RevoPemaR™

Getting Started Guide

The correct bibliographic citation for this manual is as follows: Revolution Analytics, Inc. 2014. *RevoPemaR Getting Start Guide*. Revolution Analytics, Inc., Mountain View, CA.

RevoPemaR Getting Started Guide

Revolution Analytics.
2570 W. El Camino Real
Suite 222
Mountain View, CA 94040
USA.

Revised on October 6, 2014

Contents

1 Overview 1

2 Installation 1

3 Some Tips on R Reference Classes 1

4 A Tutorial Introduction to RevoPemaR 2

4.1 Using setPemaClass to Create a Class Generator 2

4.1.2 An Overview of the *methods* for *PemaMean* 3

4.1.3 The *initialize* method 3

4.1.4 The *processData* method 4

4.1.5 The *updateResults* method 4

4.1.6 The *processResults* method 5

4.1.7 The *getVarsToUse* method 5

4.2 Creating and Using a PemaMean Reference Class Object 6

4.2.1 Instantiating and Exploring a PemaMean Object 6

4.2.2 Using a *PemaMean* Object with the *pemaCompute* Function 8

4.2.3 Using a *RevoScaleR* Data Source with the *pemaCompute* Function 9

4.2.4 Using *pemaCompute* in a Distributed Compute Context 10

5 Additional Examples Using RevoPemaR 10

5.1 Basic Text Mining Examples 10

5.2 Performing By-Group Computations 10

5.3 An Iterative PEMA Algorithm: Logistic Gradient Descent 11

6 Debugging *RevoPemaR* Code 11

1 Overview

This guide is an introduction to using the ***RevoPemaR*** package to write customized scalable and distributable analytics in R. *PEMA* stands for *P*arallel *E*xternal *M*emory *A*lgorithm. An external memory algorithm is one that does not require all of the data to be in memory at one time; that is, the data can be processed in chunks. Parallel external memory algorithms are those where the chunks of data can be processed in parallel, perhaps on different nodes of a cluster. The results are then combined and processed at the end (or at the end of each iteration). The ***RevoPemaR*** package provides a framework for writing parallel external memory algorithms in R, making use of the R reference classes introduced by John Chambers in R 2.12.

The custom PEMA functions created using the ***RevoPemaR*** framework are appropriate for small and large datasets, but are particularly useful in three common situations: 1) to analyze data sets that are too big to fit in memory, 2) to create scalable data analysis routines that can be developed locally with smaller data sets, then deployed to larger data, and 3) to perform computations distributed over nodes in a cluster,

The RevoPemaR framework is portable. Code written using R on a desktop can then be deployed using the ***RevoScaleR*** package to a high performance platform, such as Hadoop and Microsoft HPC Server.

2 Installation

The ***RevoPemaR*** package is included with ***Revolution R Enterprise 7.3*** (which also contains the ***RevoScaleR*** package). The ***RevoPemaR*** package can also be installed with a standard download of R 3.1.1.

3 Some Tips on R Reference Classes

The PEMA classes used in ***RevoPemaR*** are based on R Reference Classes. We include a brief overview of some tips for using R Reference Classes here, before moving to the specifics of the PEMA classes.

R Reference Class objects are created using a generator function. This function has four important pieces of information:

- The *name of the class*.
- The inheritance of the class, that is, the *superclasses* of the class. Fields and methods of parent reference classes are inherited.
- The *fields* or member variables. These fields are accessed by reference (as in C++ or Java), so values of the fields for an object of this class can change.
- The *methods* of the class. These are functions that can be invoked by objects of this class, which might change values of the fields.

When working with reference class, here are a few tips to keep in mind:

- Reference class generators are created using `setRefClass`. For the PEMA classes, we will use a wrapper for that function, `setPemaClass`.
- Field values are changed within methods using the non-local assignment operator (`<<-`)
- Methods are documented internally with an initial line of text, rather than in an .Rd file. This information is accessed using the $help method for the generator function.
- The reference class object can be accessed in the methods using `.self`
- The parent method can be accessed using `.callSuper`
- Use the `usingMethods` call to declare that a method will be used by another method.
- The code for a method can be displayed using an instantiated reference class object, e.g. `myRefClassObj$initialize`.

4 A Tutorial Introduction to RevoPemaR

This section contains an overview of a simple example of estimating the mean of a variable using the ***RevoPemaR*** framework. The key step is in creating a `PemaMean` reference class generator function that provides the fields and methods for computing the mean using a parallel external memory algorithm. This includes creating methods to compute the sum and number of observations for each chunk of data, to update these "intermediate results", and at the end to use the intermediate results to compute the mean.

4.1 Using setPemaClass to Create a Class Generator

Start by making sure that the ***RevoPemaR*** package is loaded:

```
library(RevoPemaR)
```

To create a PEMA class generator function, use the `setPemaClass` function. It is a wrapper function for `setRefClass`. As with `setRefClass`, we will specify four basic pieces of information when using `setPemaClass`: the class name, the superclasses, the fields, and the methods. The structure looks something like this:

```
PemaMean <- setPemaClass(
      Class = "PemaMean",
      contains = "PemaBaseClass",
      fields = list( # To be written
            ),
      methods = list( # To be written
           ))
```

The `Class` is the class name of your choice. The `contains` argument must specify `PemaBaseClass` or a child class of `PemaBaseClass`. The specification of fields and methods follows.

4.1.1 Specifying the fields for PemaMean

The fields or member variables of our class represent all of the variables we need in order to compute and store our intermediate and final results. Here are the fields we will use for our "means" computation:

```
fields = list(
      sum = "numeric",
      totalObs = "numeric",
      totalValidObs = "numeric",
      mean = "numeric",
      varName = "character"
      ),
```

4.1.2 An Overview of the *methods* for *PemaMean*

There are five methods we will specify for `PemaMean`. These methods are all in the `PemaBaseClass`, and need to be overloaded for any custom analysis.

- `initialize`: initializes field values.
- `processData`: processes a chunk of data and updates field values
- `updateResults`: updates the field values of a PEMA class object from another
- `processResults`: computes the final results from the final intermediate results
- `getVarsToUse`: the names of the variables in the dataset used for analysis

4.1.3 The *initialize* method

The primary use of the i`nitialize` method is to initialize field values. The one field that is initialized with user input in this example is the name of the variable to use in the computations, `varName`. Use of the ellipses in the function signature allows for initialization values to be passed up to the parent class using `.callSuper`, the first action in the `initialize` method after the documentation. Here is the beginning of our methods listing:

```
methods = list(
    initialize = function(varName = "", ...)
    {
      'sum, totalValidObs, and mean are all initialized to 0'
      # callSuper calls the initialize method of the parent class
      callSuper(...)
```

The `pemaSetClass` function also provides additional functionality used in the `initialize` method to ensure that all of the methods of the class and its parent classes are included when an object is serialized. This is critical for distributed computing. To use this functionality, add the following to the initialize method:

```
usingMethods(.pemaMethods)
```

(If you do not want to use this functionality you can omit this line and set `includeMethods` to FALSE in `setPemaClass`.)

Now we finish the field initialization, setting the `varName` field to the input value and setting the starting values for our computations to 0, remembering to use the double-arrow non-local assignment operator to set field values:

```
            varName <<- varName
            sum <<- 0
            totalObs <<- 0
            totalValidObs <<- 0
            mean <<- 0
        },
```

4.1.4 The *processData* method

The `processData` method is the core of an external memory algorithm. It processes a chunk of data and computes intermediate results, updating the field value(s). It takes as an argument a rectangular list of data vectors; typically only the variable(s) of interest will be included. Note that in our example code we do not compute the mean within this method; that occurs after we have processed all of the data. Here we compute and update the intermediate results: the sum and number of observations:

```
processData = function(dataList)
{
    'Updates the sum and total observations from
     the current chunk of data.'
     sum <<- sum + sum(as.numeric(dataList[[varName]]),
         na.rm = TRUE)

     totalObs <<- totalObs + length(dataList[[varName]])

     totalValidObs <<- totalValidObs +
         sum(!is.na(dataList[[varName]]))
     invisible(NULL)
},
```

4.1.5 The *updateResults* method

The `updateResults` is the key method used when computations are done in parallel. Consider the following scenario:

1) The master node on a cluster assigns each worker node the task of processing a series of chunks of data.
2) The workers do so in parallel, each with their own instantiation of a reference class object. Each worker calls `processData` for each chunk of data it needs to process. In each call, the values of the fields of its reference class object are updated.
3) Now the master process must collect the information from each of the nodes, and update all of the information in a single reference class object. This is done using the `updateResults` method, which takes as an argument another instance of the reference class. The reference class object from each of the nodes is processed by the master node, resulting in the final intermediate results in the master node's reference class object's fields.

Here is the `updateResults` method for our `PemaMean`:

```
    updateResults = function(pemaMeanObj)
    {
        'Updates the sum and total observations from
         another PemaMean object.'

        sum <<- sum + pemaMeanObj$sum
        totalObs <<- totalObs + pemaMeanObj$totalObs
        totalValidObs <<- totalValidObs + pemaMeanObj$totalValidObs

        invisible(NULL)
    },
```

4.1.6 The *processResults* method

The `processResults` performs any necessary computations to produce the final result from the accumulated intermediate results. In this case it is simple; we divide the sum by the number of valid observations (assuming we have some):

```
    processResults = function()
    {
        'Returns the sum divided by the totalValidObs.'
        if (totalValidObs > 0)
        {
              mean <<- sum/totalValidObs
        }
              else
        {
              mean <<- as.numeric(NA)
        }
        return( mean )
    },
```

4.1.7 The *getVarsToUse* method

The `getVarsToUse` method specifies the names of the variables in the dataset that are used in the analysis. Specifying this information can improve performance if reading data from disk.

```
    getVarsToUse = function()
    {
        'Returns the varName.'
        varName
    }
  ) # End of methods
) # End of class generator
```

4.2 Creating and Using a PemaMean Reference Class Object

4.2.1 Instantiating and Exploring a PemaMean Object

A version of the code in the previous section is contained within the ***RevoPemaR*** package and exported, so we can directly work with the `PemaMean` generator without first running the code. We can show the names of all the methods, including those that are explicitly overridden by the `PemaMean` class:

```
PemaMean$methods()

 [1] ".pemaMethods"                 ".pemaMethods#PemaBaseClass"
 [3] "callSuper"                    "compute"
 [5] "copy"                         "copyFields"
 [7] "createReturnObject"           "export"
 [9] "field"                        "finalizeNode"
[11] "getClass"                     "getFieldList"
[13] "getRefClass"                  "getVarsToRead"
[15] "getVarsToUse"                 "getVarsToUse#PemaBaseClass"
[17] "hasConverged"                 "import"
[19] "initFields"                   "initialize"
[21] "initialize#PemaBaseClass"     "initIteration"
[23] "outputTrace"                  "processAllData"
[25] "processData"                  "processData#PemaBaseClass"
[27] "processResults"               "processResults#PemaBaseClass"
[29] "setFieldList"                 "show"
[31] "trace"                        "untrace"
[33] "updateResults"                "updateResults#PemaBaseClass"
[35] "usingMethods"
```

Some of the methods (e.g., `initIteration`, `getFieldList`) are inherited from the `PemaBaseClass`. Others (e.g., `.callSuper`, `methods`) are inherited from the base reference class generator.

We can use the `help` method with the generator function to get help on specific methods:

```
PemaMean$help(initialize)

Call:
$initialize(varName = , ...)


sum, totalValidObs, and mean are all initialized to 0
```

Next we'll generate a default `PemaMean` object, and print out the values of its fields (including those inherited):

```
meanPemaObj <- PemaMean()
meanPemaObj

Reference class object of class "PemaMean"
Reference class object of class "PemaMean" (from the global environment)
Field ".isPemaObject":
[1] TRUE
Field ".isDistributedContext":
[1] FALSE
Field ".hasOutFile":
[1] FALSE
Field ".outFile":
NULL
Field ".append":
[1] "none"
Field ".overwrite":
[1] FALSE
Field ".onlyKeepTransformedData":
[1] FALSE
Field "traceLevel":
[1] 0
Field "iter":
[1] 0
Field "maxIters":
[1] 2000
Field "useRevoScaleR":
[1] TRUE
Field ".dataInMemory":
[1] FALSE
Field ".dataInMemoryPrepared":
[1] FALSE
Field "reportProgress":
[1] 2
Field "sum":
[1] 0
Field "totalObs":
[1] 0
Field "totalValidObs":
[1] 0
Field "mean":
[1] 0
Field "varName":
[1] ""
```

We can also print out the code for a specific method using an instantiated object. For example, the initialize method of the `PemaMean` object in the ***RevoPemaR*** package is:

```
meanPemaObj$initialize

Class method definition for method initialize()
function (varName = "", ...)
{
    "sum, totalValidObs, and mean are all initialized to 0"
    callSuper(...)
    usingMethods(.pemaMethods)
    varName <<- varName
    sum <<- 0
    totalObs <<- 0
    totalValidObs <<- 0
    mean <<- 0
}
<environment: 0x000000003148ea40>

 Methods used:
    ".pemaMethods", "callSuper", "usingMethods""
```

4.2.2 Using a *PemaMean* Object with the *pemaCompute* Function

The *pemaCompute* function takes two required arguments: an “analysis” object and a data source object. The analysis object must be generated by *setPemaClass* and inherit (directly or indirectly) from *PemaBaseClass*. The data source object must be either a data frame or a data source object supported by the ***RevoScaleR*** package if it is available. The ellipses will take any additional information used in the *initialize* method.

Let’s compute a mean of some random numbers:

```
set.seed(67)
pemaCompute(pemaObj = meanPemaObj,
      data = data.frame(x = rnorm(1000)), varName = "x")

[1] 0.00504128
```

If we again print the values of the fields of our meanPemaObj, we will see the updated values:

```
meanPemaObj

Reference class object of class "PemaMean" (from the global environment)
Field ".isPemaObject":
[1] TRUE
Field ".isDistributedContext":
[1] FALSE
Field ".hasOutFile":
[1] FALSE
Field ".outFile":
NULL
Field ".append":
[1] "none"
Field ".overwrite":
[1] FALSE
Field ".onlyKeepTransformedData":
[1] FALSE
Field "traceLevel":
[1] 0
Field "iter":
[1] 1
Field "maxIters":
[1] 2000
```

```
Field "useRevoScaleR":
[1] TRUE
Field ".dataInMemory":
[1] FALSE
Field ".dataInMemoryPrepared":
[1] FALSE
Field "reportProgress":
[1] 2
Field "sum":
[1] 5.04128
Field "totalObs":
[1] 1000
Field "totalValidObs":
[1] 1000
Field "mean":
[1] 0.00504128
Field "varName":
[1] "x"
```

By default the *pemaCompute* method will reinitialize the *pemaObj*. By setting the *initPema* flag to *FALSE*, we can add more data to our analysis:

```
pemaCompute(pemaObj = meanPemaObj,
      data = data.frame(x = rnorm(1000)), varName = "x",
      initPema = FALSE)
[1] 0.001516969

meanPemaObj$totalValidObs
[1] 2000
```

Note that the number of total valid observations is now 2000.

4.2.3 Using a *RevoScaleR* Data Source with the *pemaCompute* Function

In the previous section we analysed data in memory. The ***RevoScaleR*** package provides a data source framework that allows data to be automatically extracted in chunks from data on disk or in a database. It also provides the *.xdf* file format that can very efficiently extract chunks of data.

We can use a sample .xdf file provided with the package. First we will create a data source for this file:

```
airXdf <- RxXdfData(file.path(rxGetOption("sampleDataDir"),
    "AirlineDemoSmall.xdf"))
```

Using the *meanPemaObj* created above, we compute the mean of the variable *ArrDelay* (the arrival delay in minutes). The data in this file is stored in three blocks, with 200,000 rows in each block. The *pemaCompute* function will process these chunks one at a time:

```
pemaCompute(meanPemaObj, data = airXdf, varName = "ArrDelay")

Rows Read: 200000, Total Rows Processed: 200000, Total Chunk Time: 0.009 seconds
Rows Read: 200000, Total Rows Processed: 400000, Total Chunk Time: 0.007 seconds
Rows Read: 200000, Total Rows Processed: 600000, Total Chunk Time: 0.041 seconds
[1] 11.31794
```

You can control the amount of progress reported to the console using the *reportProgress* field of *PemaBaseClass*.

4.2.4 Using *pemaCompute* in a Distributed Compute Context

RevoScaleR provides a number of distributed compute contexts, such as IBM Platform LSF, Microsoft HPC Server, and Hadoop (Cloudera and Hortonworks). Tthe same PEMA reference class object can be used with data on those platforms just by specifying the *computeContext* in the *pemaCompute* function.

5 Additional Examples Using RevoPemaR

A number of examples are provided in the *demoScripts* directory of the *RevoPemaR* package. You can find the location of this directory by entering:

```
path.package("RevoPemaR")
```

5.1 Basic Text Mining Examples

Two PEMA text mining analyses are provided as examples.

- *PemaPopularWords* will accumulate the words used in a variable containing character data. The initialize method provides a variety of arguments to fine-tune the processing. The code for the reference class generator is provided in *PemaPopularWords.R*, and examples using it in *PemaPopularWordsEx.R*.
- *PemaWordCount* will count instances of specified words in a variable containing character data. The code for the reference class generator is provided in *PemaWordCount.R*, and examples using it in *PemaWordCountEx.R*.

If you are using ***RevoScaleR*** and are interested in exploring text mining with a large dataset, instructions for downloading and code for importing Amazon reviews of fine foods is contained in the script *finefoodsImport.R*

5.2 Performing By-Group Computations

A *PemaByGroup* class is included in *RevoPemaR* to facilitate by-group computations. Examples of using this class are provided in the *PemaByGroupEx.R* demo script. It is assumed that the relevant variables for each group can fit into memory, and are then processed by arbitrary R functions. It requires that data be pre-sorted by group before processing.

5.3 An Iterative PEMA Algorithm: Logistic Gradient Descent

A simple logistic gradient descent algorithm is provided as an example of an iterative algorithm that inherits from a parent class.

The *PemaGradDescent* class generator (in *PemaGradDescent.R*) specifies a number of important methods for iterative algorithms, for example:

- *initIteration*: initializes the appropriate field values at the beginning of each iteration
- *fn*: a placeholder for the computation of the objective (loss) function for gradient descent
- *gradientFn*: a placeholder for the computation of the gradient function for gradient descent
- *hasConverged*: checks convergence criteria

This class generator cannot be used directly. A child class generator must be created that at a minimum specifies the objective function (*fn*) and gradient function (*gradientFn*). An example is provided in *PemaLogitGD.R*, showing a logistic gradient descent. A simple example of its use is in *PemaLogitGDEx.R*.

6 Debugging *RevoPemaR* Code

The R Reference Classes provide standard R debugging tools, and *trace* and *untrace* methods are provided in the base reference class. R Reference Classes can also be debugged using the visual debugger provided by the ***R Productivity Environment*** that is included with ***Revolution R Enterprise***.

The *PemaBaseClass* provides a simple way of printing trace output that is particularly useful in debugging code in a distributed environment. Calls to the *outputTrace* method within other methods will print the specified text if the *traceLevel* field value exceeds or is equal to the *outTraceLevel* argument:

```
meanPemaObj$outputTrace

Class method definition for method outputTrace()
function (text, outTraceLevel = 1)
{
    "Prints text if the traceLevel >= outTraceLevel"
    if (length(traceLevel) == 0) {
        warning("traceLevel has not been initialized.")
    }
    else if (traceLevel >= outTraceLevel) {
        cat(text)
    }
}
```