RStorm: Developing and Testing Streaming Algorithms in R

by Maurits Kaptein

Abstract Streaming data, consisting of indefinitely evolving sequences, are becoming ubiquitous in many branches of science and in various applications. Computer scientists have developed streaming applications such as Storm and the S4 distributed stream computing platform[1] to deal with data streams. However, in current production packages testing and evaluating streaming algorithms is cumbersome. This paper presents **RStorm** for the development and evaluation of streaming algorithms analogous to these production packages, but implemented fully in R. **RStorm** allows developers of streaming algorithms to quickly test, iterate, and evaluate various implementations of streaming algorithms. The paper provides both a canonical computer science example, the streaming word count, and examples of several statistical applications of **RStorm**.

Introduction

Streaming data, consisting of indefinitely and possibly time-evolving sequences, are becoming ubiquitous in many branches of science (Chu et al., 2007; Michalak et al., 2012). The omnipresence of streaming data poses new challenges for statistics and machine learning. To enable user friendly development and evaluation of algorithms dealing with data streams this paper introduces **RStorm**.

Streaming learning algorithms can informally be described as algorithms which never "look back" to earlier data arriving at $t < t'$. Streaming algorithms provide a computationally efficient way to deal with continuous data streams by summarizing all historic data into a limited set of parameters. With the current growth of available data the development of reliable streaming algorithms whose behavior is well understood is highly important (Michalak et al., 2012). For a more formal description of streaming (or online) learning see Bottou (1998). Streaming analysis however provides both numerical as well as estimation challenges. Already for simple estimators, such as sample means and variances, multiple streaming algorithms can be deployed. For more complex statistical models, closed forms to exactly minimize popular cost functions in a stream are often unavailable.

Computer scientists recently developed a series of software packages for the streaming processing of data in production environments. Frameworks such as S4 by Yahoo! (Gopalakrishna et al., 2013), and Twitter's Storm (Storm User Group, 2013) provide an infrastructure for *real-time* streaming computation of event-driven data (e.g., Babcock et al., 2002; Anagnostopoulos et al., 2012) which is scalable and reliable.

Recently, efforts have been made to facilitate easy testing and development of streaming processes within R for example with the **stream**. **stream** allows users of R to setup (or simulate) a data stream and specify data stream tasks to analyze the stream (Hahsler et al., 2014). While **stream** allows for the development and testing of streaming analysis in R, it does not have a strong link to current production environments in which streams can be utilized. Implementations of data streams in R analogous to production environments such as Twitter's Storm are currently lacking. **RStorm** models the topology structure introduced by Storm[2], to enable development, testing, and graphical representation of streaming algorithms. **RStorm** is intended as a research and development package for those wishing to implement the analysis of data streams in frameworks outside of R, but who want to utilize R's extensive plotting and data generating abilities to test their implementations. By providing an implementation of a data stream that is extremely comparable to the production code used in Storm, algorithms tested in R can easily be implemented in production environments.

Package RStorm: Counting words

In this section **RStorm** is introduced using the canonical streaming example used often for the introduction of Storm: a streaming word count. For **RStorm** the basic terminology and concepts from Storm[3] are adapted, which are briefly explained before discussing the implementation of a streaming word count in **RStorm**. The aim of the streaming word count algorithm is to, given a stream of sentences – such as posts to a web service like Twitter – count the frequency of occurrence of each word. In Storm,

[1] Not to be confused with the S4 object system used in R.

[2] This structure is very similar to the functioning of Yahoo!'s S4.

[3] The terms differ from those used by the S4 distributed stream computing platform, despite many similarities in functionality.

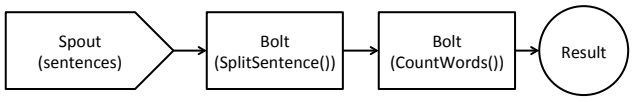

Figure 1: Graphical representation of the word count topology. This topology describes the stream that can be used to count words given an input of separate sentences.

Function	Description
`Bolt(FUNC,listen = 0,...)`	Used to create a new bolt. A bolt consists of an R function, and a specification of the bolt / spouts from which it receives tuples. The `listen` argument is used to indicate the order of bolts.
`Emit(x,...)`	Used to emit tuples from inside a bolt.
`RStorm(topology,...)`	Used to run a stream once a full topology has been specified.
`GetHash(name,...)`	Used to retrieve, inside a bolt, values from a hashmap.
`SetHash(name,data)`	Used to store, inside a bolt, values in a hashmap.
`Topology(spout,...)`	Used to create a topology by specifying the datasource (a `data.frame`) as the first spout.
`AddBolt(topology,bolt,...)`	Used to add a bolt to a stream. Once a bolt is added it receives an ID, which can be used in subsequent specification of bolts (`listen=ID`) to determine the order of the stream.
`Tuple(x,...)`	A single row `data.frame`. Used as the primary data format to be passed along the stream.

Table 1: Overview of the core functions and their primary parameters of **RStorm**.

a data stream consists of a *spout* – the data source – from which *tuples* are passed along a *topology*. The topology is a description of the spout and a series of *bolts*, which themselves are functional blocks of code. A bolt performs operations on *tuples*, the data objects that are passed between bolts in the stream. Bolts can store the results of their operations in a *local hashmap* (or database) and *emit* results (again tuples) to other bolts further down the topology. The topology, the bolts, the spout, the tuples, and the hashmap(s) together compose the most important concepts to understand a stream implemented in **RStorm**.

The *topology* is a description of the whole streaming process, and a solution to the word-count problem is given by the simple topology that is graphically presented in Figure 1. This topology describes that sentences (*tuples*) are emitted by the *spout*. These tuples – containing a full sentence – are analyzed by the first processing bolt. This first bolt, `SplitSentence(tuple)`, splits a sentence up into individual words and emits these single words as tuples. Next, these individual words are counted by the `CountWords(tuple)` bolt. The topology depicted in Figure 1 contains the core elements needed to understand the functioning of **RStorm** for a general streaming process. A topology consists of a description of the ordering of spouts and bolts in a stream. Tuples are the main data format to pass information between bolts. A call to `Emit(tuple,...)` within a bolt will make the emitted tuple available for other bolts. Table 1 summarizes the most important functions of the **RStorm** package to facilitate a stream and briefly explains their functionality.

Word count in RStorm and Java & Python

In **RStorm** the emulation of a streaming word count can be setup as follows: First, one loads **RStorm** and opens a datafile containing multiple sentences:

```
library(RStorm)  # Include package RStorm
data(sentences)
```

The data, which is a `data.frame`, will function as the spout by emitting data from it row-by-row. After defining the spout, the functional bolts need to be specified. Table 2 presents both the **RStorm** as well as the Storm implementation of the first processing bolt. The Storm implementation is done partly in Java and partly in Python. For the **RStorm** implementation the full functional code is provided, while for the Storm implementation a number of details are omitted. However, it is easy to see how an

RStorm

```
# R function that receives a tuple
# (a sentence in this case)
# and splits it into words:
SplitSentence <- function(tuple, ...)
{
  # Split the sentence into words
  words <- unlist(
    strsplit(as.character(
      tuple$sentence), " "
    ))

  # For each word emit a tuple
  for (word in words)
    Emit(Tuple(
      data.frame(word = word)),
      ...)
}
```

Java

```
/**
* A Java function which makes
* a call from the topology
* to an external Python script: */
public SplitSentence()
{
  super("Python",
    "splitsentence.py")
}

/*  The Python script (.py) */
import storm
class SplitSentenceBolt
  (storm.BasicBolt):
  def process(self, tuple)
    words =
      tuple.values[0].split(" ")
    for word in words:
      storm.emit([word])
```

Table 2: Description of the first functional bolt (`SplitSentence()`) of the word count stream in both **RStorm** (left), and Java (right).

actual Storm implementation maps to implementations in **RStorm**.

In both cases the `SplitSentence()` function receives tuples, each of which contains a sentence. Each sentence is split into words which are emitted further down the stream using the `Emit()` (or `storm.emit()`) function[4]. The second bolt is the `CountWords()` bolt, for which the **RStorm** code and the analogous Java code are presented in Table 3.

The `CountWords()` bolt receives tuples containing individual words. The **RStorm** implementation first uses the `GetHash()` function to get the entries of a hashmap / local-store called `"wordcount"`. In production systems this often is a hashmap, or, if need be, some kind of database system. In **RStorm** this functionality is implemented using `GetHash` and `SetHash` as methods to easily store and retrieve objects. If the hashmap exists, the function subsequently checks whether the word is already in the hashmap. If the word is not found, the new word is added to the hashmap with a count of 1, otherwise the current count is incremented by 1.

After specifying the two bolts the *topology* needs to be specified. The topology determines the processing order of the streaming process. Table 4 presents how this is implemented in **RStorm** and Java[5]. Each time a bolt is added to a topology in **RStorm** the user is alerted to the position of that bolt within in the stream, and the `listen` argument can be used to specify which emitted tuples a bolt should receive. Once the topology is fully specified, the stream can be run using the following call:

```
# Run the stream:
result <- RStorm(topology)
# Obtain results stored in "wordcount"
counts <- GetHash("wordcount", result)
```

The function `GetHash()` is overloaded for when the stream has finished and the function is used outside of a `Bolt`. It can be used to retrieve a hashmap once the result of a streaming process is passed to it as a second argument. The returned `counts` object is a `data.frame` containing columns of words and their associated counts and can be used to create a table of word counts.

The word count example shows the direct analogy between the implementation of a data stream in **RStorm** and in Storm. However, by focusing on an implementation that is analogous to the Storm implementation, a number of desirable R specific properties are lost. For example, the use of `for`

[4]Note that the ... argument in the **RStorm** implementation is used to manage the stream and should thus always be supplied to the processing bolt.

[5]The ... arguments in the Java implementation provide additional arguments used for managing parallelism in actual streaming applications.

RStorm

```
# R word counting function:
CountWord <- function(tuple, ...) {
  # Get the hashmap "word count"
  words <- GetHash("wordcount")
  if (tuple$word %in% words$word) {
    # Increment the word count:
    words[words$word == tuple$word,]
      $count <-
      words[words$word == tuple$word,]
        $count + 1
  } else { # If the word does not exist
    # Add the word with count 1
    words <- rbind(words, data.frame(
      word = tuple$word, count = 1))
  }

  # Store the hashmap
  SetHash("wordcount", words)
}
```

Java

```
/**
* A Java function which stores
* the word count. */
public void
  execute(Tuple tuple, ...)
{
  /* collect word from the tuple */
  String word = tuple.getString(0);

  /* get counts from hashmap */
  Integer count = counts.get(word);
  if (count == null) count = 0;

  /* increment counts */
  count++;

  /* store counts */
  counts.put(word, count);
}
```

Table 3: Description of the second functional bolt (`CountWord()`) of the word count stream in both **RStorm** (left), and Java (right).

RStorm

```
# Setting up the R topology
# Create topology:
topology <- Topology(sentences)

# Add the bolts:
topology <- AddBolt(
  topology, Bolt(
     SplitSentence, listen = 0
   )
)
topology <- AddBolt(
   topology, Bolt(
      CountWord, listen = 1
    )
)
```

Java

```
/**
* Java core topology implementation */
/* Create topology */
TopologyBuilder builder =
   new TopologyBuilder();

/* Add the spout */
builder.setSpout("sentences", ...);
/* Add the bolts */
builder.setBolt("split",
   new SplitSentence(), ... ,
   .Grouping("sentences", ...)
builder.setBolt("count",
   new WordCount(), ... ,
   .Grouping("split"), ... )
```

Table 4: Specification of the topology using **RStorm** and Java. Note: The Java code is incomplete, but used only to illustrate the similarities between the two implementations.

`(word in words) {...}` in the word count example defies the efficient vectorisation of R, and thus is relatively slow. In R one would approach the word count problem (non streaming) differently: e.g., `table(unlist(strsplit(as.character(sentences$sentence)," ")))`. The latter is much faster since it uses R properly, but the implementation in a data stream based on this code is not at all evident. Further note that while **RStorm** is modeled specifically after Storm, many other emergent streaming production packages – such as Yahoo!'s S4 – have a comparable structure. In all cases, the machinery to setup the stream can be separated from a number of functional pieces of code that update a set of parameters. These functional blocks of code are implemented in the **RStorm** bolts, and these can, after development in R, easily be implemented in production environments.

Sum of Squares method

```
var.SS <- function(x, ...) {
   # Get values stored in hashmap
   params <- GetHash("params1")
   if (!is.data.frame(params)) {
     # If no hashmap exists initialise:
     params <- list()
     params$n <- params$sum <-
       params$sum2 <- 0
   }

   # Perform updates:
   n <- params$n + 1
   S <- params$sum + as.numeric(x[1])
   SS <- params$sum2 +
     as.numeric(x[1]^2)

   # Store the hashmap:
   SetHash("params1",
     data.frame(n = n, sum = S,
       sum2 = SS))

   # Track the variance at time t:
   var <- 1/(n * (n-1)) * (n * SS - S^2) }
   TrackRow("var.SS",
     data.frame(var = var))
 }
```

Welford's method

```
var.Welford <- function(x, ...) {
   x <- as.numeric(x[1])
   params <- GetHash("params2")
   if (!is.data.frame(params)) {
     params <- list()
     params$M <- params$S <-
       params$n <- 0
   }
   n <- params$n + 1
   M <- params$M +
     (x - params$M) / (n + 1)
   S <- params$S +
     (x - params$M) * (x - M)

   SetHash("params2",
     data.frame(n = n, M = M, S = S))

   var <- ifelse(n > 1,
     S / (n - 1), 0)

   TrackRow("var.Welford",
     data.frame(var = var))
```

Table 5: Comparison of two different bolts to compute a streaming variance.

RStorm examples

The following section shows a number of streaming examples and demonstrates some of **RStorm**'s additional features.

Example 1: Comparisons of streaming variance algorithms

This first example compares two bolts for the streaming computation of a sample variance. It introduces the `TrackRow(data)` functionality implemented in **RStorm** which can be used to monitor the progress of parameters at each time point in the stream. Table 5 shows two bolts with competing implementations of streaming variance algorithms. The first bolt uses the standard Sum of Squares algorithm, while the second uses Welford's method (Welford, 1962).

After specifying the functional bolts, the topology can be specified. Creating a topology object starts with the specification of a `data.frame`. This dataframe will be iterated through row-by-row to emulate a steam.

```
t <- 1000
x <- rnorm(t, 10^8, 1)
topology <- Topology(data.frame(x = x))
```

The spout defined in the object `topology` now contains a dataframe with a single column `x`, which contains 1000 draws from a Gaussian distribution with a large mean, $\mu = 10^8$, and a comparatively small variance, $\sigma^2 = 1$. Subsequently, the bolts are added to the topology:

```
topology <- AddBolt(topology, Bolt(var.SS, listen = 0))
topology <- AddBolt(topology, Bolt(var.Welford, listen = 0))
result <- RStorm(topology)
```

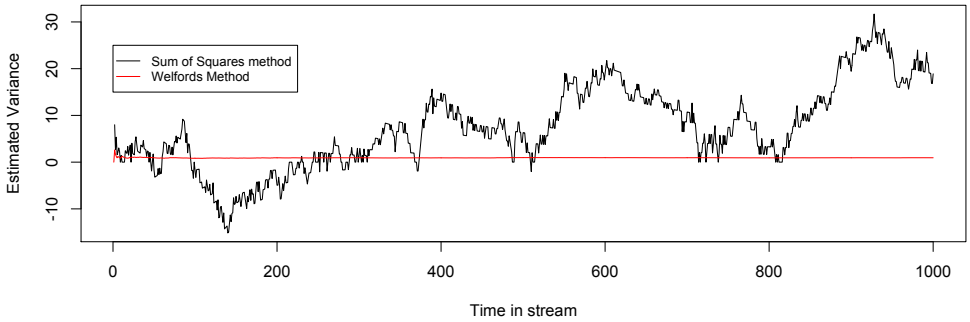

Figure 2: Comparison of two streaming variance algorithms. The sums of squares method (black) is numerically unstable when $\mu \gg \sigma^2$.

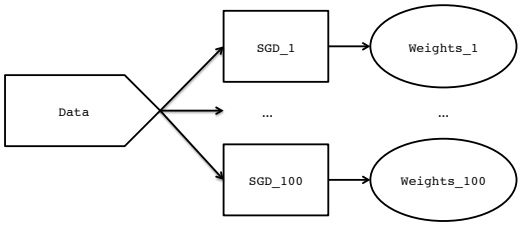

Figure 3: The DoNB SGD topology.

The `TrackRow()` function called within both functional bolts allows for inspection of the two variances at each point in time: using `TrackRow()` the values are stored for each time point. Using (e.g.,) `GetTrack("var.SS",result)` on the `result` object after running the topology allows for the creation of Figure 2.

Example 2: Online gradient descent

This example provides an implementation in **RStorm** of an logistic regression using stochastic gradient descent (SGD; e.g., Zinkevich et al., 2010), together with a Double or Nothing (DoNB; Owen and Eckles, 2012) bootstrap to estimate the uncertainty of the parameters. The functional bolt first performs the sampling needed for the DoNB bootstrap and subsequently computes the update of the feature vector $\vec{w}_t$:

```
StochasticGradientDescent <- function(tuple, learn = .5, boltID, ...) {
  if (rbinom(1, 1, .5) == 1) { # Only add the observation half of the times
    # get the set up weights for this bolt
    weights <- GetHash(paste("Weights_", boltID, sep = ""))
    if (!is.data.frame(weights)) {
      weights <- data.frame(beta = c(-1, 2))
    }
    w <- weights$beta  # get weights-vector w
    y <- as.double(tuple[1])  # get scalar y
    X <- as.double(tuple[2:3])  # get feature-vector X
    grad <- (1 / ( 1 + exp(-t(w) %*% X)) - as.double(tuple[1])) * X
    SetHash(paste("Weights_", boltID, sep = ""),
            data.frame(beta = w - learn * grad))  # save weights
  } # otherwise ignore
}
```

The dataset for this example contains 1000 dichotomous outcomes using only a single predictor:

```
n <- 1000
X <- matrix(c(rep(1, n), rnorm(n, 0, 1)), ncol = 2)
beta <- c(1, 2)
y <- rbinom(n, 1, plogis(X %*% beta))
```

The DoNB is implemented by specifying *within* the functional bolt whether or not a datapoint in the stream should contribute to the update of the weights. Using the `boltID` parameter the same functional bolt can be used multiple times in the stream, each with its own local store. The topology is specified as follows:

```
topology <- Topology(data.frame(data), .verbose = FALSE)
for (i in 1:100) {
  topology <- AddBolt(topology, Bolt(StochasticGradientDescent,
                      listen = 0, boltID = i), .verbose = FALSE)
}
```

This topology is represented graphically in Figure 3. After running the topology, the `GetHashList()` function can be used to retrieve all of the objects saved using `SetHash()` at once. This object is a list containing all the dataframes that are stored during the stream. It can be used derive the estimates of β and the 95% confidence interval: $\beta_0 = 1.33\ [.50, 2.08]$ and $\beta_1 = 2.02\ [1.34, 2.76]$ which are close to the estimates obtained using `glm`: $\vec{\beta} = \{1.25, 2.04\}$.

Example 3: The k-arm bandit

The last example presents a situation in which streaming data naturally arises: bandit problems (e.g., Whittle, 1980). In the canonical bandit problem, the *two-armed Bernoulli bandit* problem, the data stream consists of rewards $r_1, \ldots, r_t$ which are observed after playing arm $a \in \{1, 2\}$ at time t'. The goal is to find a policy to decide between the two arms at $t = t'$ such that the cumulative reward $\mathcal{R} = \sum_{i=1}^{t} r_i$ is as large as possible.

RStorm can be used to compare competing solutions to the k-armed Bernoulli bandit problem. The data is composed of the reward r at time t for each of the actions $a_1, \ldots a_k$. The function below creates such a dataframe for usage in multiple simulation runs of different policies:

```
createCounterFactuals <- function(k = 2, t = 100, p.max = .5, epsilon = .1) {
  p <- c(p.max, rep(p.max - epsilon, k - 1))
  obs <- data.frame(matrix(rbinom(t * k, 1, p), ncol = k, byrow = TRUE))
}
```

This function creates a dataframe with k arms, where arm 1 has an expected payoff of `p.max`, and the other $k - 1$ arms have an expected payoff of `p.max-epsilon`. Here we compare playing the best action (optimal play – typically unknown) to a policy called Thompson sampling (Thompson, 1933; Scott, 2010). Each datapoint z_t emitted by the spout is a vector with the possible outcome of playing arm $1, \ldots, k$ at time t.

For optimal play, the first bolt emits the reward observed by playing arm 1, and the second bolt uses a hashmap to compute the cumulative reward $\mathcal{R}_{max}$. The implementation of Thompson sampling, or Randomized Probability Matching (RPM, see Scott, 2010) uses three bolts: the first bolt (`selectRPM`) determines which arm to play given a set of estimates of the success for each arm and emits the observed reward. The second bolt (`updateRPM`) updates the estimated success of the arm that was played (using a simple beta-Bernoulli model), and the last bolt (`countRPM`) computes the cumulative reward $\mathcal{R}_{rpm}$. Both of the implementations are presented in Table 6.

The topology is graphically presented in Figure 4. The topology is initially specified using an empty dataset to enable the setup of multiple simulations:

```
topology <- Topology(data.frame())
topology <- AddBolt(topology, Bolt(selectMax, listen = 0))
topology <- AddBolt(topology, Bolt(countMax, listen = 1))
topology <- AddBolt(topology, Bolt(selectRPM, listen = 0))
topology <- AddBolt(topology, Bolt(updateRPM, listen = 3))
topology <- AddBolt(topology, Bolt(countRPM, listen = 3))
```

After specifying the bolts, the `ChangeSpout()` function is used to run the same topology with a different datasource. At each simulation run the spout is changed, and the regret, $\mathcal{R}_{max} - \mathcal{R}_{rpm}$, stored:

```
sims <- 100
regret <- rep(NA, sims)
for (i in 1:sims) {
  obs <- createCounterFactuals(k = 5, t = 10000, p.max = .5)
  topology <- ChangeSpout(topology, obs)
  result <- RStorm(topology)
  regret[i] <- GetHash("maxSum", result)$sum - GetHash("rpmSum", result)$sum
}
```

Play optimal

```
# bolt which always selects arm 1:
selectMax <- function(x, k.best = 1,
                      ...) {
  # Always select the first arm
  # and emit:
  tuple <-
    Tuple(data.frame(
      best = x[,k.best]))
  Emit(tuple, ...)
}

# bolt which counts the rewards:
countMax <- function(x, ...){
  maxSum <- GetHash("maxSum")
  if (!is.data.frame(maxSum)) {
    maxSum <- data.frame(sum = 0)
  }
  sum <- maxSum$sum + x$best
  SetHash("maxSum",
    data.frame(sum = sum))
}
```

Play Thompson

```
# bolt to select the action
selectRPM <- function(x, ...) {
  arms <- length(x)
  rpmCoefs <- GetHash("coefs")

  # if no estimates set beta priors:
  if (!is.data.frame(rpmCoefs)) {
    rpmCoefs <- data.frame(
      arm = 1:arms,
      a = rep(1, arms),
      b = rep(1, arms))
    SetHash("coefs", rpmCoefs)
  }
  # Get a random draw:
  draw <- daply(rpmCoefs, .(arm),
    .fun = function(x)
      return(rbeta(1, x$a, x$b)))
  # Determine which arm to play:
  rpm <- which.max(as.vector(draw))
  tuple <- Tuple(data.frame(
    arm = rpm, rpm = x[,rpm]))
  Emit(tuple, ...)
}

# bolt to update the estimates
updateRPM <- function(x, ...) {
  rpmCoefs <- GetHash("coefs")
  # update posteriors:
  rpmCoefs[x$arm, ]$a <-
    rpmCoefs[x$arm, ]$a + x$rpm
  rpmCoefs[x$arm, ]$b <-
    rpmCoefs[x$arm, ]$b + (1 - x$rpm)
  SetHash("coefs", rpmCoefs)
}

# bolt to count the reward
countRPM <- function(x, ...) {
  # See "countMax()" for
  # implementation.
  # Values stored in hashmap "rpmSum"
}
```

Table 6: Comparison of optimal play and Thompson sampling for the k-armed Bernoulli bandit problem.

After running 100 simulation runs with `p.max = .5` for $T = 10.000$ the average regret of Thompson sampling is 74.3, with an empirical 95% confidence interval of $[43.9, 104.5]$.

Conclusions and limitations

Datasets in all areas of science are growing increasingly large, and they are often collected continuously. There is a need for novel analysis methods which synchronize current methodological advances with the emerging opportunities of streaming data. Streaming algorithms provide opportunities to deal with extremely large and ever growing data sets in (near) real time. However, the development of

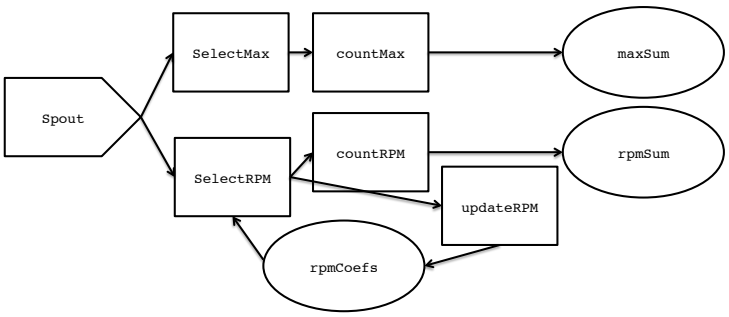

Figure 4: The *k*-armed bandit topology.

streaming algorithms for complex models is often cumbersome: the software packages that facilitate streaming processing in production environments do not provide statisticians with the simulation, estimation, and plotting tools they are used to. **RStorm** implements a streaming architecture modeled on Storm for easy development and testing of streaming algorithms in R.

In the future we intend to further develop the **RStorm** package to include a) default implementations of often occurring bolts (such as streaming means and variances of variables), and b) the ability to use, one-to-one, the bolts developed in **RStorm** in Storm. Storm provides the ability to write bolts in languages other than Java (for example Python, as demonstrated in the word count example). We hope to further develop **RStorm** such that true data streams in Storm can use functional bolts developed in R. **RStorm** is not designed as a scalable tool for production processing of data streams, and we do not believe that this is R's core strength. However, by providing the ability to test and develop functional bolts in R, and use these bolts directly in production streaming processing applications, **RStorm** aims to support users of R to quickly implement scalable and fault tolerant streaming applications.

Bibliography

C. Anagnostopoulos, D. K. Tasoulis, N. M. Adams, N. G. Pavlidis, and D. J. Hand. Online linear and quadratic discriminant analysis with adaptive forgetting for streaming classification. *Statistical Analysis and Data Mining*, 5(2):139–166, 2012. [p123]

B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems – PODS'02*, pages 1–16, New York, USA, 2002. ACM Press. [p123]

L. Bottou. Online algorithms and stochastic approximations. In D. Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. [p123]

C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems*, 19(23):281–288, 2007. [p123]

K. Gopalakrishna, F. Junqueira, M. Morel, L. Neumeyer, B. Robbins, and D. G. Ferro. S4, 2013. URL http://incubator.apache.org/s4/team/. [p123]

M. Hahsler, M. Bolanos, and J. Forrest. *stream: Infrastructure for Data Stream Mining*, 2014. URL http://CRAN.R-project.org/package=stream. R package version 1.0-0. [p123]

S. Michalak, A. DuBois, D. DuBois, S. V. Wiel, and J. Hogden. Developing systems for real-time streaming analysis. *Journal of Computational and Graphical Statistics*, 21(3):561–580, 2012. [p123]

A. B. Owen and D. Eckles. Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics*, 6(3):895–927, 2012. [p128]

S. L. Scott. A modern Bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010. [p129]

Storm User Group. Storm. Distributed and fault-tolerant realtime ccomputations, 2013. URL http://storm-project.net. [p123]

W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3–4):285–294, 1933. [p129]

B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. [p127]

P. Whittle. Multi-armed bandits and the Gittins index. *Journal of the Royal Statistical Society B*, 42(2): 143–149, 1980. [p129]

M. A. Zinkevich, A. Smola, and M. Weimer. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 23(6):1–9, 2010. [p128]

Maurits Kaptein
Department of Methodology and Statistics
Tilburg University, Tilburg, the Netherlands
Archipelstraat 13
6524LK Nijmegen, the Netherlands
+31 6 21262211
maurits@mauritskaptein.com