu therefore offers options to open and save an existing case or to create a new case. Cases are stored in the database. Cases can be exported from the database to files, or imported from files to the database. Deleting a case means to remove it from the database. The menu item `Switch editor` can be used to change from the case editor to the concept editor or back. The menu items `Print` and `Export image` allow to print the contents of the main working area (the central area) to postscript or `*.png` files.

The `Insert` menu will be explained in sections 3.2 and 3.3.

The options of the `Compile menu` are only available when the Case editor is opened; see section 3.2.

The `Tools` menu allows to change the connection settings by which MiningMart connects to the database. This should not be done when a case is opened.

The `Windows` menu offers different options. `Grid` toggles between visible and invisible states of a grid on the working pane. `Look'n'Feel` allows to change the look and feel of the graphical interface. `Show` indicates which of the possible elements for the right-hand frame are displayed. `Presentation` creates a window that gives maximal space to the contents of the working pane. The most important item in this menu is `Preferences`. When you choose it, a new window opens with three areas. The first displays all available operators (for your overview; they are also available in the `Insert` menu and the context menu, see section 3.2). The second allows you to change the verbosity level of the log output. The different levels are given names which indicate the kinds of operations the system logs if that level or a lower level is chosen. The higher you go in the list, the less output is produced in the log window and log file. Finally, the third area allows you to choose the types of links between concepts that the concept editor should display. This choice is applied from the next time that the concept editor is opened onwards. See also section 3.3.

The `Help` menu opens the MiningMart help system (menu item `Contents`) which allows you to browse through help topics. You can also start a web browser if your system is configured to allow this. The `About` option displays copyright information.

The following list provides some general hints which may help you getting familiar with the handling of MiningMart.

- Every case must have one top-level chain to which all its steps and other chains belong. So when creating a new case, please create a new chain first (using the `Insert` menu), then you can start creating steps inside this chain. Open the chain by double-clicking on it.

- When creating a new step, you can view and edit its parameters by double-clicking it. When editing a text-field parameter (by typing), please remember to press 'Enter' after editing, otherwise the new value is not recognised.

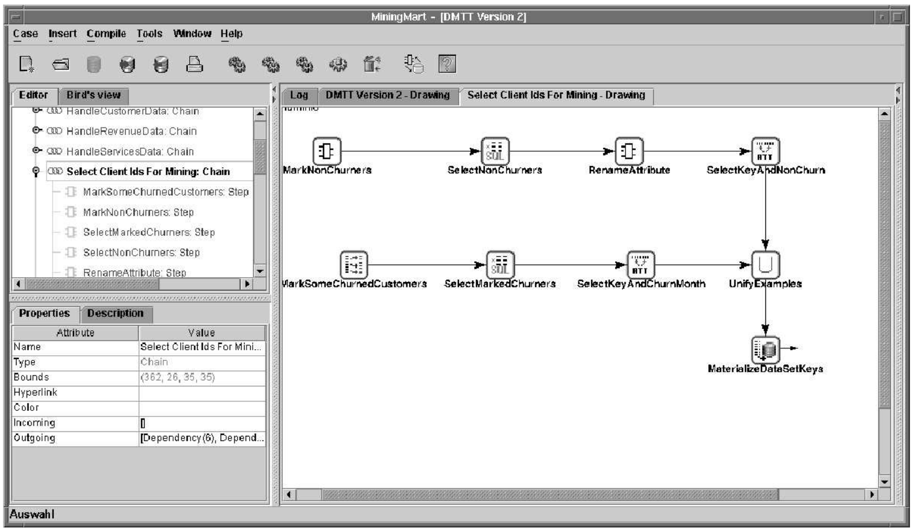
- It is a good idea to save the case you are editing regularly, in particular before starting the compiler.

- In general, icons in both editors can be freely moved around, allowing to create a clear web structure of steps or concepts. For large cases the "Bird's view" can be very helpful in both editors.

- Every MiningMart item can be documented. There is a `Description` field in the lower left corner of the working area which always applies to the currently selected element. You can enter any text here, it will be saved and displayed whenever the element is selected again. For BaseAttributes, the description field can be found in the lower right corner of the working area when a BaseAttribute is selected in the concept editor (see 3.3).

3.2 Case editor

In this editor you mainly create *Steps*. Each step uses one operator that transforms the input data and creates an output representation. The output representation can be inspected in the concept editor (section 3.3). The output is not actually created in the database until the step is *compiled*.



Attribute	Value
Name	Select Client Ids For Mini...
Type	Chain
Bounds	(362, 26, 35, 35)
Hyperlink	
Color	
Incoming	[]
Outgoing	[Dependency(6), Depend...

Figure 3.1: The case editor.

A step is created by choosing the operator it employs, either from the context menu which appears when you right-click into the working tab, or from the menu item `Insert/Operator`.

Steps are connected by *Transitions*. If two steps are not connected, the output of one is not available as input for the other. The menu item `Insert/Transition` can be used to create a transition from one step to the next. Figure 3.1 shows a screenshot of the case editor with steps and transitions.

Double-clicking on a step, or choosing `Insert/Show StepSettings`, will open the parameter view for this step on the right hand side of the working area. You can edit the parameters of the selected step here. You should read at least section 4.1 and the section in chapter 4 that corresponds to the operator your step is using, in order to understand how the parameters work and what values they should be given. The `Valid` button can be used to check the syntactical validity of the parameters; for example, if an obligatory parameter is missing, an error message will be given when this button is clicked.

Steps can be collected in *Chains*. Despite their names, chains can not only contain linear sequences of steps but any transition structure. Every case **must** have exactly **one** top level chain; when you create a new case, the first thing you have to do therefore is to create a chain using the menu item `Insert/Chain/Chain`. All other steps and chains are subelements of this chain. Double-clicking on a chain opens a new working tab in which its steps and subchains are displayed. Chains can be created by selecting any number of steps (using the mouse to mark a rectangular area of selection) and by then choosing `Insert/Chain/Merge Chain`. A chain can be dissolved by selecting it and choosing `Insert/Chain/Un-`

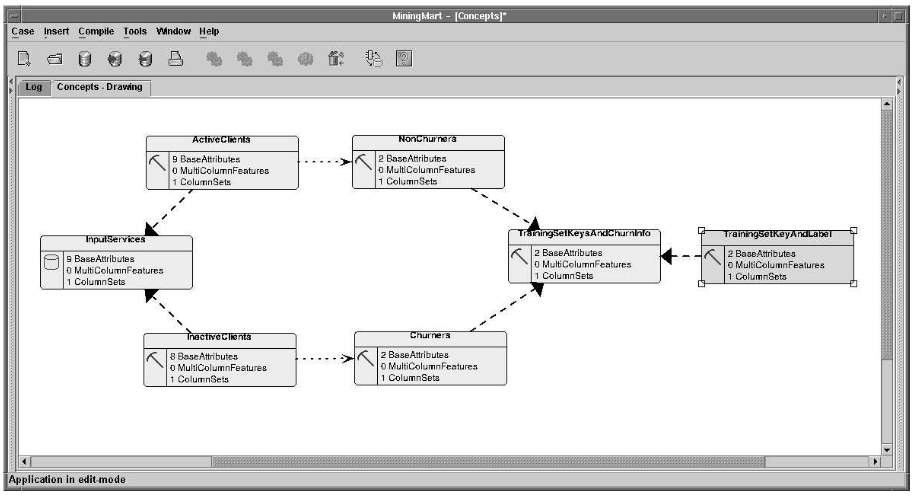


Figure 3.2: The concept editor.

merge Chain.

The OperatorTools pane on the right hand side of the working area provides shortcuts for most of these tasks.

Right-clicking on a chain and choosing Show concepts involved opens the concept editor, but displays only the concepts used as input or output in any step of that chain. This allows to keep a clearer overview of the concepts in large cases.

When a step and its parameters are fully specified, it can be *compiled.* Compilation means to create the relational level output (in the database) that corresponds to the output concepts that have been created in MiningMart. See also chapter 1 and section 3.3. The Compile menu offers various options to compile a single step or a number of steps. Compilation results are logged in the log tab and file. Error messages are displayed. An error immediately stops the compilation. Choosing the menu item Garbage collection removes the compilation results from the database. The flag Compile in lazy mode can be used to make the compiler create only one output *Columnset* when several *Columnsets* are expected; see the paragraph on compilation in section 1.2. The menu item Kill compile can be used to stop the current compiler run.

3.3 Concept editor

Concepts represent data tables or views in MiningMart and the concept editor is the tool to create, manipulate or delete them. Figure 3.2 shows a screenshot of the concept editor.

In MiningMart there are two types of concepts: `DB` and `Mining`. The former are indicated by a small database icon; the latter show a tool icon. The `DB` concepts are used to represent tables or views in the database which have existed before MiningMart was started. The `Mining` concepts represent tables or views created with MiningMart. Consequently, `DB` concepts have to be *connected* to database objects manually, while this connection is set up automatically *by the compiler* for `Mining` concepts. A concept is connected if it is associated with at least one *Columnset*; the latter directly represents a database table or view.

There are two ways of creating a `DB` concept. The first is to choose `Insert/New concept`, then give the name for the concept, and then add BaseAttributes to it. Adding BaseAttributes is done by right-clicking on the concept icon, choosing `Show BaseAttributes` and then using the `New` button to add single BaseAttributes. The second way of creating a `DB` concept is to choose `Insert/Concept from Table` and then choose directly the database table or view this concept is going to represent. Then the names of the concept and its BaseAttributes will be copied from the names of the database table or view and its columns. The advantage is that a concept created in this way is immediately connected; the disadvantage is that the names may have to be changed manually if the ones from the database objects are unsuitable.

A `DB` concept that is not connected to a columnset shows a red database icon; a connected one shows a green one. Further, the number of columnsets is indicated in the concept icon; the number 0 indicates a missing connection to columnsets. Only `DB` concepts can be manually connected to database objects (represented by columnsets). To this end, right-click on the concept icon and choose `Create connection`. This will display a tab in the right hand area which allows you to choose the database table or view for the connection. Now you have two options: (i) Select one table or view and click `Create`. This will create the columnset, and you will have to connect each BaseAttribute of the concept to zero or one column. (ii) Select one or more tables or views, select a Matcher and click `Match`. This will also create the columnset, and MiningMart will use an automatic matching method that depends on the selected matcher to try to connect the BaseAttributes to the columns automatically. This automatic matching is based on the *names* of the BaseAttributes and columns, which means that if the names are not similar at all then no suitable matching is likely to be found. But you can change the mapping of BaseAttributes to columns easily by hand afterwards, and in fact at any time by right-clicking on a concept's icon and choosing `Show mapping`. If you click on a column in the mapping tab on the right hand side, the list of available columns is displayed and the desired one can be chosen.

When a concept is connected, right-clicking on it and choosing `Show Columnsets` displays the columnsets attached to it. The buttons `Show data` and `Show statistics` can be used to inspect the data that is contained in the table or view represented by this columnset, and some statistical information about it. For views and virtual columns (the latter are created by the compiler) there is an `SQL` field that displays the SQL code which realises this view or virtual column.

`Mining` concepts are not usually created by hand but by creating a step in the case editor whose output is a concept. Such output concepts are always `Mining` concepts, and their connection to database objects can only be realised by compiling the steps whose output they are.

Depending on which operator created a `Mining` concept, there are often some semantic links to the input concept. In MiningMart the three types of semantic links between concepts are subconcept links (represented by dashed arrows), projections (represented by dotted arrows) and relationships (represented by solid arrows); see section 1.2. In the standard setting, MiningMart always displays all three types of links, but you can choose which ones to display in the `Window/Preferences/Concept links` menu item. These links can also be created by hand using the icons in the `ConceptTools` tab on the right hand side. They can be removed by right-clicking on a link and choosing `Delete`.

Relations (or relationships) represent foreign key links between tables or views. The two types “one-to-many” (1:n) and “many-to-many” (n:m) are distinguished. When creating a relationship by hand, the BaseAttributes of the two concepts involved which correspond to the database columns that realise the keys must be specified. For many-to-many relationships, also the cross table must be chosen from the available database objects, and its key columns must be specified.

Chapter 4

Operators and their Parameters

This chapter explains the operators currently available in MiningMart. Operators form the basic building blocks of the data preprocessing phase in the Knowledge Discovery process. There are a number of operators for different purposes, as explained below.

4.1 General issues

There are mainly two kinds of operators, distinguished by their output on the conceptual level: those that have an output Concept (*Concept Operators*, listed in sections 4.2 and 4.3), and those that have an output BaseAttribute (*Feature Construction Operators*, listed in section 4.4). Further there are some special operators without any output on the conceptual level, listed in section 4.5.

All operators have parameters, such as input Concept or output BaseAttribute. The parameters must be instantiated (i.e. given values) for every step that uses the operator. The name of such a parameter is fixed for the operator and cannot be changed. For instance, *TheInputConcept* is used for the input Concept for all operators.

Parameter arrays

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*). Such parameters, which may contain a list, are marked with the word *List* in the operator descriptions below.

Coordinated parameters

Sometimes several parameters are realised as arrays (see previous paragraph), and their semantics require that the values of two or more parameters are related to each other *for the same position* in the arrays. For example, the grouping operators have parameters that specify which value(s) of an input attribute are mapped to which value of the new output attribute. These operators can map several (groups of) values to new values; which values are mapped to which others is then specified by using the same position in the parameter arrays. In the GUI such parameters are highlighted by blue frames; the button "Show groups" can be used to verify that the values of the parameters are sorted in the right way. This button is only available if any coordinated parameters are available.

Loops

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. For each loop the target attribute, the output attribute and other parameters (namely those marked as "looped" below) can be different, but the input concept is the same in all loops.

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, 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.

4.2 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.

4.2.1 MultiRelationalFeatureConstruction

Takes a list of concepts which are linked by relationships, 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: Relationships 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 Relationship 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, the BaseAttributes that correspond to the Relationship 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

4.2.2 JoinByKey

Takes a list of concepts, plus attributes indicating their primary keys, and joins the concepts. In *TheOutputConcept*, only one of the keys will 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 coordination mechanism (see section 4.1). 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 "maps" as there are features in *TheOutputConcept*.

ParameterName	ObjectType	Type	Remarks
TheConcepts	CON *List*	IN	no *TheInputConcept*!
TheKeys	BA *List*	IN	
MapInput	BA or MCF	IN	coordinated
MapOutput	BA or MCF	OUT	coordinated
TheOutputConcept	CON	OUT	inherited

4.2.3 UnionByKey

Takes a list of concepts, plus attributes indicating their primary keys, and unifies the concepts. In contrast to the operator JoinByKey (section 4.2.2), the output columnset is a union of the input columnsets rather than a join. For each value occuring 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 will 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 coordination mechanism (see section 4.1). 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 "maps" as there are features in *TheOutputConcept*.

ParameterName	ObjectType	Type	Remarks
TheConcepts	CON *List*	IN	no *TheInputConcept*!
TheKeys	BA *List*	IN	
MapInput	BA or MCF	IN	"looped"!
MapOutput	BA or MCF	OUT	"looped"!
TheOutputConcept	CON	OUT	inherited

4.2.4 Pivotize

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

This operator (Pivotize) 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 optionally 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 NONE, SUM, MIN, MAX, AVG (average) and COUNT. If NONE is chosen no aggregation will take place. NONE cannot be chosen if any GroupBy-attributes are given (the parameter *TheGroupByAttributes* is optional).

The parameter *NullOrZero* specifies whether NULL or 0 should be used whereever 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 *List*	IN	optional
IndexValue	V	IN	“looped”!
MappedAttribute	BA	OUT	“looped”!
AggregationOperator	V	IN	SUM or MIN etc.
NullOrZero	V	IN	one of *Null*, *Zero*
TheOutputConcept	CON	OUT	inherited

4.2.5 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 `BaseAttribute`s 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 the corresponding 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* will contain the corresponding number of BaseAttributes (three in the example); their names will 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

4.2.6 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)

4.2.7 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

4.2.8 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

4.2.9 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). The specified attribute will not be present in the output concept.

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

4.2.10 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

4.2.11 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	*true* or *false*
ThePredictingAttributes	BA *List*	IN	
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

4.2.12 UnSegment

This operator is the inverse to any segmentation operator (see 4.2.9, 4.2.10, 4.2.11). 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 4.2.9), this parameter gives the name of the BaseAttribute that was used for stratified segmentation. Note that this BaseAttribute will 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 4.2.10), this parameter must have the value "(Random)".

If a SegmentationWithKMean operator is reversed (section 4.2.11), 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

4.2.13 RemoveDuplicates

This operator produces an output concept that is a copy of the input concept, but all duplicate entries are removed from the corresponding ColumnSet.

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

4.2.14 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

4.2.15 Materialize

This operator is a normal concept operator, 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 are connected to *TheInputConcept*, 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

4.2.16 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. The first thing you need to have 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.

The order of predicting attributes must 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 *List*	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

4.2.17 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

4.2.18 Windowing

Windowing is applicable to time series data. It takes two BaseAttributes from the input Concept; one of them 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 `BaseAttribute`s that belong to *TheInputConcept*, *OutputTimeStartBA*, *OutputTimeEndBA* and the *WindowedValuesBA*s 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 *WindowSize*
TheOutputConcept	CON	OUT	inherited

4.2.19 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 4.2.18.

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

4.2.20 WeightedMovingFunction

This operator works like SimpleMovingFunction (section 4.2.19), 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

4.2.21 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

4.2.22 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

4.2.23 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 `BaseAttribute`s 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 `BaseAttribute`s, *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

4.2.24 Feature Construction with TF/IDF

This operator calulates 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 *List*	IN	attribs to calc. TFIDF for
TheTimeStamp	BA	IN	type TIME
TheKey	BA	IN	key attribute
TheOutputConcept	CON	OUT	inherited

4.2.25 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” *TheInputConcept*, which specifies the `BaseAttributes` of the *TheOutputConcept*. If features of *TheInputConcept* are deselected, then the features will also be deselected in the output. 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 occurences 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 *List*	IN	Union compatible Concepts
DataMode	V	IN	set or multi-set mode?
TheOutputConcept	CON	OUT	inherited

4.3 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 4.3.1) 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 `Column`s 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 4.3.1.) 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.

4.3.1 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

4.3.2 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 *list*	IN	see section 4.3
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 *true*, *false*
TheKey	BA	IN	optional
SearchDirection	V	IN	one of *forward*, *backward*
TheOutputConcept	CON	OUT	inherited

4.3.3 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 *list*	IN	see section 4.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

4.3.4 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 *list*	IN	see section 4.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

4.3.5 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 *list*	IN	see section 4.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

4.3.6 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 *list*	IN	see section 4.3
TheClassAttribute	BA	IN	must be categorial
NoOfAttributes	V	IN	positive integer
SampleSize	V	IN	positive integer
TheOutputConcept	CON	OUT	inherited

4.3.7 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 4.3.1, namely when the user knows which features to select.

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

4.4 Feature construction operators

Almost all operators in this section are loopable. For loops, *TheInputConcept* remains the same while *TheTargetAttribute*, *TheOutputAttribute* and further operator-specific parameters can change from loop to loop (loop numbers start with 1). See also section 4.1.

4.4.1 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

4.4.2 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

4.4.3 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

4.4.4 AssignDefaultValue

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

4.4.5 AssignStochasticValue

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

4.4.6 Binarify

This operator is often called “Dichotomisation”. It introduces one new attribute per loop. The new attribute takes only two values, 1 and 0. The value 1 is taken whenever the target attribute takes the target value, otherwise the value 0 is taken. Thus the new attribute is a binary indicator (or a boolean flag) showing the presence or absence of the target value in the target attribute for each row in the input concept. By looping this operator, several target values can be used, so that several new boolean attributes are created.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
TargetValue	V	IN	looped!
TheOutputAttribute	BA	OUT	looped!

4.4.7 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*. You can only use it if you have the DB version of MYSVM installed.

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 *List*	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 *true*, *false*
TheKey	BA	IN	optional
TheOutputAttribute	BA	OUT	inherited

4.4.8 LinearScaling

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

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

4.4.9 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

4.4.10 SupportVectorMachineForRegression

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*. You can only use it if you have the DB version of MYSVM installed.

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 *List*	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 *true*, *false*
TheKey	BA	IN	optional
TheOutputAttribute	BA	OUT	inherited

4.4.11 SupportVectorMachineForClassification

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*. You can only use it if you have the DB version of MYSVM installed.

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 *true*, *false*
TheKey	BA	IN	optional
PositiveTargetValue	V	IN	the positive class label
TheOutputAttribute	BA	OUT	inherited

4.4.12 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 `BaseAttribute`s are used—and not the names of any `Column`s. For example, if there are two `BaseAttribute`s named “INCOME” and “TAX” in *TheInputConcept*, this operator can compute their sum if *SQL_String* is defined as “(INCOME + TAX)”.

TheTargetAttribute is needed to have a blueprint for *TheOutputAttribute*. The operator ignores *TheTargetAttribute*, except that it uses its conceptual data type, and the relational data type of its column, to specify the corresponding data types for *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

4.4.13 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 `TIME`, 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

4.4.14 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 *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*.

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

4.4.15 NumericIntervalManualDiscretization

This operator can be used to discretize a numeric attribute manually. It is very similar to the operator TimeIntervalManualDiscretization described in 4.4.14. 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* must 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

4.4.16 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 *LEFT* or *RIGHT*
TheOutputAttribute	BA	OUT	should be categorial

4.4.17 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 *LEFT* or *RIGHT*
Label	V *List*	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

4.4.18 EquifrequentDiscretizationGivenCardinality

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	*ABSOLUTE* or *RELATIVE*
Cardinality	V	IN	positive
ClosedTo	V	IN	one of *LEFT* or *RIGHT*
Label	V *List*	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

4.4.19 EquifrequentDiscretizationGivenNoOfIntervals

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 *LEFT* or *RIGHT*
Label	V *List*	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

4.4.20 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 *LEFT* or *RIGHT*
Label	V *List*	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

4.4.21 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 *LEFT* or *RIGHT*
FullMerge	V	IN	one of *YES* or *NO*
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.4.22 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	*ABSOLUTE* or *RELATIVE*
MinCardinality	V	IN	positive
ClosedTo	V	IN	one of *LEFT* or *RIGHT*
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.4.23 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 *LEFT* or *RIGHT*
Label	V *List*	IN	optional
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.4.24 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	*ABSOLUTE* or *RELATIVE*
MinCardinality	V	IN	positive
TheOutputAttribute	BA	OUT	should be categorial

4.4.25 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 *List*	IN	optional
TheOutputAttribute	BA	OUT	should be categorial

4.4.26 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

4.4.27 UserDefinedGroupingWithDefaultValue

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

4.4.28 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 *YES* or *NO*
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.4.29 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	*ABSOLUTE* or *RELATIVE*
MinCardinality	V	IN	positive
TheOutputAttribute	BA	OUT	should be categorial

4.4.30 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 *List*	IN	optional
SampleSize	V	IN	optional; positive integer
TheOutputAttribute	BA	OUT	should be categorial

4.5 Other Operators

4.5.1 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)

4.5.2 PrepareForYale

This operator has no output for MiningMart. It produces an XML file that can be read by the YALE software. The file will be given the name provided in *ExperimentFileName*. In YALE, this will provide a very basic experiment that starts by reading data from the database, that is, from the business data schema used by MiningMart. This operator makes the combination of YALE and MiningMart more convenient.

If *TheInputConcept* to this operator has more than one Columnset, the YALE file produced will contain the YALE operator *IteratingOperatorChain*, and for each Columnset a query file will be produced that is read in one iteration of the operator chain in YALE. The parameter *QueryFilePrefix* is used for this purpose; the query files produced will have a number added to the given prefix, starting with 1. If there is only one Columnset in *TheInputConcept*, the parameter *QueryFilePrefix* is ignored. Both filename parameters expect full paths.

The parameters *TheLabel* and *ThePrimaryKey* are optional. They provide information to YALE about the label (for learning) and the primary key of the data set. Of course, this information can also be added manually in YALE.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	
TheLabel	BA	IN	optional
ThePrimaryKey	BA	IN	optional
ExperimentFileName	V	IN	complete path and filename
QueryFilePrefix	V	IN	only for multiple columnsets

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. 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 case base can be found on the MiningMart web site:
`http://mmart.cs.uni-dortmund.de`
The direct link is:
`http://mmart.cs.uni-dortmund.de/caseBase/index.html`

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 level 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 ideas 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. The internet interface visualizes the conceptual elements like chains, steps and concepts as web pages, linked by HTML links if the corresponding elements are connected in the case. It is possible to navigate through the casebase and to investigate single steps, to see which operators were used on which kind of concepts, to see which concepts were used as input or output in which steps, which features belong to a concept, and so on.

To each case some free text descriptions giving an overview of the case and the application domain can be added. 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.

To use the internet case repository, please use an ordinary web browser and go to the address given at the beginning of this chapter. You can click through the structure of the cases which are already there.

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

On the MiningMart case base start page (see above) there is a list of cases. Clicking on one case name brings up the main starting page for that case. There is a bulleted item called "Exported into M4 file"; clicking on the file name will make your browser download the file.

The file may have to be unpacked if its name ends with ".zip".

After unpacking the file, you can import it into your MiningMart system. There is a menu item "Import Case" in the "Case" menu. Using it will allow you to select the file, and give a name to the case which is to be imported. Then the case is available in your MiningMart system. You should save it if you want to keep it, since importing the case does not automatically save it to your database.

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 of type `DB` (the ones with a small database symbol in their icon), use the concept editor and its "Create connection"-function. Then continue with the relationships between the concepts, if there are any. Once these items are connected to your database tables or views, 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 (see following section), there is a description field for every step, chain, concept, baseattribute etc. which can be edited directly in the GUI. It can be found in the lower left frame of the MiningMart GUI; there is a special tab "Description" which applies to the currently selected element. For the attributes of a concept there is an extra description field in the right hand frame when the BaseAttributes of a concept are displayed.

Any descriptions entered in these fields are shown in the internet repository if you publish your case there. They help other users to find out about details of your case.

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 level 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 Case” from the “Case” menu.

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 `.xml` for exported MiningMart files, since these files use an XML syntax. Please wait until all M4 objects are exported.

You can now send the exported file to the following email address:
`miningmart@ls8.cs.uni-dortmund.de`
The MiningMart team will then do some technical tests to check the consistency of the case. You will be kindly asked to provide some background information about the application domain etc. per email. Then the case will be added by MiningMart administrators to the central case base, and will be available under the web address above.