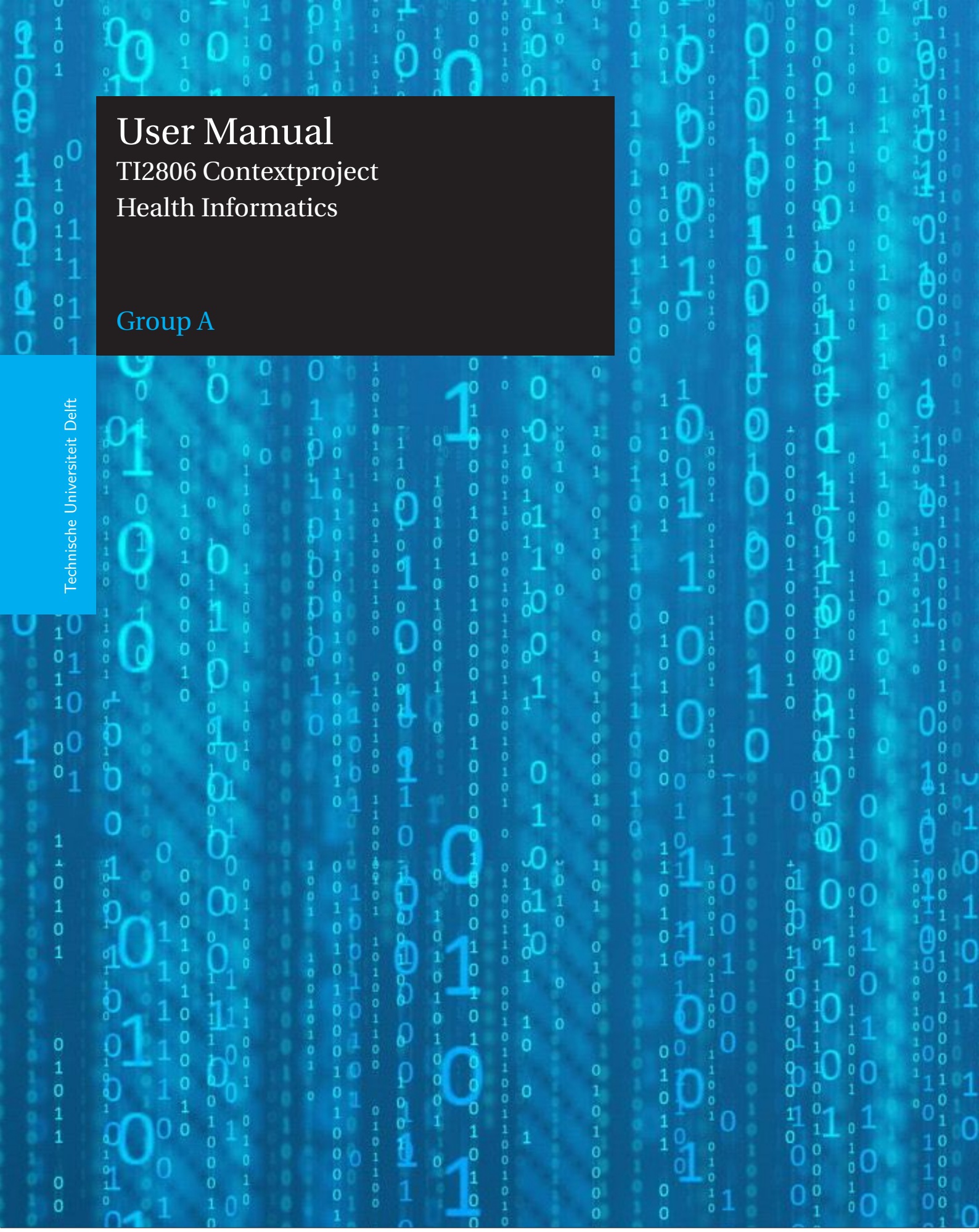
User Manual

TI2806 Contextproject
Health Informatics

Group A

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

CONTENTS

1 Introduction **1**

2 Workflow **2**

3 Input **3**

3.1 Creating and using an input description file 3

3.2 The input description file format 5

3.3 XML Wizard 6

4 Script **7**

5 Output **11**

A Script grammar **12**

B Script Examples with their explanation **15**

1

INTRODUCTION

This manual will explain and provide examples on how to use the software. The program, at its heart, is a sequential data analysis tool. Specifically, this software is meant for manipulating and analyzing data gathered in the self-monitoring process of patients who have undergone a kidney transplant. The examples and configuration files provided will primarily focus on this domain, however, the software is designed in such a way that it could be used for analyzing data for a much largery variety of contexts.

Here is a brief overview of the contents of this manual:

- *Chapter 2: Workflow* contains a step-by-step explanation on how to select input files, specify which analysis should be performed and how to start the analysis.
- *Chapter 3: Input* explains how to create an XML-document that describes the format of input data so that it can be parsed by the program.
- *Chapter 4: Script* explains in detail the scripting language used by the program and which operations can be performed with it.
- *Chapter 5: Output*
- *Appendix A: Script grammar* contains the grammar of the script, providing a formal of the syntax.
- *Appendix B: Script examples* contains example scripts complete with explanation of analyses on the self-monitoring data of kidney transplant patients.

2

WORKFLOW

This chapter will give an introduction of the steps a user has to take in order to perform an analysis.
The main steps are as follows:

1. **Input the files**. For a start, the user selects the data he wants to analyze and loads it into the application.
2. **Write the script**. Secondly, the user writes a script stating the operations that need to be performed and the visualizations that need to be generated.
3. **See the output**. Lastly, when the analysis is done the user can view the newly generated data and visualizations.

In the next chapters we will describe the steps in a more elaborate way.

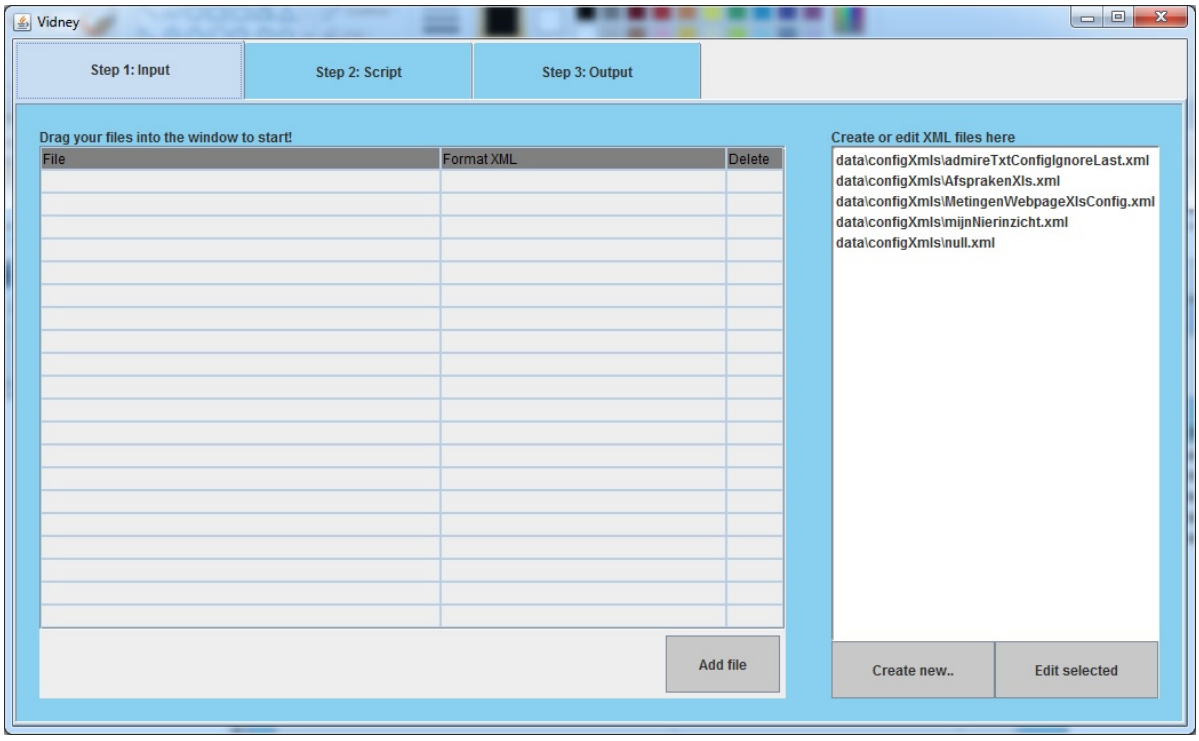
3

INPUT

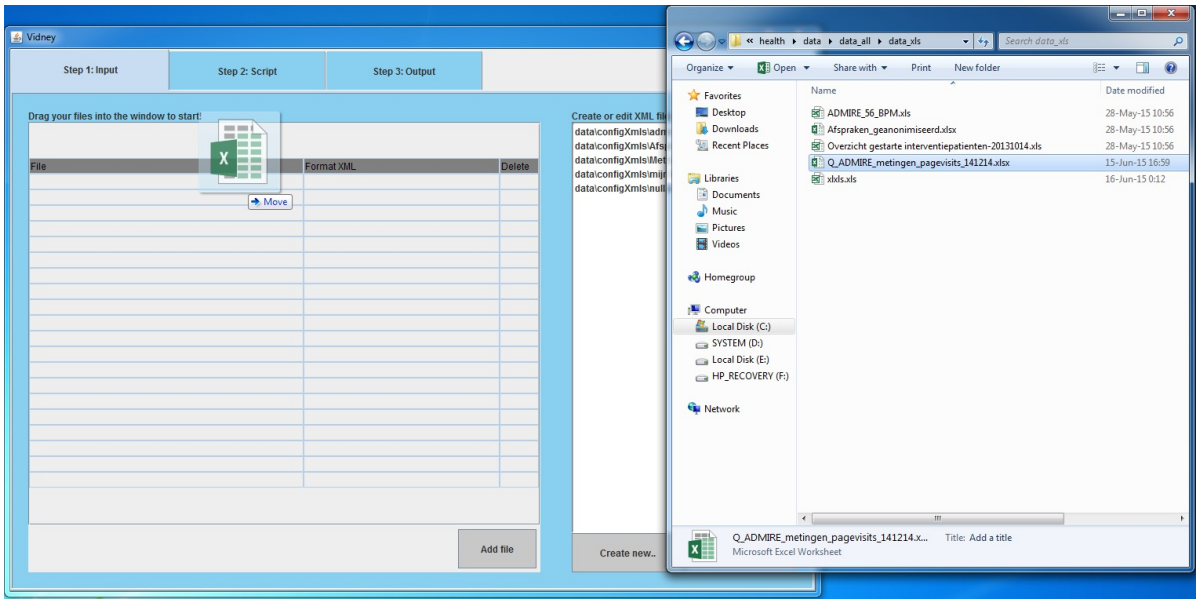
3.1. CREATING AND USING AN INPUT DESCRIPTION FILE

The input process can be pipelined and split down into two main steps, which has 4 transitions. The main steps are bold.

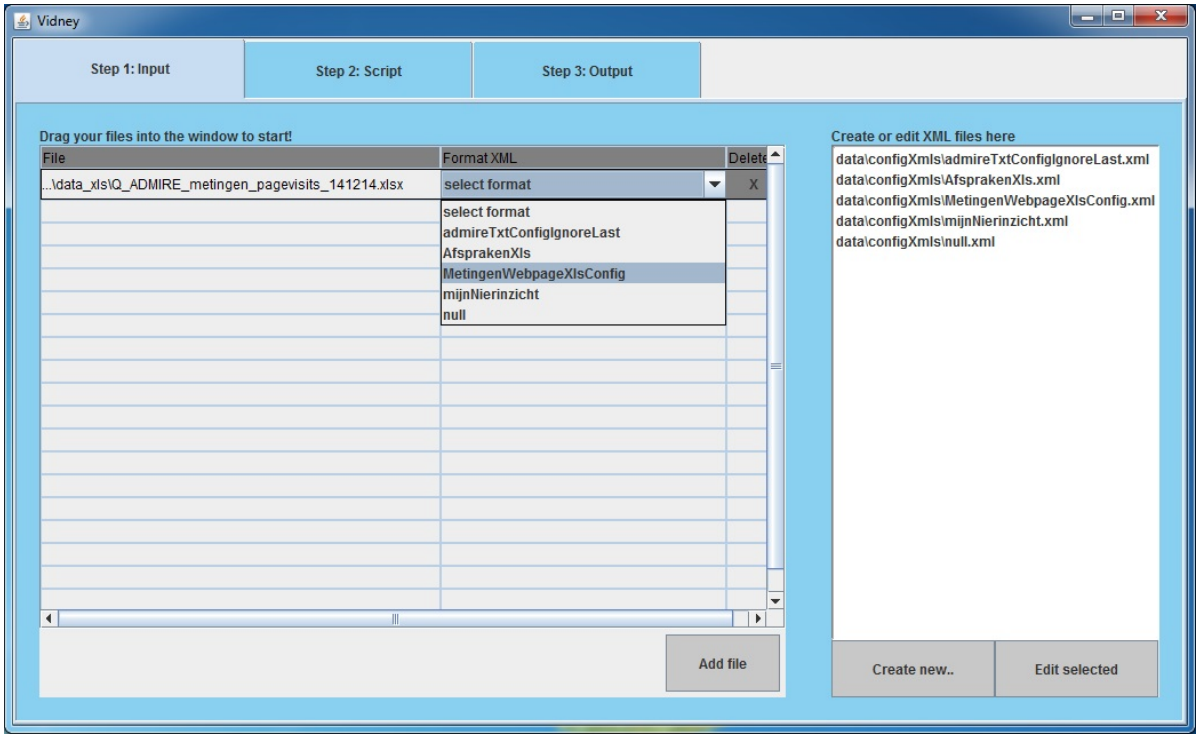
1. Starting with the standard empty GUI.



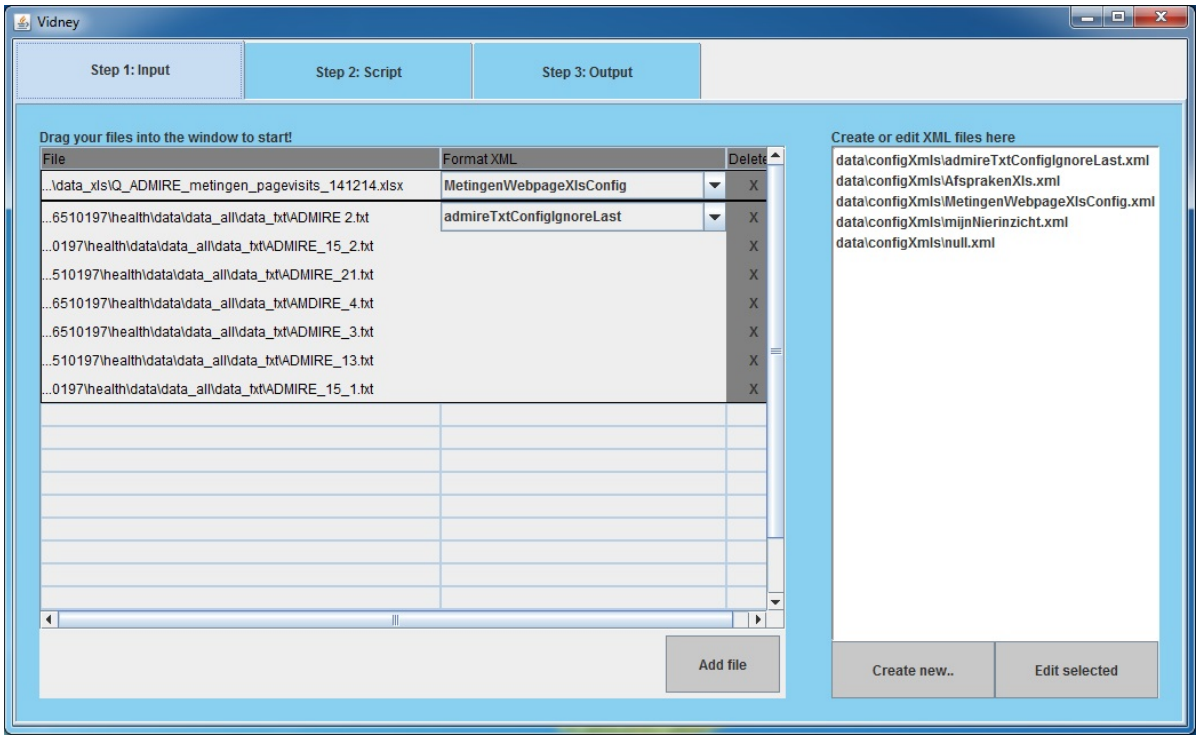
2. **Choosing your data.**



3. **Choosing your config which describes the data.**



4. Voila, the input has been chosen, and you are ready to progress.



3.2. THE INPUT DESCRIPTION FILE FORMAT

As seen in the previous section, a data set needs to be accompanied by an XML file. In this section we describe the format of such file.

The following is a description for a StatSensor data file.

```
<?xml version="1.0" encoding="UTF-8"?>
<data ignoreLast="1" delimiter="," end="]" start="[" format="text">
  <column type="String">type</column>
  <column type="Number">value</column>
  <column type="String">unit</column>
  <column type="Number">misc</column>
  <column format="yyMMdd" type="Date">date</column>
  <column type="Number">time</column>
</data>
```

Every input descriptor must have a root element named *data*. A *data* element must specify the *format* attribute, which tells the program what sort of data we are dealing with. This can be either *text*, *xls* or *xlsx*. Based on the selected format, several other attributes can be set.

For *text*, valid attributes are:

- *ignoreLast* - An option number of trailing lines to ignore.
- *delimiter* - The delimeter between columns.
- *start* - An optional string that denotes the start of the data.
- *end* - An optional string that denotes the end of the data.

For *xls* and *xlsx*, valid attributes are:

- *ignoreLast* - An option number of trailing rows to ignore.
- *startColumn* - The column at which the input data starts.
- *startRow* - The row at which the input data starts.

Within a *data* element, any number of *column* elements can be specified. As the name suggests, each *column* specifies a column. The name of a column is determined by the inner text of the *column* element. A *column* can also have a couple of attributes:

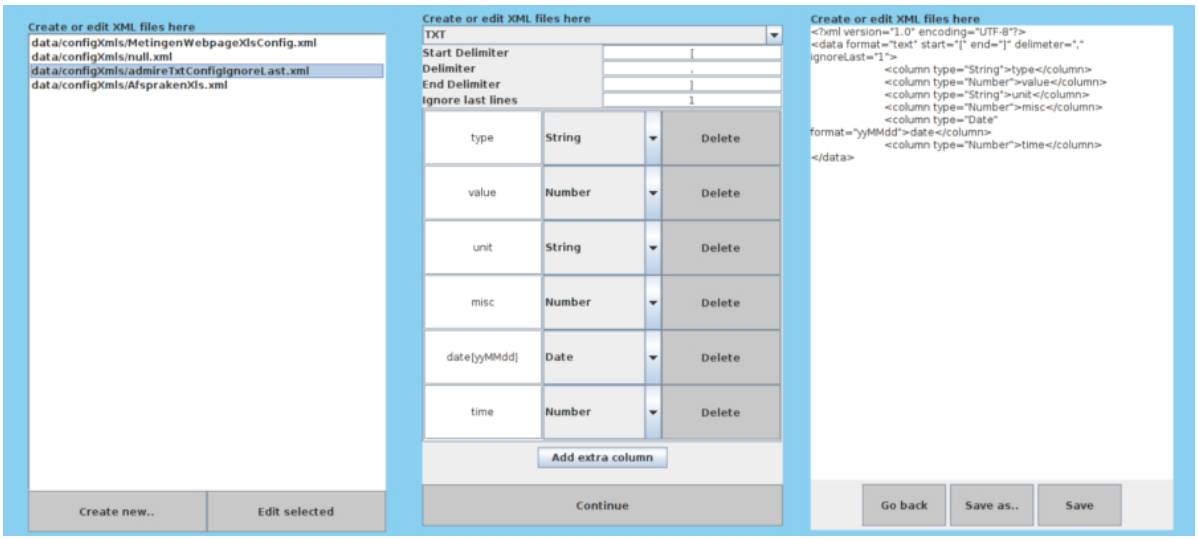
- *type* - Specifies the type of data in the column, this can be either *Number*, *String* or *Date*.
- *format* - An optional string that specifies the format of a *Date* column. This can be done by using *d* for the days, *M* for the months and *y* for the years. If the data does not always use the same same number of digits for the days, month or years, the letters can be put between brackets, for example *[d]d* would be used if it uses one digit for the first nine days of the month and two for the rest.

The following example shows an input description file for the Q_ADMIRE_metingen_pagevisits file:

```
<?xml version="1.0" encoding="UTF-8"?>
<data ignoreLast="0" startColumn="1" startRow="1" format="xlsx">
  <column type="Number">custommeasurementitem_141214Id</column>
  <column type="Number">Revision</column>
  <column type="String">Moment</column>
  <column type="Number">Date</column>
  <column type="Number">Sequence</column>
  <column type="Number">Value</column>
  <column type="Number">Value2</column>
  <column type="Number">IsMaverick</column>
  <column type="Number">RegressionCoefficient</column>
  <column type="Number">KreatinineAlgorithmRatingId</column>
  <column type="Number">KreatinineAlgorithmActionId</column>
  <column type="Number">KreatinineAlgorithmDayRatingId</column>
  <column type="String">Comment</column>
  <column type="String">CreatedBy</column>
  <column format="d[d]/MM/yy" type="Date">CreatedDate</column>
  <column type="String">ModifiedBy</column>
  <column format="d[d]/MM/yy" type="Date">ModifiedDate</column>
  <column type="Number">UserId</column>
  <column type="Number">CustomMeasurementId</column>
  <column type="String">Login</column>
  <column type="Number">custommeasurement_141214Id</column>
  <column type="String">measurementvariable_141214Name</column>
  <column type="String">measurementunit_141214Name</column>
</data>
```

3.3. XML WIZARD

Creating an XML file for your data set can be a painful job. For this reason we created the XML Wizard. Accessible from the input section, it allows for quick and easy creation and adjustment to XML files. Below you see the three steps of the XMl Wizard.



Firstly, you either select an existing file or choose to create a new. Then in the second screen you can edit all attributes such as file type, delimiters and the description of the columns.

4

SCRIPT

All the different operations and analyses in the application are achieved via a script that is provided by the user via the Script panel. This chapter will explain the structure of the script and the operations that can be done in it.

Variables A key aspect in the script is the use of variables. A variable is a symbolic name that is associated with a value and type. All data manipulation operations in the script are performed on variables. Before a variable can be used, it must first be declared. A variable declaration consists of an identifier that indicates the type of the variable, followed by an identifier that names the variable, and optionally an initializer consisting of an '=' token followed by an expression. The following example shows how to declare a variable.

```
Table table;                  // Declares a Table named table
String out = "Hello, world!"; // Declares a String named out and initializes it

// The following
Double x = 1, y, z = x;

// Is equivalent to
Double x; x = 1;
Double y;
Double z; z = x;
```

The supported variable types are *Boolean*, *Double*, *Object*, *String*, *Table*, *EventList*, *EventSequence*. After a variable is declared it is impossible to change the type of that variable. In addition to explicitly specifying the name of the type, it is also possible to declare a variable using the implicit type *var*. An implicitly typed variable *must* be initialized so that the type of the variable can still be determined.

```
// Invalid, an implicitly typed variable must be initialized.
var x;

// Valid
var x = 0.5;
```

There are some variables that are already declared by the application itself: the variables *table0* through *tableN*. *table0* corresponds to the first input file to the application and *tableN* to the last. Each of these variables is typed *Table* and contains the parsed data of their respective input file.

Expressions In addition to declaring variables, the script also allows expressions statements. An expression statement evaluates an expression, there are two kinds of expression statements: assignment expressions and function invocation expressions.

Assignment is virtually identical to the initialization of variables shown in the preceding paragraph.

```
// Given that variables x and y are declared previously,
// assigns the value of y to x.
x = y;
```

A function invocation consists of an identifier specifying the name of the function invoke, an '(' token, optionally a list of expressions that are evaluated before calling the function and the results of which are passed to the function as arguments, and finally a ')'. It is possible for functions to return a new value, so that the result can be assigned to a variable.

```
// Calls a function called 'myFunction' with arguments x and y and assigns the result to z.
z = myFunction(x, y);
```

Data manipulation expressions The script provides various special expressions for data manipulation that can be used within an expression statement.

Chunking First of all, the *chunk* expression allows user to chunk data.

```
// Given a table table0 with the at least two columns: date and value.

// Chunks table0 on rows with an identical value.
var table = chunk table0 by value;

// Chunks table0 on rows with an identical date, and selects the number of values per chunk.
var table = chunk table0 by date
  select count of value;

// Chunks table0 on rows within a seven day period of each other, and selects the date and average value
    per chunk.
var table = chunk table0 by date per week
  select date, average of value;
```

The select keyword is followed by a list of columns or aggregate operations applied on columns that are added to the resulting table. The supported aggregate functions are *count, sum, average, min, max*. The name of the resulting column in the table for each aggregate operation is given by connecting the name of the function to name of the column, for example, *count_value* for 'count of value'.

As shown in the example, chunking can also be done based on a certain period of time. The supported periods are *hour, day, week month year]* In addition to specifying a time unit, it is also possible to chunk on a multiple thereof, for example, *24 hours* or *2 years*.

Coding Next, a set of data can be coded as events.

```
// Given a table table0 with at least one column: value.

// Codes the rows of table0 into three codes: LOW, MEDIUM and HIGH based on the value of column value.
// It should be noted that the result of a code expression is an EventList and not a Table.
var events = code table0 as
 LOW : value <= 100,
   MEDIUM : value > 100 and value <= 200,
   HIGH : value > 200;
```

As shown in the example, conditions such as *value > 100* can be chained together using the *and* and *or* keywords. The result of a coding expression is a *EventList*.

Connecting It is also possible to connect tables with each other to form a new table.

```
// Given two tables:
//  table0 with at least one column: crea.
//  table1 with at least one column: Value1.

// Connects table0 and table1, merging the crea and Value1 columns into a single column named value.
var table = connect table0 with table1 where crea = Value1 as value;
```

Constraining Finally, it is possible to constrain a table, essentially filtering out rows based on a given condition.

```
// Given a table table0 with at least one column: value.

// Selects only the rows where value is between 50 and 500.
var table = constrain table0 where value >= 50 and value =< 500;
```

Data manipulation functions In addition to the above expressions, the script also provides several functions that can be used to manipulate data.

```
// Returns a new EventSequence consisting of the given codes.
sequence(String... codes);

// Returns an EventSequence with instances of the given sequence in the EventList.
findSequence(EventList events, EventSequence sequence);

// Creates a new table with an extra column day_of_week that contains
// the day of the week of the first date found in each row.
tableWithDays(Table table);

// Creates a new table with an extra column hour_of_day that contains
// the hour of the day of the first date found in each row.
tableWithHours(Table table);

// Parses the time in timeColumn and adds the time to the date in dateColumn.
addTimeToDate(Table table, String dateColumn, String timeColumn);
```

Variables of the types *EventList* and *EventSequence* also have a member function *toTable()* which converts the object into a *Table*. They can be used like this:

```
var eventsTable = events.toTable();

var sequencesTable = sequence.toTable();
```

Output functions Vital to the application is the ability to output the generated data so that it can be used as input for further analysis by statistical analysis tools or even itself. Therefor the script provides two functions for writing a set of data to a file:

```
// Writes a table to a file using a default format
write(String filename, Table table);
```

This function will print every row on a new line using a comma as separator between columns.

```
// Writes a table to a file using the given format
write(String filename, Table table, String format);
```

This function will format the rows according to the format string. The format string is an ordinary string, except that names enclosed in curly braces will be substituted by the value of the column with that name. To insert a literal curly brace character, two braces must be used. The following example demonstrates the use of this function and the format string.

```
// Given a table table0 with at least two columns: date and value.

// Writes a table to a file where each row is formatted as "date: value".
write("result.txt", table0, "{date}: {value}");
```

Visualizations functions It is also possible to create visualizations via a script. Each generated visualization will be listen in the output tab of the program, where it can be inspected. Optionally each visualization can be exported to a pdf file via the script. Currently there are four supported visualizations: a *box plot*, a *frequency bar*, a *histogram* and a *state transition matrix*. The following example shows how to generate each of these visualizations.

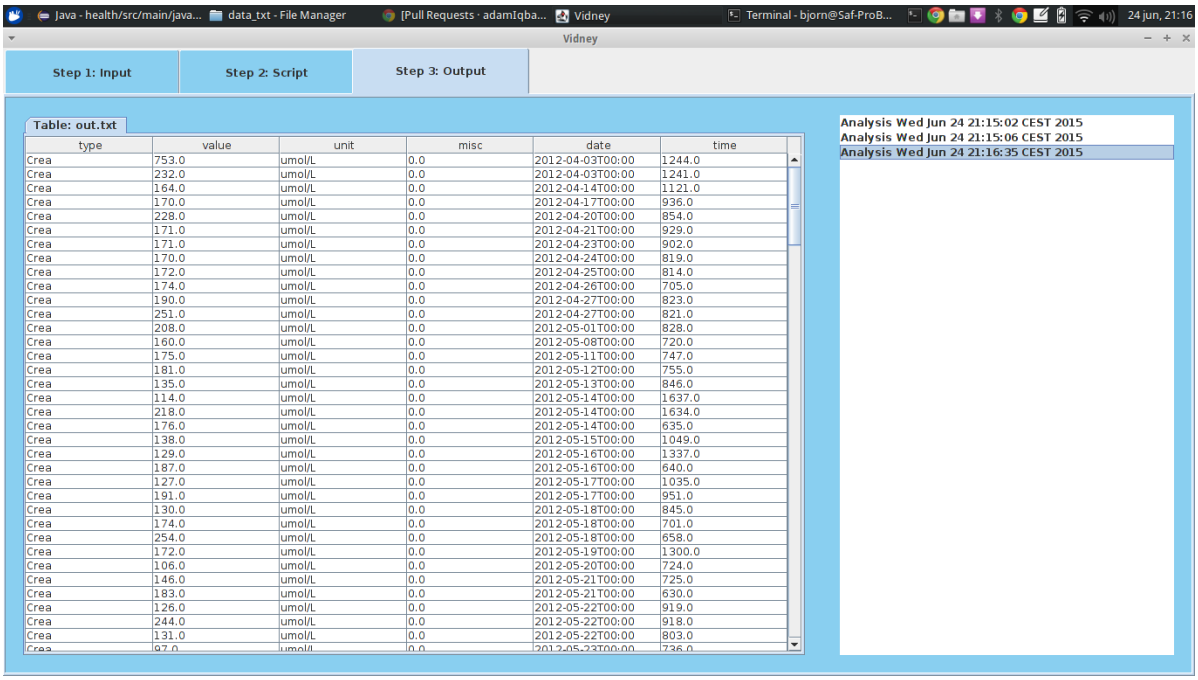
```
// Each of these visualizations will be exported as a pdf file if the optional filename is specified.

// Creates a box plot of the given column.
boxplot([String filename], Table table, String column);

// Creates a frequency bar of the given column. If no column is specified,
// the first column whose name starts with "count_" will be used instead.
freqbar([String filename], Table table, [String column]);

// Creates a histogram of the given column with the specified number of bins.
hist([String filename], Table table, String column, Double numBins);

// Creates a state transition matrix of the given events list, or if a sequence is specified,
```

```
// the occurances of that sequence in the event list.
transitionMatrix([String filename], EventList events, [EventSequence sequence]);
```

5

OUTPUT

Finally when the analysis of the data is complete, it is time to check the results. In the output section the user can find the data tables and visualizations he has asked for in the previous section. You can do this for all completed analyses performed during the current session. Below is an example of the output section and an explanation of the section's elements.



On the right there is a list containing all analyses that have been performed. When selected, the results of the analysis appear in the main section of the screen. You can browse between the generated data by clicking on the tabs on the top of this screen.

A

SCRIPT GRAMMAR

This appendix contains the grammar of the scripting language that is used for analyses.

Tokens

⟨*token*⟩ ::= ⟨*identifier*⟩
| ⟨*keyword*⟩
| ⟨*boolean-literal*⟩
| ⟨*number-literal*⟩
| ⟨*string-literal*⟩
| *Operator or punctuator character*

Identifiers

⟨*identifier*⟩ ::= ⟨*identifier-start-character*⟩⟨*identifier-rest-character*⟩*

⟨*identifier-start-character*⟩ ::= *Letter character*
| '_'

⟨*identifier-rest-character*⟩ ::= *Letter or digit character*
| '_'

Literals

⟨*literal*⟩ ::= ⟨*boolean-literal*⟩
| ⟨*number-literal*⟩
| ⟨*string-literal*⟩

⟨*boolean-literal*⟩ ::= true
| false

⟨*number-literal*⟩ ::= ⟨*number-digits*⟩ '.' ⟨*number-digits*⟩ [⟨*number-exponent*⟩]
| '.' ⟨*number-digits*⟩ [⟨*number-exponent*⟩]
| ⟨*number-digits*⟩ [⟨*number-exponent*⟩]

⟨*number-digits*⟩ ::= (⟨*digit*⟩)+

⟨*number-exponent*⟩ ::= ('e' | 'E')('+' | '-')[⟨*number-digits*⟩]

⟨*string-literal*⟩ ::= '"' ⟨*string-character*⟩* '"'

⟨*string-character*⟩ ::= *Any character except " and new-line-character*

Statements

⟨*program*⟩ ::= ⟨*statement*⟩*

⟨*statement*⟩ ::= ⟨*declaration-statement*⟩
| ⟨*expression-statement*⟩

⟨*declaration-statement*⟩ ::= ⟨*local-variable-declaration*⟩ ';'

⟨*expression-statement*⟩ ::= ⟨*statement-expression*⟩ ';'

⟨*local-variable-declaration*⟩ ::= ⟨*local-variable-type*⟩ ⟨*local-variable-declarator-list*⟩

⟨*local-variable-type*⟩ ::= 'var'
| ⟨*identifier*⟩

⟨*local-variable-declarator-list*⟩ ::= ⟨*local-variable-declarator-list*⟩ ',' ⟨*local-variable-declarator*⟩
| ⟨*local-variable-declarator*⟩

⟨*local-variable-declarator*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩ '=' ⟨*expression*⟩

Expressions

⟨*statement-expression*⟩ ::= ⟨*primary-expression*⟩
| ⟨*assignment-expression*⟩

⟨*expression*⟩ ::= ⟨*non-assignment-expression*⟩
| ⟨*assignment-expression*⟩

⟨*assignment-expression*⟩ ::= ⟨*primary-expression*⟩ '=' ⟨*expression*⟩

⟨*non-assignment-expression*⟩ ::= ⟨*primary-expression*⟩
| ⟨*chunk-expression*⟩
| ⟨*constrain-expression*⟩
| ⟨*connect-expression*⟩
| ⟨*code-expression*⟩

⟨*primary-expression*⟩ ::= ⟨*literal*⟩
| ⟨*identifier*⟩
| '(' ⟨*expression*⟩ ')'
| ⟨*primary-expression*⟩ '.' ⟨*identifier*⟩
| ⟨*primary-expression*⟩ '(' [⟨*argument-list*⟩] ')'

⟨*argument-list*⟩ ::= ⟨*argument-list*⟩ ',' ⟨*expression*⟩
| ⟨*expression*⟩

⟨*chunk-expression*⟩ ::= 'chunk' ⟨*identifier*⟩ 'by' ⟨*identifier*⟩ [⟨*period-specifier*⟩] ['select' ⟨*chunk-selection-list*⟩]

⟨*period-specifier*⟩ ::= 'per' ⟨*period*⟩

⟨*period*⟩ ::= ⟨*singular-time-unit*⟩
| ⟨*number*⟩ ⟨*plural-time-unit*⟩

⟨*singular-time-unit*⟩ ::= 'hour'
| 'day'
| 'week'
| 'month'
| 'year'

⟨*plural-time-unit*⟩ ::= 'hours'
| 'hours'
| 'days'
| 'months'
| 'years'

⟨*chunk-selection-list*⟩ ::= ⟨*chunk-selection-list*⟩ ',' ⟨*column-or-aggregate-operation*⟩
| ⟨*column-or-aggregate-operation*⟩

⟨*column-or-aggregate-operation*⟩ ::= [⟨*aggregate-operation*⟩ 'of'] ⟨*identifier*⟩

⟨*aggregate-operation*⟩ ::= 'count'
| 'average'
| 'sum'
| 'min'
| 'max'

⟨*constrain-expression*⟩ ::= 'constrain' ⟨*identifier*⟩ 'where' ⟨*conditional-expression*⟩

⟨*conditional-expression*⟩ ::= '(' ⟨*conditional-expression*⟩ ')'
| ⟨*conditional-expression*⟩ ⟨*boolean-operator*⟩ ⟨*conditional-expression*⟩
| ⟨*condition*⟩

⟨*boolean-operator*⟩ ::= 'and'
| 'or'

⟨*condition*⟩ ::= ⟨*identifier*⟩ ⟨*comparison-operator*⟩ ⟨*expression*⟩

⟨*comparison-operator*⟩ ::= '=='
| '!='
| '<'
| '<='
| '>'
| '>='

⟨*connect-expression*⟩ ::= 'connect' ⟨*identifier*⟩ 'with' ⟨*identifier*⟩ 'where' ⟨*column-connection-list*⟩

⟨*column-connection-list*⟩ ::= ⟨*chunk-selection-list 'and' <column-connection*⟩
| ⟨*column-connection*⟩

⟨*column-connection*⟩ ::= ⟨*identifier*⟩ '==' ⟨*identifier*⟩ ['as' ⟨*identifier*⟩]

⟨*code-expression*⟩ ::= 'code' ⟨*identifier*⟩ 'as' ⟨*code-list*⟩

⟨*code-list*⟩ ::= ⟨*code-list*⟩ ',' ⟨*code*⟩
| ⟨*code*⟩

⟨*code*⟩ ::= ⟨*identifier*⟩ ':' ⟨*conditional-expression*⟩

B

Script Examples with their Explanation

1. What time of the day and on what day do people measure themselves?
   This question can be split into two questions.

   (a) What time of the day do people measure themselves.

```
// this meant for the statsensor files
var table = connect table0 with table1 where date = date;
table = connect table with table2 where date = date;
table = connect table with table3 where date = date;
table = connect table with table4 where date = date;
table = connect table with table5 where date = date;
table = connect table with table6 where date = date;

addTimeToDate(table , "date", "time");

table = tableWithHoursOfDay(table);

hist(table, "hour_of_day", 24);
```

   Every table from table0 through table6 represents an input data file, like admire2.txt for example. Each of these the tables is connected on the date column to form a new table, which is stored in a variable named *table*. After that, we use the *addTimeToDate* function to add a time component to each date. Next, the *tableWithHoursOfDay* function is used to insert an extra column, *hour_of_day* to the table, which contains the hour of the time for every row. Now that we have a table containing all neccessary data, we create a frequency bar on the *hour_of_day* column of *table*, so that we obtain a histogram showing how often each hour was present in table.

   (b) What day do people measure themselves.

```
// this meant for the statsensor files

var table = connect table0 with table1 where date = date;
table = connect table with table2 where date = date;
table = connect table with table3 where date = date;
table = connect table with table4 where date = date;
table = connect table with table5 where date = date;
table = connect table with table6 where date = date;

table = tableWithDays(table);

freqbar(table, "day_of_week");
```

   Every table from table0 through table6 represents an input data file, like admire2.txt for example. Each of these the tables is connected on the date column to form a new table, which is stored in a variable named *table*. After that, the *tableWithDays* function is used to insert an extra column, *day_of_week* to the table, which contains the days of the week for every row. Now that we have

a table containing all neccessary data, we create a frequency bar on the *day_of_week* column of *table*, so that we obtain a histogram showing how often each day was present in table.

2. What time of the day and on what day do they enter measure measurement?
   This question too can be split into two questions.

   (a) What time of the day do they enter their measurement.

```
// with MijnnierInzicht Data

var table = tableWithHoursOfDay(table0);

hist(table, "hour_of_day", 24);
```

   The script above declares a variable called table. The whole mijnNierInzicht data is loaded into this table. Then we add another column to this data, which represent the *days_of_week*. After that we choose to add another column to the table, in which the days of the week are added for every column, so the day of the week per record/row translated from the date. Now that our table is completely as we want, we create a frequency bar which has table as input, and disperses it on it's the *day_of_week* column and tells us how often the days occurred. You could see it as distinct count in mySQL for example.

   (b) What day do they enter their measurement.

```
// this is meant for the mijnNierInzicht data

var table = tableWithDays(table0);

freqbar(table, "day_of_week");
```

   The script above declares a variable called table.The whole mijnNierInzicht data is loaded into this table. Then we add another column to this data, which represent the *days_of_week* After that we choose to add another column to the table, in which the days of the week are added for every column, so the day of the week per record/row translated from the date. Now that our table is completely as we want, we create a frequency bar which has table as input, and disperses it on it's *day_of_week* column and tells us how often the days occurred. You could see it as distinct count in mySQL for example.m which has table as input, and disperses it on it's the *day_of_week* column.

3. Is there a difference between StatSensor measurement and what patients enter into Mijnnierinzicht?

```
// With admireMetingen as table0 and admire4.txt measurement for example

var table = constrain table0 where Login = "admire4";

table = connect table1 with table where ModifiedDate = date as date;

var event = code table as measurement : value > 0 , entry : Value > 0;

// find sequence with same user
var seq = sequence("measurement", "entry", true);
findSequences(event , seq);

table = seq.toTable();

write("out.txt" , table);
```

   Firstly the admire data is constrained to data of user *admire4* and connected with the measurement data from the StatSensor and the ModifiedData and data columns are merged to a single column named data. Then the data is coded as either a StatSensor measurement or the entry of a measurement on mijnnierinzicht. The following step is to find all instances of the sequence *measurement->entry* and lastly these instances are converted to a table which is written to a text file named out.txt.

4. How often do patients measure themselves before they enter data into Mijnnierinzicht?

```
// with metingen as table0 and after that all admire txt measurement

var tableOriginal = constrain table0 where Login = "admire2";
```

```

tableOriginal = connect table1 with tableOriginal where ModifiedDate = date as date;

// Find instances of the sequenc
var event = code tableOriginal as measurement : value > 0 , 1meas : Value > 0;
var seq = sequence("measurement", "1meas", true);
findSequences(event , seq);
var table = seq.toTable();
table = constrain table where Value > 0;

event = code tableOriginal as measurement : value > 0 , 2meas : Value > 0;
seq = sequence("measurement", "measurement",  "2meas", true);
findSequences(event , seq);
var tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 3meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "3meas", true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 4meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "measurement", "4meas", true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 5meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "measurement", "measurement", "5meas",
     true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 6meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "measurement", "measurement", "
     measurement", "6meas", true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 7meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "measurement", "measurement", "
     measurement", "measurement", "7meas", true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

event = code tableOriginal as measurement : value > 0 , 8meas : Value > 0;
seq = sequence("measurement", "measurement", "measurement", "measurement", "measurement", "
     measurement", "measurement", "measurement", "8meas", true);
findSequences(event , seq);
tmp = seq.toTable();
tmp = constrain tmp where Value > 0;
table = connect table with tmp where date = date;

write("out.txt" , table);

freqbar(table, "code_name");
```

5. Do patient follow up advice given my Mijnnierinzicht?

```
// with metingen xlsx as table0 and afspraken afspraken anoniem xlsx as table1
var table = constrain table0 where KreatinineAlgorithmDayRatingId >= 5;
```

```
// constrain for specific user
table  = constrain table where Login = "admire3";

var tableDocAppointments = constrain table1 where admireNum = 3;

var events = code table0 as concern : KreatinineAlgorithmDayRatingId == 4 ,
  noconcern : KreatinineAlgorithmDayRatingId != 4;
var seq = sequence("concern" , "concern", true);
findSequences(events, seq);

var contactHospTable = seq.toTable();

table = connect table with contactHospTable
  where KreatinineAlgorithmDayRatingId = KreatinineAlgorithmDayRatingId;

table = connect table with tableDocAppointments where date = ModifiedDate;

write("out.txt", table);

transitionMatrix(events , seq);
```

6. What are the conditions under which people overwrite their initial data entered in Mijnnierinzicht?

```
// on Mijnnierinzicht data

var table = constrain table0 where CreatedDate != ModifiedDate;

write("out.txt", table);
```

   The records in table0 are constrained to records where the *CreatedDate* differs from the *ModifiedDate* and the resulting records are then written to out.txt.

7. Find cases where Mijnnierinzicht advice to contact the hospital

```
// with metingen xlsx as table0

var table = constrain table0 where KreatinineAlgorithmDayRatingId >= 5;

var events = code table0 as concern : KreatinineAlgorithmDayRatingId == 4;

var seq = sequence("concern" , "concern");

findSequences(events, seq);

var contactHospTable = seq.toTable();

table = connect table with contactHospTable
  where KreatinineAlgorithmDayRatingId = KreatinineAlgorithmDayRatingId;

write("out.txt", table);

transitionMatrix(events , seq);
```

8. How well do patients follow up advice of Mijnnierinzicht to re-measure again?

```
// on Q_ADMIRE_metingen_pagevisits_141214.xlsx

var t = constrain table0 where KreatinineAlgorithmActionId <= 1;

t = chunk t by KreatinineAlgorithmActionId
  select count of KreatinineAlgorithmActionId , measurementvariable_141214Name;

var t2 = constrain table0 where measurementvariable_141214Name == "Kreatinine2 (stat)";

t2 = chunk t2 by measurementvariable_141214Name
  select count of Sequence , min of KreatinineAlgorithmActionId;

t2 = connect t2 with t where count_Sequence = count_KreatinineAlgorithmActionId as count_col
  and KreatinineAlgorithmActionId = min_KreatinineAlgorithmActionId;

freqbar(t2, "measurementvariable_141214Name");
```

9. When do people deviate from their routine?

```
// with admire statsensor data of one user

var tablePerDay = chunk table0 by date per day select count of date;
write("tablePerDay.txt", tablePerDay);

var tablePer2Days = chunk table0 by date per 2 days select count of date;
write("tablePer2Days.txt", tablePer2Days);

var tablePerWeek = chunk table0 by date per week select count of date;
write("tablePerWeek.txt", tablePerWeek);
```

This does not actually show when people deviate, but does show the frequency in which people measure, knowing the routine, you could see in the written table when they deviate from it. If you would want more info on why they deviate from it, you could connect the entry table, constraint by the user you want to look at, and then select more columns in the chunk statement.