Processing Data Streams with the RapidMiner Streams Plugin

Christian Bockermann and Hendrik Blom
Technical University of Dortmund
Artificial Intelligence Group
{christian.bockermann,hendrik.blom}@udo.edu

Abstract

In various applications we face a plethora of data that is often growing continuously. Such data arize in monitoring settings such as server log files, manufacturing processes, sensor networks or high volume news feeds such as twitter. Analysis of such data is different to the traditional batch setting that RapidMiner initially has been designed for.

In this work we present the streams library – a simple and easy to use framework to continuously process streaming data. It comes with the Streams Plugin, integrating its streaming capabilities into the RapidMiner suite.

We give an overview of the architecture of the streams library and its RapidMiner integration and demonstrate its usefulness for processing very large and continuous data in several use cases.

1 Introduction

More and more applications rely on dynamic data that is produced in realtime and at a high volume. Scientific experiments, network traffic, sensor networks in manifacturing processes or message services are examples of such applications. Often the data in these applications is outdated quickly and reactions need to be applied in near to realtime. An example is given by Google's news search, which uses a dynamic index for searching even more recent news articles. In other scenarios an on-time analysis might save resources as irrelevant data can quickly be detected and discarded. Analysis in such dynamic data settings is different to the traditional batch setting that RapidMiner has initially been designed for.

Continuous data poses several challenges for data analysts: The data are often produced at large volume and require continuous processing to provide

up-to-date prediction models or summaries. Such models or statistics need to be accessible at anytime. For preprocessing that data only limited resources with regard to memory, CPU and I/O is available. Recent advances such as Google's Map/Reduce paradigm address these by large scale parallelization of batch processes [1, 2]. While this scales well with the large amounts of data at hand, it does not tackle the problem of processing data *continuously*.

To catch up with the reqirements of large scale and continuous data, online algorithms have recently received a lot of attention. Various algorithms have been proposed for online quantile computation [3, 4], frequent itemset mining [5, 6, 7], clustering [8, 9] or classification [10].

1.1 Our Contributions

In this work we introduce the streams library, a small software framework that focuses on online processing of data and its adaption into RapidMiner as the Streams Plugin. The streams framework provides a thin abstraction layer to facilitate online data processing whereas the Streams Plugin uses a generic wrapper approach[1] to build a streaming facade within RapidMiner.

The proposed library supports

1. Modelling of continuous stream processes within RapidMiner, following the *single-pass* paradigm,

2. Anytime access to services that are provided by the continuous processing and the online algorithms deployed in the process setup, and

3. Processing of large data sets using limited memory resources.

1.2 Paper Outline

The outline of this work is as follows: In Section 2 we review the problem setting and give an overview of related work and existing frameworks. Based on this we derive some basic building blocks for a modeling data stream processes (Section 3). In Section 4 we present the streams API which provides implementations to these building blocks, and present the Streams Plugin that integrates these into RapidMiner in Section 5. Finally we summarize the ideas behind the streams library and give an outlook on future work.

[1] RapidMiner operators are automatically generated using the RapidMiner Beans library [11], which allows for the implementation of operators by following the JavaBeans convention and using simple Java annotations.

2 Problem and Related Work

With nowadays data volume, the traditional batch processing model quickly reaches the resource limitations of single workstations. Even applying a previously created prediction model to a large set of examples can quickly become impossible if the example set itself does not fit into main memory. The only cumbersome solution often is to split the data into several files and process each file separately. We will refer to this setting as the *partial batch processing*. This processing typically requires the results of the processed batches to be combined, for example by computing an average.

In some cases, the data is not even static, but continuosly produced by some data generating process. In the simplest case we might be able to write batches of that data into files and fall back to the mini batch processing approach. Therefore in this work we are more interested in continuously processing that data and provide models or services in an *anytime* manner, that is the current models or statistics can be queried at any time. We will refer to this setting as the (*continuous*) *stream processing*. When dealing with a finite source of data we can consider the *stream processing* as a special case of *partial batch processing* with a batch size of 1.

The data processing model of streaming approaches share common criteria. The framing to operate on streaming data is generally given by the following constraints/requirements:

C1 continuously processing *single items* or *small batches* of data,

C2 using only a *single pass* over the data,

C3 using *limited resources* (memory, time),

C4 provide *anytime services* (models, statistics).

This contrasts to the RapidMiner batch-processing model, where a set of examples is usually processed in its entirety and during a single execution of a RapidMiner process.

Existing Frameworks

The *partial batch processing* as well as the *stream processing* have both spawned libraries and frameworks to support either of these paradigms. With some limitations[2], the partial batch processing is already natively supported by RapidMiner by looping over files or database tables. Google's Map/Reduce

[2]Essentially the key limitation we are aware of is the maintenance of a global nominal mapping when processing examples from multiple file sources. This is required to ensure that different nominal values refer to the same double value representation and essentially requires the global nominal mapping to reside in main memory.

Processing Model	**Supporting Software** (examplary)
Batch Processing	WEKA, RapidMiner
Parallel Batch Processing	Google MapReduce, Hadoop, Radoop
Continuous Stream Processing	S4, Storm, MOA, Streams Plugin

Table 1: Different software frameworks for different processing models. Packages/libraries marked as blue are related to RapidMiner (extensions, plugins).

and the RapidMiner Radoop [2] plugin are examples supporting (parallel) partial batch processing on a large cluster backend.

Parallel batch processing is addressing the setting of fixed data and is of limited use if data is non-stationary but continuously produced, for example in monitoring applications (server log files, sensor networks). A framework that provides online analysis is the MOA library [12], which is a Java library closely related to the WEKA data mining framework [13]. MOA provides a collection of online learning algorithms with a focus on evaluation and benchmarking.

Aiming at processing high-volume data streams two environments have been proposed by Yahoo! and Twitter. Yahoo!'s S4 [14] as well as Twitter's Storm [15] framework do provide online processing and storage on large cluster infrastructures, but these do not include any online learning.

In contrast to these frameworks, the streams library focuses on defining a simple abstraction layer that allows for the definition of stream processes which can be mapped to different backend infrastructures (such as S4 or Storm).

Data Streams in RapidMiner

The focus of RapidMiner so far has been batch processing. With Radoop, this has been extended to (massive) parallel partial batch processing on top of the Hadoop clustering software. The streams library proposed in this work aims at providing *continuous stream processing* of non-stationary data. Based on this, we provide a Streams Plugin for RapidMiner to extend its processing capabilities to the continuous stream setting. Table 1 summarizes the mentioned software libraries with respect to their focused processing mode.

The streams library provides a simple execution runtime by itself whereas the Streams Plugin implements an execution environment within RapidMiner, making the implemented algorithms available in the RapidMiner suite. However, the level of abstraction provided by the streams does not limit the execution of stream processes to the streams runtime, but also aims at including large scale distributed execution environments (e.g. Storm).

3 An Abstract Stream Processing Model

In this section we introduce the basic concepts and ideas that we model within the streams framework. This mainly comprises the data flow (pipes and filters [16]), the control flow (anytime services) and the basic data structures and elements used for data processing. The objective of the very simple abstraction layer is to provide a clean and easy-to-use API to implement against. The abstraction layer is intended to cover most of the use cases with its default assumptions whereas any special use cases can generally be modelled using a combination of different building blocks of the API (e.g. queues, services).

3.1 Data Items, Streams and Processors

Figure 1 illustrates an abstract data process flow following the widely accepted pipes-and-filters pattern. A stream provides access to single elements (instances, events or examples) which are sequentially processed by one or more processing units. Throughout this paper we will refer to these elements as *data items*. Each data item represents a tuple, i.e. a set of (*key*,*value*) pairs and is required to be an atomic, self contained element. Data items from a stream may vary in their structure, i.e. may contain different numbers of (*key*,*value*) pairs.

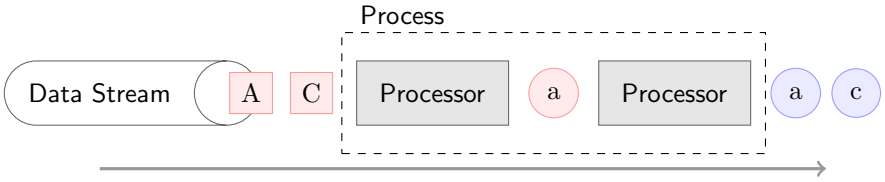

Figure 1: The general pipeline model for data processing.

A *data stream* is essentially a possibly unbounded sequence of data items. In the pipeline model, a *processor* is some processing unit that applies a function or filter to a data item. This can be the addition/removal/modification of (*key*,*value*) pairs to the current item or an update of some model/state internal to the processor. Then the outcome is delegated to the subsequent processor for further computation.

A set of processors is wrapped in a *process*, which itself is an active component that reads from a stream and applies all inner processors to each data item. The process will be running until no more data items can be read from the stream. Multiple streams and processes can be defined and executed in parallel. For communication between processes, we define *queues*. Queues can temporarily store a limited number of data items and can be fed by processors.

They do provide stream functionality as well, which allows queues to be read by other processes.

These five basic elements (*stream*, *data item*, *processor*, *process* and *queue*) already allow for modelling a wide range of data stream processes with a sequential and multi-threaded data flow. Apart from the continuous nature of the data stream source, this model of execution matches the same pipelinig idea already inherent to RapidMiner, where each processor (operator) performs some work on a complete set of data (example set).

3.2 Data Flow and Control Flow

An additional requirement of data stream processing is given by the *anytime paradigm*, which allows for querying processors for their state, prediction model or aggregated statistics at any time. We will refer to this anytime access as the *control flow*. Within the streams framework, we model these anytime functions as *services*. A service is a set of functions that is usually provided by processors and which can be invoked at any time. It is also possible to define standalone services, e.g. for lookup tables on static data. Processors may also consume services. A simple example is given by a learning algorithm, that provides predictions based on its current model as shown in Figure 2. Here, an *Add Prediction* processor acts as service consumer, adding a prediction to the data based on the prediction service provided by the learner. The data flow and control flow define two orthogonal views of the stream processing.

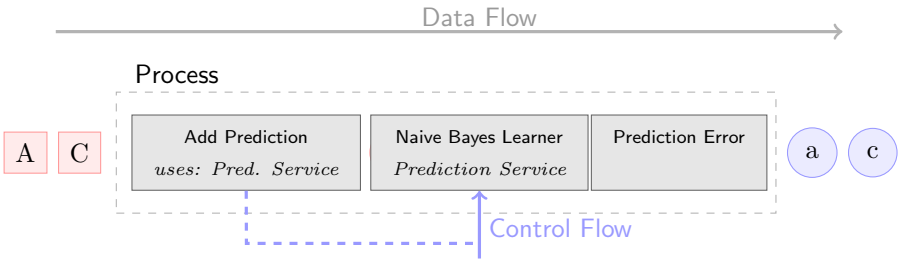

Figure 2: *The* data flow *and* control flow *in the use case of the general* test-then-train *evaluation scheme. The* Naive Bayes *processor provides a* Prediction-Service *that is used by the* Add Prediction *processor to add a prediction to each data item. The* Prediction Error *processor simply computes the error based on the prediction and the true label.*

4 The streams Library

The streams library provides a set of classes and interfaces for the elements defined in Section 3, which allows for implementing custom streams and processors. In addition it provides basic classes for reading, writing and processing data, e.g. from CSV files, SVMlight formatted data or by reading streams from an SQL database. The library consists of three packages:

1. stream-api – a small collection of interfaces and classes representing the conceptual elements outlined above.
2. stream-core – several implementations of I/O streams, processors, etc. which are of general use.
3. stream-runtime – a light-weighted execution environment that allows to define streaming processes in XML.

To a large extend we focused on developing the streams API as simple as possible using standard data structures and following design patterns and conventions like JavaBeans [17] or techniques like dependency injection [18] found in well established frameworks such as the Spring Framework [19] or Google Guice [20].

4.1 Data Items and Processors

In the stream-api data items are represented by the `stream.Data` interface, which itself is a plain Java `Map` with keys of string type and any serializable objects as values. Maps support our objective to use versatile data structures that are available and well understood in any language (e.g. dictionaries in Python or Ruby) and do provide the self-contained property. The serialization requirement allows data items to be transferred over network connections, required for running stream processes in distributed environments.

A data stream is provided by the interface `stream.io.DataStream` and basically provides a single `readNext()` method returning the next data item of the stream. In general, the data stream implementations in the streams library require a URL or a Java `InputStream` object to read from. This allows creating streams to read from file, network resources or from external data generating processes by reading from standard input.

The processor elements are defined by a simple interface `stream.Processor` that requires a single method to be implemented as shown in Figure 3.

Parameters via JavaBeans

Following the JavaBeans convention, processors are required to provide a no-args constructor and may use parameters by simply providing `get-` and `set-`

```
public interface Processor {
   /**  Method called for each item to be processed.   */
   public Data process( Data item );
}
```

Figure 3: Definition of the basic processor interface, required to implement custom processors within the streams library.

methods. The example processor shown in Figure 7 (see Appendix) outputs an alert message for every item that does not provide a (*key*,*value*) pair for a user defined key name. This simple beans convention allows for automatically registering RapidMiner operators and their corresponding parameter types. This is provided by the *RapidMiner-Beans* library.

4.2 Anytime Services

For implementing the anytime paradigm, the streams library provides a `Service` interface and a dynamic naming service which allows for registering and obtaining services or references to services. This works similarly to the standard RMI naming services included in Java, but tries to abstract from a specific implementation.

The anytime services within the streams library are implemented by extending the `Service` interface and defining any method that shall be provided in an anytime manner. As an example, the `PredictionService` is implemented by all online learning algorithms, which defines a simple `predict` method as shown in Figure 4. As soon as a processor that implements a `Service` interface is added to an experiment, it is automatically registered within the naming service.

```
public interface PredictionService extends Service {
    /** Returns the prediction for an item based
     *  on the current model                    */
    public Serializable predict( Data item );
}
```

Figure 4: A simple `PredictionService` that as is provided by all online learning algorithms that support classification.

4.3 A light-weight streams Runtime

For rapid prototyping and development purposes, the streams library implements a small multi-threaded runtime environment, which allows to define

stream processes using a simple XML document. The interpretation and structure of this XML is very similar to the notation known from frameworks like Spring [19]. A sample XML process definition of the *test-then-train* use case is provided in the Appendix (Figure 8).

The services defined within the streams API are exported via a naming service. The default naming service uses a local RMI registry, which allows for accessing services such as prediction services, or processors providing meta-data statistics (average, minimum, maximum, top-k elements) while the processes are running.

5 RapidMiner Streams-Plugin

The Streams Plugin provides RapidMiner operators for the basic building blocks of the streams API using a simple wrapper approach to directly reuse the processor implementations of most of the streams packages.

The operators of the Streams Plugin are automatically created from the processor and data stream implementations using the RapidMiner Beans library. This uses reflection and Java annotations to automatically extract and set parameter types for the wrapping operators. Figure 5 gives an overview over the Streams Plugin. The streams API serves as an abstraction layer providing implementations of the basic elements identified in Section 3.

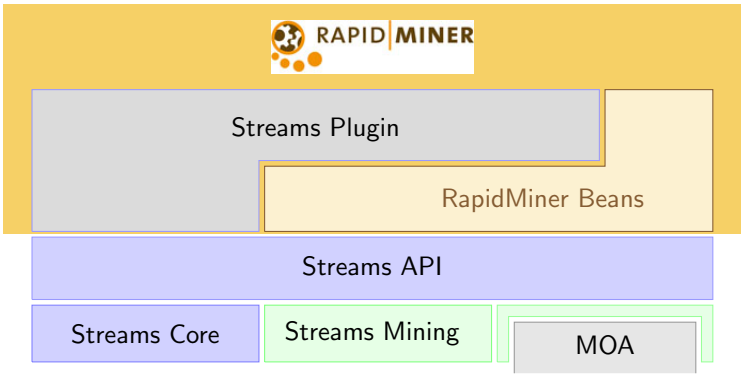

Figure 5: The architecture of the Streams Plugin, built on top of the streams API. The Streams Mining package as well as the MOA integration are work in progress and have not yet been fully integrated.

5.1 A Stream Process within RapidMiner

The elements for *streams* and *processors* are represented by RapidMiner operators. The continuous *process* is mapped to a RapidMiner subprocess. Figure 6 shows a stream process within RapidMiner.

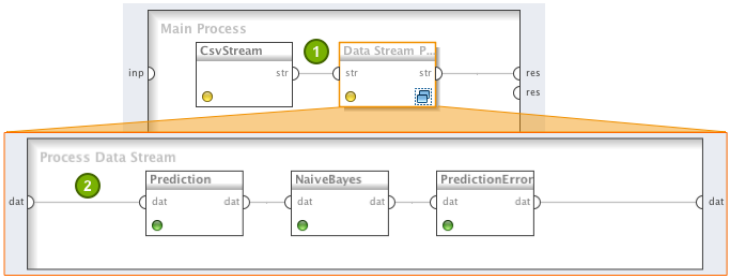

Figure 6: A continuous stream process in RapidMiner. The top process shows a stream operator and the stream process as a subprocess. The edge (1) represents a data stream object. Within the stream subprocess, IOObjects transported are single data items (edge (2)).

5.2 Control Flow and Anytime Services

Operators that relate to processors implementing a `Service` interface will be automatically registered as services within a RapidMiner naming service provided by the Streams Plugin. They can be referenced by consuming operators using a simple drop-down select box within the operator parameter view of RapidMiner.

For accessing services or monitoring from outside the continous streaming process, the Streams Plugin integrates an embedded web server that exports services via a web service interface. Currently services are exported via this embedded web server using the simple JSON-RPC protocol [21] and a local RMI registry.

6 Conclusion and Future Work

In this work we presented a simple abstraction API for modelling continuous streaming processes and implementing custom processors and services. On top of this layer of abstraction we implemented a RapidMiner Streams Plugin that integrates the stream oriented processing into the RapidMiner suite. This al-

lows for processing of continuous data or large batch data sets using sequential single item or mini batch processing.

Future work will focus on integrating MOA into the streams library, making more data mining algorithms available for online processing. In addition we seek for extending the remote access for other web service protocols like SOAP. An interesting extension will be the integration of distribution capabilities, e.g. by incorporating support for backend infrastuctures like Twitter's *Storm* framework.

Acknowledgements This work was supported by the DFG within the Collaborative Research Center on *Providing Information by Resource-Constrained Data Analysis* (SFB-876).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] Z. Prekopcsák, G. Makrai, T. Henk, and C. Gáspár-Papanek, "Radoop: Analyzing Big Data with RapidMiner and Hadoop," in *RCOMM 2011: RapidMiner Community Meeting And Conference*, Rapid-I, 2011.

[3] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *In SIGMOD*, pp. 58–66, 2001.

[4] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 286–296, ACM, 2004.

[5] M. Charikar, K. Chen, and M. Farach-colton, "Finding frequent items in data streams," pp. 693–703, 2002.

[6] T. Calders, N. Dexters, and B. Goethals, "Mining frequent itemsets in a stream," in *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, (Washington, DC, USA), pp. 83–92, IEEE Computer Society, 2007.

[7] J. Cheng, Y. Ke, and W. Ng, "Maintaining frequent itemsets over high-speed data streams," in *In Proc. of PAKDD*, 2006.

[8] M. R. Ackermann, C. Lammersen, M. Märtens, C. Raupach, C. Sohler, and K. Swierkot, "Streamkm++: A clustering algorithms for data streams," in *ALENEX* (G. E. Blelloch and D. Halperin, eds.), pp. 173–187, SIAM, 2010.

[9] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pp. 81–92, VLDB Endowment, 2003.

[10] P. Domingos and G. Hulten, "Mining High Speed Data Streams," in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '00)*, (New York, NY, USA), pp. 71–80, ACM, 2000.

[11] C. Bockermann and H. Blom, "Get some Coffee for free - Writing Operators with RapidMiner Beans," in *Proceedings of the 3rd RapidMiner Community Meeting and Conference*, 2012.

[12] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "Moa massive online analysis," 2010. `http://mloss.org/software/view/258/`.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[14] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Data Mining Workshops, International Conference on*, (CA, USA), pp. 170–177, IEEE Computer Society, 2010.

[15] N. M. et.al, "Twitter storm framework," 2011. `https://github.com/nathanmarz/storm/`.

[16] H. G. Wells, *Pattern-orientierte Software-Architektur*. Addison Wesley Verlag, 1998.

[17] "Javabeans api specification version 1.01," 1997. `http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/`.

[18] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004. `http://martinfowler.com/articles/injection.html`.

[19] "Spring framework." `http://www.springsource.org/`.

[20] "Google guice." `http://code.google.com/p/google-guice/`.

[21] "Json-rpc," 2010. `http://www.jsonrpc.org/specification`.

A Appendix

A.1 A Simple Processor

The sample code in Figure 7 shows the implementation of a very simple processor defining a `get`- and `set`-method for the `key` parameter. In addition the class is annotated using the `Description` annotation, which allows for specifying the RapidMiner Operator group into which the resulting operator will be added.

The `@Parameter` annotation marks the `setKey` method as a *mandatory* parameter and adds a description to the parameter.

```
import stream.Processor;
import stream.Data;
import stream.annotations.Description;
import stream.annotations.Parameter;

/**
 * A simple example processor
 */
@Description( group = "Data Stream.Processing.Validation" )
public class CheckMissing implements Processor {

   String key;

   @Parameter( required = true, description = "The attribute to check" )
   public void setKey( String key ){
      this.key = key;
   }

   public String getKey(){
      return key;
   }

   public Data process( Data item ){

      Serializable value = item.get( key );
      if( value == null ){
         System.err.println( "Missing value!" );
      }
      return item;
   }
}
```

Figure 7: A simple custom processor that checks if a specified key is contained within a data item. Some Java annotations has been added to the class to make it available as RapidMiner operator.

A.2 A sample XML Process Definition

```
<container>
   <stream id="data" class="stream.io.CsvStream"
           url="file:/tmp/test-data.csv" />

   <process input="data">
      <stream.learner.AddPrediction learner-ref="NaiveBayes" />

      <stream.learner.NaiveBayes id="NaiveBayes" label="play" />

      <stream.learner.evaluation.PredictionError label="play" />
   </process>
</container>
```

Figure 8: A simple experiment process in the streams runtime definition. Each XML element within the `process` element directly coresponds to a Java class implementing the `Processor` interface.

As can be seen in this simple example, the processor `NaiveBayes` is given an `id` attribute, which defines the name under which it will be registered as *PredictionService* (because it implements that service interface) in the naming system. Any other processor can now reference this services using the `id` value.

The `AddPrediction` processor in this example is a service consumer. For this it defines a single `set`-method

```
public void setLearner( PredictionService service ){
   this.predictionService = service;
}
```

which is automatically set to the service provided by the `NaiveBayes` as this is the name of the service specified in the `learner-ref` attribute. For each item, the `AddPrediction` will request a prediction using the prediction service and will add that prediction to the data item with key `@prediction`.

The naive Bayes algorithm will then update its model for each processed item, ignoring any attributes whose key starts with an "`@`" and finally the `PredictionError` processor will compute the error for keys with a prefix `@prediction`. The result will be stored in a key with prefix `@error`.