—realKD—
Java Library for Real Knowledge Discovery

Document version 1
(referring to library version 0.1)

Mario Boley
University of Bonn/Fraunhofer IAIS
Schloss Birlinghoven, Sankt Augustin

December 3, 2014

1 Introduction

This document is a developer's guide to realKD—a free open-source Java library that has been designed to help real users discovering real knowledge from real data. As such it has a strong focus on

a) a detailed data model that allows the specification of a lot of domain-dependent semantics,

b) a generic pattern model intended to capture meaningful bits of knowledge, and

c) a model to describe algorithms and their parameters in a way that makes it easy to build user-friendly applications on top of it.

In particular realKD can be used to discover associations (itemset patterns), exceptional model patterns (subgroups), and subspace outliers from tabular data. While the main purpose of realKD is to be used by other Java applications that aim to enable knowledge discovery functionality to their users, it can also be used stand-alone from the command line.

This document gives a brief introduction to the comncepts and interfaces of the library. It is targeted towards developers who want to use components of realKD to build a data analysis tool on top of it as well as those that want to extend its functionality—either to be included in a future version of this library or outside of it within other third party software.

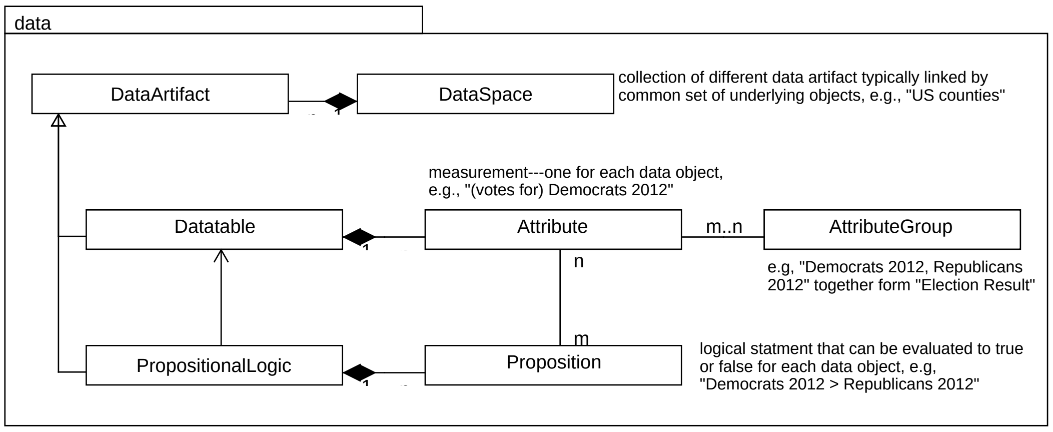
data
DataArtifact
DataSpace
collection of different data artifact typically linked by common set of underlying objects, e.g., "US counties"
measurement---one for each data object, e.g., "(votes for) Democrats 2012"
Datatable
Attribute
m..n
AttributeGroup
n
e.g, "Democrats 2012, Republicans 2012" together form "Election Result"
m
PropositionalLogic
Proposition
logical statment that can be evaluated to true or false for each data object, e.g, "Democrats 2012 > Republicans 2012"

Figure 1: Classes of the tabular data model.

2 Data Model

The basic form of input data in realKD is tabular. That is, it is given in the logical form of a table with a fixed number of rows and columns represented by the main class `Datatable`. A table contains a number of data records (rows), each of which is described by a fixed set of attributes (represented by the class `Attribute`). These attributes can be categorical (they assign to each data record one of a typically small number of fixed categories), ordinal (same as categorical but in addition there is a partial order over the categories), or metric (assign to each data record a double value). Values can be also be explicitly marked as missing. Additionally, there is attribute metadata in the form of attribute groups, which stand in an n-to-m relationship with the attributes. See Fig. 1 for a summary of the classes of the data model.

Depending on the type of attribute, different statistics can be computed: category frequencies for categoric attributes, median for ordinal attributes, and additionally mean, standard deviation, and so on for metric attributes. This is reflected in the interfaces of the different attribute sub-types.

3 Patterns

Patterns correspond to interesting observations about a dataset that can be broken down into a) a syntactical statement and b) a rationale supporting the truth of this statement on a specific dataset of interest. The most abstract part of the pattern package is the sub-package `patterns.propositions`, which defines the classes to express boolean statements about individual rows of a datatable. Many individual statements (`Proposition`) are aggregated within a `PropositionStore`.

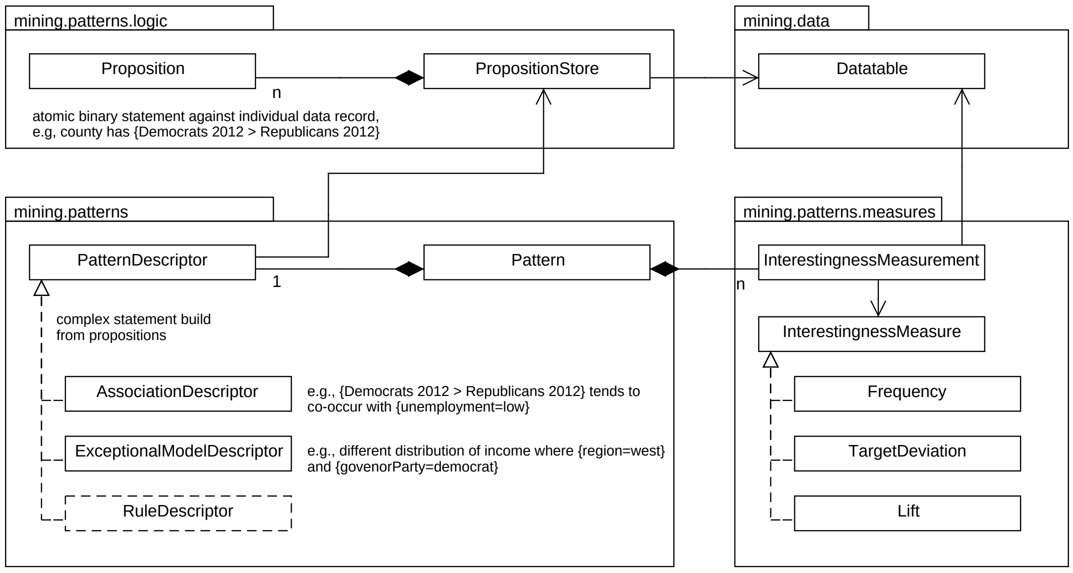
mining.patterns.logic
Proposition
PropositionStore
n
atomic binary statement against individual data record,
e.g, county has {Democrats 2012 > Republicans 2012}
mining.data
Datatable
mining.patterns
PatternDescriptor
Pattern
1
complex statement build
from propositions
AssociationDescriptor
e.g., {Democrats 2012 > Republicans 2012} tends to
co-occur with {unemployment=low}
ExceptionalModelDescriptor
e.g., different distribution of income where {region=west}
and {govenorParty=democrat}
RuleDescriptor
mining.patterns.measures
InterestingnessMeasurement
n
InterestingnessMeasure
Frequency
TargetDeviation
Lift

Figure 2: The main classes of the patterns package.

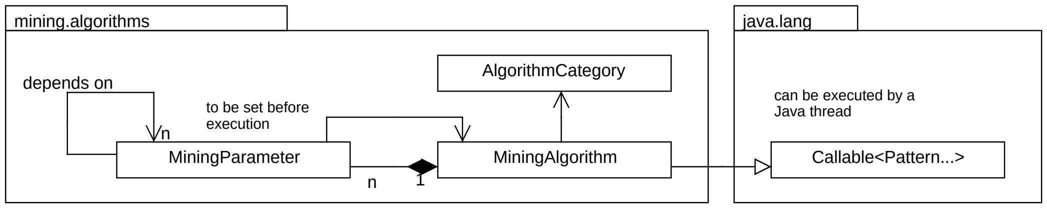

Figure 3: Main types of the algorithms package.

4 Algorithms

Algorithms for pattern discovery are represented by the type `MiningAlgorithm`, which is the main element of the package `mining.algorithms`. The central functionality of objects of this type is that they can be *called* in order to produce patterns; and typically this should be done asynchronously. Hence, the type is a `java.lang.Callable` linking it to standard Java mechanisms. Additionally, mining algorithms are organized in categories and aggregate a number of parameters (`MiningParameter`), which have to be set to valid values prior to the execution of the `call()`-method (see Fig. 3 for an ontological summary). Category and parameters provide the means to operate algorithms in a controlled and documented way to user interfaces and the one-click mining classes. Note that the contracts of the algorithm and the parameter java interfaces are coupled.

4.1 Algorithm Interfaces

Moving to the interface level (Fig. 4) we can see that the above mentioned functionality of algorithms is actually split across different interfaces corresponding to the different functionality aspects. The most basic interface is `MiningAlgorithm`, which in addition to the call-method only provides a boolean method `isRunning()`. This method provides synchronized access to a flag that indicates if `call()` currently is executed in some thread. The contract of this level is that `call()` is only executed at most in one thread at a time, which allows this thread to have an exclusive writing access to almost all of the state of the algorithm object. That is, all methods that modify the state of the object (except parts that are marked as an exception in extensions of the interface) have to throw an `IllegalStateException` if they are invoked while `isRunning()==true`—including `call()` itself.

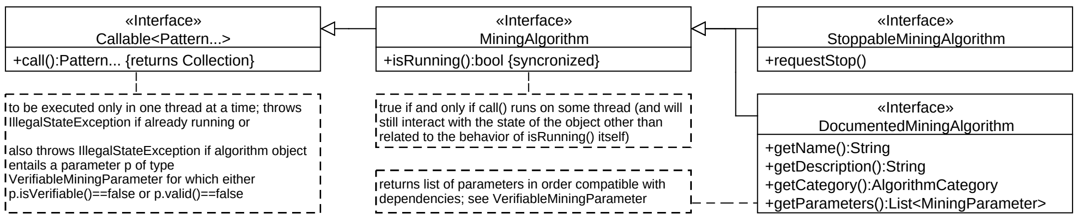
«Interface»
Callable<Pattern...>
+call():Pattern... {returns Collection}
to be executed only in one thread at a time; throws IllegalStateException if already running or
also throws IllegalStateException if algorithm object entails a parameter p of type VerifiableMiningParameter for which either p.isVerifiable()==false or p.valid()==false
«Interface»
MiningAlgorithm
+isRunning():bool {syncronized}
true if and only if call() runs on some thread (and will still interact with the state of the object other than related to the behavior of isRunning() itself)
returns list of parameters in order compatible with dependencies; see VerifiableMiningParameter
«Interface»
StoppableMiningAlgorithm
+requestStop()
«Interface»
DocumentedMiningAlgorithm
+getName():String
+getDescription():String
+getCategory():AlgorithmCategory
+getParameters():List<MiningParameter>

Figure 4: Interfaces of mining algorihms.

An optional extension of this interface is `StoppableMiningAlgorithm`, which provides the additional void-method `requestStop()`. By calling this method a client can send to an algorithm object the request to stop an execution of `call()` in another thread. Hence, this method is a declared exception to the above mentioned contract for state-modifying methods in that it does not throw an exception when the algorithm is running and is allowed to modify a designated member variable in order to communicate the stop request to the thread executing `call()`. This interface should only be implemented when also a reasonable handling of stopping requests is provided. In one-click mining we assume that this is the case for all algorithms.

The other optional extension of `MiningAlgorithm` is `DocumentedMiningAlgorithm`. This interface (along with the interfaces regarding mining parameters) bundles all the functionality needed to configure an upcoming run of the `call()` method in an informed way and to display additional meta-information about the algorithm within a graphical user interface. All non parameter-related meta-information is conveyed by the methods `getName()`, `getDescription()`, and `getCategory()`, respectively, where the first two informations are provided as user-readable strings and the latter takes on values of a small set of algorithmic categories given in the enum `AlgorithmCategory`. These categories are related to the type of patterns that the algorithm produces, i.e., at the moment they are `ASSOCIATION_ALGORITHM` and `EMM_ALGORITHM`. Parameter control and parameter-related meta-information is provided by the method `getMiningParameters`, which returns a list of objects of type `MiningParameter`. This interface is documented in the next subsection. Note, however, that the contracts of these two parts are coupled.

4.2 Parameter Interfaces

The interface `MiningParameter` and its extensions are generics with a type-parameter `T` which represent the type of values that the parameter can take (see Fig. 5). On the basic level, the interface allows to retrieve parameter-specific meta-information (methods `getName()`, `getDescription()`, and `getType()`), to retrieve the current value (method `getCurrentValue()`), and to re-set the value either by passing a `T` (`set(T)`) or by a string representation (`setByString(String)`). The last option is provided, e.g., for non Java-UIs like Web-UIs, which typically communicate with a browser via strings. Following the general contract of `MiningAlgorithm`, the two setter-methods have to throw an `IllegalStateException` if `isRunning()==true` for the associated algorithm object, which exists for all `MiningParameter` objects. Also note that the current value of a parameter can be undefined corresponding to `getCurrentValue==null`.

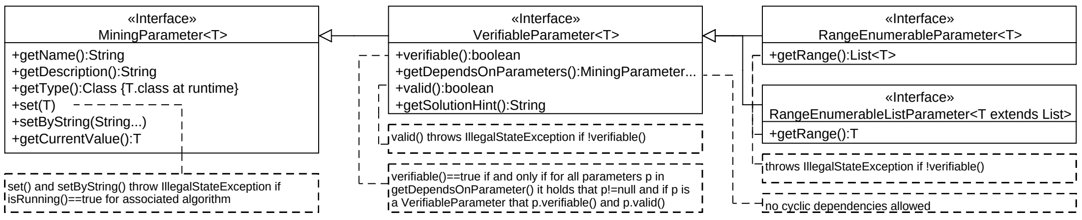
«Interface»
MiningParameter<T>
+getName():String
+getDescription():String
+getType():Class {T.class at runtime}
+set(T)
+setByString(String...)
+getCurrentValue():T
set() and setByString() throw IllegalStateException if isRunning()==true for associated algorithm
«Interface»
VerifiableParameter<T>
+verifiable():boolean
+getDependsOnParameters():MiningParameter...
+valid():boolean
+getSolutionHint():String
valid() throws IllegalStateException if !verifiable()
verifiable()==true if and only if for all parameters p in getDependsOnParameter() it holds that p!=null and if p is a VerifiableParameter that p.verifiable() and p.valid()
«Interface»
RangeEnumerableParameter<T>
+getRange():List<T>
«Interface»
RangeEnumerableListParameter<T extends List>
+getRange():T
throws IllegalStateException if !verifiable()
no cyclic dependencies allowed

Figure 5: Interfaces and implementing classes that represent the mining algorithms.

While `MiningParameter` essentially provides the functionality of a generic boxed value that respects the contract of `MiningAlgorithm`, it does not provide much support to clients for setting correct values (other than by providing parameter name and description). For this purpose, additional functionality is brought in by the interface `VerifiableMiningParameter`. The core method of this interface is the boolean-valued method `valid()` that checks the validity of the currently set value (as it can be retrieved by `getCurrentValue()`). In some use cases, however, this check can only be performed if already (valid) values have been set for some other parameters. That is, there can be *dependencies* between parameters. This dependency structure is available by the method `getDependsOnParameters()`, which returns a list of other mining parameters. The validity-check can be performed if for all these parameters `p` it holds that

a) `p.getCurrentValue()!=null` and

b) if `p instanceOf Verifiable` then `p.valid()==true`.

Condition b) implies that the validity-check can be performed for all such `p` recursively referring to their dependencies (hence, no cylcic dependencies are allowed). As a shortcut, whether the validity-check can be performed for a parameter can be checked by calling the boolean method `verifiable()`. With these concepts the contract of `MiningAlgorithm` is also extended: `call()` must throw an `IllegalStateException` if an algorithm object entails one `VerifiableParameter` which is either not verifiable or not valid.

Finally, there is another two-fold extension of `VerifiableMiningParameter` for cases in which a parameter value cannot only be checked for validity in an oracle-fashion, but an explicit list of possible values can be given. For this there are two cases depending on the nature of the type parameter `T` of the mining parameter:

a) the parameter can take on a single value from a given list of elements of type `T` or

b) `T` is a list-type itself, say `T=List<U>`; such that the parameter can take on a sub-list of a specified list of elements of type `U`.

For case a) the parameter can implement the interface `RangeEnumerableParameter<T>`, which provides `getRange()` with return type `List<T>`. For case b) there is the interface `RangeEnumerableListParameter<T extends List>`, which provides `getRange()` with return type `T` itself. In both cases, the range of valid values can depend on other parameters. Hence, `getRange()` has the same behavior as `valid()` and throws an `IllegalStateException` in both extensions if the parameter is not verifiable. Also, as an additional consistency invariant a parameter value is valid if that value is an element from the provided list (respectively a sub-list of the provided list in case b)). That is, for an instance of `RangeEnumerableParameter` with `verifiable()==true`, it holds that `valid()==getRange().contains(getCurrentValue())` (with a similar condition for the other case using a sublist check).