REVOLUTION ANALYTICS

Module 5: Transforming and Subsetting Data

Revolution Analytics

Transform and Subset Data

Sometimes your imported data is not in its proper format to conduct statistical analyses. For instance, perhaps one column in your data is not normalized, does not accurately reflect a scale that we want our model to reflect, or other such reasons. In any of these cases, we must transform our data, usually by executing a similar function on all values of data for that column, before we begin our statistical analysis.

Further, subsetting our data can be important as well. In a case that we are only examining two out of many variables for our analysis, it can be better to partition our original data set and select just those two columns. We can then create a new data set containing only these values, and conduct our analysis using the new set.

Transform and Subset Data

- Both transform and subset functions are executable in virtually all of its functions, including:
  - rxImport
  - rxDataStep (discussed below)
  - Analysis functions:
    - rxSummary
    - rxLinMod
    - rxLogit
    - rxGlm
    - rxCrossTabs
    - rxCube
    - rxCovCor
    - rxKmeans
- In all of these cases, basic procedures for transforming the data are the same.

Transforming with rxDataStep

- rxDataStep allows the user to make transformations, each of which specifies an R expression to be evaluated and typically is an assignment either creating a new variable or modifying an existing variable from the original data set. The original data set is then modified to a new data set, which may be the original data set overwritten, or an entirely new data set.
  - Locally, output requires that you have sufficient memory to hold the transformed or subsetted data (most likely not, that's why we're using Hadoop in the first place!)
  - On the Hadoop cluster, output may be stored in a file on the HDFS, and from there we can establish a data source allowing RRE functions to locate the data on the cluster.

Function: rxDataStep

The function call:

```
rxDataStep(inData = NULL, outFile = NULL, varsToKeep = NULL, varsToDrop = NULL,
    rowSelection = NULL, transforms = NULL, transformObjects = NULL, transformFunc = NULL,
    transformVars = NULL, transformPackages = NULL, transformEnvir = NULL, append = "none",
    overwrite = FALSE, removeMissings = FALSE, ...)
```

At a basic level, the function requires input and output data, specified as inData and OutFile. The inData may be a data source pointing to data on the HDFS, while the output data must be an XDF file located in a user-designated area.

Function: rxDataStep

- varsToKeep specifies the variables contained within the data set that the user wants to keep in the new, output data file.
- varsToDrop, alternately, allows the user to specify only the variables that are to be withheld in the new output file.
- rowSelection determines a subset of rows the user selects. For instance, dealing with time series data, the user may wish to only select data from within a certain time span.

Function: rxDataStep

- Transformations: Several different types of variable transformations may be executed by rxDataStep:
  - transforms: an expression of the form list representing the first round of variable transformations.
  - transformObjects: a named list containing objects that can be referencfed by transforms, transformsFunc, and rowSelection.
  - transformFunc: variable transformation function.
  - transformVars: character vector of input data set variables needed for the transformation function.
  - transformPackages: character vector defining additional R packages to be made available and preloaded for use in variable transformation functions.
  - transformEnvir: user-defined environment to serve as a parent to all environments developed internally and used for variable data transformation.

Example: Subset Data

Let's clarify some of these details with an example.

Let's subset our original Bank data set and create another data source to point to it. We'll take the columns balance and age to subset into a new data frame. Note that we have to create a data source pointing to this new data frame.

First, we have to define the Hadoop Distributed File System location, so that we can create data sources and show RRE where it may properly locate the data. We can accomplish this by the RxHdfsFileSystem command:

```
hdfsFS <- RxHdfsFileSystem(hostName = "master.local", port = 8020)
```

Example: Subset Data

Next, let's define the input directory, where the new input data will be located on the HDFS. In our case, we will store the information in the big data directory:

"/user/luba/share"

and let's specify a new directory, BankSubXDF, to store the newly subsetted data.

```
inputDir <- "/user/luba/share/BankSubXDF"
```

Because this directory does not already exist, we need to create a new directory on Hadoop, which we can do by using the rxHadoopMakeDir command:

```
rxHadoopMakeDir(inputDir)
```

Example: Subset Data

For this example, it is important to emphasize the flexibility that rxDataStep has in dealing with data sources. While we are only pointing to data using a data source, we can nevertheless transform the data contained in the location defined by the data source by only specifying the data source as an output value.

Therefore, we will copy the original data, BankXDF.xdf, to the new file directory we created above:

```
source <- "/home/Ben_Examples/BankXDF.xdf"
rxHadoopCopyFromLocal(source, inputDir)
```

Further, we will point to this data using the data source method:

```
BankSubDS <- RxXdfData(file = inputDir, fileSystem = hdfsFS)
```

Example: Subset Data

Now, we can run the rxDataStep function to subset the data. Notice that we define both the input and output data as data sources, specify the variables we wish to keep, and allow ScaleR to overwrite the data contained in the new HDFS directory we created above:

```
rxDataStep(inData = BankDS, outFile = BankSubDS, varsToKeep = c("balance", "age"),
           overwrite = TRUE)
# RxXdfData Source
# "/user/luba/share/BankSubXDF"
# File system: hdfs
```

Finally, let's check the variables contained in the new data source, just to make sure that our new data source has been properly defined.

```
rxGetVarInfo(BankSubDS)
# Var 1: balance, Type: integer, Low/High: (-8019, 102127)
# Var 2: age, Type: integer, Low/High: (18, 95)
```

Exercise: Subset Data

Using the Churn data source created in the previous module, ChurnDS, subset the variables such that you are only selecting n.family.members and n.devices as output. Don't worry about creating a new location for a data source as in the example above, simply overwrite the existing data source so that it will only contain these variables.

Exercise: Solution

Overwrite ChurnDS by equating it with the result of the rxDataStep function, similarly to the above example:

```
ChurnDS <- rxDataStep(inData = ChurnDS, varsToKeep = c("n.family.members", "n.devices"),
           overwrite = TRUE)
```

Transforms

Variable transformations are implemented in ScaleR using a list format.

- Original variables may be equated to a transformation, and rxDataStep is executed so that the variables can be modified.
- New variables may also be created, by declaring the name of the new variable in the transform list and defining the transformation based on prior variables in the original data set.

Example: Transforms

Let's transform the variable "balance" in the newly subsetted data set to have a minimum value at 1 rather than -8019 so that we can logarithmically transform the result. We will deal with the logarithmic transformation in a later section, however.

To define our new variable, we call transforms in the function, and define the variable transformation. In this case, we are still using the old variable, balance, to redefine the new variable, newBalance, by adding the minimum original balance plus one:

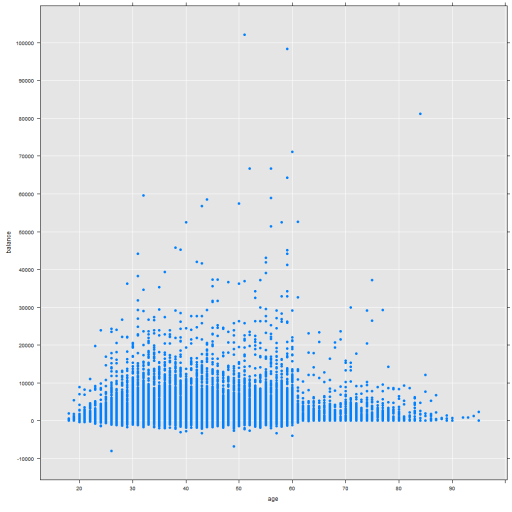
```
newDS <- rxDataStep(inData = BankSubDS,
                    transforms = list(
                      newBalance = balance + 8019 + 1))
```

Example: Transforms

Let's graphically look at the difference between balance and newBalance by age. We will use the point plotting method in the rxLinePlot call, such that a point will be plotted for every balance and age combination. First, we will look at the original balance variable:

```
rxLinePlot(balance ~ age, type = "p", data = BankSubDS)
```

Example: Transforms



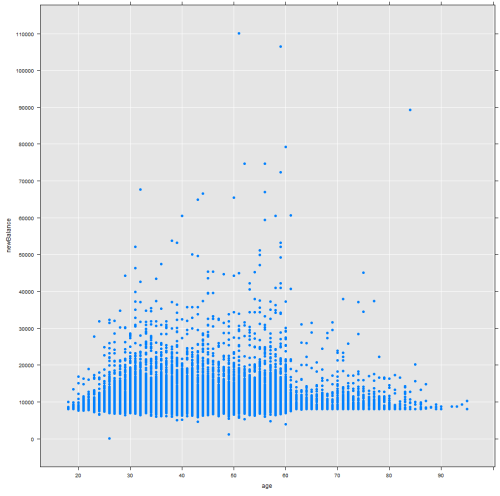
Example: Transforms

Notice that for some ages, a negative balance is declared. This is defined by the data set as owing money to the bank, most likely in the form of a loan, rather than having a positive balance. While this makes sense, it would be impossible to normalize variables using either a logarithmic or square root transformation, specifically because the functions are simply not defined to handle negative values.

Now, let's analyze our newly transformed variable, newBalance, in relation to age, again using the point plotting method:

```
rxLinePlot(newBalance ~ age, type = "p", data = newDS)
```

Example: Transforms



Example: Transforms

One may see that the bottom data values have been shifted upward by the appropriate value, and now further normalization techniques may be used to further transform the data.

Also, information on negative balances is still retained, only now a negative balance would be defined as a balance less than 8020. On the logarithmic scale, this value would simply be log(8020).

Manage Metadata and Recode Variables

Metadata is data about data, and in this instance refers to the structure or label of our big data set. How might we recode previously defined variables?

For this, we can use the function rxSetVarInfo to change variable information rather than the data values themselves.

Example: Recoding Variables

As an example for recoding previously defined variable information, let's again turn to our Bank data set. Let's redefine the variables poutcome and y to be a little more specific. Also, we can add a description to both of these variables as well:

```
newVarInfo <- list(
y = list(newName = "term.deposit",
         description = "Customer has subscribed to a term deposit"),
poutcome = list(newName = "campaign.outcome",
         description = "Outcome of prior marketing campaign"))
```

Example: Recoding Variables

Finally, using the data source BankDS, we can use the rxSetVarInfo function to redefine these variable names:

```
rxSetVarInfo(varInfo = newVarInfo, data = BankDS)
rxGetVarInfo(BankDS)
```

Sorting Data

We can sort data on the Hadoop cluster using the rxSort function, which is used to sort a data set by one or more key variables. Note that we can also subset and transform variables within this function command, as with any of the other analysis commands in ScaleR.

The function requires input data and an output file to write it to, or else writes it in memory, the variables to sort the list by, and specification on whether to sort by increasing (default) or decreasing order. Note that the function allows sorting by one or many keys, and further, a stable sorting routine is used so that, in case of ties, remaining columns are left in the same order as they were in the original data set:

```
rxSort(inData, outFile = NULL, sortByVars, decreasing = FALSE, ...)
```

Example: rxSort

As an example, let's demonstrate sorting our newly subsetted Bank data source.

We'll sort by balance and age in ascending and descending order, respectively. Therefore, the function call is defined as:

```
BankSubDS <- rxSort(inData = BankSubDS, sortByVars = c(balance, age), decreasing = c(FALSE,
    TRUE))
```

Example: rxSort

We can examine the result by looking at the first few values of the newly sorted data source:

```
head(BankSubDS)
```

Removing Duplicates While Sorting

In some cases, you may wish to just examine a sorted list of unique values (i.e. unique ID values, age values, and so forth), and therefore you may wish to remove duplicate entries from your sorted list. - When using type = “auto” or type = “mergeSort”, specify removeDupKeys = TRUE and the first record containing a unique combination of the sortByVars call is retained while subsequent matching records are omitted from the sorted results. - Nevertheless, a count of the matching records may be maintained in a new dupFreqVar output column.

Example: Removing Duplicates While Sorting

Let's again sort our subsetted Bank data source, this time removing duplicate values. We will define the function call as:

```
newDS <- rxSort(inData = BankSubDS, sortByVars = c(balance, age), decreasing = c(FALSE,
    TRUE), removeDupKeys = TRUE, dupFreqVar = "Dup_Count")
rxGetInfo(BankSubDS)
rxGetInfo(newDS)
```

The reduced data source is significantly lower than the originally subsetted data source.

Removing Duplicates While Sorting

Using the frequency counts, one may still use ScaleR analysis functions on the smaller data set specifying the weights, using the fweights argument. The same results are still obtained as using the full data set, though heavy, time consuming computations may be substantially reduced. For instance, using the rxLinMod command, we can use this method:

```
linMod <- rxLinMod(balance ~ age, data = BankSubDS, fweights = "Dup_Count")
```

Exercise: Sort

Sort the newly defined ChurnDS created above, such that n.family.members is in ascending order while n.devices is in descending order. Remove duplicate values, but define a new column, Dup_Freq, the counts the number of times a duplicate value is observed.

Exercise: Solution

Similarly to the above example, call the rxSort function using the decreasing command, specifying removeDupKeys=TRUE, and naming the new column by dupFreqVar="Dup_Freq", as shown below:

```
newDS <- rxSort(inData = ChurnDS, sortByVars=c(n.family.members, n.devices),
                decreasing = c(FALSE, TRUE), removeDupKeys=TRUE, dupFreqVar="Dup_Count")
head(newDS)
```

Recap

In this module, we examined the basics on Transforming and Subsetting data. These include

- Transforming and subsetting data with standard analyses functions
- Transforming and subsetting data with rxDataStep
- Managing Metadata and Recoding Variables with rxSetVarInfo
- Sorting data using rxSort