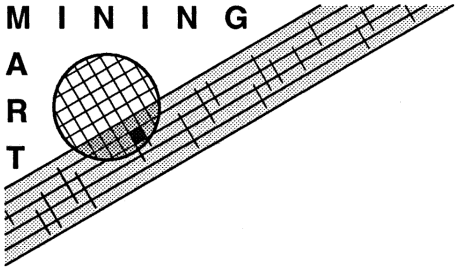

Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D18

Applicability Constraints on Learning Operators

Martin Scholz[1], Timm Euler[1],
and Lorenza Saitta[2]

[1] University of Dortmund, Computer Science
D-44227 Dortmund, Germany
{scholz,euler}@ls8.cs.uni-dortmund.de

[2] Dipartimento di Informatica, Università del Piemonte Orientale
Spalto Marengo 33, Alessandria 15100, Italy

December 20, 2002

Chapter 1

Objectives

1.1 Introduction

The MiningMart project focuses on preprocessing chains for successful data mining. Each step of a chain embeds an operator, applied to one or more input tables or views. The output of a step depends on the operator, but in all cases it is a view, a table, or a virtual column added to a view or table.

The variety ranges from so called *manual* operators to much more complex *Machine Learning* operators The former just write the result of their application as an SQL statement on the meta-level, usually without reading the business data. The latter are integrated by means of a wrapper approach. A sample of tractable size is drawn from the database and a stand-alone learning algorithm is called. The result is found in an output file or read from standard output, needs to be parsed, and is then written to the database. For most operators the result is stored as an SQL function, called by an SQL statement similar to those produced by manual operators.

For both manual and Machine Learning operators there are some conditions to a successful application, subject to this deliverable. Before the formalized operator conditions of this work package are described in the next chapters, alternative approaches together with availability of necessary information for different scenarios are sketched in section 1.2. Some remarks on learnability, efficiency and related work on sampling can be found in sections 1.3 and 1.4. Chapter 2 holds a report about the phase space transition and the border of the learnability for learning in first order logic. Chapter 3 is about the representation of operator specifications related to constraints as part of the Mining Mart Meta Model. The chapter ends with some examples on the usage of this information by different components of the MiningMart system. Appendix A contains the represented constraints, conditions and assertions of operators used by the MiningMart system. Appendix B complements this information by a short natural language description of these operators.

1.2 Available information for applicability conditions

The goal of using applicability conditions is to ease the choice of the next operator in an operator chain, by generating reasonable propositions, or by excluding poor candidates from the beginning. Furthermore, early estimates of quality measures are of interest, to decide if the application of a specific learning operator will yield the desired results.

Which kind of information addressing this tasks are available depends on which of the following settings we have in mind:

1. The online setting

   Given a specific dataset, which might be the result of several preprocessing steps, we want to choose the next operator.

2. The offline setting

   We want to define an operator chain, given just a specification at the conceptual level of M4. The general objective of this setting is to guarantee that if all constraints are met, the case is executable after adaptation to any specific dataset. Cases stored on the MiningMart internet server are a good example for this setting. There we have complete operator chain definitions not related to any database object.

1.2.1 Applicability Conditions in the Online Setting

In the online setting, the applicability of an operator can be estimated based on meta data of the specific dataset. The statistics for COLUMN, part of the M4 model (see D8/9, 2.1.2), offer a variety of information.

For instance for numerical attributes, we have the number of different values, the number of missing values, minimum and maximum, the average value, the standard deviation, and information on the distribution ("distribution blocks").

For tables and views the statistics for COLUMN_SET (see D8/9, 2.1.3) contain the total number of tupels and the number of attributes of types ORDINAL, NOMINAL, and TIME.

If all the statistics are calculated, then this information is available at "runtime", which means that it can be extracted from M4 just before deciding, whether an operator should be applied or not.

The assertions given for operators in M4 are such, that they can be verified using the meta data. These assertions about results are mostly about attributes being not null, about types of attributes and the type of the output concept. In order to be able to check such conditions before applying an operator, no information apart from M4 has to be stored.

The following information is relevant, but cannot be extracted from M4:

- To answer the question "How important is feature selection for different learning methods?" the independence of attributes (up to revealing redundancies of attribute subsets) is an important issue.

- Low data quality, especially because of noise can prohibit successful applications of learning algorithms. Some other reasons for low data quality are reflected by the statistical meta-data, for example by the number of missing values. A lack of standardization is easily detectable checking the calculated average and standard deviation.

1.2.2 Applicability Conditions in the Offline Setting

When designing a case independently of a specific dataset, then no statistics are available to guide the editing process. Available information at the conceptual level of M4 is much less precise, because it just describes the overall *structure*, namely the concepts and relations involved.

Thus an analysis of operator chains in this setting cannot be as sophisticated as in the online setting. What can be verified, however, is if all the known (and formalized) preconditions of operators match the postconditions of the prior operator chain. If this property is violated, the system can propose operators addressing this inconsistency. Care has to be taken when a case is generalized. When defining an operator chain for a specific dataset, all preconditions might hold, but this is not necessarily given for every possible application of this chain. If for example an operator A cannot handle missing values, then an operator B with explicit postcondition "Target concept has no missing values." should be applied before. Further on, this condition has to be invariant towards all operators applied between B and A. For the above example this would be given for all operators of type "RowSelection". In order to verify this, pre- and postconditions on the one hand, and invariance of conditions with respect to specific operator applications have to be formalized.

Applicability conditions based on the syntactical representation of data instead of statistics can be checked in the offline setting, as well.

- If a learning algorithm demands time series to have a specific representation, e.g. one of those listed in Deliverable D3, operators tailored to change the representation of time series can be proposed. The user can be warned, when trying to apply the operator on an incompatible representation. An example is the use of a windowing method before applying a Support Vector Machine to a univariate time series.

- In Inductive Logic Programming conditions for effective learnability can also be derived from the conceptual level, only. If "effective" means polynomial runtime, then the hypothesis space can be restricted

to ij-determinate horn clauses (see [8]). Given all the functional dependencies in the data explicitly, this restriction can be formalized based on meta data.

1.3 Analysis of Learnability

In learning theory in different paradigms criteria for learnability are analyzed. E.g. in the PAC-paradigm upper bounds on the necessary size of random samples are investigated, that allow to find an approximately correct hypothesis with arbitrary high probability. For certain learning problems it is known, that they cannot be solved by any algorithm, e.g. concept learning with a very high noise rate.

But results of learning theory do not always reflect the practical experiences. One of the reasons may be, that in the first place they tend to bound the worst case, which does not occur often. For that reason experiments on real world datasets could complement known theoretical results.

Apart from information theoretic bounds on learnability (due to the samples being of limited size), the runtime for finding "promising" hypotheses is of major interest for practical applications. In this field experiments could give estimates on the *expected* runtime based on the size of the dataset and further properties. Hopefully, as with ij-determinate horn clauses [8], more specific cases can be identified, where the runtime can be shown to be significantly smaller than the worst case runtime in the general case. For other cases it might be sufficient to use a subsample to achieve good results. A very promising approach when dealing with large datasets, as given in MiningMart, is to base estimation on statistically sound sampling at runtime (adaptive/sequential sampling, see e.g. [20], [7]). This should yield more precise results than theoretical results, bounding the worst case, and even give good results much faster than when processing the complete dataset.

1.4 Efficiency of Operator Application

The MiningMart system is meant as a tool for preprocessing large real world datasets. While for toy example datasets algorithms with a high complexity might still be applicable, for the intended use within MiningMart algorithms with a complexity of $O(n^2)$ can be considered intractable. Even for linear runtime algorithms a much more efficiently found estimate of the quality of yielded results will probably prove useful.

One of the most promising approaches for efficient online analysis, if an operator call will yield good results, is sampling. Sound applications of sampling enable us to give well founded estimates on results, without having to look and process a given large database as a whole. This can be exploited in the following ways:

- Before calling a learning operator on the whole training set (database) one or more small subsamples are drawn to estimate, if the expensive call of the operator on the large sample pays off.

  A problem with this approach is, that it is hard to foresee the accuracy of a classifier with a growing training set. In [7] John and Langley describe a heuristic measure (the *power law*) addressing this problem by extrapolating the quality measure for a specific incremental learner.

  A more cautious way of evaluation at this point is to simply use the quality value of intermediate results (trained on subsamples) as lower bounds[1] for the hypothesis chosen when all examples are used for training. If this quality bound is high enough, an application of this operator for the complete set should be recommended.

  Adaptive or *sequential sampling* methods constitute a more sophisticated form of this idea. Examples are read incrementally and the estimates for all the (remaining) different hypotheses are repeatedly compared. By relying on a so called confidence interval, based on the actual size of the subsample, probabilistic guarantees for an early identification of an approximately optimal subset of all hypotheses can be given. In contrast to usual sampling methods, which fix the sample size in order to bound the worst case before looking at the first example, the adaptive methods exploit properties of the examples already read, which helps to significantly decrease the amount of data processed.

  If it is possible to actively choose examples from the database (e.g. using index structures), examples might be chosen in a way that allows for a more precise distinction between the remaining "most promising" hypotheses. This allows to lower the error probability by reading a small amount of examples, only. An example on how active learning helps to more rapidly identify good hypotheses is given in [18].

- Quality estimates for hypotheses are often based on sampling.

  Evaluation over a random sample is a well suited method to evaluate the quality of hypotheses efficiently over large datasets. Applicability ranges from estimating the accuracy of an association rule [21] to measuring the degree of truth of universally (and with restrictions existentially) quantified formulae in tuple calculus [9].

  The computational costs of operators can be reduced by sampling, as well. This advantage is bought by having a small chance of drawing a poor sample and thus receiving poor results. A general approach is to evaluate the error of a large hypothesis space over a small sample, in

[1] It is no "bound" in the usual sense, because performance can also decrease, when the training set grows.

order to focus on the most promising ones. With high probability the hypothesis fitting the data best, or at least one hypothesis which is ϵ-close to the best one, is not excluded. For the small subset of empirically best hypotheses a more sophisticated analysis can be performed.

This or similar methods are successfully applied to APRIORI [21], MIDOS[2] ([17], [20], [19]) and TDIDT/Decision Stumps (see e.g. [4] for an application in combination with boosting).

Accordingly adapted versions of these operators could be used to report in advance[3] if an application does not appear promising. These warnings would not be based on heuristics, but on statistically sound evaluation.

- Another interesting application of sampling is to reveal dependencies between attributes, leading to problems with some learning algorithms. Redundant or irrelevant attributes might be identified, which is again fine for Feature Selection, and which can increase the performance of instance based learners. One can hope to even find implicit functional dependencies, which is of relevance for learning in logical representations.

[2]To be more precise: Subgroup discovery in combination with some measures integrated in MIDOS.

[3]Which means "quite early" compared with applying the operator to the full training set.

Chapter 2

A Monte Carlo Approach to Hard Relational Learning Problems

2.1 Introduction

In the last years there has been an increasing interest in learning with subsets of predicate logics, notably function free Horn clauses. In order to approach real-world problems, such as the ones encountered in Data Mining, one may wonder whether such kind of learning is actually applicable or, at least, under which conditions it might be.

Unfortunately, in a recent paper Giordana et al. [6] have shown that relational learning becomes very hard when the target concept requires descriptions involving more than three variables. The reason is related to the presence of a phase transition in the covering test [14, 5], i.e., an abrupt change in the probability that an inductive hypothesis covers a given example, when the hypothesis and the example sizes reach some critical values. Moreover, any top-down learner will search for discriminant hypotheses in the phase transition region [6, 5], and, finally, heuristics commonly used to guide top-down relational learning [15, 2] become useful only in the same region. The consequence is that the top-down induction process is blind in its first steps, so that the path to the correct concept definition is very easily lost.

In order to cope with the above difficulties, we have investigated two issues. The first one is how to estimate the "difficulty" of a learning problem, in terms of the density of sub-formulas of the target concept in the hypothesis space. The second is how to estimate the probability of detecting one of these, as a function of the target concept location with respect to the phase transition.

Finally, an induction algorithm, combining a Monte Carlo stochastic search [3] with local deterministic search, is proposed to (partially) avoid the pitfall that causes top-down search to fail. The new algorithm directly jumps into a region of the hypothesis space where the information gain heuristics has good chance of being successful, continuing its search exploiting a classical hill climbing strategy. The complexity of this algorithm is analyzed in a probabilistic framework, and it is proposed as a measure of the difficulty of the induction task. Finally, the algorithm has been experimentally evaluated on the set of hard induction problems provided by [6], and it shows a good agreement with the theoretical estimates.

2.2 Hard Relational Problems

Relational languages [12] are appealing for describing highly structured data. However, complex relational features may be hard to discover during the very learning process; in fact, finding substructures can be considered a learning problem of its own [10]. For the sake of exemplification, let us consider the data set in Figure 2.1. The cue discriminating structures (b) and (c) from structures (a) and (d) is the presence of at least one pentagonal ring. Such a cue can be simply described by the logical formula

> bound-to(x,y), bound-to(y,z), bound-to(z,w), bound-to(w,v), bound-to(v,x),

where variables x, y, z, w and v are instantiated to vertices, and the predicates bound-to(.,.) denote edges of the graphs. Capturing this cue in a propositional language requires that one already knows what he/she is searching for.

On the other hand, the same cue is difficult to learn also for relational/ILP learners [11, 15, 2, 12, 13]. In fact, a FOIL-like general-to-specific learning strategy requires the construction of the sequence of hypotheses: "bound-to(x,y)", "bound-to(x,y) $\wedge$ bound-to(y,z)", and so on, corresponding to growing portions of the ring (see Figure 2.1-(e)). As all elements in the sequence have a rather large number of models both in the positive and in the negative examples, heuristics such as *information gain* [15] or *MDL* [16] do not assign any special relevance to them until the last element is generated, and the number of models drops to zero on the negative examples. Then, a general-to-specific strategy is likely to be misled by the presence of other constructs, which may have a slightly higher information gain.

This problem has been detected both in real and artificial learning problems [5], and has been investigated in [6], where the results summarized in the following have been obtained. Let L be the complexity of a structured example e, measured by the number of atomic components, and let m be the number of literals in a conjunctive formula φ, in a first order logic language.

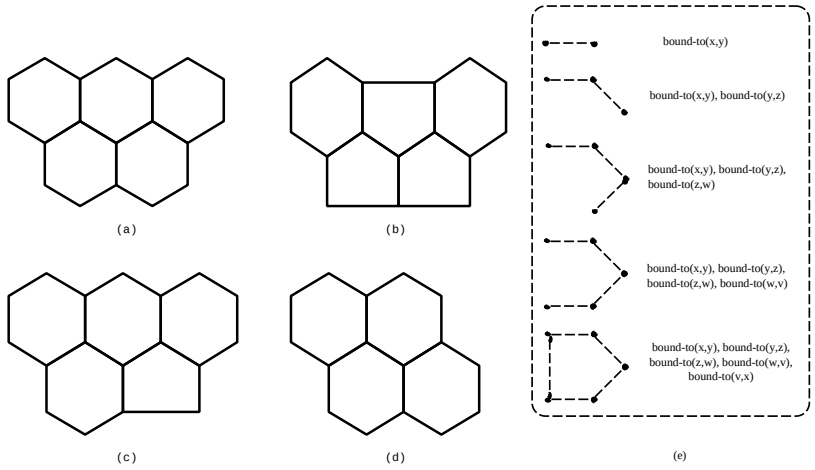

Figure 2.1: Example of data requiring relational features to be discovered. Examples (a) and (d) cannot be distinguished from (b) and (c) until a ring of at least five edges is constructed. (e) Sequence of hypotheses that a general-to-specific strategy should generate in order to learn a five edge ring description in predicated logic.

Representing the pair (e, φ) as a point (m_φ, L_e) in the plane (m, L), three regions have been found (see Figure 2.2(a)): the YES-region, the *mushy*-region, and the NO-region. In a point (m, L) located in the YES- region, it is highly probable that any formula φ with m literals has many different models on a randomly selected example e. Instead, in the mushy region (or phase transition region) a formula has, in average, 0.5 probability of having a model on e, whereas this probability is close to 0 in the NO- region. Let us now consider an inductive hypothesis φ and a set of learning examples $\mathcal{E}_L$. When the points defined by pairing every example $e \in \mathcal{E}_L$ to φ fall in the YES-region, there is no chance of deciding if it is a good or a bad hypothesis, because in both cases it will have a comparable number of models on the positives and negative examples in $\mathcal{E}_L$. Then, a learner following a general-to-specific strategy is blind until the hypotheses are complex enough to reach the border of the mushy region.

2.3 A Stochastic Approach

In absence of admissible heuristics, stochastic search may be a valid alternative. Moreover, stochastic search may be succesfully combined with deterministic search when the search space is structured into regions such that, after entering in a given region R, there exists admissible heuristics able to guide the search toward the locally best solution existing in R. Examples of how Monte Carlo search can be combined with deterministic search in

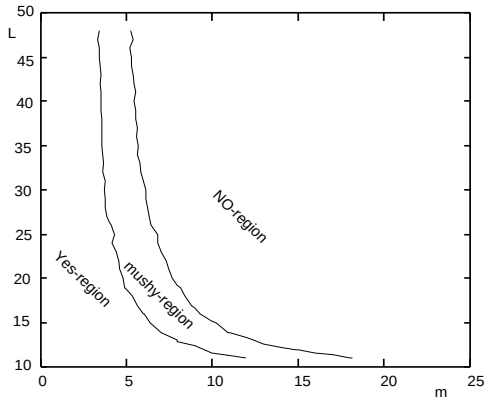

Figure 2.2: Regions characterizing the (m, L)-plane.

classical search problems, such as the k-Queens, can be found in [3].

Let P_G be the probability that φ is subformula of ω, the number τ of trials required to find at least one generalization of ω with confidence confidence $1 - \epsilon$, is

$$1 - \epsilon = 1 - (1 - P_G)^{\tau}. \tag{2.1}$$

Solving (2.1) with respect to τ we obtain

$$\tau = \frac{\log \epsilon}{\log (1 - P_G)} \tag{2.2}$$

An analytical method for estimating P_G will be supplied in the next section.

The algorithm we propose is based on a two step strategy. The first step creates a hypothesis φ_μ with a complexity m_μ sufficiently close to the border between the YES-region and the mushy region, by randomly sampling the hypothesis space. The second step performs a general-to-specific search starting from φ_μ according to a hill-climbing strategy guided by the information gain heuristics.

Algorithm T^4
let $m_\mu = \mu_0$ and let $\hat{\omega} = \emptyset$
while the learning set $\mathcal{E}_L$ is not fully covered do:

1. Generate a set Φ of τ hypotheses of complexity m_μ, randomly selected.
2. Rank hypotheses in Φ according to their information gain with respect to the formula $True$ that covers all examples, by definition.
3. Starting from the top ranked hypothesis, apply the hill-climbing specialization step to the K best ranked hypotheses.

4. Add to $\hat{\omega}$ the best hypothesis produced in the previous step.
5. Declare *covered* all the examples verified by some element in $\hat{\omega}$

It is immediate to verify that, if the target concept has a conjunctive description ω in the hypothesis space H, and τ is large enough, algorithm T^4 will find ω or at least a generalization $\hat{\omega}$ of it, correct on the learning set $\mathcal{E}_L$. Otherwise, it will produce a disjunctive description.

However, the question we want to answer is: assuming to know the complexity m_ω and the number of variables n_ω in ω what is the complexity τ we should allocate to Algorithm T^4.

2.4 Evaluation the Hypothesis Space Size

In the following we will provide methods for computing or estimating the size of the hypothesis space in dependence of the complexity of the concept description language. Moreover, we will provide an estimate of the probability of finding a generalization of the target concept by randomly sampling the hypothesis space.

2.4.1 Estimating the hypothesis space size for a function free language

Let $\mathbf{P}$ be the union of a set $\mathbf{B}$ of binary predicates and a set $\mathbf{U} = \{\beta_{\mathbf{j}}(\mathbf{x}) | \mathbf{0} \leq \mathbf{j} \leq \mathbf{p}\}$ of unary predicates.

We will first estimate the number of possible hypotheses consisting of t_B binary literals built on r variables. Let m_B be the cardinality of $\mathbf{B}$ and let m_U be the cardinality of $\mathbf{U}$. Let moreover $\mathbf{L}_{B,n}$ be the set of literals built on $\mathbf{B}$ and r variables. The cardinality of set $\mathbf{L}_{B,n}$ is $r \cdot r \cdot m_B$

Then the number of syntactically different formulas containing t_B binary literals and up to r different variables is $\begin{pmatrix} r \cdot r \cdot m_B \\ t_B \end{pmatrix}$. By subtracting, from this quantity, the number of formulas having less than r variables, the number of syntactically different formulas containing exactly r variables

$$M_s(m_B, t_B, r) = \begin{pmatrix} r \cdot r \cdot m_B \\ t_B \end{pmatrix} - \sum_{i=1}^{r-1} \binom{r}{r-i} M_s(m_B, t_B, r-i) \qquad (2.3)$$

is obtained.

Notice that syntactically different formulas can be semantically equivalent, being unifiable by properly renaming the variables. For instance, formula $\alpha_1(x_1, x_2) \wedge \alpha_2(x_2, x_3)$ is unifiable with $\alpha_1(x_2, x_3) \wedge \alpha_2(x_3, x_1)$ and hence they cover the same models in any learning instance. Actually, the

complexity of a learning task depends upon the number of semantically different hypotheses.

In order to estimate the number $M(m_B, t_B, r)$ of semantically different hypotheses with r variables, we observe that the maximum number of syntactic variants a formula may have is $r!$. Then, the following relation holds:

$$M_s(m_B, t_B, r) \geq M(m_B, t_B, r) \geq \frac{M_s(m_B, t_B, r)}{r!} \tag{2.4}$$

We will choose the central value $\hat{M}(m_B, t_B, r) = 2\frac{M(m_B,t_B,r)}{r!}$ of the interval $[M_s(m_B, t_B, r) - \frac{M_s(m_B,t_B,r)}{r!}]$ as an approximation of $M(m_B, t_B, r)$.

Let $\varphi_B(x_1, x_2, \ldots, x_r)$ a syntactic instance of a hypothesis containing only binary predicates. Let moreover β a unary literal built on one of the variables $x_1, \ldots, x_r$. It is immediate to verify that the formula $\varphi_B(x_1, x_2, \ldots, x_r) \wedge \beta$ will be the syntactic instance of a semantically different hypothesis, for any β. More in general, let $\varphi(x_1, x_2, \ldots, x_r)$ a formula containing both binary and unary predicates, any formula $\psi(x_1, x_2, \ldots, x_r) = \varphi(x_1, x_2, \ldots, x_r) \wedge \beta$, being $\beta \notin \varphi(x_1, x_2, \ldots, x_r)$, is semantically different from any other conjunction $\varphi(x_1, x_2, \ldots, x_r) \wedge \beta'$ if $\beta' \neq \beta$. On the basis of this observation, we conclude that the complexity of the hypothesis space for hypotheses $\varphi = \varphi_B \wedge \varphi_U$, being φ_U a formula of only unary literals, lies in the interval:

$$M_s(m_B, t_B, r) \cdot M_s(m_U, t_U, r) \geq M(m, t, r) \geq \frac{M_s(m_B, t_B, r) \cdot M_s(m_U, t_U, r)}{r!} \tag{2.5}$$

where $t = t_B + t_U$, $m = m_B + m_U$, and

$$M_s(m_U, t_U, r) = \sum_{i=1}^{m_U} \left(\begin{array}{c} m_U \\ i \end{array} \right) M_s(m_U, t_U - i, r - 1)$$

is the number of formulas having up to r variable made of unary literals selected from set **U** without replacement. We will approximate $M(m, t, r)$ with the value

$$\hat{M}(m, t, r) = \frac{2}{r!} M(m_B, t_B, r) \cdot M(m_B, t_U, r) \tag{2.6}$$

2.4.2 Estimating the frequency of concept generalizations

The last problem we will face is that of estimating the frequency of generalizations ψ_ω, of a concept ω, existing in a space of semantically different hypotheses. In other words, this is equivalent to estimate the probability $P(\psi_\omega|\omega)$ assuming that ω exists. Let $M_G(\omega, t, r)$ be the number of semantically different generalizations of t literals and r variables we may expect for a concept ω of t_ω literals, and r_ω variables ($r_\omega \geq r$). The number of existing

generalizations of t literals having number $\hat{M}_G(\omega, t)$ of variables between 1 and r_ω is precisely evaluated by the expression

$$\hat{M}_G = \begin{pmatrix} t_\omega \\ t \end{pmatrix}. \tag{2.7}$$

Evaluating the generalizations of exactly r variables, is impossible if ω is not known. As, for values of t, t_ω close to the mushy region, it has been found that $\hat{M}_G(\omega, t)$ is quite close to $M_G(\omega, t, r)$, we will use $\hat{M}_G(\omega, t)$ as an estimate of $M_G(\omega, t, r)$.

Then an approximation of $\hat{P}_G(\omega, m, t, r)$ can be obtained as the ratio

$$\hat{P}_G(\omega, m, t, r) = \frac{\hat{M}_G(\omega, t)}{M(m, t, r)}. \tag{2.8}$$

2.5 An Experimental Evaluation

Algorithm T^4 has been tested on the set of 451 artificial learning problems described in [6].

The artificial learning problem set has been generated by imposing that all target concept be describable using only binary predicates and exactly four variables ($n = 4$). Moreover, it has been required that all binary predicates have an extension of $N = 100$ tuples on any learning example. Given n and N, let (m, L) be a point in the plane (m, L). A set of 451 points has been sampled in the plane (m, L). Then, for each point, a learning problem $\Pi_{m,L}$ has been built up from the 4-tuple $(n = 4, N = 100, m, L)$. A target concept ω with m literals and n variables has been built up, and then a training set $\mathcal{E}_L$ and a test set $\mathcal{E}_T$ have been generated. Let Λ be a set of L constants; every example is a collection of m relational tables of size N obtained by sampling the binary table $\Lambda \times \Lambda$. In order to generate balanced training and test sets in each point of the (m, L) plane, the random generation of the examples has been modified in order to obtain examples with models also in the NO-region, and examples without models also in the YES region (see [6] for more details). The result has been the generation of training and test sets, $\mathcal{E}_L$ and $\mathcal{E}_T$, each one with 100 positive and 100 negative examples in each (m, L) point.

Some details about the problems lying in the mushy region are reported in Table 2.1. The first five columns report the number m of predicates in the concept description language, the number L of components in every single learning instance, the critical value m_c of the center of the mushy region, and the value for t_μ and r, chosen for every single run. The eleventh column reports the error rate on $\mathcal{E}_T$ for the solution selected by the algorithm. Columns from 6 to 10 report the values of $M(m, t_\mu, r)$ (computed by expression (2.6)), the real value of M_G, the estimated value $\hat{M}_G$ (assuming a concept of 4 variables lying close to the phase transition), the size

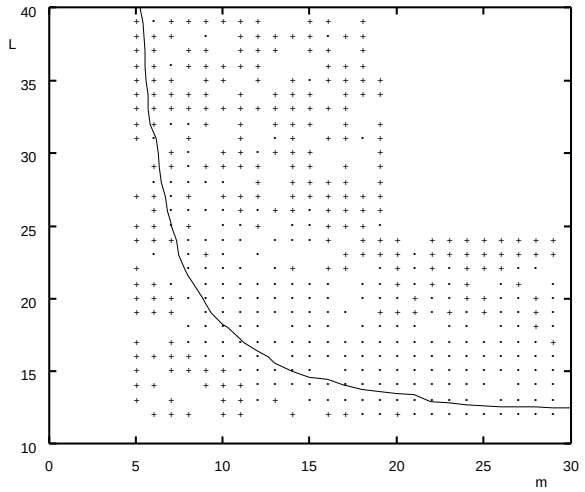

Figure 2.3: Results obtained with FOIL: *Failure region* (legend ".") and *success region* (legend "+"), for $n = 4$ and $N = 100$. The contour plot corresponds to the value $P_{sol} = 0.5$ of the probability that randomly generated covering test is positive.

$M_S(m, t_\mu, r)$ of the syntactic hypothesis space, the probability P_G computed from the values in column 6 and 7, and the number of trials τ. The results obtained by FOIL on this set of problems is summarized in Figure 2.3. It appears that most problems beyond FOILS's capability have been solved. Anyhow, for low L some problems have been found beyond the complexity affordable with the available computational resources. A problem was considered solved when the error rate on the test set of the learned concept definition was smaller than 20%. Figure 2.3 shows a large area, across the mushy region, where FOIL systematically fails.

Algorithm T^4 has been run on the problems lying in the mushy region and in the NO-region obtaining the results described in Figure 2.4. In all experiments, we started with minimal assumptions about the difficulty of the problems, scaling up until the problem T^4 was not able to find a good solution or the predicted complexity was not affordable. More specifically, in the area where the problems are easy to solve, we started with $r = 3$ and $m_\mu = 4$, and usually T^4 found the solution at the first attempt. In the more difficult region (the blind spot), we started with $r = 4$ and $m_\mu = m_c - 6$ increasing m_μ up to reach $m_c - 1$, when necessary. Parameter τ was determined using (2.6), by requiring confidence $1 - \epsilon = 0.999$. The number K of hypotheses refined at each run was chosen $K = \tau/100$. Given the high confidence required, the stochastic step always succeeded in finding at least one generalization of the target concept. Nevertheless, the hill climbing step did not succeed to find a correct generalization until t_μ was not close enough to the mushy region. Even if the t_μ and r was known in advantage, for the most complex problems, several trials have been done with smaller r and

Table 2.1: Results obtained by adding a stochastic search step to the basic hill climbing strategy. Horizontal lines separate the results reported for different problems. Rows between two horizontal lines refer to a same problem.

m	L	m_c	t_μ	r	M [10^3]	M_G [10^3]	$\hat{M}_G$ [10^3]	M_S [10^3]	P_G	τ_0.999	Err %
7	36	6	4	4	22.82	0.03	0.035	40.320	0.0011947	5778	100%
7	35	6	4	4	22.82	0.03	0.035	40.320	0.0011947	5778	100%
7	34	6	4	4	22.82	0.03	0.035	40.320	0.0011947	5778	100%
7	33	6	4	4	22.82	0.03	0.035	40.320	0.0011947	5778	100%
7	32	6	4	4	22.82	0.03	0.035	40.320	0.0011947	5778	47%
			5	4	190.68	0.02	0.021	725.760	0.0000996	69379	100%
8	31	6	4	4	45.64	0.05	0.07	80.640	0.0011758	5871	53%
			5	4	508.48	0.05	0.056	1935.360	0.0000980	70497	100%
8	30	7	4	4	45.64	0.05	0.07	80.640	0.0011758	5871	49%
			5	4	508.48	0.05	0.056	1935.360	0.0000980	70497	100%
8	29	7	4	4	45.64	0.05	0.07	80.640	0.0011758	5871	51%
			5	4	508.48	0.05	0.056	1935.360	0.0000980	70497	100%
8	28	7	4	4	45.64	0.05	0.07	80.640	0.0011758	5871	50%
			5	4	508.48	0.05	0.056	1935.360	0.0000980	70497	100%
8	27	7	4	4	45.64	0.05	0.07	80.640	0.0011758	5871	48%
			5	4	508.48	0.05	0.056	1935.360	0.0000980	70497	100%
9	26	7	4	4	82.15	0.10	0.126	145.152	0.0011665	5918	49%
			5	4	1144.08	0.11	0126	4354.560	0.0000972	71056	100%
9	24	8	4	4	82.15	0.10	0.126	145.152	0.0011665	5918	50%
			5	4	1144.08	0.11	0.126	4354.560	0.0000972	71056	100%
10	23	8	4	4	136.92	0.16	0.21	241.920	0.0011619	5941	51%
			5	4	2288.16	0.22	0.252	8709.120	0.0000968	71336	48%
			6	4	24497.76	0.20	0.21	174182.400	0.0000081	856074	100%
10	22	8	4	4	136.92	0.16	0.21	241.920	0.0011619	5941	49%
			5	4	2288.16	0.252	0.22	8709.120	0.0000968	71336	52%
			6	4	24497.76	0.20	0.210	174182.400	0.0000081	856074	100%
11	21	9	4	4	215.16	0.25	0.33	380.160	0.0011597	5953	48%
			5	4	4194.96	0.41	0.462	15966.720	0.0000966	71476	50%
			6	4	53895.07	0.43	0.462	383201.280	0.0000081	857753	100%
12	20	9	4	4	322.74	0.37	0.495	570.240	0.0011585	5959	51%
			5	4	7191.36	0.69	0.792	27371.520	0.0000965	71581	50%
			6	4	107790.14	0.87	0.924	766402.560	0.0000080	858592	100%
13	19	10	4	4	466.18	0.54	0.715	823.680	0.0011580	5961	50%
			5	4	11685.96	1.13	01.287	44478.720	0.0000965	71581	49%
			6	4	200181.70	1.61	1.716	1423319,040	0.0000080	859012	49%
			7	4	2481967.49	1.66	1.716	29889699.840	0.0000007	10308184	100%

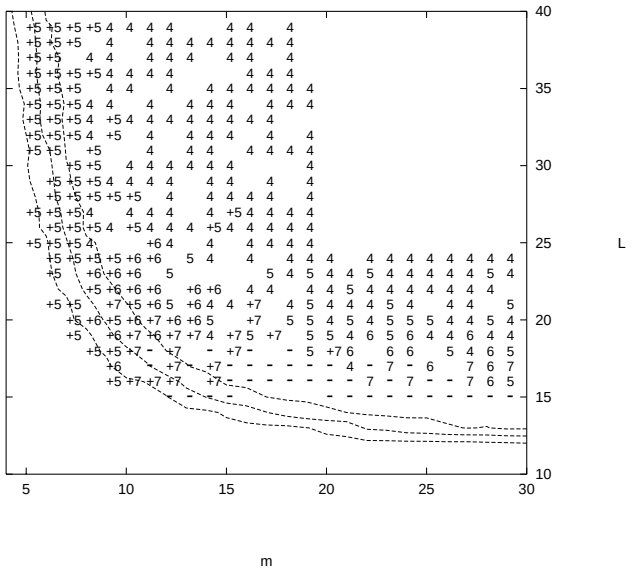

Figure 2.4: Results obtained by Algorithm T^4. The numbers denote the minimum value that was necessary to assume for m_μ in order to solve the learning problem. When the number is prefixed by "+", it means that $n = 4$ has been assumed, otherwise $n = 3$. Symbol "-" means that the problem has not been solved.

m_μ. It appears that a good generalization has been obtained always and only when the values t_μ, τ, predicted by the theory have been used.

All runs have been done using a cluster of 20 Pentium III, 800Mz.

2.6 Discussion

We have shown that combining stochastic search with local deterministic search it is possible to learn approximated concept descriptions where no known classical algorithm was successful. Even if the algorithm is used under the stringent assumption that a conjunctive concept description exists, it is not difficult to extend it in order to cope with more general concept descriptions. For instance, disjunctive descriptions can be learned by integrating T^4 with a set covering algorithm as it is made in most relational learner [15, 2].

However, this is not the fundamental result that emerges from the framework we propose. In our opinion, the most important outcome is the method for estimating the complexity of a learning problem: given a specific hypothesis about the structure of the concept, we have a method for predicting the

expected cost for testing the hypothesis. Moreover, a criterion for deciding on-line when stop testing the hypothesis is provided.

A second important result is a negative one, and concerns the possibility of learning descriptions with many variables. Even considering quite simple concept description languages, the task looks hard for many concepts requiring at least four variables. Increasing the number of variables, the complexity rises up exponentially. Considering the presence of irrelevant predicates, the analysis we performed still holds, but the the density of subformulas of the target concept close to the phase transition becomes even more tiny, and so the difficulty will increase further.

Chapter 3

Representing Constraints, Conditions and Assertions

3.1 Operator Specifications and MiningMart

Chapters 1 and 2 show different kinds of pre-conditions for efficient and successful operator applications. For operator specifications alternative formalization granularities are possible, complementing the information required and available for different scenarios and tasks. Guiding a case designer when editing a case conceptually requires formal representations of situations in which one learning method is superior to another. This task is far to complex for this work package and is addressed by other projects, for instance by the European research project MetaL[1]. A planning approach, not database oriented and too far reaching for the scope of this work package has been proposed by Bernstein, Hill, and Provost [1].

In this work-package the supported complexity has been limited to a tractable amount. The distinction between on-line (compile-time, data related) and off-line (just conceptually) scenario, described in section 1.2 is reflected by the chosen representation.

The off-line constraints focus on syntactic properties of a valid chain of steps, like admissible data types, inputs, outputs, and valid parameter settings. The amount of information allows the HCI to edit steps embedding new operators, if just their formal constraints are known. The output can also be generated by the HCI, given just this source of information.

The on-line constraints, to avoid confusion called *conditions* from now on, cover all pre-conditions of successful operators applications only available at runtime. Some operators cannot handle missing or negative values, for instance. These properties can only be evaluated after mapping the conceptual level to specific database objects, for steps inside a chain often even after compilation of the preceeding step(s). On the other hand many opera-

[1] http://www.metal-kdd.org/

tors have known data related assertions, which are also formalized, similar to conditions. In MiningMart there are several operators for replacing missing values, for example, asserting to produce an output without missing values. This information can help the M4 compiler to increase performance, because it is superfluous for example to check for missing values after an operator replacing missing values has just been applied.

3.2 Changes to the Mining Mart Meta Model (M^4)

In order to directly attach constraints, conditions and assertions to the operators in the M4 model, some modifications were necessary in the scope of this work package. According to deliverable 8 parameters are attached to the class of operators, implying that operators in the model refer to operator applications, rather than to operators as such. In the relational representation, for each step an operator is embedded in, another instance of the operator appears in the table OPERATOR_T.

It seems more intuitive to link the arguments, including inputs, outputs and parameters[2], to the steps, rather than to single operators. Thus the first change applied to the model, is to change the semantics of the class OPERATOR.

Each operator should appear exactly once in this class and thus in the table OPERATOR_T. For each step that the operator is embedded in, a foreign key reference in STEP_T is sufficient to specify the particular operator. Further foreign key references, from PARAMETER_T to STEP_T should determine the arguments and output of the operator application. No changes are necessary here, because references to STEP_T and OPERATOR_T are already foreseen in M4. As a consequence of the new structure the table OP_NAME_T will be redundant and can be removed from the M4 model, as soon as the compiler is adjusted to this change. The attribute PAR_OPID of the table PARAMETER_T will be replaced by PAR_STEPID, a foreign key reference to the step the parameters belong to. In this setting the constraints, conditions and assertions can be defined referencing the operators specified in the table OPERATOR_T. The table OPCONSTRAINT_T will be deleted. Instead four new tables will be added.

Figure one depicts the changes to the model. The constraints are represented in two tables, OP_PARAMS_T to define the inputs and outputs and OP_CONSTR_T for other constraints. Conditions and constraints will be described by two other tables. This results in the following list of new tables:

- OP_PARAMS_T defining the admissible inputs and outputs
- OP_CONSTR_T for constraints

[2]Please note that the sum of these arguments is also called *parameters* in the model!

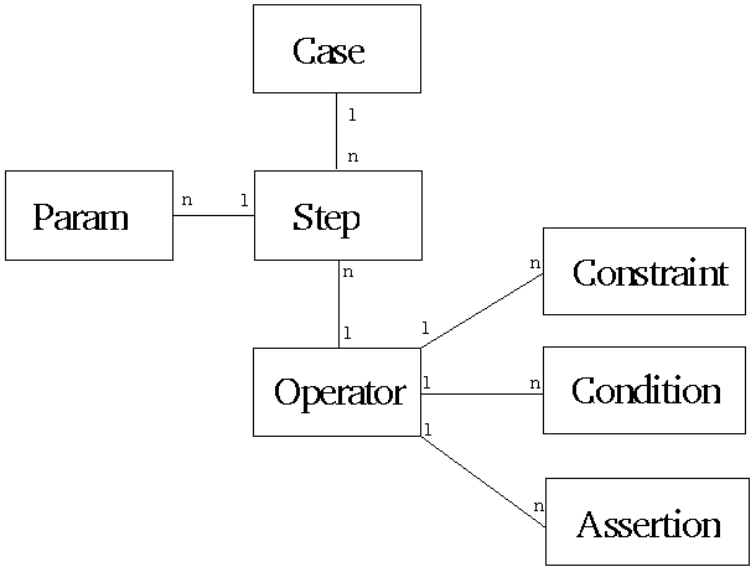

Figure 3.1: Modifications

- OP_COND_T for conditions
- OP_ASSERT_T for assertions.

The following sections will describe these tables in detail. An example scenario for using the formalism is given in section 3.7.

3.3 Specifying inputs and outputs

The first of the tables is added to be able to state restrictions on the arguments used by operators. In detail:

- type and number of input arguments
- type of output

The new table OP_PARAMS_T is the first of two tables implementing the concept "Constraint" in figure 3.1. It has the following attributes:

- PARAM_ID: ID of the parameter tuple
- OP_ID: foreign key reference to OPERATOR_T
- MINARG: minimum number of parameters of this kind - might be "0"
- MAXARG: maximum number of parameters of this kind - NULL, if unrestricted

- NAME: The name given here has to be used in table PARAMETER_T[3] in order to be able to identify parameters. The name is interpreted as a prefix, to allow more than one parameter. The ordering over the set of parameters, matching the prefix, is determined by their number given within PARAMETER_T. The identification is necessary in order to formulate applicability conditions, constraints and assertions.

- IO: Is this parameter(set) an input or the output.

- TYPE: The M4 model uses different kind of object types handled by operators. The following list shows the alternatives:

  - CON (CONCEPT)
  - REL (RELATION)
  - BA (BASE_ATTRIBUTE)
  - MCF (MULTI COLUMN FEATURE)
  - FEA (Feature, either BA or MCF)
  - V (VALUE)
  - FUNC (FUNCTION)

  The following types appear in deliverable D8/9, but seem to be deprecated, or at least not handled by software of any kind:

  - Q (QUERY)
  - TI (TIME_INTERVAL, in D8/9: just output)
  - DR (DEVIATION_RULES, in D8/9: just output)

  They are no longer supported. Time intervals can be represented by two time features and are a typical kind of multi column features, anyway. Queries do not have to be handled as an extra data type in M4 and deviation rules can be stored as database functions, similar to decision trees and SVM models.

- DOCU: Basically for the HCI, this attribute holds a brief free text description of the according parameter.

The intended usage of the above specifications of inputs and outputs is to list the admissible arguments, one after another, declaring types and names. To support arrays[4] as well, the minimum and maximum number of objects is necessary. In this case the name is interpreted as a prefix. The names of

[3] Concept PARAM in figure 3.1.

[4] By the same mechanism optional arguments could be handled, as well. However, there was no necessity, yet.

the arguments are important to identify single arguments on the instance level, namely in the table PARAMETER_T. Having these references enables a formulation of constraints, conditions and assertions.

As an example let's give the declaration of the first argument of operator no. 35, which has to be a (single) value labeled 'HEAD_W":

PARAM_ID	OP_ID	MINARG	MAXARG	NAME	IO	TYPE
1001	35	1	1	'HEAD_W'	'IN'	'V'

Let us then specify an array of inputs. For arrays the following convention is used: Given NAME="WEIGHT" all inputs with the appropriate type and name prefix "WEIGHT" will be considered part of this input array. The ordering of the array is derived from the order of the arguments in the table PARAMETER_T.

To specify a list of weights, the following statement can be used:

PARAM_ID	OP_ID	MINARG	MAXARG	NAME	IO	TYPE
1002	35	1	NULL	'WEIGHT'	'IN'	'V'

This would aggregate input arguments like "WEIGHT01" or "WEIGHTING" with data type "V" (value) for this operator. The same convention is applicable, if the output is an array.

3.4 Representing Operator Constraints

To formulate other operator constraint than expressible by the mechanism presented in section 3.3 the table OP_CONSTR_T is added to the M4 schema. It contains the attributes CONSTR_ID, CONSTR_OPID, CONSTR_TYPE, CONSTR_OBJ1, CONSTR_OBJ2, and CONSTR_SQL.
The semantics of these attributes is as follows:

- CONSTR_ID: constraint identifier
- CONSTR_OPID: foreign key reference to operator
- CONSTR_TYPE: different kinds of constraints can be used. Depending on the type, CONSTR_OBJ1 and CONSTR_OBJ2 may have different semantics
  - IS_LOOPED: if the operator is looped then parameter OBJ1 is defined per loop, which means that there is an instance of this parameter for each of the operator's loops. If OBJ1 denotes a parameter, which is an array, then there has to be such an array for each loop.

- ISA: the concept or relation (CONSTR_)OBJ1 isA concept or relation (CONSTR_)OBJ2
- SAME_FEAT: The concepts OBJ1 and OBJ2 have the same set of features, where "same" only means that feature names are equal. The assumption is made, that two different BaseAttributes may have the same name, e.g. "CUSTOMER", one time with, one time without missing values, as long as they do not share the same concept.
- SAME_TYPE: OBJ1 and OBJ2 denote BaseAttributes, which need to have the same conceptual and relational datatype. This is meant to determine the type of output attributes given the input. Additionally either OBJ1 or OBJ2 might refer to a parameter value, rather than to a BaseAttribute. In this case it needs to be checked if the value (a string in the database) can be converted into the format of the specified BaseAttribute.
- COMP:
  If OBJ1 is a CONCEPT, then OBJ2 is a relation. If OBJ1 is a relation, then OBJ2 can be either a concept or a relation. The constraint states, that the relations or the concept and the relation are of compatible type.
  * If a concept C is followed by a relation R, then $C\ isA\ domain(R)$.
  * If a relation R_1 is followed by another relation R_2, then $range(R_1)\ isA\ domain(R_2)$.
  * If a relation R is followed by a concept C, then $range(R)$ $isA\ C$. Instead of a single relation, R might also denote a set of relations, using their name prefix. The chaining of these relations is done by their ordering, using the above rules. Finally a BaseAttribute may be used, wherever a Concept is expected. In this case the BaseAttribute must be present in the according concept.
- LINK:
  This kind of constraint makes sure, that several features are deterministically and efficiently reachable ("joinable") on the level of relational tables.
  If OBJ1 and OBJ2 are BaseAttributes, then they have to refer to attributes of the same relational table, or there has to be a foreign key reference from the table containing OBJ1 to that containing OBJ2.
  If OBJ1 and OBJ2 are concepts, then all of their features have to be connected. Features are connected by sharing a relational table, by being in tables connected by a foreign key reference or by chaining these two alternatives. In case of chaining, references

are directed and all features in OBJ2 need to be reachable from OBJ1 (not necessarily vice versa).
If one object is a feature and one is a concept, then the ideas above will be canonically adopted, treating a feature like a single feature concept.
Instead of single concepts, prefixes might be used, to have concept arrays. The constraint for two concepts has to hold for each concept and its successor.

- IN:
  OBJ1 holds the name of one or the prefix for more features.
  OBJ2 is the name of a concept, containing all these features. If OBJ2 denotes the (an) output concept, then this concept needs to contain a feature (or all, if a feature set is given) with the original name(s) of the object(s) referenced by OBJ1. If there are multiple tuples of this constraint present for a fixed OBJ1, then OBJ1 has to be present in the union of OBJ2 objects.
  If OBJ1 denotes a concept, then all BaseAttributes of OBJ1 need to be contained in OBJ2.
  If the 'IN' constraint occurs more than once for the same OBJ1, then OBJ1 needs to be in only one of the corresponding OBJ2 objects.
- TYPE:
  OBJ1 holds the name of one or the prefix for more features.
  OBJ2 is the name of a conceptual data type, allowed for the according attribute(s). If more data types are supported (not correctly subsumable by one of the M4 categories) multiple tuples for a single feature are possible.
- LT, GT, LE, GE, NE:
  OBJ1 is the name of one (or more → name prefix) input parameters.
  OBJ2 is usually a number, in case of "NE" it might be a nominal constant. The conditions are "lower than", "greater than", "lower or equal", "greater or equal" and "not equal".
- ONE_OF:
  OBJ1 is the name of one (or more → name prefix) input parameters.
  OBJ2 is a comma separated string of possible values for this (these) parameter(s). An example for parameter "kernel type" could be: "linear,polynomial,radial". If necessary, a comma within a possible parameter value can be expressed by writing "\,".
- SUM:
  OBJ1 references a set of numerical parameters by their name prefix. Additionally summing over multiple parameters can be

expressed by a comma separated list, where each element might either refer to an array or to a single parameter value.
OBJ2 gives the value the sum of these parameters needs to have.

- ORDERED: OBJ1 denotes an array of values, which have to be in a specific order. OBJ2 specifies if the values have to ascending ("INC") or descending ("DEC").

- CONSTR_DOCU: This attribute holds a brief free text description of the constraint. It is intended to be used by the HCI.

- CONSTR_SQL: If possible an SQL-query implementing the test of the constraint is stored here, NULL otherwise.

The following examples demonstrate the usage[5]:

- Constraint 1: For operator no. 35 the output concept C_OUT has to be of the same type or a specialization of the input concept C_IN.

ID	OPID	TYPE	OBJ1	OBJ2
1003	35	'ISA'	'C_OUT'	'C_IN'

- Constraint 2: A given array of relationships, passed to operator no. 35, defines a domain compatible chain. The chain shall start with input concept C. The relationships should have the name prefix "REL" (e.g. $REL01, \ldots, REL20$) and have to be given in the relevant order. The concept that corresponds to the domain of the last relation should contain the BaseAttribute F.

ID	OPID	TYPE	OBJ1	OBJ2
1004	35	'COMP'	'C'	'REL'
1005	35	'COMP'	'REL'	'F'

- Constraint 3: A join on a given array of concepts, prefix C, has to be efficiently computable. Thus first of all the concepts in the array need to be stored in a single table, or to be easily joinable by exploiting foreign key references. The same holds for successive concepts. They need to be reachable via foreign key references, or might even be stored in the same table as the predecessor concept. Finally the BaseAttribute $ATTR$ should be connected equivalently easy from the last concept of the array.

ID	OPID	TYPE	OBJ1	OBJ2
1006	35	'LINK'	'C'	'ATTR'

- Constraint 4: All input attributes mentioned need to be part of the input concept C. This can be stated by using "IN"-constraints. E.g.

[5]Omitting the suffix "CONSTR_" in attribute names.

for attributes Att_1, ..., Att_10 and B:

ID	OPID	TYPE	OBJ1	OBJ2
1007	35	'IN'	'Att_'	'C'
1008	35	'IN'	'B'	'C'

- Constraint 5: A BaseAttribute D has to be of type TIME or CATEGORIAL:

ID	OPID	TYPE	OBJ1	OBJ2
1009	35	'TYPE'	'D'	'TIME'
1010	35	'TYPE'	'D'	'CATEGORIAL'

- Constraint 6: The definition of a valid interval $[0, 1]$ for a numerical parameter $PROBABILITY$ of operator no. 35 can be expressed by the following two tuples:

ID	OPID	TYPE	OBJ1	OBJ2
1011	35	'GE'	'PROBABILITY'	0
1012	35	'LE'	'PROBABILITY'	1

- Constraint 7: The sum of an array of weights, name prefix $WEIGHT$, has to be 1:

ID	OPID	TYPE	OBJ1	OBJ2
1013	35	'SUM'	'WEIGHT'	1

3.5 Representing Conditions for Operator Applications

Conditions are represented using the table OP_COND_T, which contains the attributes COND_ID, COND_OPID, COND_TYPE, COND_OBJ1, COND_OBJ2 and COND_SQL:

- COND_ID: condition identifier
- COND_OPID: foreign key reference to OPERATOR_T
- COND_TYPE: different kinds of conditions can be formulated. Depending on the type, COND_OBJ1 and COND_OBJ2 may have different semantics. For conditions taking one argument only, COND_OBJ2 needs to be NULL.

  The list below describes the set of applicable conditions. If nothing else is stated, a BaseAttribute (or multiple BaseAttributes, using name prefixes) is given by (COND_)OBJ1, while (COND_)OBJ2 has to be NULL.

- HAS_NULLS: At least one NULL value is present in the BaseAttribute.
- NOT_NULL: No values are missing. If OBJ1 specifies a concept, then in each of the concept's features no NULL entries are allowed.
- HAS_VALUES: OBJ1 refers to a BaseAttribute, OBJ2 is a number or NULL. At least OBJ2 entry of OBJ1 are not NULL. If OBJ2 is NULL at least 1 value needs to be in OBJ1.
- UNIQUE: The given BaseAttribute contains no duplicates.
- ORDERED: The concept values are ordered by the given feature, while OBJ2 determines if ascending ("INC"), descending ("DEC"), or if it does not matter (NULL).
- EQUIDIST: This condition is especially interesting for time series. It is checked, if the BaseAttribute (specified by OBJ1) is ordered and equidistant, which is the normal form for time series. OBJ2 might be NULL or specify a step size.
- LOWER_BOUND: The given BaseAttribute may not contain values below OBJ2. This condition applies to attributes of type TIME, as well.
- UPPER_BOUND: analogous for upper bounds
- AVG: The given feature has to have the average value specified by OBJ2.
- STD_DEV: The feature has a standard deviation of OBJ2.
- LE, LT, GE, GT: As for constraints, "LE" stands for "lower or equal", "GT" means "greater than" and so on. In case of such a condition it has to be checked, if the inequality between two BaseAttributes, given as arguments OBJ1 and OBJ2, holds for all instances of the according concept[6]. If the BaseAttributes belong to different concepts, the condition is not met.
  OBJ2 might also be a numerical constant, rather than a BaseAttribute.
  Further on, OBJ1 and/or OBJ2 can also refer to multiple BaseAttributes by giving their name prefixes, stating that the condition is met between all objects o_1 and o_2, with $o_1 \in OBJ1$ and $o_2 \in OBJ2$.

[6]Please note, that in the original version each BaseAttribute belongs to exactly one concept, so there is no need to provide the concept as an additional argument. In a later version there might be a n..m relation between Concepts and BaseAttributes. Even in this case it does not matter, which Concept containing both BaseAttributes will be chosen to check the condition. For all these concepts the same columns will be referenced, and the pairing of the columns values within each tuple will be the same.

- COND_DOCU: This attribute holds a brief free text description of the condition.
- COND_SQL: If possible an SQL-query implementing the test of the condition is stored here, NULL otherwise.

Examples on usage[7]:

- Condition 1: A concept TC constituting a time series that is ascendingly sorted over the time feature TA[8]:

ID	OPID	TYPE	OBJ1	OBJ2
1014	35	'ORDERED'	'TA'	'INC'

- Condition 2: A numerical BaseAttribute labeled *Attr* has to be strictly positive (e.g. for LogScaling):

ID	OPID	TYPE	OBJ1	OBJ2
1015	35	'GT'	'Attr'	0

- Condition 3: Two attributes of type TIME belong to the same concept and specify time intervals. Thus for all instances the starting time ($START$) must not be later than the end of the interval (END):

ID	OPID	TYPE	OBJ1	OBJ2
1016	35	'LE'	'START'	'END'

3.6 Representation of Assertions

Assertions are at the same level as conditions, giving guarantees about characteristics of the output. If an assertion is related to a feature of the output, while the output is a concept, then the following kind of reference is meant: The output needs to contain a feature with the same original name than the specified feature. The statement is about this feature.

Assertions are stated using the table OP_ASSERT_T containing the attributes ASSERT_ID, ASSERT_OPID, ASSERT_TYPE, ASSERT_OBJ1, ASSERT_OBJ2, ASSERT_SQL:

- ASSERT_ID: assertion identifier
- ASSERT_OPID: reference to OPERATOR_T
- ASSERT_TYPE: different kinds of conditions can be formulated. Depending on the type, ASSERT_OBJ1 and ASSERT_OBJ2 may have different semantics. For assertions taking one argument only, ASSERT_OBJ2 needs to be NULL.

[7] Omitting the suffix "COND_" in attribute names.

[8] That TA has to be a feature of concept TC is a constraint, that should be stated in the according table, using "IN".

  - SUBSET: (ASSERT_)OBJ1 is an input concept of which the output concept OBJ2 is a subset. This statement makes sense only, if the input and the output concept are of same type (same features).
  - PROJ: OBJ1 is an input concept of which the output concept OBJ2 is a projection (Complete projection of all tuples!).
  - NOT_NULL: OBJ1 denotes the name of the feature which is asserted not to be NULL in the output. Alternatively the name of the output concept may be entered here, stating that in all of the features no missing values are present.
  - LOWER_BOUND and UPPER_BOUND: The output feature OBJ1 only contains values greater or equal / lower or equal OBJ2. This kind of assertion is applicable for TIME features, as well.
  - UNIQUE, ORDERED, EQUIDIST, AVG, STD_DEV might be used as well, in the meaning described for conditions, but this is not necessary, yet.
  - LT, LE, EQ, GE, GT: As for constraints, "LE" stands for "lower or equal", "GT" means "greater than" and so on. OBJ1 is a BaseAttribute, OBJ2 is a numerical value or another BaseAttribute. The given inequality holds after applying the operator.

- ASSERT_DOCU: A brief free text description of the assertion should be entered here.
- ASSERT_SQL: If possible an SQL-query implementing a test of the assertion is stored here, NULL otherwise.

Examples for Assertions[9]:

- Assertion 1: The output concept of an operator is a projection of the input concept IN_CON. This can be stated as

ID	OPID	TYPE	OBJ1	OBJ2
1015	35	'PROJ'	'IN_CON'	NULL

- Assertion 2: An attribute A which is part of the input concept IN_CON and of the output concept OUT_CON does not have missing values in the output concept. This bundles two constraints with an assertion. The constraints, A being in IN_CON and A being in OUT_CON, can be formulated using "IN", as illustrated before.

  The tuple representing the "NOT_NULL" assertion, again for example operator no. 35, looks like this:

ID	OPID	TYPE	OBJ1	OBJ2
1017	35	'NOT_NULL'	'A'	NULL

[9] Omitting the suffix "ASSERT_" in attribute names.

3.7 A Use-Case

This section illustrates the use of the representation framework for a specific operator. Let's specify the details for the operator FeatureSelection, having ID 43 in M^4. First of all it should be recalled, that this operator with the corresponding ID is stored in table OPERATOR_T rather than in OP_NAME_T, which was deleted from the M^4 schema. The entry in this table, omitting OP_REALIZES:

OP_ID	OP_NAME	OP_LOOP	OP_MULTI	OP_MANUAL
43	'FEATURE_SEL.'	'NO'	'NO'	'YES'

This operator has the following constraints:

- There is exactly one input concept.
- A set of features specifies a subset of the concept's features.
- The output is a single concept.

Using the same variables as in deliverable 8/9 the input/output constraints (OP_PARAMS_T) would be represented as

ID	OP_ID	MINARG	MAXARG	NAME	IO	TYPE
1018	43	1	1	'TheInputConcept'	'IN'	'CON'
1019	43	1	NULL	'TheSelectedFeatures'	'IN'	'FEA'
1020	43	1	1	'TheOutputConcept'	'OUT'	'CON'

The constraints that *TheSelectedFeatures* are all present in *TheInputConcept* and in *TheOutputConcept* are still missing. They are entered in table OP_CONSTR_T:

ID	OPID	TYPE	OBJ1	OBJ2
1021	43	'IN'	'TheSelectedFeatures'	'TheInputConcept'
1022	43	'IN'	'TheSelectedFeatures'	'TheInputConcept'

To illustrate, how this is related to the corresponding instances, let's have a look at table STEP_T[10], step no. 1 of case 123, embedding this operator:

ST_ID	ST_CAID	ST_NR	ST_OPID
1023	123	1	43

The arguments for this specific step are stored in table PARAMETER_T, as shown below[11]. The attribute (PAR_)OBJID references the parameter object. For a *concept* the according concept ID of table CONCEPT_T will

[10] Loop and multistep information is omitted.

[11] The column PAR_STLOOPNR and the prefix "PAR_" of attribute names are omitted.

be given here, for relations or other types the IDs of the according M^4 tables will be referenced, instead. The type is defined in (PAR_)OBJTYPE. (PAR_)STID is a foreign key reference to the step.

ID	NAME	OBJID	OBJTYPE	OPID	TYPE	NR	STID
1024	TheInputConcept	1030	'CON'	43	IN	1	1023
1025	TheSelectedFeatures1	1031	'BA'	43	IN	2	1023
1026	TheSelectedFeatures2	1032	'BA'	43	IN	3	1023
1027	TheSelectedFeatures3	1033	'BA'	43	IN	4	1023
1028	TheSelectedFeatures4	1034	'BA'	43	IN	5	1023
1029	TheOutputConcept	1035	'CON'	43	OUT	6	1023

The names of (PAR_)NAME have to match the specifications given in the table OP_PARAMS_T. The object IDs of the arguments (PAR_OBJID) and the internal names of these objects, e.g. the CONCEPT name, can of course not be used for the formulation of constraints, etc. The arguments are referenced by the parameter names of OP_PARAMS_T, instead. Together with the above specification in table OP_PARAMS_T the following set of arguments is found from the table PARAMETER_T:

- TheInputConcept points to the object with ID 1030 in table CONCEPT_T.
- *TheAttributes* is an array of four BaseAttributes, referenced by ID:
  - TheSelectedFeatures[1] points to BA no. 1031
  - TheSelectedFeatures[2] points to BA no. 1032
  - TheSelectedFeatures[3] points to BA no. 1033
  - TheSelectedFeatures[4] points to BA no. 1034

  The ordering of these four BaseAttributes is given by their (PAR_)NR values. The elements of this array are all input arguments of type BaseAttribute, which have a name beginning with "TheSelectedFeatures".
- TheOutputConcept is the concept with ID 1035.

Now let's formulate the assertion for the FeatureSelection operator: *TheOutputConcept* is an extensionally equivalent projection of *TheInputConcept*. This can be formulated using an assertion of type projection ("PROJ"):

ID	OPID	TYPE	OBJ1	OBJ2
1036	43	'PROJ'	'TheInputConcept'	'TheOutputConcept'

3.8 Embedding the knowledge

The knowledge represented by the altered part of M4, described in the previous sections, will be necessary to maintain *case consistency*. The consistency is enforced by different parts of the system. To some extent the referential integrity constraints in the database will forbid to enter problematic data into M4, for instance steps not embedding any operator. The triggers, also part of the M4 model, constitute an even more powerful consistency checking mechanism.

However, due to limitations in representing constraints directly at the database level, the main efforts for consistency checking will be elsewhere, namely in the case editor. The constraints and conditions provided in the tables named above should be checked whenever the case designer adds a step and/or modifies a case.

Figure 3.1 shows the organization of the relevant parts of M^4. The concepts CASE, STEP and PARAM can be conceived as *instances*, while OPERATOR, CONSTRAINT, CONDITION and ASSERTION are comparable to *classes*[12] in the sense of object oriented modelling. Steps are instances of operators. If a case designer adds a step, then the case editor should automatically check, which input and output arguments are required and forbid to submit any combination of argument settings, not allowed due to *constraints* represented in M^4. In contrast to steps, the list of existing operators and the information about operator constraints, conditions and assertions may not be altered by the case designer, but can be considered as fixed. If a case designer wants to connect two steps, then assertions might be of relevance, as well.

Operator *conditions* can generally be checked at runtime, only, because before the relational tables behind the meta-data cannot be analyzed. In the extreme case of steps using a FeatureSelection operator, even the conceptual level cannot be specified before executing the operator, because the features of the output concept depend on the specific dataset. The compiler needs to check the conditions of an operator before executing it. This is necessary in order to generate runtime exceptions and meaningful messages to the user. Additionally to checking conditions, the compiler can also exploit constraints, especially for conveniently loading parameters using a single mechanism for all operators, and for not executing an operators if the set of parameters does not meet the operator's specification. However, this is less important for consistency checking, because the case editor should not allow for entering steps violating the constraints. Meta-data resulting from assertions like "NOT NULL" helps to avoid unnecessary checking of data

[12] Please note, that this distinction is different from conceptual and relational/executable level. One should also keep in mind that the distinction between information available early on (while designing a case) and that, available at runtime (executing a case) is still something else. All these distinctions are of importance for the given context!

properties at runtime.

3.9 An Example on how to use Constraints in the HCI

This section illustrates the use of the representation framework for the HCI. Let's assume that the case designer wants to add a step embedding the operator MISSING_VALUES_WITH_REGRESSION_SVM[13].
In table OPERATOR_T the following information is stored[14]:

OP_ID	OP_NAME	OP_LOOP	OP_MULTI	OP_MANUAL
53	'MISSING_V._WITH_R._SVM'	'YES'	'NO'	'NO'

As stated above, this operator is LOOPABLE, not MULTISTEPABLE and not MANUAL.

The constraints for this operator can be read from the tables OP_PARAMS_T and OP_CONSTR_T:

- There is exactly one input concept *TheInputConcept*.
- A set of BaseAttributes *ThePredictingAttributes* specifies a subset of the input concept's features.
- All attributes *ThePredictingAttributes* are of type scalar.
- A BaseAttribute specifies the target attribute *TheTargetAttribute*, for which missing values shall be replaced.
- *TheTargetAttribute* is of type scalar.
- The output is a (single) BaseAttribute *TheOutputAttribute*, attached to *TheInputConcept*. Its data type is the same as that of *TheTarget–Attribute*.
- The parameters C, $LossFunctionPos$, $LossFunctionNeg$ and *Epsilon* are numerical values. All values are strictly positive.
- The $KernelType$ is one of "dot", "polynomial", "radial", "neural" and "anova".
- The operator is loopable. The following parameters are defined per loop: *TheTargetAttribute*, *ThePredictingAttributes*, $KernelType$,

[13]The recently added feature of using a database implementation of the SVM implied some extra parameters and constraints. These aspects are rather technical and are omitted in this section.

[14]Omitting OP_REALIZES.

LossFunctionPos, *LossFunctionNeg*, *C*, *Epsilon*, *TheOutput–Attribute*

The input/output constraints (OP_PARAMS_T) would be represented as

ID	OP_ID	MIN	MAX	NAME	IO	TYPE
1040	53	1	1	'TheInputConcept'	'IN'	'CON'
1041	53	1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1042	53	1	1	'TheTargetAttribute'	'IN'	'BA'
1043	53	1	1	'C'	'IN'	'V'
1044	53	1	1	'Epsilon'	'IN'	'V'
1045	53	1	1	'LossFunctionPos'	'IN'	'V'
1046	53	1	1	'LossFunctionNeg'	'IN'	'V'
1047	53	1	1	'KernelType'	'IN'	'V'
1048	53	1	1	'TheOutputAttribute'	'OUT'	'BA'

In table OP_CONSTR_T the other constraints can be stated as

ID	OP_ID	TYPE	OBJ1	OBJ2
1049	53	'IN'	'ThePredictingAttributes'	'TheInputConcept'
1050	53	'TYPE'	'ThePredictingAttributes'	'SCALAR'
1051	53	'IN'	'TheTargetAttribute'	'TheInputConcept'
1052	53	'TYPE'	'TheTargetAttribute'	'SCALAR'
1053	53	'TYPE'	'C'	'SCALAR'
1054	53	'GT'	'C'	0
1055	53	'TYPE'	'LossFunctionPos'	'SCALAR'
1056	53	'GT'	'LossFunctionPos'	0
1057	53	'TYPE'	'LossFunctionNeg'	'SCALAR'
1058	53	'GT'	'LossFunctionNeg'	0
1059	53	'TYPE'	'Epsilon'	'SCALAR'
1060	53	'GT'	'Epsilon'	0
1061	53	'ONE_OF'	'KernelType'	'dot,polynomial,...'
1062	53	'IN'	'TheOutputAttribute'	'TheInputConcept'
1063	53	'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
1064	53	'IS_LOOPED'	'TheTargetAttribute'	NULL
1065	53	'IS_LOOPED'	'ThePredictingAttributes'	NULL
1066	53	'IS_LOOPED'	...	NULL

The most important information for the case editor will be, which inputs and outputs belong to a certain operator. As stated in OP_PARAMS_T a step embedding the MISSING_VALUES_WITH_REGRESSION_SVM needs to be provided with an input concept, a target attribute, a set of base attributes used to predict the missing target attribute and four parameters of type VALUE. The operator produces a single output of type BaseAttribute. Additionally to these constraints, the second table gives further restrictions. The parameters have to be of matching type and the numerical values need to be strictly positive. The user is guided by the HCI when entering the necessary information.

Together, these constraints provide the HCI with all the necessary in-

formation about the validity of steps, regarding the demands of operators. The output needs to be specified, too, when setting up a step. In this case the output feature is just added to the input concept, because of the constraint with ID 1062. For other operators the output is a concept, which is constructed automatically by the HCI exploiting simple constraints and conventions. For instance a constraint like

"SAME_FEAT TheOutputConcept TheInputConcept"

makes the HCI construct an output concept with the same set of features as the input concept.

After validating the set of parameters for a certain step with respect to the general constraints given for the embedded operator, the HCI enters these parameters into table PARAMETER_T. The names used in this table have to match the names used in OP_CONSTR_T.

3.10 When to check Conditions and how to exploit Assertions

The conditions can hardly be exploited by the case editor. To illustrate their runtime specific character, let's continue with the operator MISSING_VALUES_WITH_REGRESSION_SVM, having the following conditions:

- The predicting attributes *ThePredictingAttributes* do not contain any missing values.
- The target attribute has both, present and missing values.

Represented in the table OP_COND_T:

ID	OP_ID	COND_TYPE	OBJ1	OBJ2
1067	53	HAS_NULLS	'TheTargetAttribute'	NULL
1068	53	HAS_VALUES	'TheTargetAttribute'	NULL
1069	53	NOT_NULL	'ThePredictingAttributes'	NULL

In general it will not be possible to decide, if a base attribute contains a missing value, without looking at the data. This condition could rather be checked by the compiler at runtime, in order to avoid an unnecessary operator application, if there are no missing values, anyway. In other cases, maybe if a time attribute is not equidistant, although an equidistant time series is expected, a runtime exception with a meaningful message to the user should be generated.

The main advantage of formalizing assertions is to avoid checking conditions, which can already be concluded to hold or to be violated. For the

example operator the output base attribute will not contain any missing values, which is represented in table OP_ASSERT_T as follows:

ID	OP_ID	ASSERT_TYPE	OBJ1	OBJ2
1070	53	NOT_NULL	'TheOutputAttribute'	NULL

Once such an assertion is true, it can be memorized by the compiler, in order to avoid unnecessary checks at runtime.

Appendix A

This chapter shows the complete list of constraints, conditions, and assertions collected on 17th of December 2002.

A.1 MultiRelationalFeatureConstruction

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
0	NULL	'TheConcepts'	'IN'	'CON'
0	NULL	'TheRelations'	'IN'	'REL'
1	NULL	'TheChainedFeatures'	'IN'	'FEA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'COMP'	'TheInputConcept'	'TheRelations'
'COMP'	'TheRelations'	'TheConcepts'
'IN'	'TheChainedFeatures'	'TheOutputConcept'
'IN'	'TheChainedFeatures'	'TheInputConcept'
'IN'	'TheChainedFeatures'	'TheConcepts'

A.2 RowSelectionByRandomSampling

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'HowMany'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'
'TYPE'	'HowMany'	'NUMERIC'
'GE'	'HowMany'	1

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.3 DeleteRecordsWithMissingValues

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'
'IN'	'TheTargetAttribute'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.4 RowSelectionByQuery

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheLeftCondition'	'IN'	'BA'
1	1	'TheConditionOperator'	'IN'	'V'
1	1	'TheRightCondition'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'
'IN'	'TheLeftCondition'	'TheInputConcept'
'IS_LOOPED'	'TheLeftCondition'	NULL
'IS_LOOPED'	'TheConditionOperator'	NULL
'IS_LOOPED'	'TheRightCondition'	NULL

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.5 SegmentationStratified

General Properties

Loopable	Multistepable	Manual
'NO'	'YES'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheAttribute'	'IN'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheAttribute'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.6 SegmentationByPartitioning

General Properties

Loopable	Multistepable	Manual
'NO'	'YES'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'HowManyPartitions'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'
'GE'	'HowManyPartitions'	1

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.7 FeatureSelectionByAttributes

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheOutputConcept'	'IN'	'CON'
1	NULL	'TheSelectedFeatures'	'OUT'	'FEA'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheOutputConcept'	'TheInputConcept'
'IN'	'TheSelectedFeatures'	'TheInputConcept'
'IN'	'TheSelectedFeatures'	'TheOutputConcept'

Operator Assertions

Type	Object1	Object2
'PROJ'	'TheInputConcept'	'TheOutputConcept'

A.8 LinearScaling

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'NewRangeMin'	'IN'	'V'
1	1	'NewRangeMax'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'NewRangeMin'	NULL
'IS_LOOPED'	'NewRangeMax'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'NewRangeMin'	'NUMERIC'
'TYPE'	'NewRangeMax'	'NUMERIC'
'GT'	'NewRangeMax'	'NewRangeMin'

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'GE'	'TheOutputAttribute'	'NewRangeMin'
'LE'	'TheOutputAttribute'	'NewRangeMax'

A.9 LogScaling

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'LogBase'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'LogBase'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'LogBase'	'NUMERIC'
'GT'	'LogBase'	0

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'TheTargetAttribute'	NULL
'GT'	'TheTargetAttribute'	0

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.10 AssignDefault

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'DefaultValue'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'DefaultValue'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'SAME_TYPE'	'DefaultValue'	'TheOutputAttribute'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.11 AssignModalValue

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.12 AssignMedianValue

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.13 AssignAverageValue

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.14 AssignStochasticValue

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL

A.15 MissingValuesWithRegressionSVM

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'KernelType'	'IN'	'V'
0	1	'SampleSize'	'IN'	'V'
1	1	'LossFunctionPos'	'IN'	'V'
1	1	'LossFunctionNeg'	'IN'	'V'
1	1	'C'	'IN'	'V'
1	1	'Epsilon'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
0	1	'UseDB_SVM'	'IN'	'V'
0	1	'TheKey'	'IN'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'KernelType'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'LossFunctionPos'	NULL
'IS_LOOPED'	'LossFunctionNeg'	NULL
'IS_LOOPED'	'C'	NULL
'IS_LOOPED'	'Epsilon'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'UseDB_SVM'	NULL
'IS_LOOPED'	'TheKey'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'LossFunctionPos'	'NUMERIC'
'TYPE'	'LossFunctionNeg'	'NUMERIC'
'TYPE'	'C'	'NUMERIC'
'TYPE'	'Epsilon'	'NUMERIC'
'GE'	'SampleSize'	0
'GE'	'LossFunctionPos'	0
'GE'	'LossFunctionNeg'	0
'GE'	'C'	0
'GE'	'Epsilon'	0
'ONE_OF'	'KernelType'	'dot polynomial neural radial anova'
'IN'	'TheKey'	'TheInputConcept'

A.16 SupportVectorMachineForRegression

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'KernelType'	'IN'	'V'
0	1	'SampleSize'	'IN'	'V'
1	1	'LossFunctionPos'	'IN'	'V'
1	1	'LossFunctionNeg'	'IN'	'V'
1	1	'C'	'IN'	'V'
1	1	'Epsilon'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
0	1	'UseDB_SVM'	'IN'	'V'
0	1	'TheKey'	'IN'	'BA'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'KernelType'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'LossFunctionPos'	NULL
'IS_LOOPED'	'LossFunctionNeg'	NULL
'IS_LOOPED'	'C'	NULL
'IS_LOOPED'	'Epsilon'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'UseDB_SVM'	NULL
'IS_LOOPED'	'TheKey'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'LossFunctionPos'	'NUMERIC'
'TYPE'	'LossFunctionNeg'	'NUMERIC'
'TYPE'	'C'	'NUMERIC'
'TYPE'	'Epsilon'	'NUMERIC'
'GE'	'SampleSize'	0
'GE'	'LossFunctionPos'	0
'GE'	'LossFunctionNeg'	0
'GE'	'C'	0
'GE'	'Epsilon'	0
'ONE_OF'	'KernelType'	'dot polynomial neural radial anova'
'IN'	'TheKey'	'TheInputConcept'

A.17 MissingValueWithDecisionTree

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
1	1	'PruningConf'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'PruningConf'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'ThePredictingAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictingAttributes'	'ORDINAL'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PruningConf'	'NUMERIC'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

A.18 Windowing

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TimeBaseAttrib'	'IN'	'BA'
1	1	'ValueBaseAttrib'	'IN'	'BA'
1	1	'WindowSize'	'IN'	'V'
1	1	'Distance'	'IN'	'V'
1	1	'OutputTimeStartBA'	'OUT'	'BA'
1	1	'OutputTimeEndBA'	'OUT'	'BA'
1	NULL	'WindowedValuesBA'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'TimeBaseAttrib'	NULL
'UNIQUE'	'TimeBaseAttrib'	NULL
'ORDERED'	'TimeBaseAttrib'	'INC'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TimeBaseAttrib'	'TheInputConcept'
'IN'	'ValueBaseAttrib'	'TheInputConcept'
'IN'	'OutputTimeStartBA'	'TheOutputConcept'
'IN'	'OutputTimeEndBA'	'TheOutputConcept'
'IN'	'WindowedValuesBA'	'TheOutputConcept'
'TYPE'	'TimeBaseAttrib'	'TIME'
'SAME_TYPE'	'OutputTimeStartBA'	'TimeBaseAttrib'
'SAME_TYPE'	'OutputTimeEndBA'	'TimeBaseAttrib'
'SAME_TYPE'	'ValueBaseAttrib'	'WindowedValuesBA'
'TYPE'	'WindowSize'	'NUMERIC'
'TYPE'	'Distance'	'NUMERIC'
'GE'	'WindowSize'	1
'GE'	'Distance'	1

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'OutputTimeStartBA'	NULL
'UNIQUE'	'OutputTimeStartBA'	NULL
'ORDERED'	'OutputTimeStartBA'	'INC'
'NOT_NULL'	'OutputTimeEndBA'	NULL
'UNIQUE'	'OutputTimeEndBA'	NULL
'ORDERED'	'OutputTimeEndBA'	'INC'
'LT'	'OutputTimeStartBA'	'OutputTimeEndBA'

A.19 SignalToSymbolProcessing

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'InputTimeBA'	'IN'	'BA'
1	1	'InputValueBA'	'IN'	'BA'
1	1	'Tolerance'	'IN'	'V'
1	1	'AverageValueBA'	'OUT'	'BA'
1	1	'IncreaseValueBA'	'OUT'	'BA'
1	1	'OutputTimeStartBA'	'OUT'	'BA'
1	1	'OutputTimeEndBA'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'InputTimeBA'	NULL
'UNIQUE'	'InputTimeBA'	NULL
'ORDERED'	'InputTimeBA'	'INC'
'NOT_NULL'	'InputValueBA'	'INC'

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'OutputTimeStartBA'	NULL
'UNIQUE'	'OutputTimeStartBA'	NULL
'ORDERED'	'OutputTimeStartBA'	'INC'
'NOT_NULL'	'OutputTimeEndBA'	NULL
'UNIQUE'	'OutputTimeEndBA'	NULL
'ORDERED'	'OutputTimeEndBA'	'INC'
'LT'	'OutputTimeStartBA'	'OutputTimeEndBA'
'NOT_NULL'	'AverageValueBA'	NULL
'NOT_NULL'	'IncreaseValueBA'	NULL

Further Operator Constraints

Type	Object1	Object2
'IN'	'InputTimeBA'	'TheInputConcept'
'IN'	'InputValueBA'	'TheInputConcept'
'IN'	'AverageValueBA'	'TheOutputConcept'
'IN'	'IncreaseValueBA'	'TheOutputConcept'
'IN'	'OutputTimeStartBA'	'TheOutputConcept'
'IN'	'OutputTimeEndBA'	'TheOutputConcept'
'TYPE'	'InputTimeBA'	'TIME'
'TYPE'	'InputValueBA'	'SCALAR'
'SAME_TYPE'	'OutputTimeStartBA'	'InputTimeBA'
'SAME_TYPE'	'OutputTimeEndBA'	'InputTimeBA'
'SAME_TYPE'	'AverageValueBA'	'InputValueBA'
'SAME_TYPE'	'IncreaseValueBA'	'InputValueBA'
'TYPE'	'Tolerance'	'NUMERIC'
'GE'	'Tolerance'	1

A.20 SimpleMovingFunction

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'InputTimeBA'	'IN'	'BA'
1	1	'InputValueBA'	'IN'	'BA'
1	1	'WindowSize'	'IN'	'V'
1	1	'Distance'	'IN'	'V'
1	1	'OutputTimeStartBA'	'OUT'	'BA'
1	1	'OutputTimeEndBA'	'OUT'	'BA'
1	1	'OutputValueBA'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'IN'	'InputTimeBA'	'TheInputConcept'
'IN'	'InputValueBA'	'TheInputConcept'
'IN'	'OutputTimeStartBA'	'TheOutputConcept'
'IN'	'OutputTimeEndBA'	'TheOutputConcept'
'IN'	'OutputValueBA'	'TheOutputConcept'
'TYPE'	'InputTimeBA'	'TIME'
'TYPE'	'InputValueBA'	'SCALAR'
'SAME_TYPE'	'OutputTimeStartBA'	'InputTimeBA'
'SAME_TYPE'	'OutputTimeEndBA'	'InputTimeBA'
'SAME_TYPE'	'InputValueBA'	'OutputValueBA'
'TYPE'	'WindowSize'	'NUMERIC'
'TYPE'	'Distance'	'NUMERIC'
'GE'	'WindowSize'	1
'GE'	'Distance'	1

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'InputTimeBA'	NULL
'UNIQUE'	'InputTimeBA'	NULL
'ORDERED'	'InputTimeBA'	'INC'
'NOT_NULL'	'InputValueBA'	'INC'

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'OutputTimeStartBA'	NULL
'UNIQUE'	'OutputTimeStartBA'	NULL
'ORDERED'	'OutputTimeStartBA'	'INC'
'NOT_NULL'	'OutputTimeEndBA'	NULL
'UNIQUE'	'OutputTimeEndBA'	NULL
'ORDERED'	'OutputTimeEndBA'	'INC'
'LT'	'OutputTimeStartBA'	'OutputTimeEndBA'
'NOT_NULL'	'OutputValueBA'	NULL

A.21 WeightedMovingFunction

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'InputTimeBA'	'IN'	'BA'
1	1	'InputValueBA'	'IN'	'BA'
1	NULL	'Weights'	'IN'	'V'
1	1	'Distance'	'IN'	'V'
1	1	'OutputTimeStartBA'	'OUT'	'BA'
1	1	'OutputTimeEndBA'	'OUT'	'BA'
1	1	'OutputValueBA'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'IN'	'InputTimeBA'	'TheInputConcept'
'IN'	'InputValueBA'	'TheInputConcept'
'IN'	'OutputTimeStartBA'	'TheOutputConcept'
'IN'	'OutputTimeEndBA'	'TheOutputConcept'
'IN'	'OutputValueBA'	'TheOutputConcept'
'TYPE'	'InputTimeBA'	'TIME'
'TYPE'	'InputValueBA'	'SCALAR'
'SAME_TYPE'	'OutputTimeStartBA'	'InputTimeBA'
'SAME_TYPE'	'OutputTimeEndBA'	'InputTimeBA'
'SAME_TYPE'	'InputValueBA'	'OutputValueBA'
'TYPE'	'Weights'	'NUMERIC'
'TYPE'	'Distance'	'NUMERIC'
'SUM'	'Weights'	1
'GE'	'Distance'	1

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'InputTimeBA'	NULL
'UNIQUE'	'InputTimeBA'	NULL
'ORDERED'	'InputTimeBA'	'INC'
'NOT_NULL'	'InputValueBA'	'INC'

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'OutputTimeStartBA'	NULL
'UNIQUE'	'OutputTimeStartBA'	NULL
'ORDERED'	'OutputTimeStartBA'	'INC'
'NOT_NULL'	'OutputTimeEndBA'	NULL
'UNIQUE'	'OutputTimeEndBA'	NULL
'ORDERED'	'OutputTimeEndBA'	'INC'
'LT'	'OutputTimeStartBA'	'OutputTimeEndBA'
'NOT_NULL'	'OutputValueBA'	NULL

A.22 ExponentialMovingFunction

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'InputTimeBA'	'IN'	'BA'
1	1	'InputValueBA'	'IN'	'BA'
1	1	'HeadWeight'	'IN'	'V'
1	1	'TailWeight'	'IN'	'V'
1	1	'Distance'	'IN'	'V'
1	1	'OutputTimeBA'	'OUT'	'BA'
1	1	'OutputValueBA'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'InputTimeBA'	NULL
'UNIQUE'	'InputTimeBA'	NULL
'ORDERED'	'InputTimeBA'	'INC'
'NOT_NULL'	'InputValueBA'	'INC'

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'OutputTimeBA'	NULL
'UNIQUE'	'OutputTimeBA'	NULL
'ORDERED'	'OutputTimeBA'	'INC'
'NOT_NULL'	'OutputValueBA'	NULL

Further Operator Constraints

Type	Object1	Object2
'IN'	'InputTimeBA'	'TheInputConcept'
'IN'	'InputValueBA'	'TheInputConcept'
'IN'	'OutputTimeBA'	'TheOutputConcept'
'IN'	'OutputValueBA'	'TheOutputConcept'
'TYPE'	'InputTimeBA'	'TIME'
'TYPE'	'InputValueBA'	'SCALAR'
'SAME_TYPE'	'OutputTimeBA'	'InputTimeBA'
'SAME_TYPE'	'InputValueBA'	'OutputValueBA'
'TYPE'	'HeadWeight'	'NUMERIC'
'TYPE'	'TailWeight'	'NUMERIC'
'TYPE'	'Distance'	'NUMERIC'
'GE'	'Distance'	1
'SUM'	'HeadWeight,TailWeight'	1

A.23 ComputeSVMError

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetValueAttribute'	'IN'	'BA'
1	1	'ThePredictedValueAttribute'	'IN'	'BA'
0	1	'LossFunctionPos'	'IN'	'V'
0	1	'LossFunctionNeg'	'IN'	'V'
1	1	'C'	'IN'	'V'
1	1	'Epsilon'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheTargetValueAttribute'	'TheInputConcept'
'IN'	'ThePredictedValueAttribute'	'TheInputConcept'
'TYPE'	'TheTargetValueAttribute'	'SCALAR'
'TYPE'	'ThePredictedValueAttribute'	'SCALAR'
'TYPE'	'LossFunctionPos'	'NUMERIC'
'TYPE'	'LossFunctionNeg'	'NUMERIC'
'TYPE'	'C'	'NUMERIC'
'TYPE'	'Epsilon'	'NUMERIC'
'GE'	'LossFunctionPos'	0
'GE'	'LossFunctionNeg'	0
'GE'	'C'	0
'GE'	'Epsilon'	0

Operator Conditions

Type	Object1	Object2
'NOT_NULL'	'ThePredictedValueAttribute'	NULL

A.24 Unsegment

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'UnsegmentAttribute'	'OUT'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'

A.25 SegmentationWithKMean

General Properties

Loopable	Multistepable	Manual
'NO'	'YES'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'
1	1	'HowManyPartitions'	'IN'	'V'
1	1	'OptimizePartitionsNum'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheAttributes'	'TheOutputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'ThePredictingAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictingAttributes'	'ORDINAL'
'TYPE'	'TheAttributes'	'CATEGORIAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'HowManyPartitions'	'NUMERIC'
'TYPE'	'OptimizePartitionsNum'	'NUMERIC'
'GT'	'SampleSize'	0
'ONE_OF'	'OptimizePartitionsNum'	'TRUE FALSE true false'
'GT'	'HowManyPartitions'	0
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'SUBSET'	'TheInputConcept'	'TheOutputConcept'

A.26 JoinByKey

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
2	NULL	'TheConcepts'	'IN'	'CON'
2	NULL	'TheKeys'	'IN'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'
0	1	'MapInput'	'IN'	'FEA'
0	1	'MapOutput'	'OUT'	'FEA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'MapInput'	NULL
'IS_LOOPED'	'MapOutput'	NULL
'IN'	'TheKeys'	'TheConcepts'
'IN'	'MapInput'	'TheConcepts'
'IN'	'MapOutput'	'TheOutputConcept'

A.27 SpecifiedStatistics

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
0	NULL	'AttributesComputeSum'	'IN'	'BA'
0	NULL	'AttributesComputeCount'	'IN'	'BA'
0	NULL	'AttributesComputeUnique'	'IN'	'BA'
0	NULL	'AttributesComputeDistrib'	'IN'	'BA'
0	NULL	'DistribValues'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'IN'	'AttributesComputeSum'	'TheInputConcept'
'IN'	'AttributesComputeUnique'	'TheInputConcept'
'IN'	'AttributesComputeDistrib'	'TheInputConcept'
'TYPE'	'AttributesComputeSum'	'NUMERIC'

A.28 MissingValueWithDecisionRules

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
1	1	'PruningConf'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'PruningConf'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'ThePredictingAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictingAttributes'	'ORDINAL'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PruningConf'	'NUMERIC'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

A.29 AssignPredictedValueCategorial

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictedAttribute'	'IN'	'BA'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictedAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictedAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'TYPE'	'ThePredictedAttributes'	'SCALAR'
'TYPE'	'ThePredictedAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictedAttributes'	'ORDINAL'
'TYPE'	'TheOutputAttribute'	'CATEGORIAL'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'

A.30 PredictionWithDecisionTree

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
1	1	'PruningConf'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'PruningConf'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'ThePredictingAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictingAttributes'	'ORDINAL'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PruningConf'	'NUMERIC'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

A.31 PredictionWithDecisionRules

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
1	1	'PruningConf'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'PruningConf'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'ThePredictingAttributes'	'CATEGORIAL'
'TYPE'	'ThePredictingAttributes'	'ORDINAL'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PruningConf'	'NUMERIC'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

A.32 Apriori

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheOutputConcept'	'OUT'	'CON'
1	1	'CustID'	'IN'	'BA'
1	1	'TransID'	'IN'	'BA'
1	1	'Item'	'IN'	'BA'
1	1	'MinSupport'	'IN'	'V'
1	1	'MinConfidence'	'IN'	'V'
1	1	'SampleSize'	'IN'	'V'
1	1	'PremiseBA'	'OUT'	'BA'
1	1	'ConclusionBA'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IN'	'CustID'	'TheInputConcept'
'IN'	'TransID'	'TheInputConcept'
'IN'	'Item'	'TheInputConcept'
'IN'	'PremiseBA'	'TheOutputConcept'
'IN'	'ConclusionBA'	'TheOutputConcept'

A.33 StatisticalFeatureSelection

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	NULL	'TheAttributes'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'
1	1	'Threshold'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheAttributes'	'TheInputConcept'
'TYPE'	'TheAttributes'	'SCALAR'
'TYPE'	'TheAttributes'	'CATEGORIAL'
'TYPE'	'TheAttributes'	'ORDINAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'Threshold'	'NUMERIC'
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'PROJ'	'TheInputConcept'	'TheOutputConcept'

A.34 GeneticFeatureSelection

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	NULL	'TheAttributes'	'IN'	'BA'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'
1	1	'PopDim'	'IN'	'V'
1	1	'StepNum'	'IN'	'V'
1	1	'ProbMut'	'IN'	'V'
1	1	'ProbCross'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheAttributes'	'TheInputConcept'
'IN'	'TheTargetAttribute'	'TheInputConcept'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'TheAttributes'	'SCALAR'
'TYPE'	'TheAttributes'	'CATEGORIAL'
'TYPE'	'TheAttributes'	'ORDINAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PopDim'	'NUMERIC'
'TYPE'	'StepNum'	'NUMERIC'
'TYPE'	'ProbMut'	'NUMERIC'
'TYPE'	'ProbCross'	'NUMERIC'
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'PROJ'	'TheInputConcept'	'TheOutputConcept'

A.35 SGFeatureSelection

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	NULL	'TheAttributes'	'IN'	'BA'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'SampleSize'	'IN'	'V'
1	1	'TheOutputConcept'	'OUT'	'CON'
1	1	'Threshold'	'IN'	'V'
1	1	'PopDim'	'IN'	'V'
1	1	'StepNum'	'IN'	'V'
1	1	'ProbMut'	'IN'	'V'
1	1	'ProbCross'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheAttributes'	'TheInputConcept'
'IN'	'TheTargetAttribute'	'TheInputConcept'
'TYPE'	'TheTargetAttribute'	'CATEGORIAL'
'TYPE'	'TheAttributes'	'SCALAR'
'TYPE'	'TheAttributes'	'CATEGORIAL'
'TYPE'	'TheAttributes'	'ORDINAL'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'PopDim'	'NUMERIC'
'TYPE'	'StepNum'	'NUMERIC'
'TYPE'	'ProbMut'	'NUMERIC'
'TYPE'	'ProbCross'	'NUMERIC'
'TYPE'	'Threshold'	'NUMERIC'
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'PROJ'	'TheInputConcept'	'TheOutputConcept'

A.36 FeatureSelectionWithSVM

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'TheAttributes'	'IN'	'BA'
1	1	'KernelType'	'IN'	'V'
1	1	'C'	'IN'	'V'
1	1	'Epsilon'	'IN'	'V'
1	1	'SearchDirection'	'IN'	'V'
1	1	'PositiveTargetValue'	'IN'	'V'
1	1	'TheKey'	'IN'	'BA'
1	1	'TheOutputConcept'	'OUT'	'CON'
0	1	'SampleSize'	'IN'	'V'
0	1	'UseDB_SVM'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheAttributes'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'C'	'NUMERIC'
'TYPE'	'Epsilon'	'NUMERIC'
'TYPE'	'SampleSize'	'NUMERIC'
'GE'	'C'	0
'GE'	'Epsilon'	0
'GE'	'SampleSize'	0
'ONE_OF'	'KernelType'	'dot polynomial neural radial anova'
'ONE_OF'	'SearchDirection'	'backward forward'
'ONE_OF'	'UseDB_SVM'	'true false'
'IN'	'TheKey'	'TheInputConcept'
'SAME_FEAT'	'TheOutputConcept'	'TheInputConcept'

Operator Assertions

Type	Object1	Object2
'PROJ'	'TheInputConcept'	'TheOutputConcept'

A.37 SupportVectorMachineForClassification

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'KernelType'	'IN'	'V'
0	1	'SampleSize'	'IN'	'V'
1	1	'C'	'IN'	'V'
1	1	'Epsilon'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'
1	1	'PositiveTargetValue'	'IN'	'V'
0	1	'UseDB_SVM'	'IN'	'V'
0	1	'TheKey'	'IN'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'ThePredictingAttributes'	NULL
'IS_LOOPED'	'KernelType'	NULL
'IS_LOOPED'	'SampleSize'	NULL
'IS_LOOPED'	'C'	NULL
'IS_LOOPED'	'Epsilon'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'PositiveTargetValue'	NULL
'IS_LOOPED'	'UseDB_SVM'	NULL
'IS_LOOPED'	'TheKey'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'ThePredictingAttributes'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheTargetAttribute'	'TheOutputAttribute'
'TYPE'	'ThePredictingAttributes'	'SCALAR'
'TYPE'	'SampleSize'	'NUMERIC'
'TYPE'	'C'	'NUMERIC'
'TYPE'	'Epsilon'	'NUMERIC'
'GE'	'SampleSize'	0
'GE'	'C'	0
'GE'	'Epsilon'	0
'ONE_OF'	'KernelType'	'dot polynomial neural radial anova'
'IN'	'TheKey'	'TheInputConcept'

Operator Conditions

Type	Object1	Object2
'HAS_NULLS'	'TheTargetAttribute'	NULL
'HAS_VALUES'	'TheTargetAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

Operator Assertions

Type	Object1	Object2
'NOT_NULL'	'TheOutputAttribute'	NULL
'NOT_NULL'	'ThePredictingAttributes'	NULL

A.38 GenericFeatureConstruction

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'SQL_String'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'SAME_TYPE'	'TheOutputAttribute'	'TheTargetAttribute'
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'SQL_String'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'TYPE'	'SQL_String'	'NOMINAL'

A.39 UnionByKey

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
2	NULL	'TheConcepts'	'IN'	'CON'
2	NULL	'TheKeys'	'IN'	'BA'
0	1	'MapInput'	'IN'	'FEA'
0	1	'MapOutput'	'OUT'	'FEA'
1	1	'TheOutputConcept'	'OUT'	'CON'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'MapInput'	NULL
'IS_LOOPED'	'MapOutput'	NULL
'IN'	'TheKeys'	'TheConcepts'
'IN'	'MapInput'	'TheConcepts'
'IN'	'MapOutput'	'TheOutputConcept'

A.40 TimeIntervalManualDiscretization

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'IntervalStart'	'IN'	'V'
1	NULL	'IntervalEnd'	'IN'	'V'
1	NULL	'MapTo'	'IN'	'V'
1	NULL	'StartIncExc'	'IN'	'V'
1	NULL	'EndIncExc'	'IN'	'V'
1	1	'DefaultValue'	'IN'	NULL
1	1	'TimeFormat'	'IN'	'V'
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'IntervalStart'	NULL
'IS_LOOPED'	'StartIncExc'	NULL
'IS_LOOPED'	'IntervalEnd'	NULL
'IS_LOOPED'	'EndIncExc'	NULL
'IS_LOOPED'	'MapTo'	NULL
'IS_LOOPED'	'DefaultValue'	NULL
'IS_LOOPED'	'TimeFormat'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'TYPE'	'TheTargetAttribute'	'TIME'
'TYPE'	'IntervalStart'	'TIME'
'TYPE'	'IntervalEnd'	'TIME'
'TYPE'	'MapTo'	'NOMINAL'
'TYPE'	'StartIncExc'	'CATEGORIAL'
'TYPE'	'EndIncExc'	'CATEGORIAL'
'TYPE'	'DefaultValue'	'NOMINAL'
'TYPE'	'TimeFormat'	'NOMINAL'
'TYPE'	'TheOutputAttribute'	'CATEGORIAL'
'ONE_OF'	'StartIncExc'	'I E'
'ONE_OF'	'EndIncExc'	'I E'

A.41 NumericalIntervalManualDiscretization

General Properties

Loopable	Multistepable	Manual
'YES'	'NO'	'YES'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	NULL	'IntervalStart'	'IN'	'V'
1	NULL	'IntervalEnd'	'IN'	'V'
1	NULL	'MapTo'	'IN'	'V'
1	NULL	'StartIncExc'	'IN'	'V'
1	NULL	'EndIncExc'	'IN'	'V'
1	1	'DefaultValue'	'IN'	NULL
1	1	'TheOutputAttribute'	'OUT'	'BA'

Further Operator Constraints

Type	Object1	Object2
'IS_LOOPED'	'TheTargetAttribute'	NULL
'IS_LOOPED'	'TheOutputAttribute'	NULL
'IS_LOOPED'	'IntervalStart'	NULL
'IS_LOOPED'	'StartIncExc'	NULL
'IS_LOOPED'	'IntervalEnd'	NULL
'IS_LOOPED'	'EndIncExc'	NULL
'IS_LOOPED'	'MapTo'	NULL
'IS_LOOPED'	'DefaultValue'	NULL
'IN'	'TheTargetAttribute'	'TheInputConcept'
'IN'	'TheOutputAttribute'	'TheInputConcept'
'TYPE'	'TheTargetAttribute'	'NUMERIC'
'TYPE'	'IntervalStart'	'TIME'
'TYPE'	'IntervalEnd'	'TIME'
'TYPE'	'MapTo'	'NOMINAL'
'TYPE'	'StartIncExc'	'CATEGORIAL'
'TYPE'	'EndIncExc'	'CATEGORIAL'
'TYPE'	'DefaultValue'	'NOMINAL'
'TYPE'	'TheOutputAttribute'	'CATEGORIAL'
'ONE_OF'	'StartIncExc'	'I E'
'ONE_OF'	'EndIncExc'	'I E'

A.42 SubgroupMining

General Properties

Loopable	Multistepable	Manual
'NO'	'NO'	'NO'

ParameterConstraints

MINARG	MAXARG	Parameter Name	In/Out	Type
1	1	'TheInputConcept'	'IN'	'CON'
1	1	'TheTargetAttribute'	'IN'	'BA'
1	1	'TheKey'	'IN'	'BA'
1	NULL	'ThePredictingAttributes'	'IN'	'BA'
1	1	'TargetValue'	'IN'	'V'
1	1	'SearchDepth'	'IN'	'V'
1	1	'MinSupport'	'IN'	'V'
1	1	'MinConfidence'	'IN'	'V'
1	1	'NumHypotheses'	'IN'	'V'
1	1	'RuleClusters'	'IN'	'V'

Further Operator Constraints

Type	Object1	Object2
'IN'	'Target'	'InputConcept'
'IN'	'Key'	'InputConcept'
'IN'	'AttributeSpace'	'InputConcept'
'ONE_OF'	'RuleClusters'	'YES NO'

Appendix B

B.1 What this appendix is about

This appendix explains two things in detail: Firstly, section B.2 describes some details about how the MiningMart compiler expects the metadata for a case description to be set up. Secondly, section B.3 describes the current operators and their parameters.

B.2 Compiler constraints on metadata

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

B.2.1 Naming conventions

Operator names

The name of an operator (entry `op_name` in M4 table `Operator_T`) corresponds exactly (respecting case!) to the Java class that implements this operator in the compiler. This is only important to know if you want to implement additional operators. What is more generally important is that the names of the parameters of an operator are also fixed, because the compiler recognizes the type of a parameter by its name. This is described in more detail in section B.3.1.

BaseAttribute names

Some operators have as their output on the conceptual level a Concept rather than a BaseAttribute (see section B.3.1). This output Concept will generally be similar to the input Concept, in the sense that it copies some of the input BaseAttributes without changing them. To find out which BaseAttribute in the output Concept corresponds to which BaseAttribute in the input concept, their names are used. They must match exactly, ignoring case. This also means that it is necessary to give the output BaseAttribute in a feature construction operator (see section B.3.1) a name which is different

from all BaseAttribute names in the input Concept, so that no names are mixed up. If the output of the operator is a Concept, and a BaseAttribute in this output concept has no corresponding BaseAttribute in the input concept, it will be ignored by the compiler, because it may be needed for later steps. Ignoring means that no Column is created for it.

A similar mechanism is applied when Relations are used (see following section B.2.2).

B.2.2 Relations

Relations are defined by the user between the initial Concepts of a case. In a case, the Concepts may then be modified. If later in the chain an operator is applied that makes use of relations, it must be able to find the Columns that realize the keys. To this end, again the names of the BaseAttributes are used. Currently only `MultiRelationalFeatureConstruction` (MRFC) uses relations. This means that in the Concepts used by MRFC, the BaseAttributes that correspond to the key BaseAttributes in the initial Concepts must have the same name (ignoring case).
Example: Suppose there are initial Concepts *Customer* and *Product* linked by a relation *buys* which is realized by a foreign link from the *Customer* to the *Product* table. The foreign key Column in the *Customer* table is named `fk_prod` and its BaseAttribute is named *CustomerBuys*. The Concept *Customer* may be the input to a chain which results in a new Concept *PrivateCustomer*. This new Concept must still have a BaseAttribute named *CustomerBuys*, which must not be the result of a feature construction, but must be copied from Concept to Concept in the chain[15]. Then the compiler can find the Column `fk_prod` by comparing the BaseAttributes of the current input concept *PrivateCustomer* and of the Concept which is linked to the relation *buys* (this relation is an input to the MRFC operator). The Column can be used to join the two Concepts *PrivateCustomer* and *Product*, although the first is a subconcept of *Customer*.

B.3 Operators and their parameters

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

B.3.1 General issues

There are two kinds of operators, distinguished by their output on the conceptual level: those that have an output Concept (*Concept Operators*, listed

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

in section B.3.2), and those that have an output BaseAttribute (*Feature Construction Operators*, listed in section B.3.4).

All operators have parameters, such as input Concept or output BaseAttribute. The name of such a parameter is fixed, for instance *TheInputConcept* is used for the input Concept for all operators. This means that the entry for this parameter in `par_name` in the M4 table `Parameter_T` must be *TheInputConcept*, respecting case. The parameter specification for each operator is stored in the M4 table OP_PARAMS_T (see chapter 3).

Some operators have an unspecified number of parameters of the same type. For example, the learning operators take as input a number of BaseAttributes of the same concept and use them to construct their training examples. All these BaseAttributes use the same prefix for their parameter name (here *ThePredictingAttributes*) in `Parameter_T`. Since all parameters for one step are expected to have different names (for HCI use), number suffixes are added to these prefixes (*ThePredictingAttributes1, ThePredictingAttributes2*, etc). The compiler uses `ORDER BY par_nr` when reading them. Such parameters, which may contain a list, are marked with the word *List* in the operator descriptions in sections B.3.2 and B.3.4.

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

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

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

In the following sections, all current operators are listed with their exact

name (see section B.2.1), a short description and the names of their parameters. In general, all input BaseAttributes belong to the input Concept, and all output BaseAttributes belong to the output Concept.

B.3.2 Concept operators

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

MultiRelationalFeatureConstruction

Takes a list of concepts which are linked by relations, and selects specified Features from them which are collected in the output Concept, via a join on the concepts of the chain. To be more precise: Recall (section B.2.2) that Relations are only defined by the user between initial Concepts of a Case. Suppose there is a chain of initial Concepts $C_1, \ldots, C_n$ such that between all C_i and $C_{i+1}, 1 \leq i < n$, C_i is the *FromConcept* of the i-th Relation and C_{i+1} is its *ToConcept*. These Concepts may be modified in the Case being modelled, to result in new Concepts $C'_1, \ldots, C'_n$, where some C'_i may be equal to C_i. However, as explained in section B.2.2, the BaseAttributes that correspond to the Relation keys are still present in the new Concepts C'_i. By using their names, this operator can find the key Columns and join the new Concepts C'_i.

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

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

JoinByKey

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

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

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

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

UnionByKey

Takes a list of concepts, plus attributes indicating their primary keys, and unifies the concepts. In contrast to the operator JoinByKey (section B.3.2), 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 must be present. Each `BaseAttribute` specified in *TheKeys* must be a primary key of one of *TheConcepts*; thus, the number of entries in *TheConcepts* and *TheKeys* must be equal.

If several of the input concepts contain a `BaseAttribute` (or a `MultiColumnFeature`) with the same name, a special mapping mechanism is needed to relate them to different features in *TheOutputConcept*. For this, the parameters *MapInput* and *MapOutput* exist. Use *MapInput* to specify any feature in one of *TheConcepts*, and use *MapOutput* to specify the **corresponding** feature in *TheOutputConcept*. To make sure that for each *MapInput* the right *MapOutput* is found by this operator, it uses the looping mechanism. Although the parameter is not looped, the loop numbers in the parame-

ter table in M4 are used to ensure the correspondence between *MapInput* and *MapOutput*. However, these two parameters only need to be specified for every pair of equally-named features in *TheConcepts*. So there are not necessarily as many "loops" as there are features in *TheOutputConcept*.

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

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

SpecifiedStatistics

An operator which computes certain statistical values for the *TheInputConcept*. The computed values appear in a `ColumnSet` which contains exactly one row with the statistical values, and which belongs to *TheOutputConcept*.

The sum of all values in an attribute can be computed by specifying a `BaseAttribute` with the parameter *AttributesComputeSum*. There can be more such attributes; the sum is computed for each. *TheOutputConcept* must contain a `BaseAttribute` for each sum which is computed; their names must be those of the input attributes, followed by the suffix "`_SUM`".

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

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

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 `BaseAttribute`s will contain the number of entries in the input where the value was 2, 4 and 6, respectively. For each `BaseAttribute` whose value distribution is to be computed, the possible values must be given with the parameter *DistribValues*. One entry in this parameter is a comma-separated string containing the different

values; in the example, the string would be "2,4,6". Thus, the number of entries in *AttributesComputeDistrib* and *DistribValues* must be equal. *TheOutputConcept* must contain the corresponding number of BaseAttributes (three in the example); their names must be those of the input attributes, followed by the suffix "_<value>". In the example, *TheOutputConcept* would contain the BaseAttributes "inputBaName_2' ', "inputBaName_4" and "inputBaName_6".

ParameterName	ObjectType	Type	Remarks
TheInputConcept	CON	IN	inherited
AttributesComputeSum	BA *List*	IN	numeric
AttributesComputeCount	BA *List*	IN	(see
AttributesComputeUnique	BA *List*	IN	
AttributesComputeDistrib	BA *List*	IN	text)
DistribValues	V *List*	IN	
TheOutputConcept	CON	OUT	inherited

UnSegment

This operator is the inverse to any segmentation operator (see B.3.2, B.3.2, B.3.2). While a segmentation operator segments its input concept's ColumnSet into several ColumnSets, UnSegment joins several ColumnSets into one. This operator makes sense only if a segmentation operator was applied previously in the chain, because it exactly reverses the function of that operator. To do so, the parameter *UnsegmentAttribute* specifies indirectly which of the three segmentation operators is reversed:

If a SegmentationStratified operator is reversed (section B.3.2), this parameter gives the name of the BaseAttribute that was used for stratified segmentation. Note that this BaseAttribute must belong to *TheOutputConcept* of this operator, because the re-unified ColumnSet contains different values for this attribute (whereas before the execution of this operator, the different ColumnSets did not contain this attribute, but each represented one of its values).

If a SegmentationByPartitioning operator is reversed (section B.3.2), this parameter must have the value "(Random)".

If a SegmentationWithKMean operator is reversed (section B.3.2), this parameter must have the value "(KMeans)".

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

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

RowSelectionByQuery

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

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

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

RowSelectionByRandomSampling

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

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

DeleteRecordsWithMissingValues

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

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

SegmentationStratified

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

makes use of the statistics tables and the functions that compute the statistics).

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

SegmentationByPartitioning

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

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

SegmentationWithKMean

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

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

Windowing

Windowing is applicable to time series data. It takes two BaseAttributes from the input Concept; one of contains time stamps, the other values. In the output Concept each row gives a time window; there will be two time stamp BaseAttributes which give the beginning and the end of each time

window. Further, there will be as many value attributes as specified by the *WindowSize*; they contain the values for each window. *Distance* gives the distance between windows in terms of number of time stamps.

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

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 B.3.2.

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

WeightedMovingFunction

This operator works like SimpleMovingFunction (section B.3.2), but the weighted average is computed. The window size is not given explicitly, but is determined from the number of *Weights* given. The sum of all *Weights* must be 1.

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

ExponentialMovingFunction

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

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

SignalToSymbolProcessing

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

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

Apriori

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

The input format is fixed. There is one input concept (*TheInputConcept*) having a BaseAttribute for the customer ID (parameter: *CustID*), one for the transaction ID (*TransID*), and one for an item part of this customer/transaction's itemset (*Item*). The algorithm expects all entries of these `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

B.3.3 Feature selection operators

Feature selection operators are also concept operators in that their output is a `Concept`, but they are listed in their own section since they have some common special properties. All of them (except *FeatureSelectionByAttributes*, see B.3.3) use external algorithms to determine which features are taken over to the output concept. This means that at the time of designing an operating chain, it is not known which features will be selected. How can a complete, valid chain be designed then, since the input of later operators may depend on the output of a feature selection operator, which is only determined at compile time?

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

All feature selection operators have a parameter *TheAttributes* which specifies the set of features from which some are to be selected. (Again this is not true for *FeatureSelectionByAttributes*, see B.3.3.) The parameter is needed because not all of the features of *TheInputConcept* can be used, as they may include a key attribute or the target attribute for a data mining step, which should not be deselected.

FeatureSelectionByAttributes

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

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

StatisticalFeatureSelection

A Feature Selection operator. This operator uses the stochastic correlation measure to select a subset of *TheAttributes*. All of *TheAttributes* must be

present in *TheOutputConcept*. The parameter *Threshold* is a real number between 0 and 1 (default is 0.7). *SampleSize* specifies a maximum number of examples that are fed into the external algorithm.

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA *list*	IN	see section B.3.3
SampleSize	V	IN	positive integer
Threshold	V	IN	real between 0 and 1
TheOutputConcept	CON	OUT	inherited

GeneticFeatureSelection

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

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA *list*	IN	see section B.3.3
SampleSize	V	IN	positive integer
PopDim	V	IN	positive integer; try 30
StepNum	V	IN	positive integer; try 20
ProbMut	V	IN	real between 0 and 1; try 0.001
ProbCross	V	IN	real between 0 and 1; try 0.9
TheOutputConcept	CON	OUT	inherited

SGFeatureSelection

A Feature Selection operator. This operator is a combination of *StochasticFeatureSelection* (see B.3.3), which is applied first, and *GeneticFeatureSelection* (see B.3.3), applied afterwards. The parameter descriptions can be found in the sections about these operators (B.3.3 and B.3.3).

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheAttributes	BA *list*	IN	see section B.3.3
SampleSize	V	IN	
PopDim	V	IN	
StepNum	V	IN	
ProbMut	V	IN	
ProbCross	V	IN	
Threshold	V	IN	real, between 0 and 1
TheOutputConcept	CON	OUT	inherited

B.3.4 Feature construction operators

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

AssignAverageValue

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

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

AssignModalValue

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

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

AssignMedianValue

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

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

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

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

MissingValuesWithRegressionSVM

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

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

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

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

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

ParameterName	ObjType	Type	Remarks
TheInputConcept	CON	IN	inherited
TheTargetAttribute	BA	IN	inherited
ThePredictingAttributes	BA *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

LinearScaling

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

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

LogScaling

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

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

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*.

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

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*.

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

MissingValueWithDecisionRules

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

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

MissingValueWithDecisionTree

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

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

PredictionWithDecisionRules

A Feature Construction operator. Decision rules are learned using *ThePredicting-Attributes* as learning attributes and *TheTargetAttribute* as label. *TheOutputAttribute* contains the labels that the decision rules predict. The operator may be used to compare predicted and actual values, or in combination with the operator AssignPredictedValueCategorial (see section B.3.4). All *ThePredictingAttributes* must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

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

PredictionWithDecisionTree

A Feature Construction operator. A Decision Tree is learned using *ThePredicting-Attributes* as learning attributes and *TheTargetAttribute* as label. *TheOutputAttribute* contains the labels that the decision tree predicts. The operator may be used to compare predicted and actual values, or in combination with the operator AssignPredictedValueCategorial (see section B.3.4). All *ThePredictingAttributes* must belong to *TheInputConcept*. The pruning confidence level is given in *PruningConf* as a percentage.

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

AssignPredictedValueCategorial

A Missing Value operator. Any missing value of *TheTargetAttribute* is replaced by the value of the same row from *ThePredictedAttribute*. The latter may have been filled by the operator PredictionWithDecisionRules (B.3.4) or PredictionWithDecisionTree (B.3.4). It must belong to *TheInputConcept*.

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

GenericFeatureConstruction

This operator creates an output attribute on the basis of a given SQL definition (Parameter *SQL_String*). The definition must be well-formed SQL defining how values for the output attribute are computed based on one of the attributes in *TheInputConcept*. To refer to the attributes in *TheInputConcept*, the names of the `BaseAttributes` are used—and not the names of any `Columns`. For example, if there are two `BaseAttributes` named "INCOME" and "TAX" in *TheInputConcept*, this operator can compute their sum if *SQL_String* is defined as "(INCOME + TAX)". Since the operator must resolve names of `BaseAttributes`, it cannot be used if there are two or more `BaseAttributes` in *TheInputConcept* with the same name.

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

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

TimeIntervalManualDiscretization

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

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

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

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

NumericIntervalManualDiscretization

This operator can be used to discretize a numeric attribute manually. It is very similar to the operator TimeIntervalManualDiscretization described in B.3.4. The looped parameters *IntervalStart*, *IntervalEnd*, *StartIncExc*, *EndIncExc*, and *MapTo*. again specify a mapping to be performed. If an input value does not belong to any interval, it is mapped to the value *DefaultValue*. *TheTargetAttribute* needs to be of type ordinal.

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

B.3.5 Other Operators

ComputeSVMError

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

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

SubgroupMining

A special operator without output on the conceptual level. The output of the algorithm is a textual description of discovered subgroups which will be printed to the compiler output (log file). The operator is only applicable to a table which is suitable for spatial subgroup discovery. Thus, *ThePredictingAttributes* must only contain categorial data. Therefore only features with a finite (and small) number of distinct values should be selected.

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

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

Bibliography

[1] A. Bernstein, S. Hill, and F. Provost. An intelligent assitsant for the knowledge discovery process. Technical Report IS02-02, New York University, Leonard Stern School of Business, 2002.

[2] M. Botta and A. Giordana. SMART+: A multi-strategy learning tool. In *IJCAI-93, Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 937–943, Chambéry, France, 1993.

[3] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[4] C. Domingo and O. Watanabe. Scaling Up a Boosting-Based Learner via Adaptive Sampling. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 317–328, 2000.

[5] A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 41:217–251, 2000.

[6] A. Giordana, L. Saitta, and M. S. amd M. Botta. An experimental study of phase transitions in matching. In *Proceedings of th 17th International Conference on Machine Learning*, pages 311,318, Stanford, CA, 2000.

[7] G. H. John and P. Langley. Static Versus Dynamic Sampling for Data Mining. In *Proceedings of the Second International Conference on Knowledge Discovery in Databases and Data Mining*, 1996.

[8] J. U. Kietz. *Induktive Analyse relationaler Daten.* PhD thesis, Technische Universität Berlin, Berlin, oct 1996.

[9] J. Kivinen and H. Mannila. The power of sampling in knowledge discovery. In *Proc. Thirteenth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 77–85, 1994.

[10] D. L., T. H., and K. R.D. Finding frequent substructures in chemical compounds. In *Int Conf. on Knowledge Discovery and Data Mining*, pages 30–36, New York, NY, 1998.

[11] R. Michalski. A theory and methodology of inductive learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134, Los Altos, CA, 1983. Morgan Kaufmann.

[12] S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, London, UK, 1992.

[13] S. Muggleton. Inverse entailment and PROGOL. *New Gen. Comput.*, 13:245–286, 1995.

[14] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–110, 1996.

[15] R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[16] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[17] T. Scheffer and S. Wrobel. A Sequential Sampling Algorithm for a General Class of Utility Criteria. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2000.

[18] T. Scheffer and S. Wrobel. Active learning of partially hidden Markov models. In *Proc. of Workshop at ECML-2001/PKDD-2001: Active Learning, Database Sampling, Experimental Design: Views on Instance Selection*, 2001.

[19] T. Scheffer and S. Wrobel. Finding the Most Interesting Patterns in a Database Quickly by Using Sequential Sampling. Technical report, University of Magdeburg, 2001.

[20] T. Scheffer and S. Wrobel. Incremental Maximization of Non-Instance-Averaging Utility Functions with Applications to Knowledge Discovery Problems. In *Proceedings of the Eighteenth International Conference on Machine Leraning*, 2001.

[21] H. Toivonen. Sampling large databases for association rules. In A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd VLDB Conference*, pages 134–145. Morgan Kaufmann, 1996.