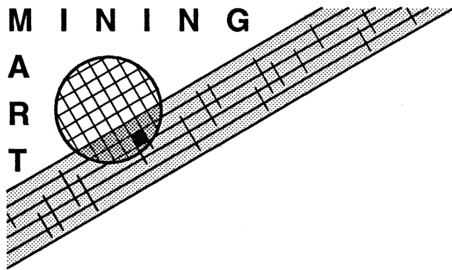

Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D3

Learning about Time

Katharina Morik and Harald Liedtke

University of Dortmund, Computer Science VIII
D-44221 Dortmund, Germany
{morik,liedtke}@ls8.cs.uni-dortmund.de
http://www-ai.cs.uni-dortmund.de

June 30, 2000

Abstract

This deliverable presents known techniques to handle time phenomena and gives meta data of those that are well suited for knowledge discovery. Techniques from statistics and machine learning that expicitly handle time phenomena are investigated. Moreover, the tricks of transforming time dependent data into a form that can be processed by methods that are incapable of explicitly handling time are described.

The methods investigated include different kinds of time phenomena (i.e., seasonal effects, sequences, cycles, time series, time intervals and relations between them). We identify six basic input formats frequently used for time related learning (section 2) and nine important preprocessing operators (section 3). For these formats and operators meta data are provided. Furthermore, eight learning methods are described in detail (section 4). These descriptions include meta data for input and output of the learning methods which will enable the integration of external learning methods to the Mining Mart.

General meta data are summarized in section 1, specific meta data are introduced in the remaining sections. All meta data described here – and the meta data from work package 1 and 5 – will have to be integrated into the meta data language that will be developed in work packages 8 and 9.

Contents

1 Meta Data **1**
1.1 Time related Meta Data 1
1.2 Meta Data required in order to apply Operators 3
1.3 Definitions and Metrics 4

2 Input Formats **7**
2.1 Multivariate Time Series 7
2.2 Univariate Time Series 8
2.3 Nominal valued Time Series 8
2.4 Sequence Vectors 9
2.5 Database Table 9
2.6 Tuples for Time Points 10
2.7 Facts 10

3 Time related Preprocessing Operators **12**
3.1 Single Process Extraction 12
3.2 Multivariate to Univariate Transformation 13
3.3 Sliding Windows 15
3.4 Summarizing 16
3.5 Multiple Learning 17
3.6 Aggregation 18
3.7 Time Segmentation 18
3.8 Time Feature Construction 19
3.9 Implicit Use of Time Information 20

4 Learning Methods **22**
4.1 Prediction Task 22
4.2 Characterization 23
4.3 Time Regions 23
4.4 Association Rules and Sequential Patterns 25
4.5 Frequent Sequences 27
4.6 Level Changes 29
4.7 Signal to Symbol Processing 29

4.8 Clustering 30

5 Experiments **32**

5.1 Proprietary Meta Data Language 32
5.2 Experiment 1 36
5.3 Experiment 2 48
5.4 Experiment 3 50

List of Figures

3.1 Concatenating data from distributed time points 20
3.2 C4.5 decision tree for $L_{H_{iT}}$ 21

4.1 Episode detection with Best Region Rules 24
4.2 Event Sequence 28
4.3 Classes of Episodes 28
4.4 Discretized time series using clusters 31

5.1 Semi-automatic operator selection by meta data matching . . 35
5.2 Sales of article 182830 in shop 055 43

Chapter 1

Meta Data

To describe input formats and meta data the following notation is used:

- Input data are described by formats L_E and identified by a name l_E. L_E comprises either one or several individuals i_l. In case of one individual this individual i_l is used in the definition of L_E, in case of a set I of individuals i_l the set I is used.
- Input formats L_E for a method (operator) producing L_H.
- Attributes A_j and attribute values a_j.
- Abstract time specifications T_j and instantiations t_j, index i always denotes the actual or the last time specification. Since the time specification may consist of one or two time attributes (explanation below) the term time specification is used here rather than *displacement variable* or *index variable* [14].

Delivery D1 introduces a first meta data scheme. Some of the meta data listed below are already part of the scheme. Other meta data will have to be integrated into the scheme. This will be part of the work packages 8 and 9 which focus on describing the meta data (model).

General meta data and meta data which are necessary for the descriptions of input formats (see section 2) are introduced in the following subsections. Additional, specific meta data that are required for describing particular preprocessing operators or learning methods will be given in the remaining chapters.

1.1 Time related Meta Data

An input l_E which includes time specifications t has to provide additional information. These meta data are mandatory for all time related data.

- the *identification of an individual.* Possibly there is none in the raw data, because all observations belong to one single individual. In some cases the individual's name is given implicitly, e.g. by the database tablename. In these cases the identification has to be added.

  This specification of an individual is necessary, e.g. to transform a multivariate time series to a univariate one (see 3.2). Of course there can be more than one identifier. For instance, consider a database table containing insurance policies. Depending on the point of view the underlying individual can be identified by the kind of insurance, the client identification, or some other attribute. For that reason a collection of individual identifiers can be specified with each individual identifier having an optional individual description.

  For each of the individual identifiers the number of different individual instantiations is required. This is one example for the meta data already included in the meta data scheme of Deliverable D1: the information is given by the attribute *MA_UNIQUE* of the table *METAATTRIBS* which denotes the number of different values for a given attribute. A time series always belongs to one individual but other representations - like sequence vectors - may describe several individuals.

- the attribute(s) containing the time specification (*time attributes*)

- the *time scale* indicated by the *scale unit* and the *point of reference*:

  - the *scale unit* (e.g. seconds, minutes, weeks, months, concrete time stamps)
  - the *point of reference*, (e.g. the start of vital sign measurement for a patient in an intensive care unit)
  - the order of time stamps (see 1.2)

- the *representation of time*: *points in time* or *time intervals* defined by a tuple $t = (t_s, t_e)$ with starting point t_s less or equal than ending point t_e.

Data that does not include explicit time information by having time attributes might include implicit time information: Is the time information given implicitly by the order of an individual's vectors? If not, we will not apply any operator considering time phenomena for the given data.

For any given input these meta data will help to decide whether the input belongs to one of the input languages (see section 2) or not. Furthermore the same meta data could support the mapping of any input to one of the input languages, and from one input language to another.

1.2 Meta Data required in order to apply Operators

Subject to the preprocessing or learning operator which should be applied some specific meta data is required.

Time series with time specifications (time-stamped time series) are classified as *uniform* or *generic* time series depending on the time-stamps' characteristics. The time specifications t_j of uniform time series are of unique length (this is always true if time specifications are instantiated by time points) and the distances between two time specifications are the same for all pairs of sequent time specifications:

$$\begin{aligned} &\textit{individual } i_l \textit{ is generic} \\ &(\textit{monotonically increasing}) :\Longleftrightarrow && (t_1, \ldots, t_i) \textit{ is monotonically increasing} \\ &\Longleftrightarrow && \forall j \in \{1, \ldots, i-1\} : t_j \leq t_{j+1} \end{aligned} \tag{1.1}$$

$$\begin{aligned} &\textit{individual } i_l \textit{ is strictly} \\ &\textit{monotonic increasing} :\Longleftrightarrow && (t_1, \ldots, t_i) \textit{ is strictly monotonic increasing} \\ &\Longleftrightarrow && \forall j \in \{1, \ldots, i-1\} : t_j < t_{j+1} \end{aligned} \tag{1.2}$$

$$\begin{aligned} &\textit{individual } i_l \textit{ is} \\ &\textit{uniformly increasing} :\Longleftrightarrow && i_l \textit{ is strictly monotonic increasing} \\ & && \exists d : \ \forall j \in \{1, \ldots, i-1\} : \\ & && \wedge t_{j+1} - t_j = d \wedge |t_j| = |t_{j+1}| \end{aligned} \tag{1.3}$$

Strictly monotonic decreasing, *monotonically decreasing*, and *uniformly decreasing* is defined likewise.

If the time points were not ordered chronologically the time series is called *unsorted* and *sorted* otherwise.

$$\begin{aligned} &\textit{individual } i_l \\ &\textit{is unsorted} :\Longleftrightarrow && \exists j, l \in \{1, \ldots, i-1\}, j \neq l : \\ & && t_j > t_{j+1} \wedge t_l < t_{l+1} \end{aligned} \tag{1.4}$$

For each of the attributes some *attribute properties* are needed: is the

attribute value numerical or nominal (*type*). Are there any *missing values*? If so, how much values are missing (in total and in percent)? An important point is that in real-world applications missing values are not only coded by the attribute value 'NULL' (using database terms). Often mappings are used to code specific attribute values like 'unknown' or 'contractor did not fill in this blank' which can possibly be of the same meaning as 'NULL'. To take this aspect into account the meta data provide an optional list of *NULL values* which identify missing values. A simple example for missing time information is using '9.9.9999' as a date that has not been specified. In many cases learning on '9.9.9999' will not be of great importance.

Additional statistics could be of interest for certain operators: total amount of values, number of different values, mean value, standard deviation, variance, and range. The meta data have to provide corresponding slots for these information.

These meta data are provided for given data as well as a requirement of attributes to apply a specific operator. It is quite important that a step of a data mining case is not only associated with a view and its attributes (or with attributes only) which provide meta data. The meta data also have to provide explicit information about the need for specific meta data as a prerequisite for the data mining step! E.g., if an attribute is associated with a step you will have to know if the type of the attribute is numerical by coincidence or if it is numerical because the step requires the attribute to be of that specific type.

1.3 Definitions and metrics useful for extracting semantic or quality related information from inputs

Because we do not want to distinguish between the two representations of time when trying to get the start or the end of a time specification t the starting and the ending point are defined as simple as

$$start(t) := \begin{cases} t & ; \; t \text{ is point in time} \\ t_s & ; \; t \text{ is time interval } (t_s, t_e) \end{cases}$$
$$end(t) := \begin{cases} t & ; \; t \text{ is point in time} \\ t_e & ; \; t \text{ is time interval } (t_s, t_e) \end{cases} \quad (1.5)$$

The relations $<$ and $\leq$ for time intervals are defined by using Allen's interval relations [4]. These new relations will be used for defining input formats later on and they were already used in (1.1) – (1.4).

$$< \ := \ \mathit{before} \ \cup \ \mathit{meets} \tag{1.6}$$

$$\leq \ := \ < \ \cup \ \mathit{overlaps} \ \cup \ \mathit{finished-by} \ \cup \ \mathit{contains} \ \cup \ \mathit{equal} \tag{1.7}$$

The distance from time specification t_1 to t_2 is:

$$t_2 - t_1 \ := \ start(t_2) - end(t_1) \tag{1.8}$$

The length $|t|$ of a time specification t
is defined as

$$|t| \ := \ end(t) - start(t) + 1 \tag{1.9}$$

where *+ 1* has the meaning of adding one unit (w.r.t to the chosen scale unit) to the distance between the two time specifications.

W. r. t. (1.3) the frequency of an uniform individual i_l is defined as

$$freq(i_l) \ := \ d \tag{1.10}$$

E.g., if a vector has been recorded every two seconds then the scale unit chosen should be seconds and the frequency is 2.

For the input languages which will be introduced in the next section some conditions are conceivable:

Given a set of instantiations of an input language L_E where $k_1, \ldots, k_l$ denote the number of attributes the condition

$$k_1 \ = \ k_2 \ = \ \ldots \ = \ k_l \tag{1.11}$$

indicate the special case of all instantiations having the same number of attributes.

Whereas the other extreme is given by the following condition:

$$k_1 \ \neq \ k_2 \ \neq \ \ldots \ \neq \ k_l \tag{1.12}$$

The meta data provide the number of attributes for each single instantiation and the last two conditions can be verified to be true or not.

Given an individual i_l whose first time specification is t_1 and whose current time specification is t_i the number of available vectors (*number of vectors*, nov) for an interval $t = (t_m, t_n)$; $start(t_1) \leq t_m \leq t_n \leq end(t_i)$ w. r. t. i_l is denoted $nov(t, i_l)$.

$$nov(t, i_l) \ := \ |\{vector | vector \ within \ t \ w. \ r. \ t. \ i_l\}| \tag{1.13}$$

A metric named *concentration of an interval* t is defined as the relation

of the interval length to the number of available vectors $nov(t, i_l)$.

$$ctr(t, i_l) := \frac{nov(t, i_l)}{|t|} \tag{1.14}$$

In consequence of (1.14) the *concentration of an individual* i_l is:

$$ctr(i_l) = ctr(\ (start(t_1), end(t_i))\ ,\ i_l\ ) \tag{1.15}$$

The concentration of i_l is equal or almost equal to the reciprocal of the frequency $freq(i_l)$. Per definition the concentration of a uniform individual i_l with time stamps $\{t_1, \ldots, t_i\}$ and $t = (start(t1), end(t_i))$ is:

$$\begin{aligned} &i_l \ with\ time\ stamps\ \{t_1, \ldots, t_i\}\ is\ uniform \\ &\Rightarrow t = (start(t1), end(t_i)) :\ ctr(i_l) = \frac{ceil(\frac{|t|}{freq(i_l)})}{|t|}(\simeq \frac{1}{freq(i_l)}) \end{aligned} \tag{1.16}$$

The (interval) concentration provides information whether i_l is more or less 'sparse'. This would be useful if we have got domain knowledge which includes heuristics for the applicability of a certain learning algorithm on 'sparse' data. nov, ctr, and $freq$ could be included in the meta data depending on the domain kowledge identified in deliverable D5 of work package 5.

Chapter 2

Input Formats

A set of frequently used representations for the input of time related learning was introduced by Morik [11].

For the further use of these input formats in the succeeding chapters, the input languages are described in the following subsections. Furthermore, some variations of the languages and meta data for each language are added here.

2.1 Multivariate Time Series

Multivariate time series are based on one single process, they describe one individual, and they have k numerical attributes $A_1, \ldots, A_k$. The individual identifier is given implicitly (e.g. by the table name of the table containing the data set).

The time specification consists of points in time which are given by t_n and which are uniformly increasing. In case of time omitted multivariate time series the time specification is given implicitly with the same order. The type of the remaining attributes is numerical.

$$
\begin{array}{rll}
L_{E_1} \textit{ multivariate} & & \\
\textit{time series (time omitted)} & & \\
& i_l: & < a_{1_1}, a_{1_2}, \ldots, a_{1_k} >, \\
& & < a_{2_1}, a_{2_2}, \ldots, a_{2_k} >, \\
& & \ldots, \\
& & < a_{i_1}, a_{i_2}, \ldots, a_{i_k} >
\end{array}
\tag{2.1}
$$

L_{E_1} *multivariate time series*

$$\begin{aligned} i_l : \quad & < t_1, a_{1_1}, a_{1_2}, \ldots, a_{1_k} >, \\ & < t_2, a_{2_1}, a_{2_2}, \ldots, a_{2_k} >, \\ & \ldots, \\ & < t_i, a_{i_1}, a_{i_2}, \ldots, a_{i_k} > \end{aligned} \tag{2.2}$$

2.2 Univariate Time Series

Univariate time series are of the same representation as multivariate time series with the condition of having only one numerical attribute ($k = 1$) based on one single process.

$L_{E_{1'}}$ *univariate time series (time omitted)*

$$\begin{aligned} i_l : \quad & a_1, \\ & a_2, \\ & \ldots, \\ & a_i \end{aligned} \tag{2.3}$$

$L_{E_{1'}}$ *univariate time series*

$$\begin{aligned} i_l : \quad & < t_1, a_1 >, \\ & < t_2, a_2 >, \\ & \ldots, \\ & < t_i, a_i > \end{aligned} \tag{2.4}$$

2.3 Nominal valued Time Series

Univariate and multivariate time series are restricted to numerical attributes. By loosening this restriction nominal valued time series are defined as

L_{E_2} *nominal valued time series*

$$\begin{aligned} i_2 := \quad & \textit{time series } L_{E_1} \textit{ or } L_{E_{1'}} \\ & \textit{having attributes of} \\ & \textit{nominal (or numerical) value.} \end{aligned} \tag{2.5}$$

Each vector of nominal valued attributes can be considered events which, e.g. can be used as input for best region rules from ZHANG ([11], [22]).

2.4 Sequence Vectors

A large set of sequence vectors with nominal or numerical attribute values is the input to finding frequent sequences. The scheme of the vectors is similar to univariate time series, but the example set always consists of a large number of individuals that are generic and described by the attributes.

The time span is fixed to the given number of fields in the vector $(T_1, \ldots, T_i)$ which can contain points in time or time intervals. The scheme I:[...] is instantiated by all individuals about which data are stored in the database. The time points can vary from instance to instance, but the ordering is fixed [11].

$$L_{E_3}\ \textit{sequence vectors}$$

$$\begin{aligned} I: \quad & < T_1, A_1 >, \\ & < T_2, A_2 >, \\ & \ldots, \\ & < T_i, A_i > \end{aligned} \tag{2.6}$$

An example of sequence vectors is given in 4.5: a univariate time series is the source for sequence vectors computed with the sliding windows III operator (see 3.3).

2.5 Database Table

Each database table (= one individual) may describe several other individuals.

The chronological order of the rows is not determined, that means the table might be unsorted (see (1.4)).

An input is given by a set of individuals (database tables) with each individual (instantiation of I) containing another set of (sub-)individuals. Each of these (sub-)individuals (e.g. specified by individual identifier A_1) is a nominal valued, multivariate time series having the same set of time attributes $\{t_1, \ldots, t_i\}$ as all the others. The time specifications are either points in time or time intervals and their order is not predefined. In addition condition (1.11) holds.

For each (sub-)individual multiple rows for one time point as well as no row for one of the time points can be available.

$$
\begin{aligned}
&L_{E_{DB1}}\ database\ tables \\
&\qquad I: \quad < T_1, A_1, A_2, \ldots, A_k >, \\
&\qquad\qquad < T_2, A_1, A_2, \ldots, A_k >, \\
&\qquad\qquad \ldots, \\
&\qquad\qquad < T_i, A_1, A_2, \ldots, A_k >
\end{aligned} \tag{2.7}
$$

2.6 Tuples for Time Points

Tuples for time points are based on multiple individuals and the chronologically order of the rows is not determined, that means the table might be unsorted (see (1.4)). The time specifications T are not predefined: both representations, points in time and time intervals, could be applied to different data sets.

Given a set of individuals the database table stores a nominal valued, multivariate time series for each of them. All these time series have the same number of attributes (see (1.11)) and they are put into one single table, which is an instantiation of $L_{E_{DB2}}$.

$$
\begin{aligned}
&L_{E_{DB2}}\ tuples\ for\ time\ points \\
&\qquad I: \quad < T, A_1, A_2, \ldots, A_k >,
\end{aligned} \tag{2.8}
$$

Contrary to the other input schemes an individual (of an input $l_{E_{DB2}}$) can have several rows. That means the measurements of one individual are no longer stored in one row (as was the case for L_{E_3}) but in several rows with each row containing the measurements for just one point in time.

The number of vectors is not necessarily the same for all individuals.

This format provides access to all vectors for one specific time point (or time interval) belonging to one or more individuals. An operator for extracting vectors for one individual will be introduced in 3.1.

It might be of interest to provide an input of format $L_{E_{DB2}}$ which is sorted on the individuals and on the time specifications for each individual (see 4.4):

$$
L_{E_{DB2'}} := \quad L_{E_{DB2}} \wedge I\ and\ T\ are\ in\ ascending\ order \tag{2.9}
$$

2.7 Facts

All of the representations introduced so far can be mapped to facts. That means each individual with k attributes of numerical or nominal value can

be mapped to a corresponding fact with k terms. T is the time point or time interval. The order of the time specifications is unimportant in general. In experiment 1 (5.2) the order of the facts used as input for the learning method STSP is generic (time points, $d = 1$, $t_1 = 1$).

$$\begin{aligned} &L_{E_4}\ facts \\ &\qquad I: \quad P(T, A_1, \ldots, A_k) \end{aligned} \tag{2.10}$$

Subject to the selected mapping the facts can have a different number of terms than attributes are available, e.g. the predicate name P can be used to code one of the attributes. That way an event name, a (sequence) classification or any other attribute value can be stored by the predicate name. The coding of an attribute's value as the predicate name is used in section 4.7.

Of course the predicate name could store the individual identifier instead. Especially when several rows are given for each individual this coding is valuable.

Chapter 3

Time related Preprocessing Operators

The time related preprocessing operators introduced in this section are table-to-table converters which operate within the data warehouse of the KDD-SE. They will be implemented in Oracle 8i SQL and SQL/Plus. Each of the operators needs one or more specific input tables and produces one or several output tables. An example operator producing several output tables is given in section 3.5. A manual operator expecting more than one input table is the multi-relational attribute construction operator of deliverable D1.

Therefore it is quite important that the meta data take into account how to define the *input tables* and *output tables*. In some cases the number of output tables does not depend on the operator itself but on the content of the input tables. That is why the meta data should support the dynamical assignment of an unknown number of output tables as input for the succeeding preprocessing step. Alternatively all members of the set of output tables are applied to the succeeding step one by one, as it is the case for the multiple learning operator (see 3.5).

3.1 Single Process Extraction

If information from multiple individuals (each individual representing a process) are given, the vectors corresponding to one of the processes can be extracted to derive the information for this single process.

$L_{E_{sP}}$: Let $T_{mp} := \{t_{mp_1}, \ldots, t_{mp_j}\}$ be the set of time specification for the input from which tuples for one single process *sp* should be extracted. Time specifications for selected vectors of *sp* are denoted as $T_{sp} := \{t_1, \ldots, t_i\} \subseteq T_{mp}$. At least one attribute is the same (or holds some other specific condition) for all tuples from one single process. This attribute value can be denoted as the *process identifier*:

$$
\begin{aligned}
&\exists\{attrID_1, \ldots, attrID_i\}: \\
&(\forall j \in \{1, \ldots, i\}: attrID_j \in \{1, \ldots, k_j\}) \\
&\wedge\ \forall t_m, t_n \in T_{sp}: a_{t_{m_{attrID_m}}} = a_{t_{n_{attrID_n}}} \\
&\Longrightarrow\ \textit{process identifier} := a_{t_{m_{attrID_m}}}
\end{aligned} \tag{3.1}
$$

Since the meta data for the input include all individual identifiers we just need to know the $individual\ identifier \in individual\ identifiers$ and its value (= process identifier) to extract the data for. The time specification is always expected to be the first (two) attribute(s).

The output table's first column (resp. the first two columns in case of time intervals) is the time specification followed by the process identifier. E.g., if the input is an instantiated database table ($L_{E_{DB1}}$, (2.7)) we will get an output of the following form:

$$
\begin{aligned}
&L_{H_{sP}} \\
&\quad i_l: \quad < t_1, a_{t_{1_{attrID_1}}}, a_{t_{1_1}}, \ldots, a_{t_{1_{attrID_1-1}}}, a_{t_{1_{attrID_1+1}}}, \ldots, a_{t_{1_k}} >, \\
&\qquad\quad < t_2, a_{t_{1_{attrID_2}}}, a_{t_{2_1}}, \ldots, a_{t_{2_{attrID_2-1}}}, a_{t_{2_{attrID_2+1}}}, \ldots, a_{t_{2_k}} >, \\
&\qquad\quad \ldots \\
&\qquad\quad < t_i, a_{t_{1_{attrID_i}}}, a_{t_{i_1}}, \ldots, a_{t_{i_{attrID_i-1}}}, a_{t_{i_{attrID_i+1}}}, \ldots, a_{t_{i_k}} >,
\end{aligned} \tag{3.2}
$$

With the notation used in deliverable D1 the implementation of this operator could be done as:

```
Single Process Extraction
  Select time specification, individual identifier,
         remaining attributes
  From source
  where individual identifier = given process/individual

view -> view (row reduction)
```

3.2 Multivariate to Univariate Transformation

For the simplest form of multivariate to univariate transformation (I) all we need to know for any given input $L_{E_{m2u}}$: is the attribute which names the value of the univariate time series to produce. The output will then consist of the original time specification and a second attribute containing the unvariate time series' values.

If all possible univariate time series should be created, we will iterate with all attributes of the multivariate time series.

The output $L_{H_{m2uI}}$ is described with:

```
Multivariate to Univariate Transformation I
  Select time attributes, attribute
  From Source

view -> view (column reduction)
```

A second feasible transformation (II) reads the values from a multivariate time series from the left to the right and from the top to the bottom and creates one univariate time series.

An example data set contained energy consumption data which was measured every 15 minutes for several days. The first attribute of the table is the day of the measurement, the remaining columns denote the time of the day when the value was measured. They are named "0:00 a.m.", "0:15 a.m.", ..., "11:45 p.m.".

The time specifications for the univariate time series will start with value 1 and increase one by one for each value of the series. The original time attributes were given by the values of the column named *day* and by the names of the remaining columns. They are to be integrated in the output table, too.

If the time specification is not needed feature selection (see deliverable D1) can be used to delete it.

$L_{H_{m2uII}}$:

```
Multivariate to Univariate Transformation II
  For all combinations of time specifications and attributes do
    Select time specification, attribute name, attribute value
    From source

view -> view
```

The output table will have equal or more rows than the input table.

A third multivariate to univariate transformation (III) is needed to compute frequent sequences (see 4.5). The input rows of a multivariate time series are considered events. Each event is defined by the values from a row with k attributes (the time attributes are not considered for defining event types). Thus each new combination of attribute values denotes a new event type. The output is a nominal valued univariate time series of format L_{E_2} based on L'_{E_1}. The single attribute A_1 of this output denotes the event type that occured at that point in time. Therefore the event types are mapped to integers starting with the value 1.

$L_{H_{m2uIII}}$:

```
Multivariate to Univariate Transformation III
  Select time specification,
```

```
        map attribute values (attribute 1, ..., attribute k)
  From Source

view -> view (feature construction)
```

A second table is created. It gives information about the mapping of event types to integers. The meta data scheme should include such mappings.

```
Multivariate to Univariate Transformation III
  Select attribute 1, ..., attribute k,
        map attribute values (attribute 1, attribute k)
  From Source

view -> view (feature construction)
```

Alternatively the event types could be added as a constructed feature to the original view. Then feature selection has to be applied to derive the appropriate table for the frequent sequences approach.

3.3 Sliding Windows

The input table to the sliding windows operator consists of a time specification and one more attribute. In case of multiple attributes multivariate to univariate transformation should be applied first to provide the correct input table.

$$L_{E_{slW}} := \quad L_{E'_1} \tag{3.3}$$

In particular the multivariate to univariate transformation II produces appropriate input for the computation of sliding windows of a multivariate time series and thus windows which include values of two rows from the original multivariate time series.

For a *window size* w the observations within this window are combined to one new vector (sliding window I)[1]:

$$\begin{array}{ll} L_{H_{slWI}} & \\ \quad i_l: & t_{1_1}, a_{1_1}, \ldots, t_{1_k}, a_{1_k}, \ \ldots \ , t_{i_1}, a_{i_1}, \ldots, t_{i_k}, a_{i_k} \end{array} \tag{3.4}$$

The window is moved by *window movement* v steps and the new vector is computed. This will be repeated until the end of the input is reached.

[1]Since this operation will consist of direct cursor implementation on database tables rather than of SQL-Statements the output is not specified with SQL-like statements as in the preceeding sections.

Another sliding windows variation (II) compresses the time information to a new time interval:

$$L_{H_{slWII}} \quad i_l: \quad t = (start(t_{1_1}), end(t_{i_k})), a_{1_1}, \ldots, a_{1_k}; \ \ldots \ ; a_{i_1}, \ldots, a_{i_k} \tag{3.5}$$

Each vector contains only one time specification.

If the windows movement is less than the windows size then the sliding is called *overlapping*, and *non-overlapping* otherwise.

The third sliding window operator (III) is needed for computing frequent epsiodes (see 4.5). This sliding window operator ensures attribute values at the beginning and at the end of the univariate time series to be put in as many windows as the values in the middle of the series. Therefore the first and the last window, and in some cases some more windows, extend outside the sequence!

$L_{H_{slWIII}}$: The output format is the same as the one of sliding window I, but it will consist of more output vectors.

All of the sliding window variations belong to:

```
Sliding Window
 View -> View
```

with the output table having less or equal rows than the input table.

3.4 Summarizing

$L_{E_{sum}}$: Attribute values $a_j, \ldots, a_{j+w-1}$ within a window of *past observations* w are summarized by a *function* $f(a_1, \ldots, a_w)$ (e.g. average, variance). The original time series is replaced by the discretized one.

$L_{H_{sum}}$: E.g. a univariate time series of representation (2.4) is replaced by:

$$\begin{aligned} &L_{H_{sum}} \\ &i_l: \quad t = (start(t_1), end(t_{w+1})), f(a_1, \ldots, a_w), \\ &\qquad \ldots \\ &\qquad t = (start(t_{i-w+1}, end(t_i)), f(a_{i-w+1}, \ldots, a_i) \end{aligned} \tag{3.6}$$

Each row of the input table contains one window of past observations. If this format were not given already it has to be created with sliding windows.

E.g. for a univariate time series sliding windows with window size w and movement v ($v \in \{1, \ldots, w-1\}$ for overlapping summarizing, $v \geq w$ for non-overlapping summarizing) can be used to generate output rows with each row containing the elements of the past observations w.

Beneath the input table we need a second meta date: the function f. MiningMart will at least provide the moving average functions *simple moving average* (SMA), *weighted moving average* (WMA), and *exponential moving average* (MVA) [14]. Basically all of this moving functions behave the same: they do some computation on observations which are given with the sliding window operator. The window size w used for computing the sliding windows has great influence on the moving function. E.g. a windows size $w = 6$ is used for a lag-six SMA (SMA6). This parameter is implicitly given by the number of attributes of the input table passed to the summarizing operator.

With using weights WMA decreases the drawback of each value having the same contribution as the other values to the average of the observation window. EMA uses a tail weight and a head weight for weighing the previous average value and the current value. For both WMA and EMA the weights sum up to 1.

The weights have to be passed as meta date to the summarizing operator. In case of EMA the number of weights is exactly 2, in case of WMA the number of weights depends on the window size of each observation.

3.5 Multiple Learning

Instead of handling diverse individuals in one learning run a learning can be started for each individual. In the drug store example for each branch office and each item a separate signal to symbol processing was started (see 5.2).

$L_{E_{multL}}$: The splitting of the input table is controlled by one of the individual identifiers of the table which is denoted *disjunctive attribute* in this context. This operator is almost equal to the single process extraction operator: here we will produce a set of tables (not only one table), one for each value of the individual identifier.

$L_{H_{multL}}$:

```
Multiple Learning
  for all individuals
    select individual, time specification, remaining attributes
    where individual = name of the current individual
    from source;

(view -> views)
```

The output tables are specified by a separate table containing the names of all of the output tables.

3.6 Aggregation

$L_{E_{aggr}}$: Aggregation sums up several inputs vector by vector. For the given inputs l_E condition (1.11) should hold and the set of time specifications should be the same for all inputs. If a time specification does not occur in all of the input tables then only the available data for that time specification will be summed up.

$L_{H_{aggrI}}$: An input vector is read from each of the inputs, all vectors having the same time specification. One output vector is created by summing up all values for each of the attributes.

```
Aggregation I

views -> view
```

$L_{H_{aggrII}}$: A second aggregation operator (II) takes into account that not only the time specifications should be the same for creating a new vector but also any other combination of attributes for a given input table.

E.g. considering the drug store chain data it is of interest to sum up all sales of one article at one point in time in all shops or to sum up all sales of one article in one shop at any time. There are some more combinations which might be of interest. The key question is: which attributes stay the same for one new output vector (*join attributes*)? In the drug store example (see 5.2) these join attributes are the article number and the point in time respectively the article number and the shop number.

If the time specification does not belong to the join attributes then the output will include the start of the earliest time specification and the end of the latest time specification at the beginning of the table.

```
Aggregaton II

view(s) -> view
```

3.7 Time Segmentation

An important task of preprocessing within the kdd-process is the segmentation of a given input according to certain time information. In the context of time related learning two important segmentation operators are given with:

- the extraction of time specifications which fulfill a certain segmentation-pattern, e.g. any time specification t_i should be greater than t_1 and less than t_2
- the segmentation to reduce the input data set to those records whose time specification is a specific weekday, e.g. 'Tuesday'. Especially if

the domain knowledge tells about the usefulness of learning about a single day rather than over the whole time series this operator will be of great benefit. Consider the sale of newspapers. It could depend on several properties, e.g. loose inserts on Wednesdays, articles with specific subjects on Tuesdays and Thursdays, extended sport reporting on Mondays, that motivate special interest groups to buy newspapers on specific weekdays only.

Both of these segmentation operators are supported by the segmentation operator introduced in deliverable D1. But the second operator requires the time feature construction of the next section. This feature can be processed with the segmentation operator of D1 afterwards.

3.8 Time Feature Construction

When learning about time the transformation of a given time specification into another representation can be useful. E.g. for a given date, the weekday or the weekday index (e.g. 1=Monday, ..., 7=Sunday) could be of great interest in order to extract a corresponding time segment (see 3.7).

Therefore a special time feature construction operator is introduced. It operates on any input language ($L_{E_{TFC}}$) whose time information is represented by time points that are given as dates.

The output contains the whole input concatenated with the weekday

$$L_{H_{TFC1}} := \quad L_{E_{TFC}}, Weekday \tag{3.7}$$

```
Time feature construction 1
  select time specification, remaining attributes,
         weekday(time specification)
  from source;

(view -> view)
```

or concatenated with the weekday index.

$$L_{H_{TFC2}} := \quad L_{E_{TFC}}, WeekdayIndex \tag{3.8}$$

```
Time feature construction 2
  select time specification, remaining attributes,
         weekdayIndex(time specification)
  from source;

(view -> view)
```

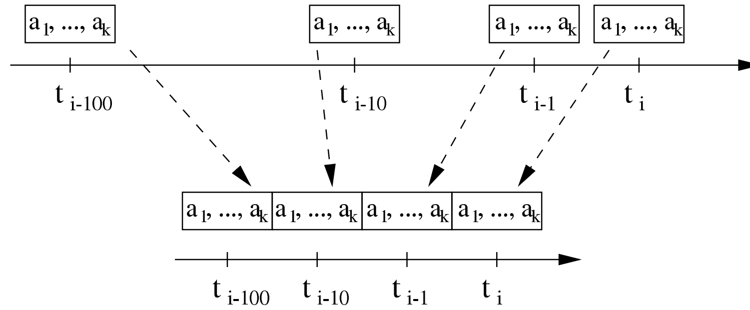

Figure 3.1: Concatenating data from distributed time points

3.9 Implicit Use of Time Information

Time dependent data can be transformed into a form that can be processed by methods that are incapable of explicitly handling time. In general concatenating data from several past observations produces appropriate input vectors for approaches that do not explicitly handle time. The sliding window operators and the summarizing operators can be applied for that reason (see 3.3, 3.4). Usually distributed time specifications are taken into consideration rather than contiguous time specifications as produced by the sliding window operators. An example is given in Figure 3.1.

Consider an input of format L_{E_1}. E.g. an operator transforms this input to several 'windows' of format $L_{H_{iT}}$ (implicit time). Basically the operator output is of the same format as the input (L_{E_1}). $L_{H_{iT}}$ specifies how to compute the output for a given input:

$$
\begin{aligned}
&L_{H_{iT}} \\
&I := \quad T_I, A_{I_1}, \ldots, A_{I_k}; \\
&\qquad\quad T_{I-1}, A_{I-1_1}, \ldots, A_{I-1_k}; \\
&\qquad\quad T_{I-10}, A_{I-10_1}, \ldots, A_{I-10_k}; \\
&\qquad\quad T_{I-100}, A_{I-100_1}, \ldots, A_{I-100_k}; \\
&\qquad\quad \ldots \\
&\qquad\quad T_{I-10^n}, A_{I-10^n{}_1}, \ldots, A_{I-10^n{}_k}
\end{aligned}
\tag{3.9}
$$

```
Implicit time
  For each set of (10^n)+1 contiguous individuals
    select *
    from source
    where the time specification index
          is an element of {I, I-1, ..., I-(10^n)}
          and I is the index of the youngest
```

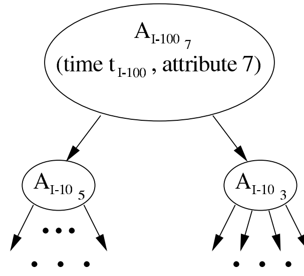

Figure 3.2: C4.5 decision tree for L_{H_iT}

```
time specification of the current set;
```

```
(view -> view)
```

I is instantiated with the index i of the time specifications of L_{E_1} (except the last 10^n time specifications). If the data set contained enough historical data then it is likely to set n to at least 2 or 3.

For instance, L_{H_iT} can be used as input to C4.5 from *Quinlan* [15]. Without loss of generalization consider A_{I-1} the class identifier. Despite the fact that C4.5 does not handle time in an explicit manner the resulting decision tree will yield time related information.

In Figure 3.2 the root decision node refers to time specification T_{I-100}, the decision nodes on the next level refer to T_{I-10}. Irrespective of the set of mutual exclusive outcomes for each decision node it can easily be seen that the main decision depends on the time specification that is 100 time points in the past.

Chapter 4

Learning Methods

Important methods which learn on time related data will be introduced in this section. Contrary to the operators explained in section 3 the following methods are usually supposed to access flat files directly. Thus table-to-file operators are required here to export data from the data warehouse to those learning methods that are not an integrated part of the KDD-SE.

We will give a quite detailed example of the input file format for signal to symbol processing (see 4.7 and 5.2).

4.1 Prediction Task

Excellent performances were obtained in time series prediction applications using the support vector machine (SVM) [18] [16]. Given a sequence of elements until timepoint t_i the task is to predict the element that will occur at time point T_{i+n}. n is called the horizon.

The input consists of the parametrizing of the SVM and the training set:

$$\begin{array}{ll} L_{E_{SVM}} & \\ \quad I & 1 :< A_1 > 2 :< A_2 > \ldots n :< A_n > \end{array} \tag{4.1}$$

I is instantiated with the target attributes of each training case (that is the element to predict at time point T_{i+n}). This target attribute will be predicted for new cases with the SVM after learning on the training set.

$L_{H_{SVM}}$: The SVM outputs a file including the whole data of the training set and additionally learned parameters, which can be used for new inputs to predict a_{i+n}. For a given implementation a mapping of the result to a database table has to be provided in order to use the output for further predictions.

4.2 Characterization

Time series have an implicit pattern even if this pattern is not repetitive. The identification and the description of this pattern is the main goal of the characterization preprocessing step which can be done in several ways. The characterization of certain parts of the series can be expressed by approaches like level change detection (see 4.6) whereas the characterization of the whole series - which is of concern in this section - can be done by several statistical approaches: detecting different kind of trends (linear trends, superlinear trends, e.g. exponential ones, sublinear trends, e.g. logarithmic, logistic trends, see [19]), detecting seasonal increasing or decreasing peaks, or detecting cyclic orders of elements.

The cyclic order can be described by a function, e.g. phase shifted sine or cosine functions or waveforms defined by some other functions (e.g. random walk waveform), as used for *Fourier analysis* or *Spectral analysis* [14]. *Classical decomposition* decomposites time series into cycles and three more elements: trend, seasonality, and noise. Irrespective of the characterization chosen the characterization relies basically on its parametrization and on a specific input format of the implemeted approach derived from a univariate time series.

Thus the input is described by $L_{E_{Ch}} := L_{E_1{}'}$ and the output is:

$$\begin{array}{ll} L_{H_{Ch}} & \\ \quad i_l : & function\ f_1, \ldots, function\ f_m \end{array} \tag{4.2}$$

The order of elements in terms of frequent patterns but not necessarily cyclic patterns is discussed in section 4.5. Frequent patterns are not the same as the cyclic patterns discussed here.

If $L_{E_{Ch}}$ is not given already then it can be computed by another operator, e.g. the multivariate to univariate transformation could transform a multivariate time series (L_{E_1}) to $L_{E_{Ch}}$.

4.3 Time Regions

The *Best Region Rules* approach from Zhang discovers temporal structures in event sequence data in a large time horizon [22].

The input language for the time region approach is given with

$$L_{E_{Zhang}} := \quad L_{E_2} \tag{4.3}$$

with generic points in time (see Figure 4.1, [22]). Consider the input sequence of events normalized: each attribute a_n specifies whether the event associated with this attribute occurs or does not occurs at a given time

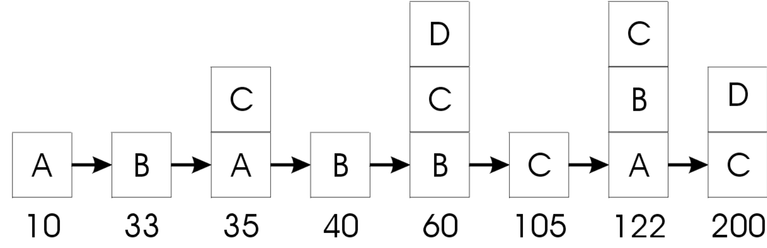

Figure 4.1: Episode detection with Best Region Rules

specification. Additional meta data have to be passed to the algorithm: a minimum score and the weights $w_1, \ldots, w_6$ used for computing the score of a rule as weighted sum of the prediction accuracy ($AccP$), the recall accuracy ($AccR$), the prediction bonus ($BnsP$), the recall bonus ($BnsR$), the range (Rng), and the coverage (Cov). These meta data parametrize the approach.

The output is a set of rules and their scores:

$$
\begin{aligned}
&L_{H_{Zhang}} \\
&\quad I: \qquad A_v \rightarrow_{[T_b, T_e]} A_z, Score
\end{aligned}
\tag{4.4}
$$

with $v, z \in \{1, \ldots, k\}$ and $b, e \in \{1, \ldots, i\}$. The rule states that A_z will occur if event A_v occured in the last $[b, e]$ time steps, which can refer to a very large time horizon[1].

The approach generates $k^2 - k$ rules with k being the number of event types, e.g. a rule looks like $C \Rightarrow A$. For each of the rules a *lag set* S_{lag} is computed specifying the distances of the two events of the rule occuring in the event sequence with respect to a minimal distance $min-lag$ and a maximum distance $max-lag$. W.r.t. the example in Figure 4.1 and to the rule $C \Rightarrow A$ the lag set S_{lag} is $\{0, 17, 62, 87\}$. Afterwards *minimal temporal regions* for each lag set are computed specifying all possible distances between the two events C and A, e.g. $\{[0, 0], [0, 17][0, 62], [0, 87], [17, 17], [17, 62], [17, 87], [62, 62], [62, 87], [87, 87]\}$. For each of the minimal temporal regions a rule is generated, e.g. $C \Rightarrow_{[0,0]} A$, and each rule is associated with a score that is computed by the score function explained above. Finally the rule with the best score (above a given threshold min_{score}) is selected.

Zhang introduced an extension to the Best Region Rules algorithm which is called *Multiple Rules* [23] . Best Region Rules selects only (at most) one rule for each pair of events. To avoid this drawback Multiple Rules segments the temporal region space of a rule into several segments and selects the best rule for each of these segments. In addition it selects

[1]The table representation of such rules is always generated by reading rules from 'left to right'. In case of $L_{H_{Zhang}}$ the table attributes are: 1. A_v, 2. T_b, 3. T_e, 4. A_z, ...

one more rule for each segment, except the last one, across the boundary of the segment. It solves the problem of how to determine the segments (uniform selection, clustering) and the number of segments depending on the application problem.

Because Multiple Rules increases computational costs by far in case long event sequences are given, ZHANG introduces heuristic pruning for rule selection: the lag set is pruned on the basis of a measure δ denoting the granularity of the rule selection process. δ determines the minimum difference between a lag value's next smaller value and its next larger value to let the lag value be mandatory. If the difference is less than δ a heuristic is used to decide whether the lag value will be skipped or not. More details are given in [23].

Multiple Rules does not affect the format of $L_{E_{Zhang}}$ and $L_{H_{Zhang}}$.

4.4 Association Rules and Sequential Patterns

The original version of *Apriori* already operated on time related data (even if it did not take the order into account that is given with the transaction identifications of basket data). For that reason the original version of Apriori for discovering association rules is introduced first. Afterwards the modified versions for mining for frequent episodes - also denoted as sequential patterns - is explained. The difference to the frequent sequences (see 4.5) will be explained, too.

The *Apriori* algorithm (and its extensions *AprioriTid* and *AprioriHybrid*) considers the problem of discovering association rules between items in a database of sales items (basket data) [1] [2] [21]. A basket or itemset consists of a non-empty set of items each of which is represented by a binary variable indicating whether the item was bought or not without looking at the quantities.

The input to the algorithm is given with:

$$L_{E_{Apriori}} := \quad L_{E_{DB2'}} \tag{4.5}$$

I is instantiated only once by the identifier of the basket data set. A_n is a binary value representing whether item n was bought at time point T by the customer or not. Since apriori does not take into account the customer identifications, I could be omitted if the customer identifications of the transactions were not given in the data set.

The output is a set of rules of the form

$$L_{H_{Apriori}} \qquad I: \quad A_{prem_1}, \ldots, A_{prem_k} \rightarrow_{[Conf,Supp]} A_{concl_1}, \ldots, A_{concl_k} \tag{4.6}$$

with $\forall i \in \{1, \ldots, k\} : A_{prem_i} = 1 \rightarrow A_{concl_i} = 0$ (items that occur as bought items in the premise will not occur as bought items in the conclusion). The rule states: if items of the premise with binary value 1 are bought then all items in the conclusion with binary value 1 will be bought in the same transaction, too (with confidence $Conf$ and support $Supp$). That means $Conf\%$ of the transactions that include the bought items of the premise also include the bought items of the conclusion and $Supp\%$ of the transactions contain all bought items of the rule.

The problem of mining for frequent episodes in sequence data was first introduced by AGRAWAL and SRIKANT in 1995. They adopted their association rule algorithm for unordered data to mine for frequent episodes (or frequent/ sequent patterns) over time stamped basket data and introduced the algorithms *AprioriAll*, *AprioriSome*, and *DynamicSum* [3].

A sequence is an ordered list of itemsets. One sequence is given for each of the customers. The maximal sequence among all sequences with a minimum user-specified support is denoted *sequential pattern*. Thus a sequential pattern consists of elements with the elements being single items or a set of items. The problem is to find this sequential pattern for a given dataset.

These algorithms mine for *all* patterns or episodes in contrast to the approach from MANNILA ET AL. which requires the specification of a certain class of episodes to look for (see 4.5), and in contrast to the approach from GURALNIK ET AL.: they use the same term *sequential pattern* but allow to specify episodes of interest in a more complex way than MANNILA ET AL. do. There a pattern language is introduced to define more powerful combinations of partially ordered event specifications that is not restricted to serial and parallel order but also supports the definition of constraints of higher complexity (using order, selection, and join constraints) [8].

The input of *AprioriAll*, *AprioriSome*, and *DynamicSum* stays the same as before except for a minor modification: here I is instantiated by the customer identifications. The output has changed to:

1. $L_{H_{AprioriMLS}}$, a set of *maximal large sequences* each of which contains one or several *large itemsets* identifications.

$$\begin{array}{ll} L_{H_{AprioriMLS}} & \\ \quad I: & A_1, \ldots, A_k \end{array} \tag{4.7}$$

   I is instantiated by maximal large sequences, that means by 1-sequences, ..., n-sequences. There can be several rows for one instantiation. A_n is the identification of a large itemset.

2. $L_{H_{AprioriLI}}$, a collection of *large itemsets*.

$$L_{H_{AprioriLI}} \qquad I: \quad A_1, \ldots, A_k \tag{4.8}$$

I is instantiated by the large itemsets. A_1, ..., A_k are the items of the large itemset. Condition 1.12 holds for $L_{H_{AprioriLI}}$ and $L_{H_{AprioriMLS}}$.

$L_{H_{AprioriMLS}}$ contains maximal large sequences. The sequential pattern is given with the longest one.

4.5 Frequent Sequences

The approach from MANNILA ET AL. discovers all frequently occuring episodes of a given class with an episode containing several events with a certain order (e.g. parallel episodes containing events occuring at the same point in time or serial episodes containing succeeding events) [10].

The input data is basically a monotonically increasing, multivariate time series. Each row of the time series represents an event having k attributes and occuring at a certain point in time.

In the first step this multivariate time series is transformed to a nominal valued uniform time series using the multivariate to univariate preprocessing operator III.

To produce rules representing dependencies between episodes and subepisodes within a window the derived data of format L_{E_2} (univariate) is preprocessed one more time with the sliding window operator III with a window movement of $v = 1$. Thus we get a set of windows of size w which are of input format

$$L_{E_{freqSeq}} \quad := L_{E_3} \text{ (sequence vectors)} \tag{4.9}$$

The number i of attributes is the same as the window size taken for the sliding window preprocessing step. The window movement v that was used to generate the sequence vectors is important to the algorithm: the calculation of the candidate episodes for the next window (given the frequent episodes of the previous window) is optimized for $v = 1$ [10].

In addition the beginning and the end of the original time series (S, T) have to be passed to the algorithm. S and T are not necessarily the same as $start(t_1)$ and $end(t_i)$ with t_1 being the time specification of the first sequence vector and t_i being the last. Have a look at Figure 4.2 describing the nominal valued time series which was the input to the sliding window operator: the first event E occurs at time point $t_1 = 31$, but the whole event sequence already started at time point $S = 29$ [10].

Instead of generating all possible rules and pruning them afterwards (like Best Region Rules from ZHANG does, see 4.3) this approach requires to

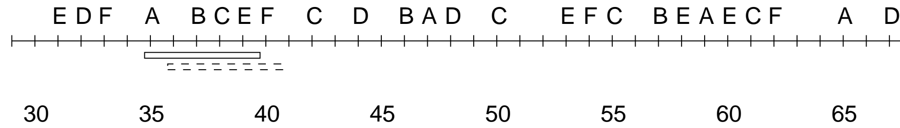

Figure 4.2: Event Sequence

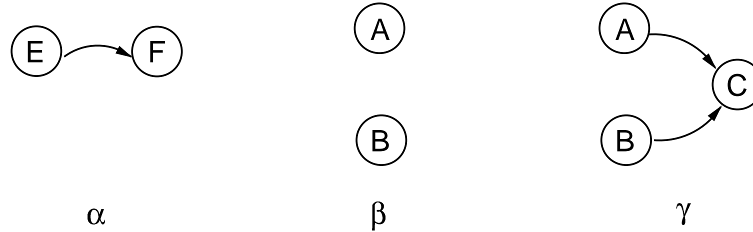

Figure 4.3: Classes of Episodes

specify a class of episodes first, e.g. serial or parallel episodes, to compute valid rules. Figure 4.3 shows a serial episode α, a parallel episode β, and an episode γ which is neither serial nor parallel [10]. Parameters required for this approach are the class of episodes, a frequency threshold, and a confidence threshold.

The output is a set of rules of the form

$$L_{H_{freqSeq}} \quad I: \quad \beta \rightarrow_{[i]} \alpha, confidence\ value\ of\ the\ rule \tag{4.10}$$

with β and α being episodes and β being a subepisode of α. That means all events in β are also part of α with the same order (if any). The detailed formal definitions are given in [10]. The rule states that α occurs within the same window (of size $w = i$) as β does with the confidence of the rule. Please note that rules of this kind do not state that all events belonging to β occur *before* the events of α!

The algorithm explained above is called *WINEPI*. An extension is called *MINEPI* and computes episode rules with two time bounds:

The output is a set of rules of the form

$$L_{H'_{freqSeq}} \quad I: \quad \beta_{[win_1]} \rightarrow \alpha_{[win_2]}, confidence\ value\ of\ the\ rule \tag{4.11}$$

and states that if all events in β occured within win_1 seconds, then α follows within win_2 seconds.

4.6 Level Changes

Any level change algorithm detects time points in a sequence of elements, where the elements are no longer homogenous to some measure. For instance BAUER has implemented a level change detection with the SAS System from SAS Institute Inc. [5].

The input for level change detection is a univariate time series of format

$$L_{E_{levCh}} \quad := L_{E_{1'}} \tag{4.12}$$

The output is of the same format as the input:

$$L_{H_{levCh}} \quad := L_{E_{levCH}} \tag{4.13}$$

but the attribute values are replaced by values defining if the attribute's level has changed or not. Depending on the level change algorithm the value will tell about the magnitude of the level change or it will just indicate the occurence of a level change.

4.7 Signal to Symbol Processing

The signal to symbol processing approach (STSP) from MORIK and WESSEL is a strongly incremental approach that bridges the gap between numerical sensor data and symbolic approaches to both learning and planning [13] [20]. The approach transforms a stream of numerical sensor measurements from a mobile robot's sensor into a sequence of symbolic descriptions for further processing, e.g. for the automatic guiding of a robot [9]. An experiment is described in detail in section 5.2.

STSP works on inputs of format L_{E_4}. For producing such input the table-to-file operator described here needs a univariate time series of format

$$L_{E_{STSP}} \quad := L'_{E_1} \; (univariate\ time\ series) \tag{4.14}$$

with the order of time specification supposed to be uniform.

One of the attributes of the input table providing $L_{E_{STSP}}$ needs to have the same value for all input rows (this one is used as the trace identification of STSP). A third attribute is required where the type has to be numerical (this is interpreted as the numerical data from the robot's sensor).

Please note that $L_{E_{STSP}}$ describes the input format of the preprocessing operator and not the input format of STSP itself. The STSP approach itself expects time points with the first time point's value equal to 1 and the succeeding time points increasing one by one.

The output of the STSP algorithm is of form L_{E_4}, facts that describe

which symbol has been chosen for which time interval. The symbols of the intervals, e.g. decreasing, increasing, stable, and the parameters deciding which symbol name belongs to which kind of measurements is coded in the program source. These parameters have not to be passed to the external tool STSP but it should be stored by the meta data: a set of symbol names with each symbol name having two numerical values defining the interval for the gradient's value. The gradient specifies the relation between the moved distance of the robot and the measured distances.

Thus the resulting table of STSP has five attributes: the symbol name (nominal), the trace identification (numerical), the start and the end of the time interval (both numerical) and the gradient (numerical).

The preprocessing operator transforms this output to the output table

$$L_{H_{STSP}} \quad := L_{E_2} \tag{4.15}$$

The time specifications are time intervals and the remaining attributes are taken from the output of the STSP algorithm explained above.

4.8 Clustering

Given subsequences in a sequence of events clusters of similar sequences are detected. The rule discovery approach from DAS ET AL. finds rules relating patterns in a time series to other patterns in the same or a second series [7]. Therefore the approach forms subsequences by sliding a window of size w through the time series in a first step. The window movement is not restricted to be of value 1. These subsequences should be computed using the sliding window operator (II).

These sliding windows are the input for the clustering approach described here: a time interval specification given by two attributes and $k = \textit{window size } w$ numerical attributes. The time specification is uniform.

$$L_{E_{clust}} \quad := L_{E_{slWIII}} \tag{4.16}$$

After that these subsequences are clustered and the orginal time series is transformed to a discretized one using the clusters (see Figure 4.4, [7]). The clusters $C_1, \ldots, C_k$ are formed by using a suitable measure of time series similarity, e.g. the simplest one is the euclidean distance in combination with a normalized version of the subsequences which is based on the mean and the standard deviation to force the mean to be 0 and the variance to be 1. The clustering itself can be done, for example, with a greedy-algorithm ($\textit{cluster radius} \leq d \wedge \textit{distance between cluster centers} > d$) or k-means.

In the last step a simple rule finding method is applied to obtain rules from the sequence: $m * k^2$ rules are generated with k being the number of clusters used to discretize the series and m being the number of different

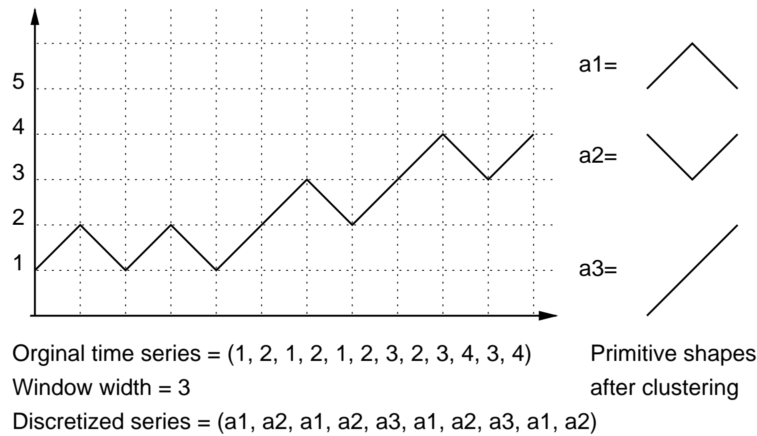

Figure 4.4: Discretized time series using clusters

possibilities for t for rule generation of the form:

$$L_{H_{Clust}} \quad I: \quad A \rightarrow_{[t]} B, confidence\ value\ of\ the\ rule \tag{4.17}$$

The rule states that if A occurs, then B will occur within time t. Rule pruning is done on the basis of frequency (e,g,. 1%) and confidence (e.g. 50%).

The resulting rules are mapped to two tables. The first table holds the clusters: each cluster is defined by a cluster identification (id), and a number of numerical attributes. This number of attributes is defined by the window size w used for forming the subsequences. The second table holds the rules and has three attributes: two cluster identifications representing the left and the right side of the rule and the time specification t.

Guiding the pruning with the confidence value yields the drawback of rules with high significance but low frequency to go undetected. To avoid this some other pruning criterion like the branch-and-bound properties of the J-measure can be used afterwards to prune the search space one more time. E.g. this was used in the ITRULE algorithm of SMYTH and GOODMAN [17].

Chapter 5

Experiments

Performing experiments helped identifying preprocessing operator requirements. Of course operators to transform data from one representation to another are reasonable, e.g. the multivariate to univariate transformation (see 3.2). Additionally other requirements have been discovered. An example is splitting up one input format to multiple output files of another format, e.g. multiple learning (see 3.5). This can be realized in different ways. A second example is generating attribute values in a specific way to substitute time specifications with new ones (as used for STSP). Gained insights were already integrated and introduced in the description of meta data in the preceeding sections. They had great influence on the requirements for the meta data (language).

5.1 Proprietary Meta Data Language

A proprietary meta data language will be introduced in this section. It is used for supporting chains of preprocessing steps and for specifying the experiments of the succeeding sections: the steps of the experiments are described using input descriptions and output descriptions.

This proprietary meta data language (you will identify some meta data elements we have defined before) is used here to give an explicit specification of input files and output files which have been processed for the experiment.

These descriptions will not be used for describing files later on in the project because the project is mainly about preprocessing within the data-warehouse (delivery D6.1 will provide an excerpt of Swiss Life's data warehouse). The participants developing the meta data language agreed, that external tools which will be integrated to the Mining Mart and which work on flat files will have an associated table-to-file converter that accesses a database table and takes care for producing an output file of whatever format the external tool expects.

Meta data for reading the input are given by a row description which

describes the scheme of one input vector. In some cases the input might be read from database tables, in other cases the input is read from raw data files. The latter requires an excact mapping of the raw data to the input languages described in this paper. Therefore *line delimiter*, *input row description*, and *input descriptors* are introduced:

An input file has a meta date

- *Line delimiter.* Per default this end of line string is a string with the single character 'line feed' but the line delimiter may be of arbitrary length.

An *input row description* has an ordered collection containing *input descriptors*:

- Attribute input descriptor identifies an attribute of interest which will probably be processed and put in an output file,
- Unknown string descriptor describes a string of unknown content and length,
- String descriptor describes a certain collection of characters,
- Field delimiter descriptor is a kind of special string descriptor used for separating other descriptors.

Please note that the field delimiter descriptor is not specified once but as often as it appears in one row. That is because an input row needs not necessarily to consist of alternating <attribute, field delimiter> pairs. Thus any raw input can be mapped to the input languages. The only constraint is to not use two successive descriptors of {attribute input descriptor, unknown string descriptor}, because the ending of the first and the beginning of the second descriptor within the input line would not be well-defined.

For describing the output of preprocessing operators we will need an *output description*: an *output row description* which is based on an input row description and a line delimiter.

An output row description is based on attribute input descriptors of an input row description denoted *source row*. It is also possible to name more than one input row descriptions (*source rows*) and thus operate on several input rows to compute one output row. The output row description has a meta date *line delimiter* and it has an ordered collection containing *output descriptors*:

- An attribute output descriptor is based on a *source row*'s attribute input descriptor (*source attribute*). If no condition or computation is specified for the attribute output descriptor then the attribute value specified by the attribute input descriptor will be taken as is.

- An attribute output descriptor has an optional *condition* which decides whether its input row is processed and possibly transformed to an output row. A condition specifies that the source row will be processed only in case the condition holds. E.g. the source attribute's value has to be greater than 13 but less than 21, or the attribute value should be equal to "married". Otherwise the whole input vector will be skipped.

- The *computation* of an attribute input descriptor is good for computing the output value on base of the source attribute's value instead of just passing the value from the input row to the output row.
  E.g. a factor can be specified if the attribute is numerical or appending "This is an additional information!" to the source attribute's value if it is of nominal value.
  A quite more important, specific computation is the defintion of a *mapping* that maps certain source attribute values to new ones.
  The computation also grants access to the values of the other descriptors of the same input row and to the values of previous input rows (*previous rows*). This is very important for operators like summarizing (see 3.4).

- The *enumerator descriptor* generates ascending or descending numbers, one for each output row. It has two properties *starting with* and *step* which define the starting number and the amount to increase for each new output row.

- As input row descriptors do output row descriptors also support *string descriptors* and *field delimiter descriptors.*

An output row descriptions can have a *disjunctive attribute descriptor* which is a reference to one of the attribute input descriptors. If specified a separate instantiation of the output format will be generated for each different value of the attribute that is defined by the attribute input descriptor. E.g., if the disjunctive attribute descriptor is the attribute specifying the article number of a sales figures, a separate output will be created for each different article. This meta date causes the splitting of the input data (see 3.5) and it is more flexible than single process extraction explained in 3.1 which extracts vectors for defined individual identifiers.

With the help of these descriptors and because line delimiters and field delimiters are not restricted to contain only one character quite complex transformation can be done. Examples will be given in the following sections.

The execution of several steps from a preprocessing chain of steps is shown in Figure 5.1.

In the first step the user specifies the input that has to be preprocessed: this is the raw input data plus meta data and an input description giving explicit information about the syntactical format.

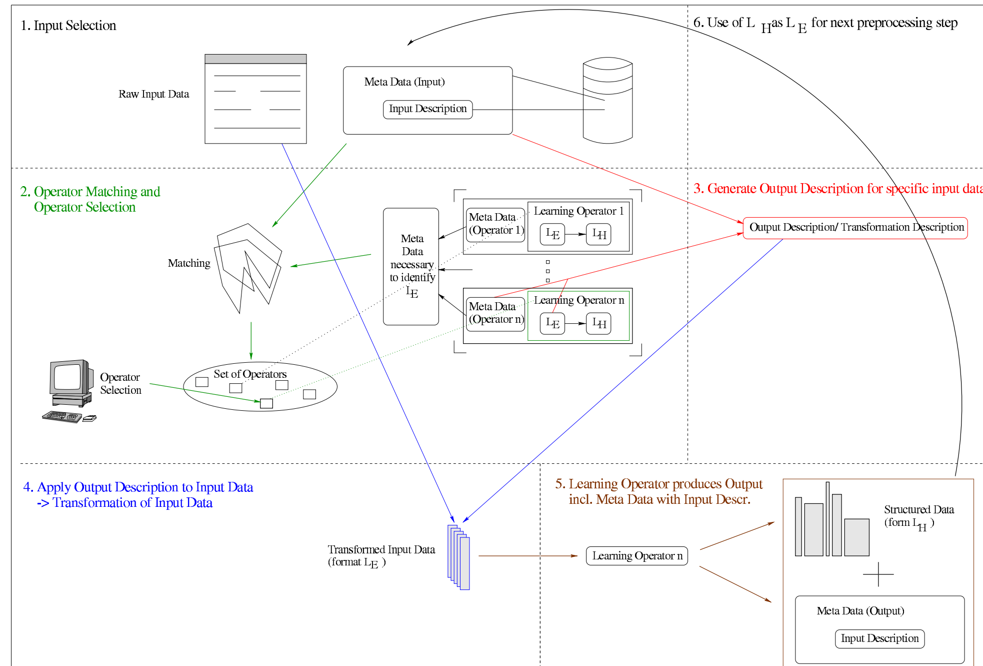

Figure 5.1: Semi-automatic operator selection by meta data matching

In the second step Mining Mart matches the meta data of the input with the meta data of all available learning operators, e.g., if the order of the time specifications of the input is uniform then all operators expecting uniform or generic time specification could be applied. The result of this matching is a set of applicable operators. The HCI displays these operators and the user selects one of them.

In the third step a transformation description (which is an output description based on the input's input description) will be computed automatically. It is possible to compute this transformation description for all defined operators automatically. Thus one implementation to apply the transformation description is implemented instead of implementing each operator separately.

In the fourth step the transformation description produces the appropriate input for the selected learning operator, which is applied in the fifth step. Since the operator knows which output L_H will be produced it provides not only the output but also meta data and an input description that tells about the syntax of the output.

The output including the meta data is the result of one step as part of a preprocessing chain and it can be used as input for the next preprocessing step (this is the sixth step in Figure 5.1).

5.2 Experiment 1: Drug Store Chain Data processed by RDT/DB

The first experiment transforms data from a drug store chain *Drogerie Markt* and processes it with *signal to symbol processing* (see 4.7) and *RDT/DB* afterwards [12]. The whole experiment consists of several steps:

1. definition of meta data (*MD DM*) for drug store chain data (*DM*)
2. definition of meta data (*MD STSP input*) for input of *signal to symbol processing* (*STSP input*)
3. definition of operator (*OP DM-to-STSP*) to transform DM to STSP input
4. definition of meta data (*MD STSP output*) for output of STSP (*STSP output*)
5. definition of meta data (*MD RDT/DB input*) for input of RDT/DB (*RDT/DB input*)
6. definition of operator (*OP STSP-to-RDT/DB*) to transform STSP output to input of RDT/DB (*RDT/DB input*)
7. definition of meta data (*MD RDT/DB output*) for output of RDT/DB
8. performing experiment
   (a) transform DM to STSP input using OP DM-to-STSP
   (b) process STSP input with STSP $\rightarrow$ STSP output
   (c) transform STSP output to RDT/DB input using OP STSP-to-RDT/DB
   (d) process RDT/DB input with RDT/DB $\rightarrow$ RDT/DB output

definition of meta data (*MD DM*) for drug store chain data (*DM*) The drug store chain data comprises sales figures for twenty branch offices. The sales summed up for each of 110 consecutive weeks and for each of around 4000 articles have been recorded separately for each branch office.

Thus 20 files resp. tables of representation $L_{E_{DB1}}$ (2.7) with variation (1.4) are provided[1]. The number of attributes is $k = 3$:

1. A_1 is the time specification with scale unit week. The week and the year are coded to one numeric value whose first 4 digits represent the year and whose last two digits represent the week within the specified year, e.g. '199939' codes the 39th week in 1999,

[1] Other representations of these data are available. For illustration purposes we have chosen this one.

2. A_2 specifies sales summed up for the specified week, and

3. A_3 is the article number to whom the other attribute values belong to.

The following proprietary meta data language is used to define input and output descriptions. The grammar of the language is not explained in detail here, but it should be understandable without further explanations because of its simple syntax and because it corresponds to the meta data language of section 5.1. This language could be a starting point for a more formal specification, e.g. for a XML document type definition (DTD).

```
Metadata

<Name>         branchSalesFigure
<Type>         InputDescription
<Description>  AttributeInDescriptor withName: 'YYYYWW';
               FieldDelimiterDescriptor with: ' ';
               AttributeInDescriptor withName: 'sale';
               FieldDelimiterDescriptor with: ' ';
               AttributeInDescriptor withName: 'article no.'
<AdditionalProperties>
               lineDelimiter: Character lf
```

The default line delimiter is the line feed character. Any string of arbitrary length and content can be used as line delimiter. The last line of metadata does not change the default setting and could be omitted.

An example DM file starts with these data:

```
199548 4 592536
199549 19 592536
199550 17 592536
199551 5 592536
199552 6 592536
199601 12 592536
199602 19 592536
199603 13 592536
199604 12 592536
199605 16 592536
199606 12 592536
199607 9 592536
...
```

definition of meta data (*MD STSP input*) for input of *signal to symbol processing* (*STSP input*) STSP has been developed to deal with numerical sensor data from robot sensors measured from a mobile robot's movements. The approach transforms this stream of numerical data into a sequence of symbolic descriptions for further processing and automatic guiding of the robot. This is done for each of the 24 sensors of the robot. Since the implementation is somehow specialised on the robot domain the input format is specified for general use of the implemented approach in this paper.

The input format is of form L_{E_4} (2.10) where predicates P_1 and P_2 with $k_1 = 13$ and $k_2 = 5$ are used in pairs:

- *messung* instantiates P_1: the attributes A_{1_1} to $A_{1_{13}}$ denote processID (for DM the processID for each branch office file is the article number), point in time, sensor id, measurement, unused, unused, unused, 0, unused, unused, unused, unused, unused
- *robot_position* instantiates P_2: the attribute A_{2_1} always has to be of the same value as attribute A_{1_1}, the attributes A_{2_2} and A_{2_3} always have to be of the same value as A_{1_2}, unused, unused

The point in time (attribute values a_{1_2}, a_{2_2}, and a_{2_3}) always starts with value 1 and increases by one (as specified in section 4.7).

```
Metadata

<Name>        stspInput
<Type>        InputDescription
<Description> StringDescriptor
                with: 'messung(';
              AttributeInDescriptor
                withName: 'Trace Id. Messung';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'Time Messung';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'Sensor Id.';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'Value';
              StringDescriptor
                with: (',nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).',
                       (String with: Character lf),
                       'robot_position(');
              AttributeInDescriptor
                withName: 'Trace Id. Robot Position';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'Time Robot Position';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'Time Robot Position2';
              StringDescriptor
                with: ',0,0).'
```

In addition to the syntactical description some more meta data is needed for the input file (as shown in Figure 5.1, step 1):

```
<individual identifiers>
  whole input (this is one shop with
              the identifier given by the filename);
  Metadata
    named: 'branchSalesFigure'
    attributeInDescriptor: 'article no'
<time attribute>
   Metadata
     named: 'branchSalesFigure'
     attributeInDescriptor: 'YYYYWW';
<time scale>
  <scale unit>
    week
  <point of reference>
    48. week of 1999, start of selling
<order of time stamps>
  uniform
<representation of time>
  time points
<line delimiter>
  Character lf
<attribute properties>
  <attribute>
    Metadata
      named: 'branchSalesFigure'
      attributeInDescriptor: 'YYYYWW'
  <type>
    nominal
  <total amount of values>
    110
  <attribute>
    Metadata
      named: 'branchSalesFigure'
      attributeInDescriptor: 'sale'
  <type>
    numerical
  <attribute>
    Metadata
      named: 'branchSalesFigure'
      attributeInDescriptor: 'article no.'
  <type>
    numerical
  <total amount of values>
    1
<missing values>
  0
```

definition of operator *OP DM-to-STSP* to transform DM to STSP input Considering MD DM and MD STSP input two ways of transforming DM to STSP can be identified: firstly the twenty branch office identifications

could be mapped to the sensors of the robot. Hence we have got about 9 million records we assume problems to occur when trying to feed in all of the DM to STSP. Thus the second approach - where each branch office is mapped to one of the sensors and is passed to STSP separately - seems to be advantageous.

Because the number of different articles for each branch office (file) ranges from 3875 to 4344 it is even more helpful to split up the records for each branch and pass the data for each article and branch to STSP separately. After splitting up each of the branch office files to separate all of its articles we have got 82.721 output files, overall containing 9.099.310 records to process with STSP.

```
Metadata
<Name>        stspInputFromBranchSalesFigure
<Type>        OutputDescription
<Description> StringDescriptor
               with: 'messung(';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                   named: branchSalesFigure
                                   attributeInDescriptor: 'article no.');
              FieldDelimiterDescriptor
                with: ',';
              EnumeratorDescriptor
                startingWith: 1
                step: 1;
              StringDescriptor
                with: ',s1,';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                   named: branchSalesFigure
                                   attributeInDescriptor: 'sale');
              StringDescriptor
                with: (',nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).',
                       (String with: Character lf),
                       'robot_position(');
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                   named: branchSalesFigure
                                   attributeInDescriptor: 'article no.');
              FieldDelimiterDescriptor
                with: ',';
              EnumeratorDescriptor
                startingWith: 1
                step: 1;
              FieldDelimiterDescriptor
                with: ',';
              EnumeratorDescriptor
                startingWith: 1
                step: 1;
              StringDescriptor
                with: ',0,0).'
```

```
<AdditionalProperties>
              disjunctiveAttributeDescriptor:
                (Metadata
                   named: branchSalesFigure
                   attributeInDescriptor: 'article no.')
```

In addition to the output description of the operator we need some more meta data for the automatic guiding of the preprocessing, that means in order to enable the automatic decision whether this operator is applicable to the given input or not:

```
<required meta data>
  <time attribute>
    <identifier>
      'T1'
  <order of time stamps>
    uniform (start: 1, step: 1)
  <attribute>
    <identifier>
      'ID'
    <semantics>
      'This is the TraceId of STSP.'
    <total amount of values>
      1
  <attribute>
    <identifier>
      'Value'
    <semantics>
      'This is the attribute with the measured numerical values for STSP.'
    <type>
      numerical
```

After converting DM with *OP DM-to-STSP* to STSP input an example output file starts with the following data:

```
messung(592536,1,s1,4,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,1,1,0,0).
messung(592536,2,s1,19,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,2,2,0,0).
messung(592536,3,s1,17,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,3,3,0,0).
messung(592536,4,s1,5,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,4,4,0,0).
messung(592536,5,s1,6,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,5,5,0,0).
messung(592536,6,s1,12,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,6,6,0,0).
messung(592536,7,s1,19,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,7,7,0,0).
messung(592536,8,s1,13,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,8,8,0,0).
messung(592536,9,s1,12,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
```

```
robot_position(592536,9,9,0,0).
messung(592536,10,s1,16,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,10,10,0,0).
messung(592536,11,s1,12,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,11,11,0,0).
messung(592536,12,s1,9,nd5,nd6,nd7,0,nd9,nd10,nd11,nd12,nd13).
robot_position(592536,12,12,0,0).
...
```

definition of meta data (*MD STSP output*) for output of STSP (*STSP output*) The output is of form L_{E_4} (2.10) with forms P_1 to P_7 all of which have $k = 6$ attributes:

- the symbolic descriptions named *stable*, *incr_peak*, *decr_peak*, *very_decreasing*, *very_increasing*, *increasing*, and *decreasing* instantiate P_1 to P_7. The names and the number of these symbolic descriptions can be changed and configured by doing minor modifications to the implementation of STSP.
- the six attributes denote the processID (=MD STSP input attribute A_{1_1}), 0, sensor id (=input attribute A_{1_3}), start of time interval, end of time interval, gradient (numerical value for the relation between moved distance and measured distance of the robot's sensor)

The description for reading the output is given by this input description:

```
Metadata
<Name>        stspOutput
<Type>        InputDescription
<Description> AttributeInDescriptor
                withName: 'interval classification');
              StringDescriptor
                with: '(';
              AttributeInDescriptor
                withName: 'article no.');
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'orientation (always zero)');
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'sensor (always s1)');
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'time interval begin');
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'time interval end');
```

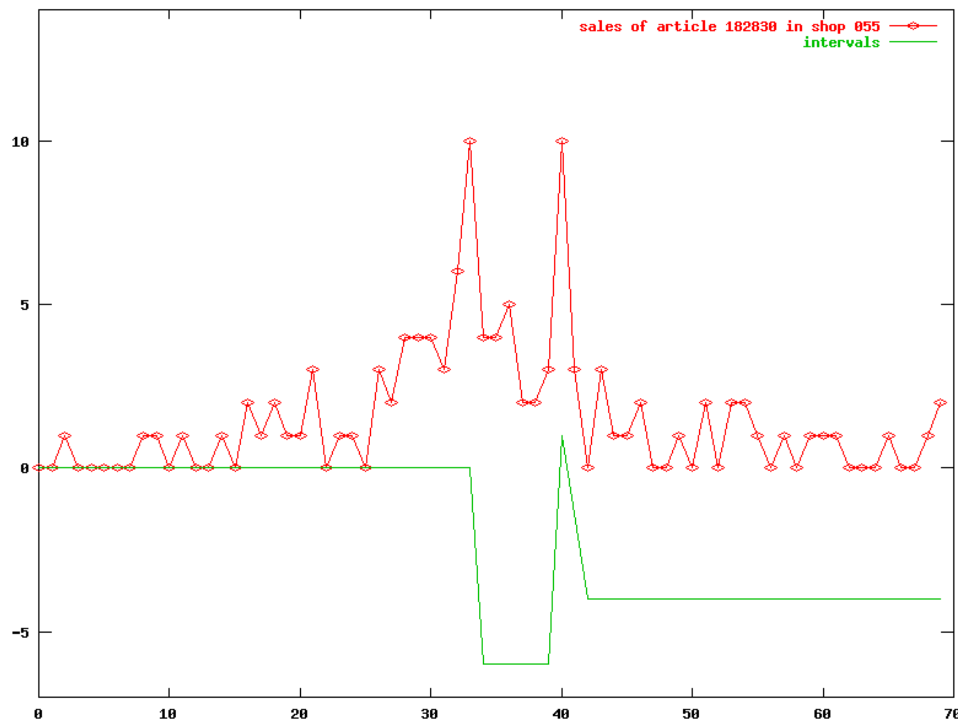

Figure 5.2: Sales of article 182830 in shop 055

```
FieldDelimiterDescriptor
  with: ',';
AttributeInDescriptor
  withName: 'gradient');
StringDescriptor
  with: ').'
```

The STSP output for branch no. 055 and article no. 182830 consists of six intervals (see Figure 5.2):

```
stable(182830,0,s1,1,33,0).
decreasing(182830,0,s1,33,34,-6).
stable(182830,0,s1,34,39,0).
increasing(182830,0,s1,39,40,7).
decreasing(182830,0,s1,40,42,-5).
stable(182830,0,s1,42,108,0).
```

definition of meta data (*MD RDT/DB input*) for input of RDT/DB (*RDT/DB input*) Since RDT/DB does access the RDBMS Oracle directly STSP output has to be loaded into the database. This can easily be done with the oracle tool *Oracle Loader*. The definition for input of RDT/DB

is given with the definition of the Oracle Loader input format which uses comma separated value. The meta data definition for our example is:

```
Metadata
<Name>        oracleLoaderInput
<Type>        InputDescription
<Description> AttributeInDescriptor
                withName: 'branch office identification';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'article no.';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'interval classification';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'time interval begin';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'time interval end';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'gradient'
```

definition of operator (*OP STSP-to-RDT/DB*) **to transform STSP output to input of RDT/DB** (*RDT/DB input*) The STSP output has to be transformed to the input format of the Oracle Loader which is given by the meta data above. The following meta data specifies the output description for the Oracle Loader input format. This output description is based on the input description given with meta data stspOutput. Thus the output description defines the requested transformation operator:

```
Metadata
<Name>        oracleLoaderInputFromStspOutput
<Type>        OutputDescription
<Description> StringDescriptor
                with: '//place branch identification here//';
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                  named: 'stspOutput'
                                  attributeInDescriptor: 'article no.');
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
```

```
                          named: 'stspOutput'
                          attributeInDescriptor: 'interval classification')
          map: ('increasing' -> 'I';
                'stable' -> 'S';
                'decreasing' -> 'D';
                'veryIncreasing' -> 'vI';
                'veryDecreasing' -> 'vD';
                'increasingPeak' -> 'iP';
                'decreasingPeak' -> 'dP');
        FieldDelimiterDescriptor
          with: ',';
        AttributeOutDescriptor
          withDescriptor: (Metadata
                          named: 'stspOutput'
                          attributeInDescriptor: 'time interval begin');
        FieldDelimiterDescriptor
          with: ',';
        AttributeOutDescriptor
          withDescriptor: (Metadata
                          named: 'stspOutput'
                          attributeInDescriptor: 'time interval end');
        FieldDelimiterDescriptor
          with: ',';
        AttributeOutDescriptor
          withDescriptor: (Metadata
                          named: 'stsptOutput'
                          attributeInDescriptor: 'gradient')
```

The mapping of the interval symbols to abbreviations, e.g. 'stable' to 'S', is included here to use the data produced with the operator OP STSP-to-RDT/DB for other purposes. E.g. generating a graph which shows the interval and places the interval symbols at the corresponding part of the graph requires abbreviated symbols. This kind of mapping can be useful in several other transformation operations.

The input for the Oracle Loader computed for branch no. 055 and article no. 182830 is:

```
branch055,182830,S,1,33,0
branch055,182830,D,33,34,-6
branch055,182830,S,34,39,0
branch055,182830,I,39,40,7
branch055,182830,D,40,42,-5
branch055,182830,S,42,108,0
```

The metadate for the operator *OP STSP-to-RDT/DB* is given with the meta data oracleLoaderInputFromStspOutput. These data were loaded into one single relation *miningmart.stspOutput* which was created by the following SQL-statement:

```
CREATE TABLE miningmart.STSPoutput
(
```

```
        BRANCH CHAR(9) NOT NULL,
        ARTICLE NUMBER(7) NOT NULL,
        INTERVAL CHAR(2) NOT NULL,
        BEGIN NUMBER(3) NOT NULL,
        END NUMBER(3) NOT NULL,
        GRADIENT NUMBER(2) NOT NULL
)
TABLESPACE MININGMART
STORAGE (
        initial 20m
        next 4m
        minextents 4
        pctincrease 0
)
parallel (degree 4);
```

The meta data for this table could also be described by the meta data scheme of deliverable D1 using multiple (meta data) tables.

definition of meta data (*MD RDT/DB output*) for output of RDT/DB RDT/DB produces metafacts which depend on the predicate names and the model used for learning (and on the database scheme, relation, etc.). Because this is parametrizing of the learning algorithm and belongs to the learning step already we do not consider this as part of the preprocessing. Basically the output is of representation L_{E_4} (2.10). An example of meta data w.r.t. a model is given below.

For exemplifying the benefit of this preprocessing experiment we will process some learning on the preprocessed data and check whether some valuable insights could be gained: for the final phase of the experiment we use the *const-to-pred* mapping 3 where the attribute *interval symbol* is supposed to be of constant value ('S', 'I', 'D', 'vI', 'vD', 'iP', 'dP') [12].

The mapping defines the predicates `s/5`, `i/5`, `d/5`, `vI/5`, `vD/5`, `iP/5`, `dP/5` where the predicate names were derived from the interval symbols of the relation miningmart.stspoutput and the five terms/arguments are the remaining attributes of the relation.

For generating hypotheses of great significance very complex models can be used. The following simple model is used for learning dependencies between two consecutive intervals:

```
serial(IntervalSymbol1,IntervalSymbol2) :
  IntervalSymbol1(Branch,Article,Start,End,Unused2)
  --> IntervalSymbol2(Branch,Article,End,Unused3,Unused4).
```

These are the corresponding meta data for the hypotheses tested by RDT/DB resp. for the facts that RDT/DB will produce for this model (if any):

```
Metadata
```

```
<Name>        rdtDBOutput
<Type>        InputDescription
<Description> StringDescriptor
                with: 'serial(';
              AttributeInDescriptor
                withName: 'interval symbol premise';
              FieldDelimiterDescriptor
                with: ',';
              AttributeInDescriptor
                withName: 'interval symbol conclusion';
              StringDescriptor
                with: ')'
```

An example hypothesis stating that an increasing peak is followed by a decreasing interval is:

```
serial(MININGMART.STSPOUTPUT_INTERVAL8_:iP,MININGMART.STSPOUTPUT_INTERVAL8_:D)
```

Lerning runs with different number of positive examples (from 5.000 to 500.000) proved the following hyptotheses, e.g. $\{I, vI]\} \rightarrow vD$ comprises two hypotheses $serial(I, vD)$ and $serial(vI, vD)$).

```
5000
{I, vI} -> vD
{P, vD} -> vI
{S, I, D} -> S
{S, I, vI, D} -> D
{S, I, D, vD} -> I

12000
{I} -> vD
{S, I, D} -> S
{S, I, D} -> D
{S, I, D, vd} -> I

20000
{I, D} -> S
{S, I, D} -> D
{S, I, D} -> I

200000/ 300000
{I} -> D
{D} -> D

400000
{I} -> D

above 500000
%
```

5.3 Experiment 2: Multivariate to Univariate Time Series Transformation II

The multivariate to univariate operator II is applied to an input of format L_{E_1} which contains energy consumption data. The format of this data has already been introduced in section 3.2 and is described with the following input description.

It may be surprising that this operator can be described without extending the input and output descriptions introduced in 5.1.

```
Metadata
<Name>        energyConsumptionMultivariate
<Type>        InputDescription
<Description> AttributeInDescriptor
                withName: 'date');
              FieldDelimiterDescriptor
                with: ';');
              AttributeInDescriptor
                withName: '0:15');
              FieldDelimiterDescriptor
                with: ';');
              AttributeInDescriptor
                withName: '0:30');
              FieldDelimiterDescriptor
                with: ';');
...
              AttributeInDescriptor
                withName: '23:45');
              FieldDelimiterDescriptor
                with: ';');
              AttributeInDescriptor
                withName: '0:00')
```

The output of this operator is of format $L_{E_1'}$ (2.4) and is transformed by the following output description.

```
Metadata
<Name>        energyConsumptionUnivariateFromMultivariate
<Type>        OutputDescription
<Description> EnumeratorDescriptor
                startingWith: 1
                step: 96;
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                   named: energyConsumptionMultivariate
                                   attributeInDescriptor: '0:15');
              EnumeratorDescriptor
                startingWith: 2
                step: 96;
```

```
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                  named: energyConsumptionMultivariate
                                  attributeInDescriptor: '0:30');
...
              EnumeratorDescriptor
                startingWith: 95
                step: 96;
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                  named: energyConsumptionMultivariate
                                  attributeInDescriptor: '23:45);
              EnumeratorDescriptor
                startingWith: 96
                step: 96;
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                  named: energyConsumptionMultivariate
                                  attributeInDescriptor: '0:00');
```

Instead of writing all attributes of the input vectors to the new univariate time series any requested subset of the input attributes can be selected. E.g. the following meta data specifies to process the attribute named '10:30' only:

```
Metadata
<Name>        energyConsumptionUnivariateSingleColumnFromMultivariate
<Type>        OutputDescription
<Description> EnumeratorDescriptor
                startingWith: 1
                step: 1;
              FieldDelimiterDescriptor
                with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                  named: energyConsumptionMultivariate
                                  attributeInDescriptor: '10:30');
```

This can be easily described with the input and output descriptions introduced for doing the experiments. That proves that the meta data (language) used here is capable of describing some of the manual preprocessing operators. Later on in this project the selection of attributes for reducing the number of columns will be done with the manual operator feature selection described in deliverable D1.

5.4 Experiment 3: Univariate to Multivariate Time Series Transformation

This experiment is not covered by any of the operators of section 3, but it points up that the input and output descriptions are powerful enough to express an operation like this.

The input is a univariate time series (representation $L_{E'_1}$, (2.4) or (2.3)). The following example transforms the univariate time series generated in 5.3 back to a multivariate time series. Again, we do not need to extend the given meta data (language): 96 (physical) input vectors of the univariate time series are defined to be one vector for the multivariate time series.

```
Metadata
<Name>        energyConsumptionUnivariate
<Type>        InputDescription
<Description> UnknownStringDescriptor; "This is the time specification
                                        which is not of interest"
              FieldDelimiterDescriptor
                with: ',');
              AttributeInDescriptor
                withName: '0:15');
              FieldDelimiterDescriptor
                with: (String
                         with: (Character lf));
...
              UnknownStringDescriptor;
              FieldDelimiterDescriptor
                with: ',');
              AttributeInDescriptor
                withName: '23:45');
              FieldDelimiterDescriptor
                with: (String
                         with: (Character lf));
              UnknownStringDescriptor;
              FieldDelimiterDescriptor
                with: ',');
              AttributeInDescriptor
                withName: '0:00')
```

The output of the operator is a multivariate time series (representation L_{E_1}, (2.2) or (2.1):

```
Metadata
<Name>        energyConsumptionMultivariateFromUnivariate
<Type>        OutputDescription
<Description> EnumeratorDescriptor
                startingWith: 1
                step: 1);
              FieldDelimiterDescriptor with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
```

```
                                named: 'energyConsumptionUnivariate'
                                attributeInDescriptor: '0:15');
...
              FieldDelimiterDescriptor with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                named: 'energyConsumptionUnivariate'
                                attributeInDescriptor: '23:45');
              FieldDelimiterDescriptor with: ',';
              AttributeOutDescriptor
                withDescriptor: (Metadata
                                named: 'energyConsumptionUnivariate'
                                attributeInDescriptor: '0:00')
```

Now we have got the original multivariate time series from 5.3. The only difference is that the time specifications of the new multivariate time series are increasing numbers. That is because the original time information was substituted already when we computed the univariate time series (which was the data source for this experiment).

Acknowledgements

This work has been partially funded by the European Commission, IST-1999-11993 (Mining Mart).

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large data bases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pages 478–499, Santiago, Chile, sep 1994.

[2] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 12, pages 307–328. AAAI Press/The MIT Press, Cambridge Massachusetts, London England, 1996.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *International Conference on Data Engineering*, Taipei, Taiwan, mar 1995.

[4] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[5] Marcus Bauer. *Simultane Ausreißer- und Interventionsidentifikation bei Online-Monitoring-Daten*. PhD thesis, University of Dortmund, 1997.

[6] Pavel Brazdil. Data transformation and model selection by experimentation and meta-learning. In C.Giraud-Carrier and M. Hilario, editors, *Workshop Notes – Upgrading Learning to the Meta-Level: Model Selection and Data Transformation*, number CSR-98-02 in Technical Report, pages 11–17. Technical University Chemnitz, April 1998.

[7] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule Discovery from Time Series. In Rakesh Agrawal, Paul E. Stolorz, and Gregory Piatetsky-Shapiro, editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 16 – 22, Ney York City, 1998. AAAI Press.

[8] Valery Guralnik, Duminda Wijesekera, and Jaideep Srivastava. Pattern Directed Mining of Sequence Data. In Rakesh Agrawal, Paul E.

Stolorz, and Gregory Piatetsky-Shapiro, editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 51 – 57, Ney York City, 1998. AAAI Press.

[9] Volker Klingspor. *Reaktives Planen mit gelernten Begriffen.* PhD thesis, Univ. Dortmund, 1998.

[10] Heikki Mannila, Hannu Toivonen, and A.Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–290, November 1997.

[11] Katharina Morik. The representation race - preprocessing for handling time phenomena. In *Proceedings of the European Conference on Machine Learning 2000*, Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, New York, 2000. Springer Verlag.

[12] Katharina Morik and Peter Brockhausen. A multistrategy approach to relational knowledge discovery in databases. In Ryszard S. Michalski and Janusz Wnek, editors, *Proceedings of the Third International Workshop on Multistrategy Learning (MSL-96)*, pages 17–27, Palo Alto, may 1996. AAAI Press.

[13] Katharina Morik and Stephanie Wessel. Incremental signal to symbol processing. In K.Morik, M. Kaiser, and V. Klingspor, editors, *Making Robots Smarter – Combining Sensing and Action through Robot Learning*, chapter 11, pages 185 –198. Kluwer Academic Publ., 1999.

[14] Dorian Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann Publishers, 1999.

[15] John Ross Quinlan. *C4.5: Programs for Machine Learning.* Machine Learning. Morgan Kaufmann, San Mateo, CA, 1993.

[16] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. Technical report, NeuroCOLT2 Technical Report Series, 1998.

[17] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Transactions on Knowledge and data Engineering 4*, 4:301–316, 1992.

[18] V. Vapnik. *Statistical Learning Theory.* Wiley, Chichester, GB, 1998.

[19] Claus Weihs and Jutta Jessenberger. *Statistische Methoden zur Qualitätssicherung und -optimierung in der Industrie.* Wiley-VCH, 1999.

[20] Stefanie Wessel. Lernen qualitativer Merkmale aus numerischen Robotersensordaten. Diplomarbeit, Fachbereich Informatik, Universität Dortmund, mar 1995.

[21] F. Wiechers. Verwaltung grosser datenmengen für die effiziente anwendung des apriori-algorithmus zur wissensentdeckung in datenbanken. Diplomarbeit, Universität Dortmund, Lehrstuhl 8, nov 1997.

[22] Wei Zhang. A region-based approach to discovering temporal structures in data. In Ivan Bratko and Saso Dzeroski, editors, *Proc. of 16th Int. Conf. on Machine Learning*, pages 484 – 492. Morgan Kaufmann, 1999.

[23] Wei Zhang. Some Improvements on Event-Sequence Temporal Region Methods. In Ramon López de Mántaras and Enric Plaza, editors, *Proceedings of the 11th Conference on Machine Learning (ECML 2000)*, pages 446 – 458, Berlin, 2000. Springer-Verlag.