REVOLUTION ANALYTICS

Introduction to Revolution R Enterprise Module 6: Transform Functions

Transform Functions

Our final section for this course involves an overview of data transformations involving functional commands. We have already considered one type of data transformation earlier in the Transforming and Subsetting Module; however, this section will discuss additional methods on variable normalization.

A transformation function is a function that takes as input a list of variables and returns a list of possibly different variables. If defined and specified, a transform function is evaluated before any transformations specified in the transforms, rowSelection, or formula arguments. The list of variables to be passed to the transform function is specified as the transformVars argument.

Variable Transformations

First, let's discuss some reasons we might have for wanting to transform data initially. Perhaps our data is substantially skewed or does not represent a normal distribution, consistent with 95% of the data being contained within two standard deviations from the mean. In this case much of the theory behind our statistical tools fails, since many of these functions are based on the assumption of our data following a normal distribution.

In this case, a variable transformation which attempts to normalize the data would be in order. There are several options that we have to accomplish this, but in this module we will focus on the logarithmic data transformation.

Logarithm

Let's interpret the meaning of a data transformation. If we transform our balance variable in our Bank data, for instance, how might we interpret the result?

In this case, let's review the definition of the logarithm:

$$y = e^x, \quad x = log(y)$$

Logarithm

In this case, we must refer to Euler's number,

$$e = 2.71828$$

where each new value for balance now represents the exponent with a base equal to Euler's number.

Example: Logarithm

For example, the first customer has a balance equal to 2143 euros. To compute the log transform, we take the logarithm:

$$\log(2143) = 7.669962$$

To obtain the original, non-transformed balance value, raise Euler's number to our solution (note that in R, Euler's number may be accessed using the exp() function command):

$$e^{7.669962} = 2143$$

```
exp(7.669962)
```

```
## [1] 2143
```

Logarithmic Transformation

Now that we understand the basics of logarithms, let's consider the logarithmic transformation.

We will repeat the summary computation on the BankSubDS data source, using a logarithmic transformation. We will use the rxDataStep function again, setting transforms equal to a list of logarithmically transformed newBalance values. Remember to use the newBalance variable created earlier, since the domain of the log function is only defined for positive values:

```
infile <- file.path("data", "BankSubXDF.xdf")
BankSubDS <- RxXdfData(file = infile)

rxDataStep(inData = BankSubDS, outFile = BankSubDS, transforms = list(logBalance = log(newBalance
    overwrite = TRUE)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.006 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.005 seconds
```

Simple v. More Elaborate Transforms

We just saw an example of a very simple transformation: create a new variable that is a row-wise transformation of another. Simple, single-line transofrmations like this are appropriate for the transforms argument.

Simple v. More Elaborate Transforms

Examples of similarly simple tranformations include creating a new variable…

- that is based on binning another (factor/categorical var)
- that takes on yes/no 1/0 based on values of another,
- that is a a simple transformation of another (such as log, as.integer() or rounding),
- that is a row-wise transformation of multiple other variables

Simple v. More Elaborate Transforms

Not an example: normalizing a variable. normaliziation depends on min max of entire dataset – computation for a single row does depend on values of other rows.

Such a normalization can be executed using tarnsformFunc argument.

SImple v. More Elaborate Transforms

Tranforms argument works well for simple transformations but for more complicated transformations you'll want to use transform Functions. These are used for more elaborate transformations than can't be squeezed into a single line of code. Instead of a single list, the idea is to write a function that accomplishes all of the desired transformations and returns the results as a list.

Transform Functions

Some examples of when you might use transform functions include - merging in data - create a random sample from massive dataset - calculating a moving average - normalizing a variable

Transform Functions

Recall that normalizing a variable is a poor use case for the transforms argument. Why?

In this example, we will work though normalizing a variable as an example of transform Function. We'll normalize balance in our BankSubDS.

Transform Functions

First, we need to obtain the global min and max of the variable.

```
(minBal <- rxGetVarInfo(BankSubDS)$balance[[3]])
```

```
## [1] -8019
```

```
(maxBal <- rxGetVarInfo(BankSubDS)$balance[[4]])
```

```
## [1] 102127
```

Transform Functions

We will write a function that creates three new variables: a new normalized balance, a logical variable that is true if the normalized balance is greater than .5 and a variable that has the constant value 23 for every observation.

```
simplyNormalize <- function(dataList) {
    dataList[["normBalance"]] <- (dataList[["balance"]] - minBalance)/(maxBalance -
        minBalance)
    dataList[["moreThanHalf"]] <- dataList[["normBalance"]] > 0.5
    dataList[["newConst"]] <- rep(23, length.out = length(dataList[[1]]))
    return(dataList)
}
```

Transform Functions

Our function operates on a list, which contains one chunk of data at a time.

Elements of this list correspond with variables in our dataset. Notice the double bracket indexing to access each elemnt of the list (each avriable in our chunk of data).

Our function operates on one chunk of data at a time. Thus, to return the number 23 repeated for the length of the data, we must return 23 repeated for the length of each chunk.

Our function returns the list.

Transform Functions

We call the function that we just wrote, simplyNormalize, in our data step by using the parameter transformFunc. We use the transformObjects parameter to get the min and max values into the "sterile" environment in which the data step executes.

```
rxDataStep(inData = BankSubDS, outFile = BankSubDS, transformFunc = simplyNormalize,
    transformObjects = list(minBalance = minBal, maxBalance = maxBal), overwrite = TRUE)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.007 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.005 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.006 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.009 seconds
```

Transform Functions

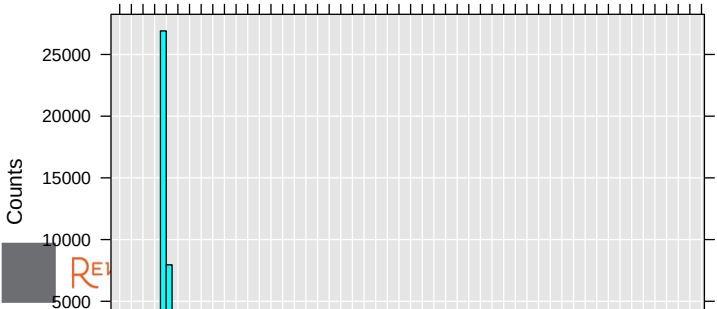
```
rxGetInfo(data = BankSubDS, getVarInfo = TRUE)
```

```
## File name: /AcademyR/Revolution_Course_Materials/modules/IntroToR/Transform_Functions_in_RRE/d
## Number of observations: 45211
## Number of variables: 7
## Number of blocks: 5
## Compression type: zlib
## Variable information:
...
```

Transform Functions

```
rxHistogram(~normBalance, data = BankSubDS, xnNumTicks = 15)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.004 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.005 seconds
## Computation time: 0.027 seconds.
```



Transform Functions

```
rxSummary(~normBalance, data = BankSubDS)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: Less than .001 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: Less than .001 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: Less than .001 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: Less than .001 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: Less than .001 seconds
## Computation time: 0.006 seconds.
```

```
## Call:
## rxSummary(formula = ~normBalance, data = BankSubDS)
##
## Summary Statistics Results for: ~normBalance
## Data: BankSubDS (RxXdfData Data Source)
## File name: data/BankSubXDF.xdf
...
```

Transform Functions Alternate

Instead of passing the min and max variables into the transformation, we could have explicitly specified the values.

```
minBal
```

```
## [1] -8019
```

```
maxBal
```

```
## [1] 102127
```

```
rxDataStep(inData = BankSubDS, outFile = BankSubDS, transforms = list(normBalance3 = (balance -
    (-8019))/(102127 - (-8019))), overwrite = TRUE)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.006 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.007 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.005 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.006 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.011 seconds
```

Transform Functions Alternate

```
plot1 <- rxHistogram(~normBalance, data = BankSubDS, xnNumTicks = 15)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.006 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.004 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.004 seconds
## Computation time: 0.028 seconds.
```

```
plot2 <- rxHistogram(~normBalance3, data = BankSubDS, xnNumTicks = 15)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.006 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.005 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.004 seconds
## Computation time: 0.029 seconds.
```

```
print(plot1, split = c(1, 1, 2, 1), more = TRUE)
print(plot2, split = c(2, 1, 2, 1), more = FALSE)
```

Exercise: Transformations

Here you get a taste of how to create a new factor variable by cutting an existing continuous variable.

Use rxDataStep() and the tranformFunc argument to create a new file while has the following properties:

- includes only Young observations with positive balance
- has a recoded factor variable that buckets observations into balance ranges of 10000 (<=10000, 10001-20000, 20001-30000, …).
- has a character variable variable stringAge which takes on the same values as age.

Exercise: Solutions

```
newTransformFunc <- function(dataList) {

    dataList[["F_balance"]] <- cut(x = dataList[["balance"]], breaks = seq(0,
        110000, 10000))
    dataList[["stringAge"]] <- as.character(dataList[["age"]])
    return(dataList)

}

smallBank <- file.path("data", "smallerBanks.xdf")

rxDataStep(inData = BankSubDS, outFile = smallBank, rowSelection = (Young ==
    "TRUE") & (balance > 0), transformFunc = newTransformFunc, overwrite = TRUE)

## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.028 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.025 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.025 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.017 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.021 seconds

rxGetInfo(smallBank, getVarInfo = TRUE, numRows = 3, startRow = 1000)

## File name: /AcademyR/Revolution_Course_Materials/modules/IntroToR/Transform_Functions_in_RRE/d
## Number of observations: 23765
```

Transforms: Factor Variables

Working with factor variables can be especially vexing in situations where you are only working with one chunk of data at a time. Let's take a look at what can go wrong.

Here we see R's factor function in a data step fail due to not seeing all levels as we process one chunk at a time

```
manualFactorRecoding <- function(dataList) {
    dataList[["F_n.devices"]] <- factor(dataList[["n.devices"]])
    return(dataList)
}
```

Let's work with the Churn Data

```
infile <- file.path("data", "ChurnData.xdf")
ChurnDS <- RxXdfData(file = infile)
```

Transforms: Factor Variables

This fails. Why?

```
rxDataStep(inData = ChurnDS, outFile = ChurnDS, transformFunc = manualFactorRecoding,
    overwrite = TRUE)

# Rows Read: 500000, Total Rows Processed: 500000, Total Chunk Time: 0.686
# seconds Existing and new value labels must be the same. There are fewer
# value labels in the new data for variable F_n.devices.  Error in
# rxCall('RxDataStep', params) :
```

Hint1: The Error message is helpful! …

Transforms: Factor Variables

This fails. Why?

```
rxDataStep(inData = ChurnDS, outFile = ChurnDS, transformFunc = manualFactorRecoding,
    overwrite = TRUE)

# Rows Read: 500000, Total Rows Processed: 500000, Total Chunk Time: 0.686
# seconds Existing and new value labels must be the same. There are fewer
# value labels in the new data for variable F_n.devices.  Error in
# rxCall('RxDataStep', params) :
```

Hint1: The Error message is helpful! Hint2: We succeeded in processing first chunk but not the second.

Exercise: Transforms: Factor Variables

We can add one argument to our call to factor() and fix this problem.
What is that one argument?

Exercise: Solution

```
manualFactorRecoding <- function(dataList) {
    dataList[["F_n.devices"]] <- factor(dataList[["n.devices"]], levels = seq(1,
        10))
    return(dataList)
}

rxDataStep(inData = ChurnDS, outFile = ChurnDS, transformFunc = manualFactorRecoding,
    overwrite = TRUE)


## Rows Read: 10000, Total Rows Processed: 10000, Total Chunk Time: 0.139 seconds
```

Transforms: Factor Variables

ScaleR provides two functions for working with factor variables: F() and rxFactors.

```
rxFactors(inData = ChurnDS, outFile = ChurnDS, factorInfo = list(F_n.devices2 = list(levels = seq
    varName = "n.devices")), overwrite = TRUE)
```

```
## Rows Read: 10000, Total Rows Processed: 10000, Total Chunk Time: 0.013 seconds
```

```
rxGetInfo(ChurnDS, getVarInfo = TRUE)
```

```
## File name: /AcademyR/Revolution_Course_Materials/modules/IntroToR/Transform_Functions_in_RRE/dc
## Number of observations: 10000
## Number of variables: 30
## Number of blocks: 1
## Compression type: zlib
## Variable information:
...
```

Transforms: Factor Variables

A rxCrossTabs run shows that the resulting levels will match those created by the Big F() function. For these two little ways, theyre essentially the same thing, but for most complicated situations, rxFactors is the way to go - more options more control.

Transforms: Factor Variables

```
rxCrossTabs(~F(n.devices):F_n.devices, data = ChurnDS)
```

```
## Rows Read: 10000, Total Rows Processed: 10000, Total Chunk Time: 0.003 seconds
## Computation time: 0.007 seconds.
```

```
## Call:
## rxCrossTabs(formula = ~F(n.devices):F_n.devices, data = ChurnDS)
##
## Cross Tabulation Results for: ~F(n.devices):F_n.devices
## Data: ChurnDS (RxXdfData Data Source)
## File name: data/ChurnData.xdf
...
```

Transforms: Factor Variables

```
rxCrossTabs(~F(n.devices):F_n.devices2, data = ChurnDS)
```

```
## Rows Read: 10000, Total Rows Processed: 10000, Total Chunk Time: 0.003 seconds
## Computation time: 0.007 seconds.
```

```
## Call:
## rxCrossTabs(formula = ~F(n.devices):F_n.devices2, data = ChurnDS)
##
## Cross Tabulation Results for: ~F(n.devices):F_n.devices2
## Data: ChurnDS (RxXdfData Data Source)
## File name: data/ChurnData.xdf
...
```

Row Selection Variable

Another option for using a transformation function relates to row selection. For instance, suppose you want to create a random sample from a massive data set. Transforming the data and creating a new random variable and using it for row selection.

We'll create a uniformly distributed random variable (ranging from 1 to 10) that will allow us to randomly select 10% of the observation.

The internal variable (.rxNumRows) gives the size of the chunk, equivalently, we could also use length() function along with one of the variables in the dataset.

Row Selection Variable

```
rxDataStep(inData = BankSubDS,
           outFile = BankSubDS,
           transforms=list(urns = as.integer(runif(.rxNumRows,1,11))),
           # OR:
           # transforms=list(urns = as.integer(runif(length(DayOfWeek),1,11))),
           overwrite=TRUE)
```

```
## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.008 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.005 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.005 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.005 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.005 seconds
```

```
rxGetInfo(BankSubDS,getVarInfo=TRUE,numRows=3)
```

```
## File name: /AcademyR/Revolution_Course_Materials/modules/IntroToR/Transform_Functions_in_RRE/do
## Number of observations: 45211
## Number of variables: 7
## Number of blocks: 5
## Compression type: zlib
## Variable information:
...
```

```
rxHistogram(~urns, BankSubDS, xNumTicks=10)
```

Exercise: Row Selection Variable

Based on this new urns variable, create a data frame in memory (BankSubDF) consisting of a random 10% subset of BankSubDS. How many observation does this data frame have?

Exercise: Solution

```
BankSubDF <- rxDataStep(inData = BankSubDS, rowSelection = urns == 1, overwrite = TRUE)

## Rows Read: 9043, Total Rows Processed: 9043, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 18086, Total Chunk Time: 0.004 seconds
## Rows Read: 9043, Total Rows Processed: 27129, Total Chunk Time: 0.006 seconds
## Rows Read: 9043, Total Rows Processed: 36172, Total Chunk Time: 0.004 seconds
## Rows Read: 9039, Total Rows Processed: 45211, Total Chunk Time: 0.004 seconds

rxGetInfo(BankSubDF, getVarInfo = TRUE, numRows = 3)

## Data frame: BankSubDF
## Number of observations: 4475
## Number of variables: 7
## Variable information:
## Var 1: balance, Type: integer, Low/High: (-3372, 102127)
## Var 2: age, Type: integer, Low/High: (18, 95)
...

rxHistogram(~urns, BankSubDF, xNumTicks = 10)

## Rows Read: 4475, Total Rows Processed: 4475, Total Chunk Time: 0.001 seconds
## Computation time: 0.004 seconds.
```

Transformations to Avoid

Transform functions are very powerful, and we recommend their use. However, there are four types of transformation that you should tread lightly when executing:

1. transformations that change the length of a variable (this includes, naturally, most model-fitting functions)
2. transformations that depend upon all observations simultaneously (because RevoScaleR works on chunks of data, such transformations will not have access to all the data at once). Examples of such transformations are matrix operations such as poly or solve.
3. transformations that have the possibility of creating different mappings of factor codes to factor labels.
4. transformations that involve sampling with replacement. Again,

Recap

- Why might one want to perform variable transformations using transform functions?
- What is so useful about the logarithmic transformation?
- How might one use transform functions for row selection?

REVOLUTION ANALYTICS

Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com, 1.855.GET.REVO, Twitter: @Revo-lutionR