Technical Report

The streams Library for Processing Streaming Data

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

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.

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

Todays applications produce a plethora of data on a continuous basis: In the field of astrophysics, telescopes produce raw data that quickly exceeds hundreds of gigabytes in a single day [?]. In the area of social networks, Twitter announced recently to have observed a peak of 150,000 messages per minute during the TV duell between president Obama and his challenger Romney []. In computing farms and large scale networks there is a large pile of log data about system state, system activities or working load gathered and stored that is far from being analyzed by any human, but may embody valuable information about the system [?].

The data we are considering here, is non-stationary, in the sense that new data is continously produced. This is fundamentally different to the traditional batch-data processing paradigm that has been dominating the data-analysis and data-mining world for the past decades. That paradigm shift has triggered the evolving area of stream or online analysis, which investigates algorithms for analysis of continuous data.

In order to support the design of processes that allow for online analysis we propose the *streams* library. It is a Java based abstraction layer for defining stream processes by means of simple XML definitions.

2 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.

2.1 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 4. 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 1: A simple container, defining a stream that is created from a CSV file.

The core XML elements used in the simple example of Figure 4 are `stream` and `process`, which correspond to the same conceptual elements that have previously been defined in Section 4.

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

2.1.2 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 4.2.

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 1.

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 $f \in P$ do
            $d' := f(d)$
            if $d' = null$ then return $null$
            else
                $d := d'$
            end if
        end for
    end while
end function
```

2.1.3 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 1.

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 5.

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

Figure 2: The `Processor` interface that needs to be implemeted to create new processor elements.

3 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.

3.1 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).

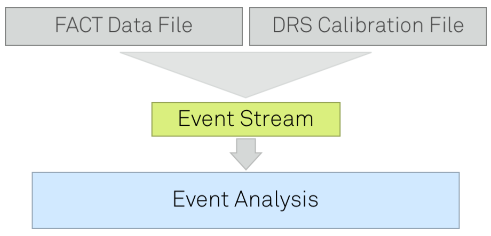

Figure 3: Stream-lined processing of events.

3.1.1 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 7 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 4: 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 1 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.

3.1.2 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