Debellor: A Data Mining Platform with Stream Architecture

Marcin Wojnarski

Warsaw University, Faculty of Mathematics, Informatics and Mechanics
ul. Banacha 2, 02-097 Warszawa, Poland
mwojnars@ns.onet.pl

Abstract. This paper introduces Debellor (www.debellor.org) – an open source extensible data mining platform with stream-based architecture, where all data transfers between elementary algorithms take the form of a stream of samples. Data streaming enables implementation of *scalable* algorithms, which can efficiently process large volumes of data, exceeding available memory. This is very important for data mining research and applications, since the most challenging data mining tasks involve voluminous data, either produced by a data source or generated at some intermediate stage of a complex data processing network.

Advantages of data streaming are illustrated by experiments with clustering time series. The experimental results show that even for moderate-size data sets streaming is indispensable for successful execution of algorithms, otherwise the algorithms run hundreds times slower or just crash due to memory shortage.

Stream architecture is particularly useful in such application domains as time series analysis, image recognition or mining data streams. It is also the only efficient architecture for implementation of online algorithms.

The algorithms currently available on Debellor platform include all classifiers from Rseslib and Weka libraries and all filters from Weka.

Keywords: Pipeline, Online Algorithms, Software Environment, Library.

1 Introduction

In the fields of data mining and machine learning, there is frequently a need to process large volumes of data, too big to fit in memory. This is particularly the case in some application domains, like computer vision or mining data streams [1,2], where input data are usually voluminous. But even in other domains, where input data are small, they can abruptly expand at an intermediate stage of processing, e.g. due to extraction of windows from a time series or an image [3,4]. Most of ordinary algorithms are not suitable for such tasks, because they try to keep all data in memory. Instead, special algorithms are necessary, which make efficient use of memory. Such algorithms will be called *scalable*.

J.F. Peters et al. (Eds.): Transactions on Rough Sets IX, LNCS 5390, pp. 405–427, 2008.

Another feature of data mining algorithms – besides scalability – which is very desired nowadays is *interoperability*, i.e. a capability of the algorithm to be easily connected with other algorithms. This property is more and more important, as basically all newly created data mining systems – whether experimental or end-user solutions – incorporate much more than just one algorithm.

It would be very worthful if algorithms were both scalable and interoperable. Unfortunately, combining these two features is very difficult. Interoperability requires that every algorithm is implemented as a separate module, with clearly defined input and output. Obviously, data mining algorithm must take data as its input, so the data must be fully *materialized* – generated and stored in a data structure – just to invoke the algorithm, no matter what it actually does. And materialization automatically precludes scalability of the algorithm.

In order to provide scalability and interoperability at the same time, algorithms must be implemented in a special software architecture, which do not enforce data materialization. Debellor[1] – the data mining platform introduced in this paper – defines such an architecture, based on the concept of *data streaming*. In Debellor, data are passed between interconnected algorithms sample-by-sample, as a stream of samples, so they can be processed on the fly, without full materialization. The idea of data streaming is inspired by architectures of database management systems, which enable fast query execution on very large data tables.

It should be noted that Debellor is not a library, like e.g., Rseslib[2] [5,6,7] or Weka[3] [8], but a data mining *platform*. Although its distribution contains implementations of a number of algorithms, the primary goal of Debellor is to provide not algorithms themselves, but a common architecture, in which various types of data processing algorithms may be implemented and combined, even if they are created by independent researchers. Debellor can handle a wide range of algorithm types: classifiers, clusterers, data filters, generators etc. Moreover, extendability of data types is provided, so it will be possible to process not only ordinary feature vectors, but also images, text, DNA microarray data etc.

It is worth mentioning that Debellor's modular and stream-oriented architecture will enable easy parallelization of composite data mining algorithms. This aspect will be investigated elsewhere.

Debellor is written in Java and distributed under GNU General Public License. Its current version, Debellor 0.5, is available at www.debellor.org. The algorithms currently available include all classifiers from Rseslib and Weka libraries, all filters from Weka and a reader of ARFF files. There are also several algorithms implemented by Debellor itself, like Train&Test evaluation procedure. The algorithms from Rseslib and Weka, except the ARFF reader, are not scalable – this is enforced by architectures of both libraries.

[1] The name originates from Latin *debello* (to conquer) and *debellator* (conqueror).
[2] http://rsproject.mimuw.edu.pl/
[3] http://www.cs.waikato.ac.nz/ml/weka/

2 Related Work

There is large amount of software that can be used to facilitate implementation of new data mining algorithms. A common choice is to use an environment for numerical calculations: R[4] [9], Matlab[5], Octave[6] [10,11] or Scilab[7] and implement the algorithm in a scripting language defined by the environment. Many data mining and machine learning algorithms are available for each of these environments, usually in a form of external packages, so the environments can be seen as common platforms for different data mining algorithms. However, they do not define common architecture for algorithms, so they do not automatically provide interoperability. Moreover, the scripting languages of these environments have low efficiency, no static typing and only weak support for object-oriented programming, so they are suitable for fast prototyping and running small experiments, but not for implementation of scalable and interoperable algorithms.

Another possible choice is to take a data mining library written in a general-purpose programming language (usually Java) – examples of such libraries are Weka[8] [8], Rseslib[9] [5,6,7], RapidMiner[10] [12] – and try to fit the new algorithm into the architecture of the library. However, these libraries preclude scalability of algorithms, because the whole training data must be materialized in memory before they can be passed to an algorithm.

The concept of data streaming, called also pipelining, has been used in database management systems [13,14,15,16] for efficient query execution. The elementary units capable of processing streams are called *iterators* in [13,14].

The issue of scalability is related to the concept of online algorithms. In machine learning literature [17,18], the term *online* has been used to denote training algorithms which perform updates of the underlying decision model after every single presentation of a sample. The algorithms which update the model only when the whole training set has been presented are called *batch*.

Usually online algorithms can be more memory-efficient than their batch counterparts, because they do not have to store samples for later use. They are also more flexible, e.g., they can be used in incremental learning or allow for the training process to be stopped anytime during scan of the data. This is why extensive research has been done to devise online variants of existing batch algorithms [19,20,21,22,23]. Certainly, online algorithms are the best candidates for implementation in stream architecture. Note, however, that many batch algorithms also do not have to keep all samples in memory and thus can benefit from data streaming. In many cases it is enough to keep only some statistics calculated during scan of the data set, used afterwards to make the final update of the model. For example, standard k-means [17,24,25] algorithm performs batch

[4] http://www.r-project.org
[5] http://www.mathworks.com
[6] http://www.octave.org
[7] http://www.scilab.org
[8] http://www.cs.waikato.ac.nz/ml/weka
[9] http://rsproject.mimuw.edu.pl
[10] http://rapid-i.com

updates of the model, but despite this it can be scalable if implemented in stream architecture, as will be shown in Sect. 5.8.

3 Motivation

3.1 Scalability

Scalable algorithms are indispensable in most of data mining tasks – every time when data become larger than available memory. Even if initially memory seems capacious enough to hold the data, it may appear during experiments that data are larger and memory smaller than expected. There are many reasons for this:

1. Not the whole physical memory is available to the data mining algorithm at a given time. Some part is used by operating system and other applications.
2. The experiment may incorporate many algorithms run in parallel. In such case, available memory must be partitioned between all of them. In the future, parallelization will become more and more common due to parallelization of hardware architectures, e.g., expressed by increasing number of cores in processors.
3. In a complex experiment, composed of many elementary algorithms, every intermediate algorithm will generate another set of data. Total amount of data will be much larger than the amount of source data alone.
4. For architectural reasons data must be stored in memory in some general data structures, which take more memory than would be necessary in a given experiment. For example, data may be composed of binary attributes and each value could be stored on a single bit, but in fact each value takes 8 bytes or more, because every attribute – whether it is numeric or binary – is stored in the same way. Internal data representation used by a given platform is always a compromise between generality and efficient memory usage.
5. Data generated at intermediate processing stages may be many times larger than source data. For example:
   - Input data may require decompression, e.g. JPEG images must be converted to raw bitmaps to undergo processing. This may increase data size even by a factor of 100.
   - In image recognition, a single input image may be used to generate thousands of subwindows that would undergo further processing [4,26]. An input image of 1MB size may easily generate windows of 1GB size or more. Similar situation occurs in speech recognition or time series analysis, where the sliding-window technique is used.
   - Synthetic attributes may be generated, e.g. by taking all multiplications of pairs of original attributes, which leads to quadratic increase in the number of attributes.
   - Synthetic samples may be generated, in order to increase the size of training set and improve learning of a decision system. For example, this method is used in [27], which studies the problem of Optical Character Recognition. Training images of hand-written characters are randomly

distorted by planar affine transformations and added to the training set. Every image undergoes 9 random distortions, which leads to 10-fold increase in the training set size (from 60 to 600 thousand images).

6. In some applications, like mining data streams [1], input data are potentially infinite, so scalability obviously becomes an issue.
7. Even if the volume of data is small at the stage of experiments, it may become much bigger when the algorithm is deployed in a final product and must process real-world instead of experimental data.

The above arguments show clearly that memory is indeed a critical issue for data mining algorithms. Every moderately complex experiment will show one or more of the characteristics listed above. This is why we need scalable algorithms and – for this purpose – an architecture that will enable algorithms to process data on the fly, without full materialization of a data set.

3.2 Interoperability

Nowadays, it is impossible to solve a data mining task or conduct an experiment using only one algorithm. For example, even if you want to experiment with a single algorithm, like a new classification method, you at least have to access data on disk, so you need an algorithm that reads a given file format (e.g. ARFF[11]). Also, you would like to evaluate your classifier, so you need an algorithm which implements an evaluation scheme, like cross-validation or bootstrap. And in most cases you will also need several algorithms for data preprocessing like normalization, feature selection, imputation of missing values etc. – note that preprocessing is an essential step in knowledge discovery [28,29] and usually several different preprocessing methods must be applied before data can be passed to a decision system.

To build a data mining system, there must be a way to connect all these different algorithms together. Thus, they must possess the property of interoperability. Without this property, even the most efficient algorithm is practically useless.

Further on, the graph of data flow between elementary algorithms in a data mining system will be called a *Data Processing Network* (DPN). In general, we will assume that DPN is a directed acyclic graph, so there are no loops of data flow. Moreover, in the current version of Debellor, DPN can only have a form of a single chain, without branches. An example of DPN is shown in Figure 1.

Fig. 1. Example of a Data Processing Network (DPN), composed of five elementary algorithms (boxes). Arrows depict data flow between the algorithms.

[11] http://www.cs.waikato.ac.nz/ml/weka/arff.html

4 Data Streaming

To provide interoperability, data mining algorithms must be implemented in a common software *architecture*, which specifies:

- a method for connecting algorithms,
- a model of data transfer,
- common data representation.

Architectures of existing data mining systems utilize the *batch* model of data transfer. In this model, algorithms must take the whole data set as an argument for execution. To run composite experiment, represented by a DPN with a number of algorithms, an additional *supervisor* module is needed, responsible for invoking consecutive algorithms and passing data sets between them. Figure 3 presents a UML *sequence diagram* [30] with an example of batch processing in a DPN composed of three algorithms. DPN itself is presented in Fig. 2.

Batch data transfer enforces data materialization, which precludes scalability of algorithms and DPN as a whole. For example, in Weka, every classifier must be implemented as a subclass of `Classifier` class (in `weka.classifiers` package). Its training algorithm must be implemented in the method:

```
buildClassifier(Instances) :  void
```

The argument of type `Instances` is an array of training samples. This argument must be created before calling `buildClassifier`, so the data must be fully materialized in memory just to invoke training algorithm, no matter what the algorithm actually does.

Similar situation takes place for clustering methods, which must inherit from `weka.clusterers.Clusterer` class and overload the method:

```
buildClusterer(Instances) :  void
```

Rseslib and RapidMiner also enforce data materialization before a training algorithm can be invoked. In Rseslib, classifiers must be trained in the class constructor, which takes an argument of type `DoubleDataTable`. In RapidMiner, training of any decision system takes place in the method `apply(IOContainer)` of the class `com.rapidminer.operator.Operator`. Both Rseslib's `DoubleDataTable` and RapidMiner's `IOContainer` represent materialized input data.

If a large data set must be materialized, execution of the experiment is practically impossible. If data fit in virtual memory [31], but exceed available physical memory, operating system temporarily swaps [31] part of the data (stores it in

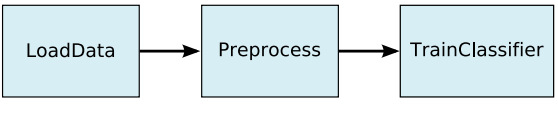

Fig. 2. DPN used as an example for analysis of data transfer models

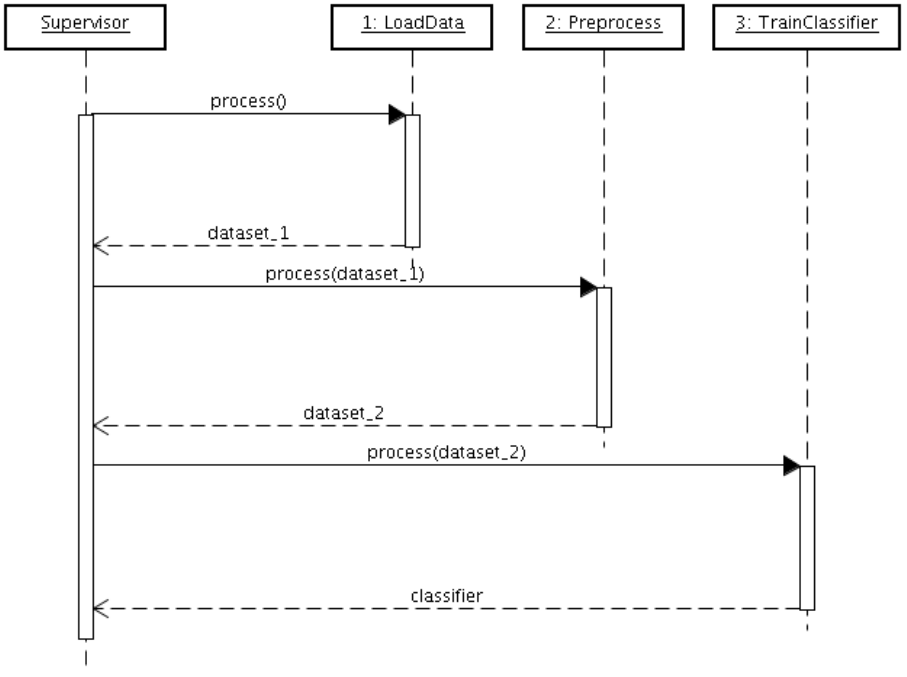

Fig. 3. UML diagram of *batch* data transfer in a DPN composed of three algorithms: LoadData, Preprocess and TrainClassifier, controlled by the Supervisor module. Supervisor invokes the algorithms (methods `run`) and pass data between them. All samples of a given data set are generated and transferred together, so available memory must be large enough to hold all data. Vertical lines denote life of modules, with time passing down the lines. Horizontal lines represent messages (method calls and/or data transfers) between the modules. Vertical boxes depict execution of the module's code.

the swap file on disk), which makes the execution tens or hundreds times slower, as access to disk is orders of magnitude slower than to memory.

If the data set is so large that it even exceeds available virtual memory, execution of the experiment is terminated with an out-of-memory error. This problem could be avoided if the class that represents a data set (e.g., `Instances` in Weka) implemented internally the buffering of data on disk. Then, however, the same performance degradation would occur as in the case of system swapping, because swapping and buffering on disk are actually the same things, only implemented at different levels: of operating system or data mining environment.

The only way to avoid severe performance degradation when processing large data is to generate data iteratively, sample-by-sample, and instantly process created samples, as presented in Fig. 4. In this way, data may be generated and consumed on the fly, without materialization of the whole set. This model of data transfer will be called *iterative*.

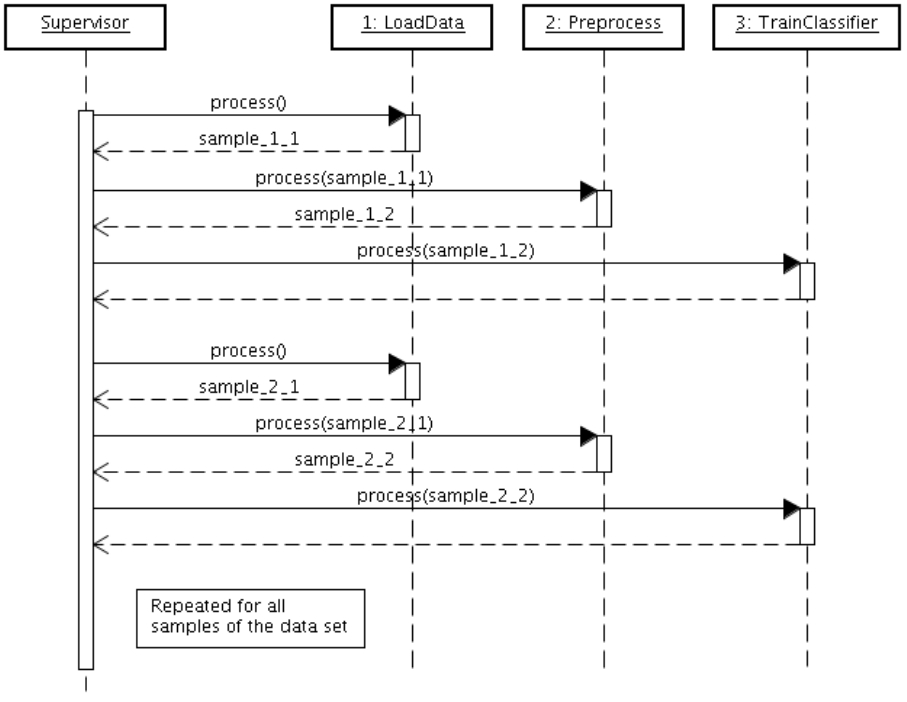

Fig. 4. UML diagram of *iterative* data transfer. The supervisor invokes the algorithms separately for each sample of the data set (sample_x_y denotes sample no. x generated by algorithm no. y). In this way, memory requirements are very low (memory complexity is constant), but supervisor's control over data flow becomes very difficult.

Iterative data transfer solves the problem of high memory consumption, because memory requirements imposed by the architecture are constant – only a fixed number of samples must be kept in memory in a given moment, no matter how large the full data set is. However, another problem arises: the supervisor becomes responsible for controlling the flow of samples and the order of execution of algorithms. This control may be very complex, because each elementary algorithm may have different input-output characteristics. The number of possible variants is practically infinite, for example:

1. Preprocessing algorithm may filter out some samples, in which case more than one input sample may be needed to produce one output sample.
2. Preprocessing algorithm may produce a number of output samples from a single input sample, e.g. when extracting windows from an image or time series.
3. Training algorithm of a decision system usually have to scan data many times, not only once.

4. Generation of output samples may be delayed relatively to the flow of input samples, e.g. an algorithm may require that 10 input samples are given before it starts producing output samples.
5. Input data to an algorithm may be infinite, e.g. when they are generated synthetically. In such case, the control mechanism must stop data generation in appropriate moment.
6. Some algorithms may have more than one input or output, e.g. an algorithm for merging data from several different sources (many inputs) or an algorithm for splitting data into training and test parts (many outputs). In such case, the control of data flow through all the inputs and outputs becomes even more complex, because there are additional dependencies between many inputs/outputs of the same algorithm.

Note that the diagram in Fig. 4 depicts a simplified case when DPN is a single chain of three algorithms, without branches; preprocessing generates exactly one output sample for every input sample; and training algorithm scans data only once.

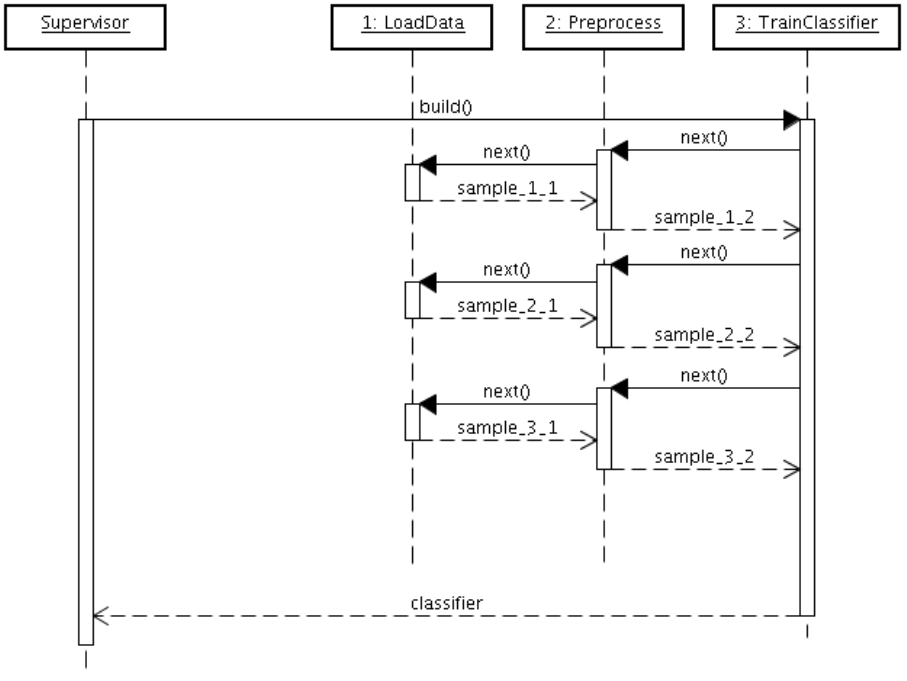

Fig. 5. UML diagram of control and data flow in the *stream* model of data transfer. The supervisor invokes only method build() of the last component (TrainClassifier). This triggers a cascade of messages (calls to methods next()) and transfers of samples, as needed to fulfill the initial build() request.

The way how data flow should be controlled depends on what algorithms are used in a given DPN. For this reason, the algorithms themselves – not the supervisor – should be responsible for controlling data flow. To this end, each algorithm must be implemented as a *component* which can communicate with other components without external control of the supervisor. Supervisor's responsibility must be limited only to linking components together (building DPN) and invoking the last algorithm in DPN, which is the final receiver of all samples. Communication should take the form of a stream of samples: (i) sample is the unit of data transfer; (ii) samples are transferred sequentially, in a fixed order decided by the sender. This model of data transfer will be called a *stream* model. An example of control and data flow in this model is presented in Fig. 5.

Component architecture and data streaming are the features of Debellor which enable scalability of algorithms implemented on this platform.

5 Debellor Data Mining Platform

5.1 Data Streams

Debellor's components are called *cells*. Every cell is a Java class inheriting from the base class `Cell` (package `org.debellor.core`). Cells may implement all kinds of data processing algorithms, for example:

1. Decision algorithms: classification, regression, clustering, density estimation etc.
2. Transformations of samples and attributes.
3. Removal or insertion of samples and attributes.
4. Loading data from file, database etc.
5. Generation of synthetic data.
6. Buffering and reordering of samples.
7. Evaluation schemes: train&test, cross-validation, leave-one-out etc.
8. Collecting statistics.
9. Data visualization.

Cells may be connected into DPN by calling the `setSource(Cell)` method on the receiving cell, for example:

```
Cell cell1 = ..., cell2 = ..., cell3 = ...;
cell2.setSource(cell1);
cell3.setSource(cell2);
```

The first cell will usually represent a file reader or a generator of synthetic data. Intermediate cells may apply different kinds of data transformations, while the last cell will usually implement a decision system or an evaluation procedure.

DPN can be used to process data by calling methods `open()`, `next()` and `close()` on the last cell of DPN, for example:

```
cell3.open();
sample1 = cell3.next();
sample2 = cell3.next();
sample3 = cell3.next();
...
cell3.close();
```

The above calls open communication session with `cell3`, retrieve some number of processed samples and close the session. In order to realize each request, `cell3` may communicate with its source cell, `cell2`, by invoking the same methods (`open`, `next`, `close`) on `cell2`. And `cell2` may in turn communicate with `cell1`. In this way it is possible to generate output samples on the fly. The stream of samples may flow through consecutive cells of DPN without buffering, so input data may have unlimited volume.

Note that the user of DPN does not have to control sample flow by himself. To obtain the next sample of processed data it is enough to call `cell3.next()`, which will invoke – if needed – a cascade of calls to preceding cells.

Moreover, different cells may control the flow of samples differently. For example, cells that implement classification algorithms will take one input sample in order to generate one output sample. Filtering cells will take a couple of input samples in order to generate one output sample that matches the filtering rule. The image subwindow generator will produce many output samples out of a single input sample. We can see that the cell's interface is very flexible. It enables implementation of various types of algorithms in the same framework and allows to easily combine the algorithms into a complex DPN.

5.2 Buildable Cells

Some cells may be *buildable*, in which case their *content* must be built before the cell can be used. Building procedure is invoked by calling method

```
build() :  void
```

on the cell object. This method is declared in the base class `Cell`.

Building a cell may mean different things for different types of cells. For example:

- training a decision system of some kind (classifier, clusterer, ...),
- running an evaluation scheme (train&test, cross-validation, ...),
- reading all data from input stream and buffering in memory.

Note that all these different types of algorithms are encapsulated under the same interface (method `build()`). This increases simplicity and modularity of the platform.

Usually, the cell reads input data during building, so it must be properly connected to a source cell before `build()` is invoked. Afterwards, the cell may be reconnected and used to process another stream of data.

Some buildable cells may also implement `erase()` method, which clears the content of the cell. After erasure, the cell may be built once again.

5.3 State of the Cell

Every cell object has a *state* variable attached, which indicates what cell operations are allowed in a given moment. There are three possible states: EMPTY, CLOSED and OPEN. Transitions between them are presented in Fig. 6. Each transition is invoked by call to an appropriate method: `build()`, `erase()`, `open()` or `close()`.

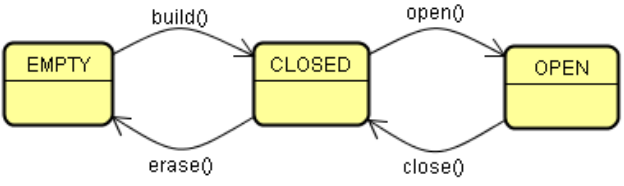

Fig. 6. Diagram of cell states and allowed transitions

Only a part of cell methods may be called in a given state. For example, `next()` can be called only in OPEN state, while `setSource()` is allowed only in EMPTY or CLOSED state. It is guaranteed by the base class implementation that disallowed calls immediately end with an exception thrown. Thanks to this automatic state control, connecting different cells together and building composite algorithms becomes easier and safer, because many possible mistakes or bugs related to inter-cell communication are detected early. Otherwise, they could exist unnoticed, generating incorrect results during data processing. Moreover, it is easier to implement new cells, because the authors do not have to check correctness of method calls by themselves.

5.4 Parametrization

Most of cells require a number of parameters to be set before the cell can start working. Certainly, every type of a cell requires different parameters, but for the sake of interoperability and simplicity of usage, there should be a common interface for passing parameters, no matter what number and types of parameters are expected by a given cell. Debellor defines such an interface.

Parameters for a given cell are stored in an object of class `Parameters` (package `org.debellor.core`), which keeps a dictionary of parameter names and associated `String` values (in the future we plan to extend permitted value types, note however that all simple types can be easily converted to `String`). Thanks to the use of a dictionary, the names do not have to be hard-coded as fields of cell objects, hence parameters can be added dynamically, according to requirements of a given cell.

The object of class `Parameters` can be passed to the cell by calling `Cell`'s method:

```
setParameters(Parameters) :  void
```

It is also possible (and usually more convenient) to pass single parameter values directly to the cell, without an intermediate `Parameters` object, by calling:

```
set(String name, String value) :  void
```

This method call delegates to analogous method of `Cell`'s internal `Parameters` object.

5.5 Data Representation

The basic unit of data transfer between cells is *sample*. Samples are represented by objects of class `Sample`. Every sample contains two fields, `data` and `label`, which hold input data and associated decision label, respectively. Any of the fields can be `null`, if corresponding information is missing or simply not necessary at the given point of data processing.

Cells are free to use whichever part of input data they want. For example, `build()` method of a classifier (i.e. training algorithm) would use both `data` and `label`, interpreting `label` as a target classification of `data`, given by a supervisor. During operation phase, the classifier would ignore input `label`, if present. Instead, it would classify `data` and assign the generated label to the `label` field of the output sample.

Data and labels are represented in an abstract way. Both `data` and `label` fields reference objects of type `Data` (package `org.debellor.core`). `Data` is a base class for classes that represent data items, like single features or vectors of features. When the cell wants to use information stored in `data` or `label`, it must downcast the object to a specific subclass, as expected by the cell. Thanks to this abstract method of data representation, new data types can be added easily, by creating a new subclass of `Data`. Authors of new cells are not limited to a single data type, hard-coded into the platform, as for example in Weka.

Data objects may be nested. For example, objects of class `DataVector` (in `org.debellor.core.data`) hold arrays of other data objects, like simple features (classes `NumericFeature` and `SymbolicFeature`) or other `DataVector`s.

5.6 Immutability of Data

A very important concept related to data representation is *immutability*. Objects which store data – instances of `Sample` class or `Data` subclasses – are *immutable*, i.e. they cannot be modified after creation. Thanks to this property, data objects can be safely shared by cells, without risk of accidental modification in one cell that would affect operations of another cell.

Immutability of data objects yields many benefits:

1. Safety – cells written by different people may work together in a complex DPN without interference.
2. Simplicity – the author of a new cell does not have to care about correctness of access to data objects.

3. Efficiency – data objects do not have to be copied when transferred to another cell. Without immutability, copying would be necessary to provide a basic level of safety. Also, a number of samples may keep references to the same data object.
4. Parallelization – if DPN is executed concurrently, no synchronization is needed when accessing shared data objects. This simplifies parallelization and makes it more efficient.

5.7 Metadata

Many cells have to know some basic characteristics ("type") of input samples before processing of the data starts. For example, training algorithm of a neural network has to know the number of input features, to be able to allocate arrays of weights of appropriate size. To provide such information, method `open()` returns an object of class `MetaSample` (static inner class of `Sample`), which describes common properties of all samples generated by the stream being open. Similarly to `Sample`, `MetaSample` has separate fields describing input data and labels, both of type `MetaData` (static inner class of `Data`).

Metadata have analogous structure and properties as the data being described. The hierarchy of metadata classes, rooted at `MetaData`, mirrors the hierarchy of data classes, rooted at `Data`. The nesting of `MetaData` and `Data` objects is also similar, e.g. if the stream generates `DataVector`s of 10 `SymbolicFeature`s, corresponding `MetaData` object will be an instance of `MetaDataVector`, containing an array of 10 `MetaSymbolicFeature`s describing every feature.

Similarly to `Data`, `MetaData` objects are immutable, so they can be safely shared by cells.

5.8 Example

To illustrate the usage of Debellor, we will show how to implement standard k-means algorithm in stream architecture and how to employ it to data processing in a several-cell DPN.

K-means [17,24,25] is a popular clustering algorithm. Given n input samples – numeric vectors of fixed length, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ – it tries to find cluster centers $\mathbf{c}_1, \ldots, \mathbf{c}_k$ which minimize the sum of squared distances of samples to their closest center:

$$\mathcal{E}(\mathbf{c}_1, \ldots, \mathbf{c}_k) = \sum_{i=1}^{n} \min_{j=1,\ldots,k} \|\mathbf{x}_i - \mathbf{c}_j\|^2 \ . \tag{1}$$

This is done through iterative process with two steps repeated alternately in a loop: (i) assignment of each sample to the nearest cluster and (ii) repositioning of each center to the centroid of all samples in a given cluster. The algorithm is presented in Fig. 7. As we can see, the common implementation of k-means as a function is non-scalable, because it employs batch model of data transfer: training data are passed as an array of samples, so they must be generated and accumulated in memory before the function is called.

```
function kmeans(data) returns an array of centers

    Initialize array centers
    repeat
        Set sum[1], ..., sum[k], count[1], ..., count[k] to zero
        for i = 1..n do                    /* assign samples to clusters */
            x = data[i]
            j = clusterOf(x)
            sum[j] = sum[j] + x
            count[j] = count[j] + 1
        end
        for j = 1..k do                    /* reposition centers */
            centers[j] = sum[j]/count[j]
        end
    until no center has been changed
    return centers
```

Fig. 7. Pseudocode illustrating k-means clustering algorithm implemented as a regular stand-alone function. The function takes an array of n samples (*data*) as argument and returns k cluster centers. Both samples and centers are real-valued vectors. The function clusterOf(x) returns index of the center that is closest to x.

```
class KMeans extends Cell
method build()

    Initialize array centers
    repeat
        Set sum[1], ..., sum[k], count[1], ..., count[k] to zero
(*)     source.open()
        for i = 1..n do
(*)         x = source.next()
            j = clusterOf(x)
            sum[j] = sum[j] + x
            count[j] = count[j] + 1
        end
(*)     source.close()
        for j = 1..k do
            centers[j] = sum[j]/count[j]
        end
    until no center has been changed
```

Fig. 8. Pseudocode illustrating implementation of k-means as Debellor's cell. Since k-means is a training algorithm (generates a decision model), it must be implemented in method build() of a Cell's subclass. Input data are provided by the *source* cell, the reference *source* being a field of Cell. The generated model is stored in the field *centers* of class KMeans, method build() does not return anything. The lines of code inserted or modified relatively to the standard implementation are marked with asterisk (*).

```
class KMeans extends Cell
method next()
    x = source.next()
    if x == null then
        return null
    return x.setLabel(clusterOf(x))
```

Fig. 9. Pseudocode illustrating implementation of method next() of KMeans cell. This method employs the clustering model generated by build() and stored inside the KMeans object to label new samples with identifiers of their clusters.

```
/* 3 cells are created and linked into DPN */
Cell arff = new ArffReader();
arff.set("filename", "iris.arff");          /* parameter filename is set */

Cell remove = new WekaFilter("attribute.Remove");
remove.set("attributeIndices", "last");
remove.setSource(arff);                  /* cells arff and remove are linked */

Cell kmeans = new KMeans();
kmeans.set("numClusters", "10");
kmeans.setSource(remove);

/* k-means algorithm is executed */
kmeans.build();

/* the clusterer is used to label 3 training samples with cluster identifiers */
kmeans.open();
Sample s1 = kmeans.next(),
       s2 = kmeans.next(),
       s3 = kmeans.next();
kmeans.close();

/* labelled samples are printed on screen */
System.out.println(s1 + "\n" + s2 + "\n" + s3);
```

Fig. 10. Java code showing sample usage of Debellor cells: reading data from an ARFF file, removal of an attribute, training and application of a k-means clusterer

Stream implementation of k-means – as Debellor's cell – is presented in Fig. 8. In contrast to the standard implementation, training data are *not* passed explicitly, as an array of samples. Instead, the algorithm retrieves samples one-by-one from the source cell, so it can process arbitrarily large data sets. In addition,

Fig. 9 shows how to implement method `next()`, responsible for application of the generated clustering model to new samples.

Note that despite the algorithm presented in Fig. 8 employs *stream* method of data transfer, it employs *batch* method of updating the decision model (the updates are performed after all samples have been scanned). These two things – the method of data transfer and the way how model is updated – are separate and independent issues. It is possible for batch (in terms of model update) algorithms to utilize and benefit from stream architecture.

Listing in Fig. 10 shows how to run a simple experiment: train a k-means clusterer and apply it to several training samples, to label them with identifiers of their clusters. Data are read from an ARFF file and simple preprocessing – removal of the last attribute – is applied to all samples. Note that loading data from file and preprocessing is executed only when the next input sample is requested by the `kmeans` cell – in methods `build()` and `next()`.

6 Experimental Evaluation

6.1 Setup

In existing data mining systems, when data to be processed are too large to fit in memory, they must be put in *virtual* memory. During execution of the algorithm, parts of data are being swapped to disk by operating system, to make space for other parts, currently requested. In this way, portions of data are constantly moving between memory and disk, generating huge overhead on execution time of the algorithm. In the presented experiments we wanted to estimate this overhead and the performance gain that can be obtained through the use of Debellor's data streaming instead of swapping.

For this purpose, we trained k-means [17,24,25] clustering algorithm on time windows extracted from the time series that was used in EUNITE[12] 2003 data mining competition. We compared execution times of two variants of the experiment:

1. *batch*, with time windows created in advance and buffered in memory,
2. *stream*, with time windows generated on the fly.

Data Processing Networks of both variants are presented in Fig. 11 and 12.

In both variants, we employed our stream implementation of k-means, sketched in Sect. 5.8 (KMeans cell in Fig. 11 and 12). In the first variant, we inserted a buffer into DPN just before the KMeans cell – in this way we effectively obtained a batch algorithm. In the second variant, the buffer was placed earlier in the chain of algorithms, before window extraction. We could have dropped buffering at all, but then the data would be loaded from disk again in every training cycle, which was not necessary, as the source data were small enough to fit in memory.

[12] EUropean Network on Intelligent TEchnologies for Smart Adaptive Systems, http://www.eunite.org

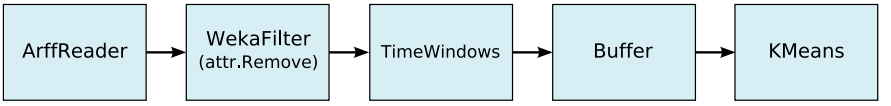

Fig. 11. DPN of the first (batch) variant of experiment

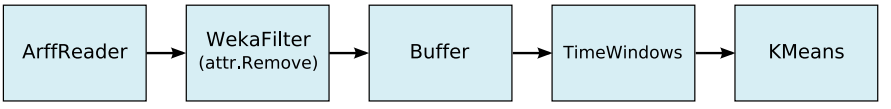

Fig. 12. DPN of the second (stream) variant of experiment

Source data were composed of a series of real-valued measurements from glass production process, recorded in 9408 different time points separated by 15-minute intervals. There were two kinds of measurements: 29 "input" and 5 "output" values. In the experiment we used only "input" values, "output" ones were filtered out by Weka filter for attribute removal (WekaFilter cell).

After loading from disk and dropping unnecessary attributes, the data occupied 5.7MB of memory. They were subsequently passed to TimeWindows cell, which generated time windows of length W, on every possible offset from the beginning of the input time series. Each window was created as a concatenation of W consecutive samples of the series. Therefore, for input series of length T, composed of A attributes, the resulting stream contained $T - W + 1$ samples, each composed of $W * A$ attributes. In this way, relatively small source data (5.7MB) generated large volume of data at further stages of DPN, e.g. 259MB for $W = 50$.

In the experiments, we compared training times of both variants of k-means. Since the time effectiveness of swapping and memory management depends highly on the hardware setup, the experiments were repeated in two different hardware environments: (A) a laptop PC with Intel Mobile Celeron 1.7 GHz CPU, 256MB RAM; (B) a desktop PC with AMD Athlon XP 2100+ (1.74 GHz), 1GB RAM. Both systems run under Microsoft Windows XP. Sun's Java Virtual Machine (JVM) 1.6.0_03 was used. The number of clusters for k-means was set to 5.

6.2 Results

Results of experiments are presented in Table 1 and 2. They are also depicted graphically in Fig. 13 and 14.

Different lengths of time windows were checked, for every length the size of generated training data was different (given in the second column of the tables). In each trial, training time of k-means was measured. These times are reported in normalized form, i.e. the total training time in seconds is divided by the number of training cycles and data size in MB. Normalized times can be directly

Table 1. Normalized training times of k-means for batch and stream variant of experiment and different lengths of time windows. Corresponding sizes of training data are given in the second column. Hardware environment A.

Window length	Data size [MB]	Normalized execution time (batch variant)	Normalized execution time (stream variant)
10	53	3.1	5.6
20	104	3.2	5.3
30	156	3.1	5.0
40	208	5.1	4.9
50	259	244.4	5.0
60	311	326.9	8.3
70	362	370.6	10.7
80	413	386.0	10.9
90	464	475.3	11.1

Table 2. Normalized training times of k-means for batch and stream variant of experiment and different lengths of time windows. Corresponding sizes of training data are given in the second column. Hardware environment B.

Window length	Data size [MB]	Normalized execution time (batch variant)	Normalized execution time (stream variant)
50	259	4.0	5.3
100	515	4.0	5.4
120	617	4.0	6.5
150	769	5.3	8.7
170	869	6.3	8.8
180	919	23.8	8.8
190	969	36.4	8.8
200	1019	50.7	8.8
210	1069	71.3	8.8
220	1119	85.1	8.8
230	1168	100.4	9.1
240	1218	111.1	9.1
250	1267	140.2	9.4
260	1317	**crash**	9.3

compared across different trials. Every table and figure presents results of both variants of the algorithm.

Time complexity of a single training cycle of k-means is linear in the data size, so normalized execution times should be similar across different values of window length. However, for the batch variant, the times are constant only for small sizes of data. At the point when data size gets close to the amount of physical memory installed on the system, execution time suddenly jumps to a very high value, many times larger than for smaller data sizes. It may even

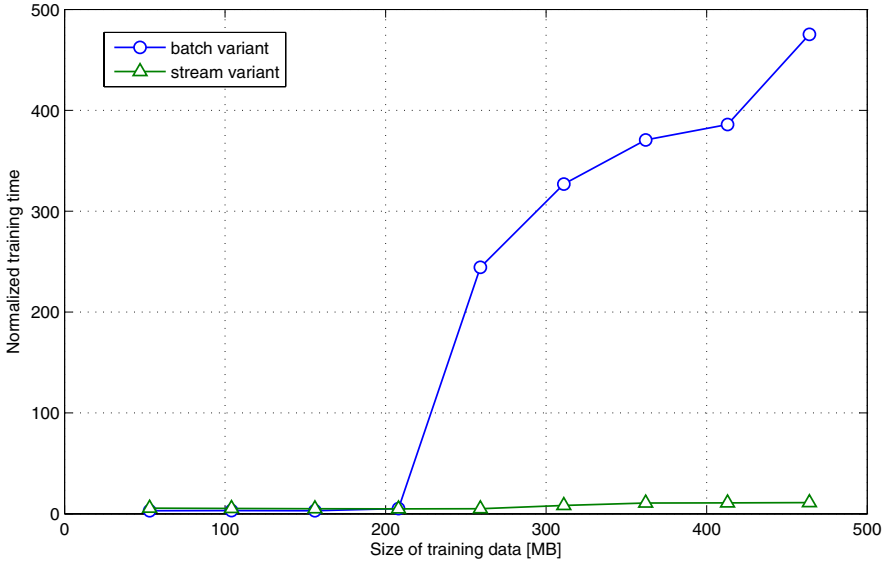

Fig. 13. Normalized training times of k-means for batch and stream variant of experiment and different lengths of time windows. Hardware environment A.

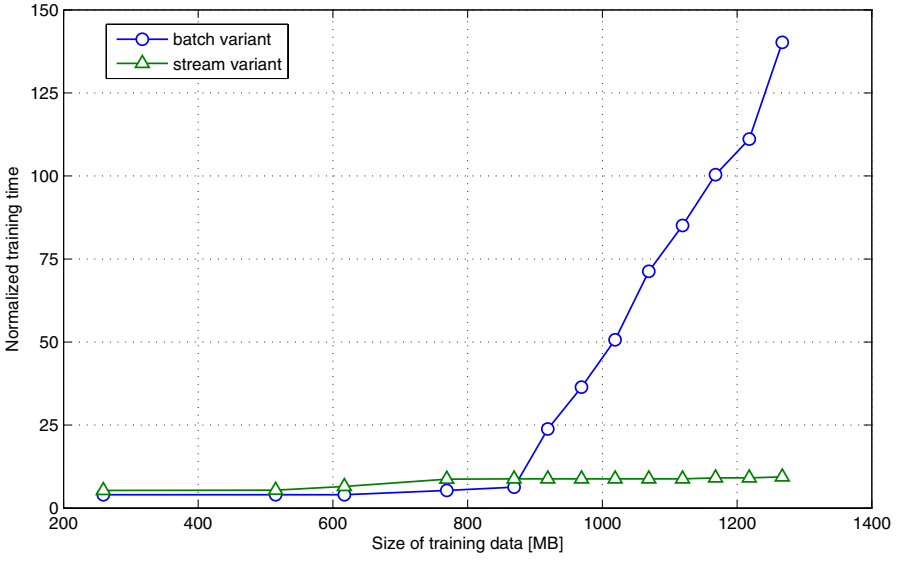

Fig. 14. Normalized training times of k-means for batch and stream variant of experiment and different lengths of time windows. Hardware environment B. Note that the measurement which caused the batch variant to crash (last row in Table 2) is not presented here.

happen that from some point the execution crashes due to memory shortage (see Tab. 2), despite JVM heap size being set to the highest possible value (1300 MB on a 32-bit system). This is because swapping must be activated to handle this large volume of data. And because access to disk is orders of magnitude slower than to memory, algorithm execution becomes also very slow.

This dramatic slowdown is not present in the case of the stream algorithm, which requires always the same amount of memory, at the level of 6MB. For small data sizes this algorithm runs a bit slower, because training data must be generated in each training cycle from the beginning. But for large data sizes it can be 40 times better, or even more (the curves in Figures 13 and 14 rise very quickly, so we may suspect that for larger data sizes the disparity between both variants is even bigger). The batch variant is actually not usable.

What is also important, every stream implementation of a data mining algorithm can be used in batch manner by simply preceding it with a buffer in DPN. Thus, the user can choose the faster variant, depending on the data size. On the other hand, batch implementation *cannot* be used in stream-based manner, rather the algorithm must be redesigned and implemented again.

7 Conclusions

In this paper we introduced Debellor – a data mining platform with stream architecture. We presented the concept of data streaming and proved through experimental evaluation that it enables much more efficient processing of large data than the currently used method of batch data transfer. Stream architecture is also more general. Every stream-based implementation can be used in batch manner. Opposite is not true. Thanks to data streaming, algorithms implemented on Debellor platform can be scalable and interoperable at the same time. We also analysed the significance of scalability issue for the design of composite data mining systems and showed that even when source data are relatively small, lack of memory may still pose a problem, since large volumes of data may be generated at intermediate stages of data processing network.

Stream architecture has also weaknesses. Because of sequential access to data, implementation of algorithms may be conceptually more difficult. Batch data transfer is more intuitive for the programmer. Moreover, some algorithms may inherently require random access to data. Although they can be implemented in stream architecture, they have to buffer all data internally, so they will not benefit from streaming. However, these algorithms can still benefit from interoperability provided by Debellor – they can be connected with other algorithms to form a complex data mining system.

Development of Debellor will be continued. We plan to extend the architecture to handle multi-input and multi-output cells as well as nesting of cells (e.g., to implement meta-learning algorithms). We also want to implement parallel execution of DPN and serialization of cells (i.e., saving to a file).

Acknowledgement. The research has been partially supported by the grant N N516 368334 from Ministry of Science and Higher Education of the Republic

of Poland and by the grant "Decision support – new generation systems" of Innovative Economy Operational Programme 2008-2012 (Priority Axis 1. Research and development of new technologies) managed by Ministry of Regional Development of the Republic of Poland.

References

1. Aggarwal, C.C. (ed.): Data Streams: Models and Algorithms. Springer, Heidelberg (2007)
2. Gama, J., Gaber, M.M. (eds.): Learning from Data Streams: Processing Techniques in Sensor Networks. Springer, Heidelberg (2007)
3. Gonzalez, R.C., Woods, R.E.: Digital Image Processing. Prentice Hall, Englewood Cliffs (2002)
4. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. IEEE Computer Vision and Pattern Recognition 1, 511–518 (2001)
5. Bazan, J.G., Szczuka, M.: RSES and RSESlib – A collection of tools for rough set computations. In: Ziarko, W.P., Yao, Y. (eds.) RSCTC 2000. LNCS, vol. 2005, pp. 106–113. Springer, Heidelberg (2001)
6. Bazan, J.G., Szczuka, M.S., Wojna, A., Wojnarski, M.: On the evolution of rough set exploration system. In: Tsumoto, S., Słowiński, R., Komorowski, J., Grzymała-Busse, J.W. (eds.) RSCTC 2004. LNCS, vol. 3066, pp. 592–601. Springer, Heidelberg (2004)
7. Wojna, A., Kowalski, L.: Rseslib: Programmer's Guide (2008), `http://rsproject.mimuw.edu.pl`
8. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)
9. R Development Core Team: R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria (2005)
10. Eaton, J.W.: Octave: Past, present, and future. In: International Workshop on Distributed Statistical Computing (2001)
11. Eaton, J.W., Rawlings, J.B.: Ten years of Octave – recent developments and plans for the future. In: International Workshop on Distributed Statistical Computing (2003)
12. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: Yale: Rapid prototyping for complex data mining tasks. In: KDD 2006: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (2006)
13. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation. Prentice Hall, Englewood Cliffs (1999)
14. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall, Englewood Cliffs (2001)
15. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The stanford data stream management system (2004), `http://dbpubs.stanford.edu:8090/pub/2004-20`
16. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: ACM (ed.) Symposium on Principles of Database Systems, pp. 1–16. ACM Press, New York (2002)
17. Ripley, B.D.: Pattern recognition and neural networks. Cambridge University Press, Cambridge (1996)

18. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer, Heidelberg (2006)
19. Bradley, P.S., Fayyad, U.M., Reina, C.: Scaling clustering algorithms to large databases. In: Knowledge Discovery and Data Mining (1998)
20. Balakrishnan, S., Madigan, D.: Algorithms for sparse linear classifiers in the massive data setting. Journal of Machine Learning Research 9, 313–337 (2008)
21. Amit, Y., Shalev-Shwartz, S., Singer, Y.: Online learning of complex prediction problems using simultaneous projections. Journal of Machine Learning Research 9, 1399–1435 (2008)
22. Furaoa, S., Hasegawa, O.: An incremental network for on-line unsupervised classification and topology learning. Neural Networks 19, 90–106 (2006)
23. Kivinen, J., Smola, A.J., Williamson, R.C.: Online learning with kernels. IEEE Transactions On Signal Processing 52, 2165–2176 (2004)
24. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Computing Surveys 31, 264–323 (1999)
25. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs (1995)
26. Wojnarski, M.: Absolute contrasts in face detection with adaBoost cascade. In: Yao, J., Lingras, P., Wu, W.-Z., Szczuka, M.S., Cercone, N.J., Ślęzak, D. (eds.) RSKT 2007. LNCS, vol. 4481, pp. 174–180. Springer, Heidelberg (2007)
27. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324 (1998)
28. Kriegel, H.P., Borgwardt, K.M., Kröger, P., Pryakhin, A., Schubert, M., Zimek, A.: Future trends in data mining. Data Mining and Knowledge Discovery 15, 87–97 (2007)
29. Pyle, D.: Data Preparation for Data Mining. Morgan Kaufmann Publishers, San Francisco (1999)
30. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide. Addison-Wesley, Reading (2005)
31. Silberschatz, A., Galvin, P., Gagne, G.: Operating System Concepts, 7th edn. Wiley, Chichester (2004)