LISp-Miner Control Language
description of scripting language implementation

Milan Simunek
University of Economics Prague, Faculty of Informatics and Statistics, Czech Republic
simunek@vse.cz

DOI: 10.20470/jsi.v5i2.193

Abstract: This paper introduces the LISp-Miner Control Language – a scripting language for the LISp-Miner system, an academic system for knowledge discovery in databases. The main purpose of this language is to provide programmable means to all the features of the LISp-Miner system and mainly to automate the main phases of data mining – from data introduction and preprocessing, formulation of analytical tasks, to discovery of the most interesting patterns. In this sense, the language is a necessary prerequisite for the EverMiner project of data mining automation. Language will serve other purposes too – for an automated verification of the LISp-Miner system functionality before a new version is released and as an educational tool in advanced data mining courses.

Key words: Knowledge discovery in databases, automation, EverMiner, LISp-Miner, scripting language, Lua

1. Introduction

The goal of this paper is to introduce the LISp-Miner Control Language (LMCL) and describe its implementation. LMCL opens an access to LISp-Miner system core objects and functionality to be used on a higher level of abstraction in user-created scripts written in an understandable programming language. Its syntax allows for all the common programming concepts (as variables, expressions evaluation) and execution control constructs (if-then, loops or functions calls). Scripts are executed automatically and could perform sequences of operations much faster than if initiated manually through user interface. Thus, an algorithm could be implemented in LMCL syntax to automate some data mining process phases (and possible all of them).

There are several unique features implemented in the LISp-Miner system, as mentioned later in this paper and in more details in cited references. The most important of them is a rich syntax of several types of mined patterns and theoretically well-founded inclusion of domain knowledge across the whole data mining process. Those features are becoming available through LMCL implementation and together they provide a necessary prerequisite for achieving the goal of data mining automation.

Data mining process automation was included in the well known paper “10 Challenging Problems in Data Mining Research” (Yang&Wu, 2006) back in the 2006. There are mentioned challenges regarding automation of data mining operations, under the problem number 8, together with a need for special care that should be given to the pre-processing phase and data cleaning namely. Paper concluded that significant costs saving could be implied from successful mastering of automation.

On the other hand, we are well aware that the whole problem is much wider than just crawling through data. There are other business-oriented steps that pre-cede or follow-up the actual data mining analysis. These are hardy to automate and possibly not suitable for automation at all (e.g. a problem identification and definition from managerial point of view, trust establishment between data owner and data analyzer or practical deployment of knowledge obtained through data mining analysis). Nevertheless, there are clear benefits from automation of computer-aided phases of data mining – be it speed-up in time of analysis, an automatic deployment of know-how and best-practices or a new value added through permanent update of known patterns.

Data mining automation is a long term research goal of the EverMiner project (Šimůnek&Rauch, 2011), (Rauch, 2012a). It is built upon the LISp-Miner system platform (see the next section) and on theoretical results in areas of formalization of domain knowledge, formulation of analytical questions, observational calculi and synthesis of new knowledge from found patterns (Rauch, 2011), (Rauch, 2012b), (Rauch&Šimůnek, 2012). The idea of the EverMiner is inspired by the project GUHA80 (Hájek&Havránek, 1982, Hájek&Ivánek, 1982) that has been but never realized. The architecture,
particular software, theoretical components and the principles of the mining process management used in EverMiner differ from that used in GUHA80. However both projects are based on the application of GUHA data mining procedures (see later).

This paper is organized as follows. Two basic stones the LMCL was built upon – LISp-Miner system and Lua scripting language – are shortly described in the next section and section 3 respectively. A general concept of implemented solution, detailed description of LMCL language syntax and of the way it connects scripts to LISp-Miner Core is presented in section 4. The LM Exec module to interpret scripts is described in section 5. A special section 6 was dedicated to an automatically generated programmer’s documentation to LMCL. Achieved results and a proof of the concept in terms of the EverMinerSimple demo example is in section 7. They are other existing languages used for data mining and for algorithms description mentioned in section 8. Finally, a summary ends this paper.

2. LISp-Miner System

The LISp-Miner system is an academic system used mainly for data mining research and teaching. The system is developed at University of Economics, Prague since 1996. It is freely available at http://lispminer.vse.cz and is used at several universities in Czech Republic, Finland, France and USA and for real data analyses. For more details see (Šimůnek, 2003). LISp-Miner consists now of ten data mining analytical procedures plus thirteen other modules supporting e.g. the Business understanding and Data preprocessing phases of the data mining process, parallel processing or communication with other systems.

The system is based on many decades of related research of the GUHA method, an original Czech method of exploration analysis. Theoretical foundations were published in books and papers since 1960’s – see e.g. (Hájek et al., 1966), (Hájek, 1974), (Hájek&Havránek, 1978), ( Holeňa, 1996), (Rauch, 2005), (Rauch, 2009), (Piche&Turunen, 2010), (Rauch&Šimůnek, 2008). A complex overview could be found in (Rauch, 2013) and a summary of the GUHA method in (Hájek et al., 2010).

There are several types of patterns the LISp-Miner could mine for: 4ft-association rules – we would like to stress that we do not mine for simple association rules derived from shopping baskets in the sense of (Agrawal et al., 1993), but for more complex types of patterns (Rauch&Šimůnek, 2005) – action rules (Rauch&Šimůnek, 2009), conditional histograms of single attribute (Hájek et al., 2010), conditional frequencies of two multi-categorical attributes (Lin et al., 2005), decision- and exploration-trees (Berka, 2011), clusters or even for pairs of patterns trying to compare two subsets of original data (so called set-difference rules). All types of patterns use a rich syntax of so called Derived Boolean Attributes – automatically generated conjunctions and disjunctions of Basic Boolean Attributes. Basic Boolean Attribute is an expression A(α) where A is an attribute and α is a subset of its possible values, again automatically generated. An Basic Boolean attribute A(α) is true in a row of analyzed data matrix if the value of A in this row belongs to α. – for details see (Rauch&Šimůnek, 2005). This is an important feature which distinguish the LISp-Miner from most of other systems where only expressions of A(α) are allowed, where α is one of possible values of A.

Highly optimized algorithms allow for mining of these automatically constructed complex patterns in a reasonable time. Parallel processing of tasks is available through distributed grid or cloud (Šimůnek&Tammisto, 2010). Achieved theoretical results based on observational calculi and logic of association rules namely, for details see (Rauch, 2010), were subsequently implemented and used to filter-out already known facts from the found patterns (Rauch&Šimůnek, 2011).

Syntactical richness, together with LISp-Miner data preprocessing features allow for wide range of interesting patterns to be automatically found in analyzed data. We would like to mention, that LISp-Miner is a closed system from point of view of implemented objects, analytical procedures or operators. This approach makes “tight” code and thorough optimizations within the whole system possible. The LMCL proposed here opens these features to everybody to build something upon them on a higher level of abstraction.

3. Lua Scripting Language

First of all, an option to develop an own scripting language was considered. It has become clear that this is a no-way solution due to time and effort it would have been necessary to spend on such an adventure. Moreover, the final result would not be on par with already existing languages and valuable developers capacity would have been blocked in maintenance of just another weird syntax language.
Therefore existing scripting languages with a possibility to be embedded by a C++ application (such as LISp-Miner) were considered to base the LM Control Language syntax on them. Two have emerged as run-off competitors – Lua (Ierusalimschy et al., 1996) and JavaScript (based on the ECMA-262 standard, see (Ecma), respectively its Google Chrome V8 Engine implementation (V8).

Finally, the Lua was chosen based on history of its development, more traditional syntax (with probably steeper learning curve and understandability of code), easy installation of development platform, more straightforward embedding of LISp-Miner functionality into script syntax and last but not least on learning materials provided by the Lua community.

Lua (according to its official pages) is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection. For more detail see http://www.lua.org.

Moreover, Lua is free and compact. Its language is widely used, was primary developed for embedding in different types of applications and it is supported by a large community of developers. Its syntax is relatively simple (Ierusalimschy et al., 2006), so script-authors could concentrate on theirs algorithms implementation instead.

Lua has been chosen even despite of unavailability of objects in it (it is based on the pure C not C++), compared to JavaScript natural integration of objects and classes. This drawback was overcome with concept of tables and meta-tables in Lua (Ierusalimschy, 2013). Tables and meta-tables are a neatly way of integration of objects and classes into scripts without any complications to its syntax. This is also demonstrated in examples and in recommendations found on Lua community pages.

4. LISp-Miner Control Language

The purpose of the LISp-Miner Control Language (LMCL) is to allow for calling of LISp-Miner internal functions and accessing user's meta-data in an automated manner. The main goal is to provide a script-like mean to import data, to preprocess them, to formulate reasonable analytical tasks, to process those tasks and finally to digest results (found patterns) and to report only the interesting ones to the user. In this sense, it is a necessary prerequisite for the automation of data mining process in realm of the EverMiner.

The basic concept of the LM Control Language integration into the LISp-Miner system is in Fig. 1.
There is a **LM Control Language script** (which could further include several other scripts and libraries) on the input. This script is executed by the **LM Exec** module (see later) using the Lua language interpreter (Ierusalimschy et al., 1996), of the 5.2 version currently.

The main **LiSp-Miner Core** object classes and high-level functions were embedded into the Lua interpreter, so were calls to the main task processing modules **LM ProcPooler** and **LM GridPooler**. Therefore the **LMCL scripts** have access to the **LiSp-Miner** internal object model to manipulate with meta-data structures (like discretized attributes, analytical tasks settings or found patterns) and to call functions (like to automatically create equi-frequent discretize bins for values in a given database column). Specialized modules for processing analytical tasks could be called from within scripts with option to choose between the parallel processing on multiple cores of the local computer (**LM ProcPooler**) or distributed parallel processing on computer grid or cloud (**LM GridPooler**).

The script execution is logged, so the whole history of execution is available. The log is simultaneously displayed in the **LM Exec** window to interactively inform users about the progress of the execution (including possible warnings or errors). He or she could pause or stop the execution as necessary.

There were two main problems identified design phase – how to allow the Lua script syntax to use the LiSp-Miner objects and functions (**Lua to LM Exec binding**) and how to expose the LiSp-Miner to the **LM Exec** module (**LM Exec to LiSp-Miner Core binding**).

Though both the problems are interconnected, they were solved separately due to their nature and different levels of familiarity and therefore self-confidence to find out a solution on the LiSp-Miner part (seventeen years of in-house development) and on the part of Lua (initially, with only limited knowledge and practically no experience with it) – see subsection 4.2.

### 4.1 Language Syntax

**LMCL** follows Lua naming conventions as far as they were identified in Lua examples. Names of variables, namespaces, functions, methods and named-function-parameters start with lower letters. Only the names of classes, theirs properties and predefined global constants start with capital letters. Names compounded from two or more words follow the “camel” convention of starting capital letters for the second and every other word.

There is an example of **LMCL** script syntax in Fig. 2. An array of all database tables in analyzed data database is retrieved (line 28) and individual tables are iterated using a **for-loop** (line 31). Each table is initialized (line 36) and presence of primary-key is checked and updated if necessary (lines 38 to 47). Data caching is enabled also to speed-up future analytical task processing (line 54). There are also examples of user-defined messages to be included in the execution log (lines 24, 33 and 50–51).
LMCL implementation follows Lua convention of namespaces. All the LISp-Miner related functions and classes are placed in the lm namespace. It is moreover subdivided into additional namespaces as is shown in Fig. 3.

There is lm.database namespace containing the analyzed data related functions, namely the importTXT function to import text/CSV files. There is the lm.metabase namespace for functions to create and associate of a meta-data database (storing data preprocessing parameters, analytical tasks settings and found patterns). Classes and functions for data exploration (e.g. to collect information about database tables and columns of analyzed data) are in the lm.explore namespace. Preprocessing and data transformations related classes and functions are in the lm.prepro namespace. The most important classes are categorized Attributes and its Categories. Finally, there is lm.tasks namespace for classes and function for analytical tasks settings and browsing results (found patterns). This namespace is divided into two sub-namespaces for more clarity – lm.tasks.settings and lm.tasks.results. Respective classes for these two sub-namespaces (for the 4ft-Miner procedure) are shown in Fig. 4.

There are unified name prefixes of set- and get- for functions reading and setting properties of objects. The setter function has a single unnamed parameter of the same type as the property it is modifying. Of the same type is the return of a corresponding getter function.

```lua
dataColumn.getName() -- returns column name as string
attribute.setName("age") -- changes name of the attribute
```

Unnamed parameters for function calls are used only for setter functions and for few other simple functions (mainly the logging functions from the lm namespace). In all other cases the named parameters are used for enhancing readability and clarity of the script codes.

Fig. 2 – LMCL script syntax example
A simplified Hungarian notation is used to identify four basic types of named parameters.

- **nTaskSubTypeCode**: an integer number expected
- **dParamP**: a floating-point number (double precision) is expected
- **bForceRunFinished**: a Boolean value is expected
- **pParentGroup**: a reference to an LMObject is expected

A special function name prefix `prepare-` is used to identify functions or methods returning an array of objects as a table data-type. Some `prepare-` functions allow passing an optional parameter to filter only some objects from the internal array (e.g. only tasks of a given type).

```text
dataColumnArray= dbtable:prepareDataColumnArray()
taskArray= lm.tasks:prepareTaskArray(
    nTaskSubTypeCode= lm.codes.TaskSubType.CFMiner)
```
Another prefix of find- is used to identify functions looking-up an object in an internal array by one of its unique properties (usually by the ID or Name object properties). The identifier is passed as a named parameter.

```ruby
dataColumn= dataTable:findDataColumn( name= "District")
task= lm.tasks:findTask( nID= 7)
```

There is another special namespace dedicated to list code-tables and constants for identifying LISp-Miner related types. It is automatically generated from the LISp-Miner Core source codes – Fig. 5 and LMCL Reference Pages section below.

<table>
<thead>
<tr>
<th>TestingType</th>
<th>Key</th>
<th>Name</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TrainingSet</td>
<td>Use training set</td>
<td>Use same records as for training</td>
</tr>
<tr>
<td></td>
<td>CrossValidation</td>
<td>Cross-validation</td>
<td>Cross-validating each of n-folds</td>
</tr>
<tr>
<td></td>
<td>RandomSplit</td>
<td>Random split</td>
<td>Random split of available data in given ration</td>
</tr>
</tbody>
</table>

Fig. 5 – Example of generated codes for values of “TestingType”
4.2 Lua to LISp-Miner Binding

LISp-Miner objects are available within Lua scripts as instances of lua userdata objects with metatables attached to keep necessary information about each class properties and methods.

There is an abstract class CLuaBind on the top of the class hierarchy to implement basic Lua to LISp-Miner binding functionality (see Fig. 6). A general class CLuaLMMnamespace serves as an ancestor for all the implemented namespaces. There are two general classes to wrap LISp-Miner Core classes and to bind theirs properties and methods – CLuaLMWrap and CLuaLMWrapName. The distinction is in that all the descendants from the CLuaLMWrapName class have Name and Note properties. All the objects derived from the CLuaLMWrap class have the ID property for a unique identification (e.g. in the find-functions).

![Diagram of Lua to LM Exec Binding classes hierarchy](image)

Fig. 6 – Lua to LM Exec Binding classes hierarchy diagram

The other side of the binding (the above-mentioned LM Exec to LISp-Miner Core binding) was implemented by the ordinary pointers to the LM Object Layer containing the real “production” LISp-Miner classes. Thus, the CLuaBind objects are pure proxies to represent LISp-Miner objects in Lua scripts and are lacking any functionality. Any request to access some object properties or functions originated in the script is handled by a proxy object and propagated to its corresponding real LISp-Miner Core object in the LM Object Layer, see also the LMCL conceptual schema in Fig. 1.

A special care has to be given to a proper memory management of both CLuaBind-derived proxy objects and LM Object Layer objects. Lua implements a garbage collector to manage dynamic memory allocations and de-allocations. LM Exec and LISp-Miner Core libraries are but implemented in C++ and use the standard unmanaged way of allocating and releasing dynamic memory.

There were three situations identified, where problems could occur:
• multiple requests from the Lua code for the same LM Object Layer object should return the same proxy object to allow for proper equality test results;
• a Lua reference has gone out scope, but the corresponding impostor object stays in memory till the Lua garbage collector has time to release it;
• living proxy objects has to be notified when its corresponding real object is deleted internally in the LM Object Layer.

The first situation has been solved by a reverse HashMap with pointers to real objects as keys and pointers to proxy objects as values. So the same proxy object is returned if already exists, otherwise it is created first and added into the HashMap for future look-up.

In respect to the second situation, no proxy objects are de-allocated and wait till they are released by the Lua garbage collector. Moreover, an internal counter was implemented to remember number of passed Lua script references. Proxy object becomes “dead” only after this counter returns back to zero.

There was a call-back function implemented into LISp-Miner Core objects destructor in the LM Object Layer (more precisely, to the destructor of the LMOBject – the topmost class of hierarchy). After the call-back registration from the LM Exec module is established, it allows for an arbitrary code to be called during internal de-allocation of any LISp-Miner Core object. So a proxy object is notified that its corresponding real object no longer exists and the link is destroyed. Nevertheless, the proxy object has to stay alive to catch all possible calls from the still existing Lua script references. If such a call occurs, a run-time error is reported and script execution is aborted.

An example of situation where this behavior takes effect is deleting of task settings represented by an object of the Task class. Not only the reference to task settings itself is invalid from this point, so are all the possible references to objects the task settings is composed from – partial cedents, literals, quantifiers and so on (see Fig. 4 above). They all were deleted internally by the LM Object Layer together with the given task settings. Therefore it is necessary to notify all theirs existing proxy objects as described above.

5. LM Exec Module

The LM Exec module executes LMCL scripts. It is freely available also on the LISp-Miner download page, but has to be downloaded separately (http://lispminer.vse.cz/files/exe/LM.Exec.zip) and extracted into the LISp-Miner root directory. It has a simple interface (see Fig. 7) to open a script, execute it and to see history of progress.

 Scripts could be opened using the Open button and executed by pressing the Start button. If an error occurs during execution, it is possible to switch to a text editor, change script at given position and save changes. The updated version of script could be re-started without a need to open the script again or even to restart the LM Exec module. Script execution could be cancelled or paused.

The center space of the LM Exec windows occupies a text log describing execution of the script. All errors and warnings are logged and displayed during the script execution. So are main Lua-script function calls and selected time-consuming LISp-Miner operations (depending on the log verbosity level).

If an error is encountered, the script is aborted and a description of the error appears in the log together with the line-number on which this error had occurred. User-defined debug-messages could be logged also using the log function (or its variants) from the lm namespace.

The whole log could be copied into clipboard, but is also automatically saved into a file (with user-predefined name).
6. Programmers Reference

LMCL is meant to be used by researchers to automate data mining process and by students in advanced courses. There must be a proper documentation for the LM Control Language to be understandable for both groups of users.

The main reason for so much time was spent in looking-up a good solution for a proper Lua to LISp-Miner binding was necessity of a complete reference manual to be generated automatically from multiple source codes of the LISp-Miner Core and the LM Exec module especially. Thus, any change to the implementation of Lua to LM Exec binding (e.g. a change to name or type of a function-parameter to be called from LMCL scripts) does automatically update the corresponding description of reference pages. Similarly, a list of LISp-Miner code-tables items included into scripts is updated after a code-table is added or values are modified. Automatic updates are the only viable solution to avoid future de-synchronization of the reference pages and the actual code implementation.

Although already available solutions were studied (namely the LuaDoc (LuaDoc)) and an inspiration was taken from the JavaDoc (JavaDoc), they are not intended for documenting Lua embedding application implementation. Decision finally taken was to implement an own implementation of reference pages generation to tailor its needs to LMCL and to contain not only the Lua-script binding related part, but also other descriptions taken directly from the LISp-Miner Core source codes (namely the code-tables items). An example of the chosen solution is in Fig. 8.

There are prepared C++ macros for several types of function depending on their input and output parameters structure. There is a function queryTaskGenerationStatusAll returning single long integer value registered by the METHOD_REF_RETURN_LONG macro. Similarly, the METHOD_REF_GETSET_STR macro is used to register setter and getter methods to manipulate with string value of the Note property of any given object derived from the CLMLuaLMWrapName class.
The above presented examples demonstrate a single-place registration of a function both for the **LM Exec** implementation of the *Lua-script* binding and for the reference pages generation. Therefore the implementation of embedded function calls from the *Lua-script* exactly matches its description in the reference pages. An example of automatically generated reference page descriptions is in Fig. 9.

![Class Task](image)

**Fig. 9 – Example of automatically generated description for class Task**

An example of detailed descriptions for functions is in Fig. 10.

```
queryTaskGenerationStatusAll() : long
    Re-reads tasks states from metabase and returns number of tasks for which theirs state has changed from `isTaskGenerationStatusInProcess` to `isTaskGenerationStatusFinished`
    RETURNS
    • long

runAllAndWaitForResults()
    Starts all tasks (with not `isTaskGenerationStatusInProcess`) in this metabase and waits till all of them have finished
    PARAMETERS
    • luaTable - to store named parameters
    OPTIONAL NAMED PARAMETERS
    • nTargetPlatform : long - `TargetPlatform` code
    • bForceRunFinished : boolean - if true, forces all tasks to be run again even if they are already finished
    RETURNS
    • long - number of launched tasks
```

**Fig. 10 – Example of automatically created detailed descriptions for functions**

There are special-purpose comments directly in the source code for more complex functions with several input obligatory and optional parameters (like is the `runAllAndWaitForResults` function in Fig. 10). See the first line in Fig. 11 starting with `@@` prefix. These hints are parsed directly from the *C++* source code during the generation of reference pages.
7. Results and Examples

The LISp-Miner Control Language was successfully implemented using Lua script interpreter library of version 5.2. Any user-defined script could be now executed by the LM Exec module (interactively or as a batch in background). The Lua interpreter used is really lightweight and proved to be fast, so far tested with medium-sized scripts (up to thousands of code-lines). Script parsing and execution overhead costs are insignificant compared to data mining task solution times or to data transfers from database. Performance of LMCL scripts therefore depends solely on ability of the LISp-Miner system modules to compute data mining tasks. It has been proved already (Rauch & Šimůnek, 2005) that the algorithms and optimizations techniques implemented in the LISp-Miner system lead to solution times linearly dependant on number of rows (objects) in analyzed data.

There are several examples included in the LM Exec installation package. They range from the obligatory “Hello, world!” example to a more complex one called EverMinerSimple.

Fig. 11 – Example of in-code description of an input parameter

Fig. 12 – EverMinerSimple algorithm overview

The EverMinerSimple demo is a really simplified version of the EverMiner concept, rather a prototype of it. Its only purpose is to proof that the LM Control Language is really able to automate the data mining process.
This prototype solution implements only one iteration of the main phases of data mining process with no new domain knowledge inferred yet. But it already incorporates the inner cycle of fine-tuning tasks parameters to obtain an acceptable number of patterns in results (this number is an input parameter – see below). Only one type of pattern is used for now – 4ft-association rule.

There is a conceptual diagram of EverMinerSimple steps in Fig. 12.

Few user-defined parameters provide all the necessary input to the whole process. The first group of parameters defines the text file with analyzed data to import, destinations to store the created database with analyzed data and the database with meta-data. Finally, it defines the ODBC DataSourceName to identify this data + meta-data pair within the operating system.

The second group of parameters provides a bit of domain knowledge – groups of attributes the analyzed data columns should be grouped into. This information is important for analytical tasks construction, where all the possible combinations of groups of attributes in antecedents and succedents of patterns to be mined are created.

<table>
<thead>
<tr>
<th>EverMinerSimple Demo Analytical Report</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input parameters</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Data Exploration</strong></td>
</tr>
<tr>
<td><strong>Hotel</strong></td>
</tr>
<tr>
<td>- Age: Long integer</td>
</tr>
<tr>
<td>- Nationality: String</td>
</tr>
<tr>
<td>- ...</td>
</tr>
<tr>
<td>Derived columns:</td>
</tr>
<tr>
<td>- VisitFrom.DayOfWeek: Long integer</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Data Preprocessing</strong></td>
</tr>
<tr>
<td><strong>Guest</strong></td>
</tr>
<tr>
<td>Age: c18;23, c23;31, c31;38, c38;43, c43;49, c49;54, c54;59, c59;65, c65;70, c70;82</td>
</tr>
<tr>
<td>Nationality: AT, CZ, GE, PL, SK</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Tasks**

**Calendar, Guest, Visit -> Weather (08)**

Task finished but an acceptable number of patterns has not been achieved
Could not reduce the search space
Number of iterations: 8
Found patterns: 36
The most interesting ones:

- **Nationality**(SK) & **Persons**(2) & **VisitType**(Tourist) >\< Weather(Cloudy)
- **Nights**(7) & **Persons**(1) & **VisitType**(Tourist) >\< Weather(Sunny)

<table>
<thead>
<tr>
<th>Fig. 13 – Example of an automatically created EverMinerSimple analytical report (shortened)</th>
</tr>
</thead>
</table>
| The most interesting input parameters are the minimal and maximal number of patterns to mine, regardless of combination of groups this particular task is concerning. There are several ways how to reduce (or enlarge) task search space to influence the number of found patterns, but they are out of scope of this paper. Nevertheless, they are exploited in the Parallel processing phase to ensure the number of found patterns is within given range. (A check for maximal number of iteration is implemented to avoid a never-ending cycle.)

An important feature is that no task settings are changed after it was processed. Every time a change is necessary to task settings, its exact clone is created first and the desired change is made to this cloned task settings instead. Therefore, a complete history of task settings evolution, together with a number and an exact form of found patterns is preserved in the meta-data database. It could be used later for investigations of steps and decisions taken during automated data mining process – either for debug purposes or to help with a proper interpretation of found patterns.
The last input parameter specifies the path and name of file for the analytical report to be written into. In this example, a HTML report is prepared and opened in operating system default web browser. A manually shortened version is in Fig. 13.

In this particular case, amount of found patterns for an analytical task Calendar, Guest, Visit implying Weather type was not reduced to number requested by input parameters (even after reaching an arbitrary limit of eight iterations of task settings changes). This failure is mentioned in the report and only two of found patterns are shown. Please remember also, that EverMinerSimple example is just a prototype with no ambition to provide real results yet.

8. Other Approaches

The most popular programming language for data mining is R with 52.5 % of respondents reporting its usage for a data mining related task within past 12 months, according to 2012 poll results presented on KDnuggets.com (KDnuggets Poll, 2012). It is followed by Python (36.1 %), a general scripting language, and by SQL (32.1 %), a structured query language designed for relational database data retrieval and manipulation. Java, a general programming language, closes the top-five list with 21.2 %.

R is a programming language and (environment) developed primary for statistical computing, see http://www.r-project.org/. It is an implementation of S language (Becker&Chambers, 1984). R is a GNU project and is supported by a large community, including the R Foundation. Its wide focus is both its advantage and disadvantage. LISp-Miner and LMCL is not so capable and has not so many different modules and add-ons but its more narrow focus and compactness makes research in the area of data mining automation more feasible.

An interesting fact is a wide adoption of general programming languages for data mining analysis. We suppose that these general languages are involved in data import and data preprocessing phases where they general (or system oriented) abilities could be exploited. It is very laborious to implement any specialized data mining technique (like clustering or decision trees) in those languages from scratch.

It has much more sense for data mining automation to step above to a higher level of abstraction and to use a data mining system and its features as building blocks (as LMCL allows).

Another commonly used method of describing steps of data mining and of data flowing from one step to another is visualization of boxes and links among them, popularized first by the Clementine system (later SPSS Modeller and now IBM SPSS Modeller, see http://www.spss.com/clementine). A large number of data mining tools re-implemented this graphical approach, e.g. Ferda (Ralbovský, 2007), including the RapidMiner (http://www.rapidminer.com) system.

There is a clear benefit of a graphical representation of any algorithm regardless of its nature – there were already made several attempts to replace traditional programming based on writing source-texts by a mouse-driven positioning of graphical boxes on desktop to visually describe underlying algorithm, usually called "visual programming languages" (no connection to Microsoft’s Visual XX family of programming language products). An example of visual programming language is Simulink (http://www.mathworks.com/products/simulink). The most important advantage is understandability and clarity of graphs with not too many nodes and links among them. Anyone could see then at glance the whole algorithm and could visually trace its execution. This could be suitable for beginners because software learning curve is steep and for relatively small problems.

Unfortunately, this approach is not scalable. If number of nodes exceeds relatively low threshold of 10 nodes (or there is too many links among nodes) human ability to mentally absorb information included in the graph rapidly decreases, (Miller, 1956). There are practical problems also with choosing a suitable position for each of boxes (mainly to minimize links overlapping). Finally, graphs could grow very large and problems emerge with parts of graph being outside the working space on screen or boxes on printed graph being too small when scaled to fit to size of paper.

Despite of a higher effort and a longer time that must be spent initially to become familiar with syntax of chosen scripting language, the traditional approach of written source-texts makes possible to implement an algorithm regardless of its complexity and number of steps it includes.
9. Summary

The LISp-Miner Control Language is a necessary prerequisite for data mining automation in the EverMiner concept. But it could serve for other purposes as well. Firstly, it will be used to prepare pre-release testing scripts of a new version of the LISp-Miner system to prove that no bugs were unintentionally introduced by adding a new functionality. Secondly, the scripting language will become a part of advanced teaching courses to allow students to better understanding of implementation details of data mining algorithms. They could possibly implement some add-ons or new features to existing LISp-Miner functionality.

References


Hájek, P., 1974. Automatic listing of important observational statements I-III. Kybernetika 9, 1973, s. 187-205, s. 251-271 a Kybernetika, 10, 1974, s. 95-124


Lua scripting language home page. (cit. 2014-01-08) Available at: http://www.lua.org

LuaDoc project home page. (cit. 2014-01-08) Available at: http://keplerproject.github.io/luadoc


Yang, Qiang; Wu, Xindong. 2006. 10 Challenging Problems in Data Mining Research. *International Journal of Information Technology & Decision Making*. Vol. 5, No. 4: 597–604

**JEL Classification:** C60, D83