Recent Advances in Intelligent Information Systems
ISBN 978-83-60434-59-8, pages 669–682

Debellor: Open Source Modular Platform for Scalable Data Mining

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-oriented 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. Due to its scalability and modularity Debellor was chosen as the basis for TunedTester application – one of three pillars of TunedIT (tunedit.org) system for automatic evaluation of machine learning algorithms. The current version of Debellor is 0.6.2.

Keywords: pipeline, online algorithm, framework, knowledge discovery

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 (Aggarwal, 2007; Gama and Gaber, 2007), 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 (Viola and Jones, 2001). 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*.

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.

Debellor is written in Java and distributed under GNU General Public License. The most recent version 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.

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 implement the algorithm in an environment for numerical calculations: R[2] (R Development Core Team, 2005), Matlab, Octave[3]. The problem is that they do not define common architecture for algorithms, so they do not automatically provide interoperability. Moreover, the scripting languages of these environments are suitable rather for fast prototyping and running small experiments than 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[4] (Witten and Frank, 2005) or Rseslib[5] (Bazan *et al.*, 2004; Wojna and Kowalski, 2008) – and try to fit the new algorithm into the architecture of the library. However, these libraries preclude scalability of algorithms, because the

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

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 (Garcia-Molina *et al.*, 2001) for efficient query execution. Elementary units capable of processing streams are called *iterators* in Garcia-Molina *et al.* (2001).

The issue of scalability is related to the concept of *online* algorithms – the training algorithms which perform updates of the underlying decision model after every single presentation of a sample (Ripley, 1996; Bishop, 2006). The algorithms which update the model only when the whole training set has been presented are called *batch*.

3 Motivation

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. Experiment may incorporate many algorithms run in parallel – 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 general data structures, which take more memory than really needed in a given task. 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 (cf. Wojnarski, 2007). 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.

- Samples may be extended with synthetic (derived) features, e.g. multiplications of all pairs of original features, which leads to quadratic increase in the size of every sample.
- Training set may be extended with synthetic (derived) samples to increase the amount of training data and improve learning of a decision system. For example, this method is used in LeCun *et al.* 1998.

6. In some applications, like mining data streams (Aggarwal, 2007), 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.

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. Sample DPN is shown in Fig. 1.

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

4 Data Streaming

Architectures of existing data mining systems utilize the *batch* model of data transfer, where 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. Batch data transfer enforces data materialization, which precludes scalability of algorithms and DPN as a whole. For example, in Weka, every classifier must be a subclass of `Classifier` class. 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 it actually does. Similar situation takes place for clustering methods. 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. 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*.

Iterative data transfer solves the problem of high memory consumption, because only a fixed number of samples must be kept in memory in a given moment. However, another problem arises: supervisor becomes responsible for controlling 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. For this reason, algorithms themselves should be responsible for controlling data flow. They must be implemented as *components* which can communicate with others without external control. Supervisor's responsibility must be limited to linking components together into DPN and invoking the last algorithm – final receiver of all samples. Communication should take the form of a stream of samples, transferred sequentially, in a fixed order decided by the sender. This model of data transfer will be called a *stream* model. 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.
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 Trainable Cells

The cell may be *trainable* – before it can be used it must be trained how to process data. Training a cell consists typically of connecting it with a source of training data, setting parameter values and invoking its *learning procedure* declared in the base class `Cell` as:

```
learn() :  void
```

The behavior of learning procedure is specific to the `Cell`'s subclass actually used.

The term of "learning procedure" as used in Debellor has very wide meaning. It includes not only generation of a decision system that could be used subsequently for data processing, but also any other data-driven operation that only accumulates some information (*knowledge*) internally in the cell, without generation of output samples. For example, learning procedure may implement:

1. Calculation of data-driven parameters for a preprocessing algorithm, e.g., attribute means for the normalization algorithm.
2. Calculation of data statistics, like a histogram of attribute values, for subsequent visualization.
3. Evaluation of another cell (Train&Test, Cross-validation, Leave-one-out, ...).
4. Buffering of input data, so that later on they can be delivered repeatedly to other cells without recalculation.

Knowledge gained by the cell during learning can be erased by a call to `erase()`. After erasure the cell can be trained 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. 2. Each transition is invoked by call to an appropriate method: `learn()`, `erase()`, `open()` or `close()`.

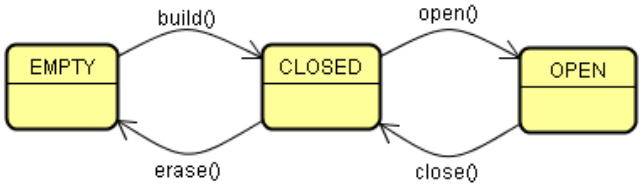

FIGURE 2: Diagram of cell states and allowed transitions

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 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 for access to shared data objects, so parallelization is simpler and more efficient.

```
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
```

FIGURE 3: 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

5.5 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 (Jain *et al.*, 1999; Ripley, 1996) 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. 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. 3. 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.

Stream implementation of k-means – as Debellor's cell – is presented in Fig. 4. 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. Note that despite this algorithm 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). Thus, it is possible for batch (in terms of model update) algorithms to utilize and benefit from stream architecture.

```
class KMeans extends Cell
method learn()

    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
```

FIGURE 4: Pseudocode illustrating implementation of k-means as Debellor's cell. Since k-means is a learning algorithm (generates a decision model), it must be implemented in method learn() 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 learn() does not return anything. The lines of code inserted or modified relatively to the standard implementation are marked with asterisk (*)

Listing in Fig. 5 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 `learn()` 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.

```
/* 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.learn();

/* 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);
```

FIGURE 5: 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

For this purpose, we trained k-means (Jain *et al.*, 1999; Ripley, 1996) clustering algorithm on time windows extracted from the time series that was used in EUNITE[6] 2003 data mining competition. We compared execution times of two variants of 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. 6. In each of them, we employed our stream implementation of k-means, sketched in Sect. 5.5 (KMeans cell in Fig. 6). 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

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

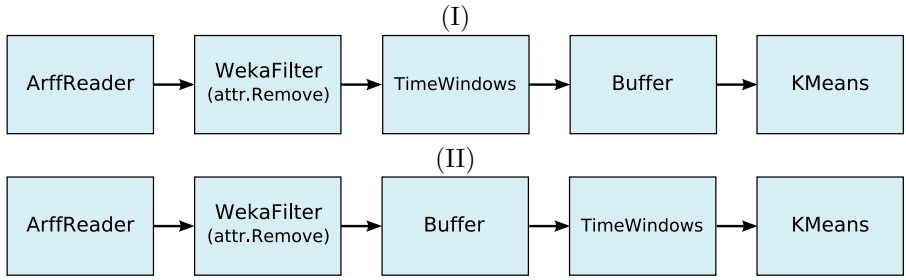

FIGURE 6: DPN of the first (batch) and second (stream) variant of experiment

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.

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 data at further stages of DPN, e.g. 259MB for $W = 50$.

Since time effectiveness of swapping and memory management depends highly on hardware setup, 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 was used. Number of clusters in k-means was 5.

6.2 Results

Results of experiments are presented in Table 1 and Fig. 7. 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 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 a 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

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 environments A and B

Window length	Data size [MB]	Normalized execution time (batch variant)	Normalized execution time (stream variant)
Hardware environment A			
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
Hardware environment B			
50	259	4.0	5.3
100	515	4.0	5.4
120	617	4.0	6.5
150	769	5.3	8.7
180	919	23.8	8.8
200	1019	50.7	8.8
220	1119	85.1	8.8
240	1218	111.1	9.1
250	1267	140.2	9.4
260	1317	**crash**	9.3

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 happen that from some point the execution crashes due to memory shortage, despite JVM heap size set to the highest possible value (1.3 GB on 32-bit system).

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 Fig. 7 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.

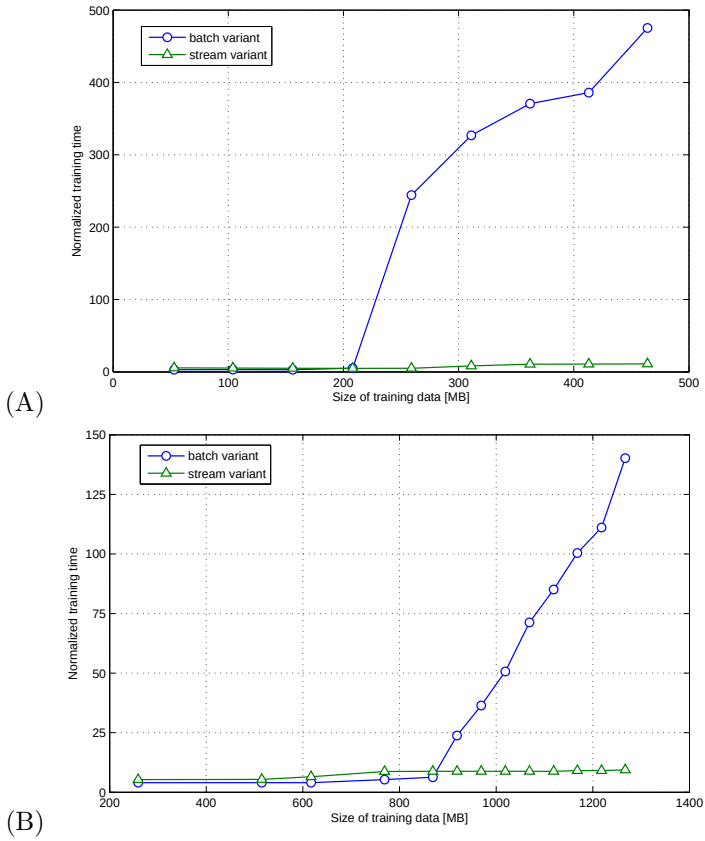

FIGURE 7: Normalized training times of k-means for batch and stream variant of experiment and different lengths of time windows. Hardware environments A and B

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 both scalable and interoperable.

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.

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 serialization of cells (i.e., saving to a file) and parallel execution of DPN.

It is also worth to mention that Debellor was chosen as the basis for TunedTester – an application for automatic evaluation of machine learning algorithms, being a part of TunedIT (tunedit.org) integrated system which facilitates execution of reproducible experiments, comparison of results and collaboration between researchers in the fields of data mining and machine learning.

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.

References

Charu C. Aggarwal, editor (2007), *Data Streams: Models and Algorithms*, Springer-Verlag.

Jan G. Bazan, Marcin S. Szczuka, Arkadiusz Wojna, and Marcin Wojnarski (2004), On the Evolution of Rough Set Exploration System, in *Rough Sets and Current Trends in Computing*, volume 3066 of *Lecture Notes in Computer Science*, pp. 592–601, Springer.

Christopher M. Bishop (2006), *Pattern Recognition and Machine Learning*, Springer.

Joao Gama and Mohamed Medhat Gaber, editors (2007), *Learning from Data Streams: Processing Techniques in Sensor Networks*, Springer.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom (2001), *Database Systems: The Complete Book*, Prentice Hall.

A. K. Jain, M. N. Murty, and P. J. Flynn (1999), Data clustering: a review, *ACM Computing Surveys*, 31(3):264–323, ISSN 0360-0300.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner (1998), Gradient-Based Learning Applied to Document Recognition, *Proceedings of the IEEE*, 86(11):2278–2324.

R Development Core Team (2005), *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, URL `http://www.R-project.org`.

Brian D. Ripley (1996), *Pattern recognition and neural networks*, Cambridge University Press, Cambridge.

P. Viola and M. Jones (2001), Rapid object detection using a boosted cascade of simple features, in *IEEE Computer Vision and Pattern Recognition*, volume 1, pp. 511–518.

Ian H. Witten and Eibe Frank (2005), *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, San Francisco, 2 edition.

Arkadiusz Wojna and Lukasz Kowalski (2008), *Rseslib: Programmer's Guide*, uRL: http://rsproject.mimuw.edu.pl.

Marcin Wojnarski (2007), Absolute Contrasts in Face Detection with AdaBoost Cascade, in *Rough Sets and Knowledge Technology*, volume 4481 of *Lecture Notes in Computer Science*, pp. 174–180, Springer.