Technical Report

The streams Library for Processing Streaming Data

Christian Bockermann
Lehrstuhl für künstliche Intelligenz
Technische Universität Dortmund
christian.bockermann@udo.edu

technische universität dortmund

SFB 876 Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project C3.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

Contents

Abstract

In this report, we present the *streams* library, a generic Java-based library for designing data stream processes. The *streams* library defines a simple abstraction layer for data processing and provides a small set of online algorithms for counting and classification. Moreover it integrates existing libraries such as MOA. Processes are defined in XML files which follow the semantics and ideas of well established tools like Maven or the Spring Framework.

The *streams* library can be easily embedded into existing software, used as a standalone tool or be integrated into the RapidMiner tool using the *Streams Plugin*.

1 Introduction

In todays applications, data is continuously produced in various spots ranging from network traffic, log file data, monitoring of manufacturing processes or scientific experiments. The applications typically emit data in non-terminating data streams at a high rate, which poses tight challenges on the analysis of such streams.

Several projects within the Collaborative Research Center SFB-876 deal with data of large volume. An example is given by the FACT telescope that is associated to project C3. This telscope observes cosmic showers by tracking light that is produced by these showers in the atmosphere with a camera. These showers last about 20 nanoseconds and are recorded with a camera of 1440 pixels at a sampling rate of 2 GHz. As about 60 of these showers are currently recorded each second, a 5-minute recording interval quickly produces several gigabytes of raw data.

Other high-volume data is produced in monitoring system behavior, as performed in project A1. Here, operating systems are monitored by recording fine grained logs of system calls to catch typical usage of the system and optimize its resource utilization (e.g. for energy saving). System calls occur at a high rate and recording produces a plethora of log entries.

The project B3 focuses on monitoring (distributed) sensors in an automated manufacturing process. These sensors emit detailed information of the oven heat or milling pressure of steel production and are recorded at fine grained time intervals. Analysis of this data focuses on supervision and optimization of the production process.

1.1 From Batches to Streams

Traditional data analysis methods focus on processing fixed size batches of data and often require the data (or large portions of it) to be available in main memory. This renders most approaches useless for continuously analyzing data that arrives in steady streams. Even procedures like preprocessing or feature extraction can quickly become challenging for continuous data, especially when only limited resources with respect to memory or computing power are available.

To catch up with the reqirements of large scale and continuous data, online algorithms have recently received a lot of attention. The focus of these algorithms is to provide approximate results while limiting the memory and time resources required for computation. The constraints for the data stream setting are generally defined by allowing only a single pass over the data, and focusing on approximation schemes to deal with the inbalance of data volume to computing resources. In addition, models computed on streaming data are expected to be queriable at any time.

Various algorithms have been proposed dedicated to computational problems on data streams. Examples include online quantile computation [**?**, **?**], distinct counting of elements, frequent itemset mining [**?**, **?**, **?**], clustering [**?**, **?**] or training of classifiers on a stream [**?**].

1.2 Designing Stream Processes

In this work we introduce the *streams* library, a small software framework that provides an abstract modelling of stream processes. The objective of this framework is to establish a layer of abstraction that allows for defining stream processes at a high level, while providing the glue to connect various existing libraries such as MOA [**?**], WEKA [**?**] or the RapidMiner tool.

The set of existing online algorithms provides a valuable collection of algorithms, ideas and techniques to build upon. Based on these core elements we seek to design a process environment for implementing stream processes by combining implementations of existing online algorithms, online feature extraction methods and other preprocessing elements.

Moreover it provides a simple programming API to implement and integrate custom data processors into the designed stream processes. The level of abstraction of this programming API is intended to flawlessly integrate into existing runtime environments like *Storm* or the RapidMiner platform [**?**].

Our proposed framework supports

1. Modelling of continuous stream processes, following the *single-pass* paradigm,
2. Anytime access to services that are provided by the modeled processes and the online algorithms deployed in the process setup, and
3. Processing of large data sets using limited memory resources
4. A simple environment to implement custom stream processors and integrate these into the modelling
5. A collection of online algorithms for counting and classification
6. Incorporation of various existing libraries (e.g. MOA [**?**]) into the modeled process.

The rest of this report is structured as follows: In Section **??** 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 **??**). In Section **??** we present the *streams* API which provides implementations to these building blocks. Finally we summarize the ideas behind the *streams* library and give an outlook on future work.

2 Problem and Related Work

The traditional batch data processing aims at computations on fixed chunks of data in one or more passes. The results of these computations again form a fixed outcome that can further be used as input. A simple example is given by the computation of a prediction model based on some fixed training set. After the determination of a final model, the learning step is finished and the model is applied to deliver predictions based on the learning phase. Similar situations arise for the computation of statistics, creation of histograms, plots and the like. From a machine learning perspective, this has been the predominant approach of the last years.

Two aspects have been changed in the data we are facing today, which requires a paradigm shift: The size of data sets has grown to amounts intractable by existing batch approaches, and the rate at which data changes demands for short-term reactions to data drifts and updates of the models.

2.1 Processing Masses of Data

The former problem has been addressed by massive parallelism. With the drop of hardware prizes and evolving use of large cloud setups, computing farms are deployed to handle data at a large scale. Though parallelism and concepts for cluster computing have been studied for long, their applicability was mostly limited to specific use cases.

One of the most influential works to use computing clusters in data analysis is probably Google's revival of the *map-and-reduce* paradigm [**?**]. The concept has been around in functional programming for years and has now been transported to large-scale cluster systems consisting of thousands of compute nodes. Apache's open-source *Hadoop* implementation of a map-and-reduce platform nowadays builds a foundation for various large-scale systems.

With the revival of map-and-reduce, various machine learning algorithms have been proven to be adjustable to this new (old) way of computing.

2.2 The Problem of Continuous Data

Whereas the massive parallelism addresses the batch computation of large volumes of data, it still requires substantial processing time to re-compute prediction models, statistics or indexes once data has been changed. Therefore it does not fully reflect the demands for reacting to short-term drifts of data.

Within this work we will refer to this as the setting of *continuous data*, i.e. we consider an unbound source D of data that continuously emits data items d_i. In the following, we model that data stream as a sequence

$$D = \langle d_0, d_1, \ldots, d_i, \ldots \rangle$$

with $i \to \infty$. At any time t we want to provide some model that reflects the analysis of the items d_i with $i \leq t$. Typical tasks to compute on S are

- Given $d_i \in \mathbb{N}$ - finding the top-k most frequent values observed until t.
- For $d_i \in \mathbb{N}^p$ - find the item sets $I \subset \mathbb{N}^p$ which most frequently occurred in the d_i.
- With $d_i \subset X$, provide a classifier $c : X \to Y$, that best approximates the real distribution of labeled data $X \times Y$ (classification).
- Provide a clustering C for the data item d_i observed so far (clustering).
- Find indications on when the overall distribution of the d_i changes within the stream (concept drift detection).

Often, these tasks are further refined to models that focus on a recent sliding window of the last w data items observed, e.g. we are interested in the top-k elments of the last 5 minutes.

Algorithms for solving these tasks on static data sets exists. However, the challenging requirements in the continuous data setting are the tight limits on the resources available for computation. This can for example be realtime constraints, such as a fixed limit on the time available for processing a data item, or a bound on the memory available for computation.

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).

Several algorithms to the task mentioned above with respect to these requirements have been proposed. Regarding the counting of elements and sets of items, a variety of different approximate count algorithms based on sketches of the data have been developed in [?, ?, ?]. For statistical model, estimators for quantiles have been presented in [?, ?].

Whereas a wide range of different methods have been provided for various streaming tasks, this work aims at providing an abstract framework to integrate these different approaches into a flexible environment to build a streaming analysis based upon the existing algorithms.

Existing Frameworks

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 [**?**], which is a Java library closely related to the WEKA data mining framework [**?**]. 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* [**?**] as well as Twitter's *Storm* [**?**] 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*).

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 [**?**]), 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 Streams and Items

We consider the case of continuous streaming data being modeled as a sequence of data items. Throughout this work, we will denote a *data stream* by D and a family of such streams as D_l. A data stream D is a sequence

$$D = \langle d_0, d_1, \ldots, d_i, \ldots \rangle$$

of data items d_i. Let $A = \{A_1, \ldots, A_p\}$ be a set of attributes, then a data item d_i is a function mapping attributes to values of a domain associated with each attribute. In database notation, each d_i is a tuple of $M^p = M_{A_1} \times \ldots \times M_{A_p}$ for sets M_j. The M_j can be any discrete sets of a fixed domain as well as $M_j \subseteq \mathbb{R}$. The values of each data item are further denoted by $d_i(k)$, i.e.

$$d_i = (d_i(A_1), \ldots, d_i(A_p)).$$

The index i may reflect some time unit or a (monotonically increasing) time-like dimension, constituting the sequence of tuples. For $p = 1$ and $M_1 = \mathbb{R}$ this models a single value series with index i.

3.2 Stream Processes and Processors

A stream provides access to single *data item* (also referred to as instances, events or examples) which are sequentially processed by one or more processing units. As noted above, 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 (e.g. for sparse elements).

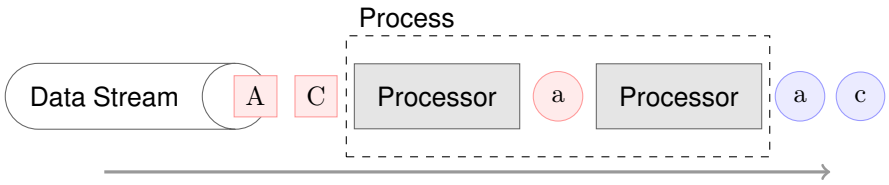

Figure 1: The general pipeline model for data processing.

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. Figure **??** illustrates an abstract data process flow following the widely accepted pipes-and-filters pattern.

Functional Representation of Processes

As a more abstract concept, each processor is a function $f : M^p \times C \to M^{p'} \times C'$, where C is some (global) state. This state C exhibits the fact, that processors may make use of a state, e.g. a learnt model. A list of processors $f_1, \ldots, f_m$ can now be modeled as a sequence of function applications

$$f = f_m \circ \ldots \circ f_1.$$

Processors are applied to single data items. A list of processors is wrapped into a *process*, which handles a stream of data (sequence). A process P can be defined as a function on sequences, built upon a list of processors

$$\begin{aligned} P &= f_m \circ \ldots \circ f_1 \\ \langle d_0, d_1, \ldots \rangle &\mapsto \langle P(d_0, C_0), P(d_1, C_1), \ldots \rangle \end{aligned}$$

for an initial state C_0 of the process. The implicit sequence of states $C_0, C_1, \ldots$ represents the evolving state of the process.

Using multiple Processes

In the *streams* framework, a process 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 executing in parallel. For communication between processes, the environment provides the notion of *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 from by other processes.

Figure **??** shows two processes being connected by a queue. The processor f_i in the first process is a simple *Enqueue* function that pushes a copy of the current data item into the queue Q. The second process constantly reads from this queue, blocking while the queue is empty.

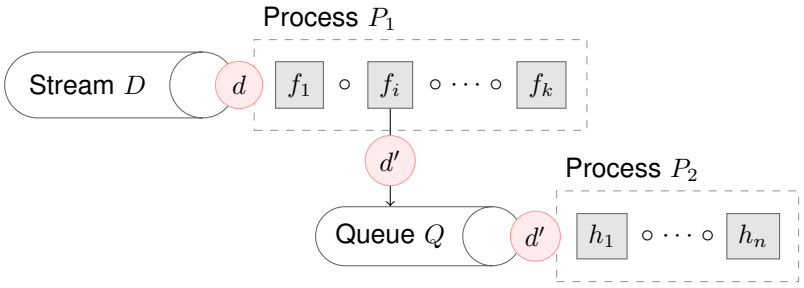

Figure 2: Two Processes P_1 and P_2 communicating via queues.

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 pipelining idea known from tools like RapidMiner, where each processor (operator) performs some work on a complete set of data (example set).

3.3 Data Flow and Control Flow

A fundamental 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, these anytime available functions are modeled as *services*. A service is a set of functions that is usually provided by processors and which can be invoked at any time. Other processors may consume/call services.

This defines a control flow that is orthonogal to the data flow. Whereas the flow of data is sequential and determined by the data source, the control flow represents the anytime property as the functions of services may be called asynchronuous to the data flow. Figure **??** shows the flow of data and service access.

Examples for services my be classifiers, which provide functions for predictions based on their current state (model); static lookup services, which provide additional data to be merged into the stream or services that can be queried for current statistical information (mean, average, counts).

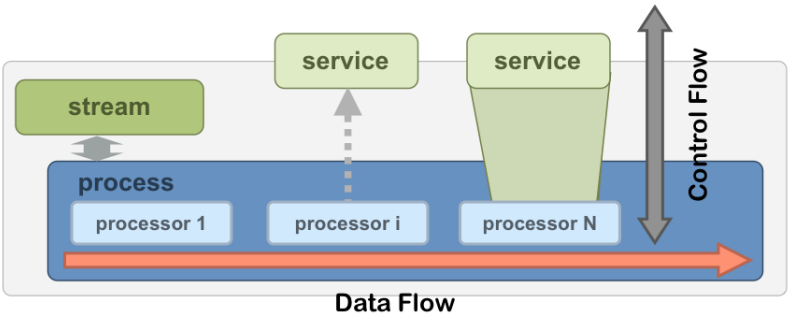

Figure 3: Orthogonal *data* and *control flow*. Processors may use services as well as export functionality by providing services.

Service References and Naming Scheme

In order to define the data flow as well as the control flow, a naming scheme is required. Each service needs to have an unique identifier assigned to it. This identifier is available within the scope of the experiment and will be used by service consumers (e.g. other processors) to reference that service.

At a higher level, when multiple experiments or stream environments are running in parallel, each experiment is associated with an identifier by itself. This imposes a hierarchical namespace of experiments and services that are defined within these experiments. The *streams* library constitutes a general naming scheme to allow for referencing services within a single experiment as well as referring to services within other (running) experiments.

A simple local reference to a service or other element (e.g. a queue) is provided by using the identifier (string) that has been specified along with the service definition. Following a URL like naming format, services within other experiments can be referenced by using the experiment identifier and the service/element identifier that is to be referred to within that experiment, e.g.

`//experiment-3/classifier-2`.

Such names will be used by the *streams* library to automatically resolve references to services and elements like queues.

4 Designing Stream Processes

The former section introduced the main conceptual elements of the *streams* library. These basic elements are used to define stream processes that can be deployed and executed with the *streams* runtime environment.

The definition of stream processes is based on simple XML files that define processes, streams and queues by XML elements that directly correspond to the elements presented in Section **??**.

4.1 Layout of a Process Environment

The *streams* library follows the concept of runtime containers, known from Java's servlet specification and similar architectures. A single container may contain multiple processes, streams and services, which are all executed in parallel. A contains is simply providing the abstract environment and a joint namespace for these elements to execute.

An example for a container definition is provided in Figure **??**. This example defines a process environment with namespace `example` which contains a single data stream with identifier `D` and a process that will be processing that stream.

```
<container id="example">
    <stream id="D" url="file:/test-data.csv" />

    <process input="D">
        <!--
            The following 'PrintData' is a simple processor that outputs each
            item to the standard output (console)
          -->
        <stream.data.PrintData />
    </process>
</container>
```

Figure 4: A simple container, defining a stream that is created from a CSV file.

The core XML elements used in the simple example of Figure **??** are `stream` and `process`, which correspond to the same conceptual elements that have previously been defined in Section **??**.

4.1.1 Defining a Stream Source

As you can see in the example above, the `stream` element is used to define a stream object that can further be processed by some processes. The `stream` element requires an `id` to be specified for referencing that stream as input for a process.

In addition, the `url` attribute is used to specify the location from which the data items should be read by the stream. There exists Java implementations for a variety of data formats that can be read. Most implementations can also handle non-file protocols like `http`. The class to use is picked by the extension of the URL (`.csv`) or by directly specifying the class name to use:

```
<stream id="D" url="http://download.jwall.org/stuff/test-data.csv"
               class="stream.io.CsvStream" />
```

Additional stream implementations for Arff files, JSON-formatted files or for reading from SQL databases are also part of the *streams* library. These implementation also differ in the number of parameters required (e.g. the database driver for SQL streams). A list of available stream implemenations can be found in Appendix **??**. The default stream

implementations also allow for the use of a `limit` parameter for stopping the stream after a given amount of data items.

4.1.2 A Stream Process

The `process` element of an XML definition is associated with a data stream by its `input` attribute. This references the stream defined with the corresponding `id` value. Processes may contain one or more *processors*, which are simple functions applied to each data item as conceptually shown in **??**.

A process will be started as a separate thread of work and will read data items from the associated stream one-by-one until no more data items can be read (i.e. `null` is returned by the stream). The general behavior of a process is shown in the pseudo-code of Algorithm **??**.

Algorithm 1 Pseudo-code for the behavior of a simple `process` element.

Require: A data stream S and a sequence $P = \langle f_1, \ldots, f_k \rangle$ of processors

```
function ProcessStream(S)
    while true do
        d := readNext(S)
        for all f ∈ P do
            d' := f(d)
            if d' = null then return null
            else
                d := d'
            end if
        end for
    end while
end function
```

4.1.3 Processing Data Items

As mentioned in the previous Section, the elements of a stream are represented by simple tuples, which are backed by a plain hashmap of keys to values. These items are the smallest units of data within the *streams* library. They are read from the stream by the process and handled as shown in Algorithm **??**.

The smallest unit of work that the *streams* library defines is a simple *processor*. A *processor* is essentially a function that acts upon a data item. The processors already available in the library are provided by Java classes, which implement the simple `stream.Processor` interface. The interface defines a single function as shown in Figure **??**.

```
public interface Processor {
    public Data process( Data item ){
        return item;
    }
}
```

Figure 5: The `Processor` interface that needs to be implemeted to create new processor elements.

5 Example Applications

In this section we will give a more detailed walk-through of some applications and use-cases that the *streams* library is used for. These examples come from various domains, such as pre-processing of scientific data, log-file processing or online-learning by integrating the MOA library.

Most of the use-cases require additional classes for reading and processing streams, e.g. stream implementations for parsing domain-specific data formats. Due to the modularity of the *streams* library, domain-specific code can easily be added and directly used within the process design.

5.1 FACT Data Analysis

The first use-case we focus on is data pre-processing in the domain of scientific data obtained from a radio telescope. The FACT project maintains a telescope for recording cosmic showers in a fine grained resolution. The telescope consists of a mirror area which has a 1440-pixel camera mounted on top. This camera is recording electric energy-impulses which in turn is measured by sampling each pixel at a rate of 2 GHz.

Based on trigger-signals, small sequences (a few nanoseconds) of these energy-impulses are recorded and stored into files. Each sequence is regarded as a single *event*. The electronics of the telescope are capable of recording about 60 events per second, resulting in a data volume of up to 10 GB of raw data that is recorded over a 5 minute runtime. The captured data of those 5-minute runs is stored in files.

The long-term objective of analyzing the FACT data comprises several tasks:

1. Identify events that represent showers.
2. Classify these events as Gamma or Hadron showers.
3. Use the Gamma events for further analysis with regard to physics stuff.

Besides the pure analysis the data has to be pre-processed in order to filter out noisy events, calibrate the data according to the state of the telescope electronic (e.g. stratify voltages over all camera pixels).

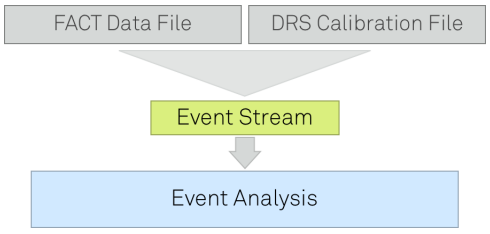

Figure 6: Stream-lined processing of events.

5.1.1 Reading FACT Data

The *fact-tools* library is an extension of the *streams* framework that adds domain specific implementations such as stream-sources and specific processors to process FACT data stored in FITS files. These processors provide the code for calibrating the data according to previously observed parameters, allow for camera image analysis (image cleaning) or for extracting features for subsequent analysis.

The XML snippet in Figure **??** defines a simple process to read raw data from a FITS file and apply a calibration step to transform that data into correct values based upon previously recorded calibration parameters.

```
<container>
    <stream id="factData" url="file:/data/2011-09-13-004.fits.gz"
        class="fact.io.FACTEventStream" />
    <process input="factData">
        <fact.io.DrsCalibration file="/data/2011-09-13-001.fits.drs.gz" />
        <!--  add further processors here  -->
    </process>
</container>
```

Figure 7: Basic process definition for reading raw FACT data and calibrating that data.

Each single event that is read from the event stream, contains the full raw, calibrated measurements of the telescope. The attributes of the data items reflect the image data, event meta information and all other data that has been recorded during the observation. Table **??** lists all the attributes of an event that are currently provided by the `FACTEventStream` class.

Name (key)	Description
`EventNum`	The event number in the stream
`TriggerNum`	The trigger number in the stream
`TriggerType`	The trigger type that caused recording of the event
`NumBoards`	
`Errors`	
`SoftTrig`	
`UnixTimeUTC`	
`BoardTime`	
`StartCellData`	
`StartCellTimeMarker`	
`Data`	The raw data array ($1440 \cdot 300 = 432000$ float values)
`TimeMarker`	
`@id`	A simple identifier providing date, run and event IDs
`@source`	The file or URL the event has been read from

Table 1: The elements available for each event.

The `@id` and `@source` attributes provide meta-information that is added by the FACT-stream implementation itself, all the other attributes are provided within the FITS data files. The `@id` attribute's value is created from the `EventNum` and date, e.g. `2011/11/27/42/8`, denoting the event 8 in run 42 on the 27th of November 2011.

5.1.2 Processors for FACT Data

Any of the existing core processors of the *streams* library can directly be applied to data items of the FACT event stream. This already allows for general applications such as adding additional data (e.g. whether data from a database).

The *fact-tools* library provides several domain specific processors that focus on the handling of FACT events. The `DrsCalibration` processor for calibrating the raw data has already been mentioned above.

Other processors included are more specifically addressing the image-analysis task:

- `fact.data.CutSlices`
  Which can be used to select a subset of the raw data array for only a excerpt of the region-of-interest[1] (ROI).
- `fact.data.SliceNormalization`
  As there is a single-valued series of floats provided for each pixel, this processor allows for normalizing the values for these series to $[0, 1]$.
- `fact.data.MaxAmplitude`
  This processor extracts a float-array of length 1440, which contains the maximum amplitude for each pixel.
- `fact.image.DetectCorePixel`
  This class implements a heuristic strategy to select the possible core-pixels of a shower, that may be contained within the event.

Installing the FACT-Plugin

The FACT-Plugin is an extension for RapidMiner. The RapidMiner open-source software is written in Java and is available for multiple environments (Windows, Unix). It can be downloaded from `http://rapid-i.com`.

The FACT-Plugin is a simple Java archive (*jar*-file) that can be found at

`http://sfb876.tu-dortmund.de/FACT/`

To install the plugin simply copy the latest `FACT-Plugin.jar` to your RapidMiner `lib/plugins` directory. After restarting RapidMiner, the plugin will be loaded and all of its additional operators will show up in the operators list.

[1] The *region of interest* is the length of the recorded time for each event, usually 300 nanoseconds, at most 1024 nanoseconds

A API Reference

A.1 Package stream.io

Reading data is usually the first step in data processing. This package provides a set of data stream sources for data files/resources in various formats.

Most of the streams provided by this package do read from URLs, which allows reading from files as well as from network URLs such as HTTP urls.

The streams provide an iterative access to the data and use the default `DataFactory` for creating data. They do usually share some common parameters supported by most of the streams such as `limit` or `username` and `password`.

A.1.1 ArffStream

This stream provides access to reading ARFF files and processing them in a stream based fashion. ARFF is a standard format for data in the machine learning community with its root in the (WEKA)[http://en.wikipedia.org/wiki/Weka_(machine_learning)] project.

Parameter	Type	Description	Required
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 8: Parameters of processor `ArffStream`

A.1.2 BlockingQueue

The *BlockingQueue* provides a simple DataStream that items can be enqueued into and read from. This allows inter-process communication between multiple active processes to be designed using data items as messages.

TODO: Write details about the queuing behavior!

A.1.3 CSVStream

This data stream source reads simple comma separated values from a file/url. Each line is split using a separator (regular expression).

Parameter	Type	Description	Required
size	int		?
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 9: Parameters of processor `BlockingQueue`

Lines starting with a hash character (`#`) are regarded to be headers which define the names of the columns.

The default split expression is `(;|,)`, but this can changed to whatever is required using the `separator` parameter.

Parameter	Type	Description	Required
keys	String[]		?
separator	String		true
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 10: Parameters of processor `CsvStream`

A.1.4 CsvUpload

This processor simply uploads all processed items to a remote (HTTP) URL. The upload consists of a single POST request for each item. The POST request contains a header line and the data line as produced by the *CsvWriter*.

A.1.5 CsvWriter

This processor appends all processed data items to a file in CSV format. The processor either adds all keys of the items or only a set of previous defined keys.

As first line, the writer emits a header line with a comma separated list of column names. This line is prepended with a `#` character.

The processor supports the creation of files with varying numbers of keys/attributes. If an item is processed with a different (larger) number of keys and the set of keys has not been defined in the `keys` parameter, a new header will be inserted into the file, signaling the header for the next items to be written.

By default the writer uses `,` as separator, which can be changed by the `separator` parameter.

The following example shows a *CsvWriter* writing to `/tmp/test.csv` using the `;` as separator. Only keys `@id` and `name` will be written:

```
<CsvWriter url="file:/tmp/test.csv" keys="@id,name"
           separator=";" />
```

Parameter	Type	Description	Required
keys	String[]	The attributes to write out, leave blank to write out all attributes.	false
url	String	The url to write to.	true
separator	String	The separator to separate columns, usually ','	false
attributeFilter	String		?
condition	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Figure 11: Parameters of processor `CsvWriter`

A.1.6 JSONStream

This data stream reads JSON objects from the source (file/url) and returns the corresponding Data items. The stream implementation expects each line of the file/url to provide a single object in JSON format.

A.1.7 JSONWriter

This processor writes the processed items to a file/url. The output is written in JSON format, where a single line of JSON output is produced for each processed item.

Parameter	Type	Description	Required
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 12: Parameters of processor `JSONStream`

The result can be parsed/read using the (JSONStream)[JSONStream.html].

Parameter	Type	Description	Required
keys	String[]	The attributes to write out, leave blank to write out all attributes.	false
url	String	The url to write to.	true
separator	String	The separator to separate columns, usually ’,’	false
attributeFilter	String		?
condition	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Figure 13: Parameters of processor `JSONWriter`

A.1.8 LineStream

This is a very simple stream that just reads from a URL line-by-line. The content of the line is stored in the attribute determined by the `key` parameter. By default the key `LINE` is used.

It also supports the specification of a simple format string that can be used to create a generic parser to populate additional fields of the data item read from the stream.

The parser format is:

```
%{IP} [%{DATE}] "%{URL}"
```

This will create a parser that is able to read line in the format

```
127.0.0.1 [2012/03/14 12:03:48 +0100] "http://example.com/index.html"
```

The outcoming data item will have the attribute `IP` set to `127.0.0.1` and the `DATE` attribute set to `2012/03/14 12:03:48 +0100`. The `URL` attribute will be set to `http://example.com/ind`
In addition, the `LINE` attribute will contain the complete line string.

Parameter	Type	Description	Required
format	String false	The format how to parse each line. Elements like key	String
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 14: Parameters of processor `LineStream`

A.1.9 LineWriter

This processor simply writes out items to a file in text format. The format of the file is by default a single line for each item. Any occurrences of new lines in the values of each item are escaped by backslash escaping.

The following example will create a single line for each item starting with some constant string, followed by the value of the items `@id` attribute, a constant string `->` and the items `name` attribute:

```
<LineWriter format="UserId: %{data.@id} -> %{data.name}"
            file="/tmp/example.out.txt"/>
```

A.1.10 ListDataStream

This class implements the DataStream interface and can be used to create a stream instance based on a collection of already available data items.

The purpose of this class is to programmatically provide a data stream implementation, e.g. for testing.

Parameter	Type	Description	Required
format	String	The format string, containing macros that are expanded for each item	true
file	File	Name of the file to write to.	true
append	boolean	Denotes whether to append to existing files or create a new file at container startup.	false
escapeNewlines	boolean	Whether to escape newlines contained in the attributes or not.	false

Figure 15: Parameters of processor `LineWriter`

A.1.11 ProcessStream

This processor executes an external process (programm/script) that produces data and writes that data to standard output. This can be used to use external programs that can read files and stream those files in any of the formats provided by the stream API.

The default format for external processes is expected to be CSV.

In the following example, the Unix command `cat` is used as an example, producing lines of some CSV file:

```
<Stream class="stream.io.ProcessStream"
        command="/bin/cat /tmp/test.csv"
        format="stream.io.CsvStream" />
```

The process is started at initialization time and the output will be read from standard input.

Parameter	Type	Description	Required
format	String	The format of the input (standard input), defaults to CSV	true
command	String	The command to execute. This command will be spawned and is assumed to output data to standard output.	true

Figure 16: Parameters of processor `ProcessStream`

A.1.12 SQLStream

This class implements a DataStream that reads items from a SQL database table. The class requires a `jdbc` URL string, a username and password as well as a `select` parameter that will select the data from the database.

The following XML snippet demonstrates the definition of a SQL stream from a database table called `TEST_TABLE`:

```
<Stream class="stream.io.SQLStream"
        url="jdbc:mysql://localhost:3306/TestDB"
        username="SA" password=""
        select="SELECT * FROM TEST_TABLE" />
```

The database connection is established using the user `SA` and no password (empty string). The above example connects to a MySQL database.

As the SQL database drivers are not part of the streams library, you will need to provide the database driver library for your database on the class path.

Parameter	Type	Description	Required
url	String	The JDBC database url to connect to.	true
select	String	The select statement to select items from the database.	true
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 17: Parameters of processor `SQLStream`

A.1.13 SQLWriter

This processor inserts processed items into a SQL database table. At initialization time, it checks for existence of the table and creates the table based on the keys of the first item if the table does not exist.

If the table exists beforehand, the table schema will be extracted and only keys with corresponding table columns will be inserted.

The following example shows the configuration of the SQLWriter to insert the keys `@id` and `attr1`, `attr2` into the table `DATA`:

```
<SQLWriter keys="@id,attr1,attr2"
           url="jdbc:hsqldb:file:/tmp/test.db"
           username="SA" password=""
           table="DATA" />
```

The parameters `url`, `username` and `password` define the connection to the database, whereas the `table` parameter defines the table into which data is to be inserted.

Dropping existing Tables The *SQLWriter* also allows to drop existing tables at initialization time by specifying the parameter `dropTable` as `true`.

Parameter	Type	Description	Required
table	String	The database table to insert items into.	true
keys	String[]	A list of attributes to insert (columns), empty string for all attributes.	false
dropTable	boolean		false
password	String	The password used to connect to the database.	false
username	String	The username used to connect to the database.	false
url	String	The JDBC database url to connect to.	true

Figure 18: Parameters of processor `SQLWriter`

A.1.14 SvmLightStream

This stream implementation provides a data stream for the SVMlight format. The SVMlight format is a simple `key:value` format for compact storage of high dimensional sparse labeled data.

It is a line oriented format. The keys are usually indexes, but this stream implementation also supports string keys.

The following snippet shows some line of an SVMlight formatted file:

```
-1.0 4:3.3 10:0.342 44:9.834 # some comment
```

In the example line above, the first column `-1.0` is the label attribute, followed by `key:value` pairs. The `#` character starts a comment that can be provided to each line.

A.1.15 SvmLightWriter

The SvmLightWriter is a simple processor implementation that writes all processed items to a file or URL using the SVMlight format. The format supports sparse vectors and a single label attribute.

The format is described in (SvmLightStream)[SvmLightStream.html].

Parameter	Type	Description	Required
sparseKey	String		?
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 19: Parameters of processor `SvmLightStream`

Parameter	Type	Description	Required
includeAnnotations	boolean		?
keys	String[]	The attributes to write out, leave blank to write out all attributes.	false
url	String	The url to write to.	true
separator	String	The separator to separate columns, usually ','	false
attributeFilter	String		?
condition	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Figure 20: Parameters of processor `SvmLightWriter`

A.1.16 TimeStream

This is a very simple stream that emits a single data item upon every read. The data item contains a single attribute `@timestamp` that contains the current timestamp (time in milliseconds).

The name of the attribute can be changed with the `key` parameter, e.g. to obtain the timestamp in attribute `@clock`:

```
<Stream class="stream.io.TimeStream" key="@clock" />
```

Parameter	Type	Description	Required
interval	String		?
password	String	The password for the stream URL (see username parameter)	false
prefix	String	An optional prefix string to prepend to all attribute names	false
limit	Long	The maximum number of items that this stream should deliver	false
username	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 21: Parameters of processor `TimeStream`

A.1.17 TreeStream

This stream is a line oriented stream that expects each line to provide a tree structure in the format

```
( ROOT ( A1 ( A1.1 ) ( A1.2 ) ) ( A2 ) )
```

Parameter	Type	Description	Required
treeAttribute	String		?

Figure 22: Parameters of processor `TreeStream`