Microsoft

Data Cleaning and Manipulation with Microsoft R Server

Contents

Overview

Prerequisites

In this lab we explore some data cleansing and manipulation techniques with Microsoft R Server. This module assumes the following:

- You are proficient with the R language
- You understand the concept of working with big data and R (i.e. chunking of data).
- You have completed the Introduction to Microsoft Server Module

Summary

This lab takes you through the following eclectic mix of data cleaning and manipulation methods:

- Removing unhelpful columns (i.e. high number of missing values and factor levels)
- Normalization
- Imputation
- Moving averages
- Cleaning strings (with a slant for text mining at scale) and date conversion
- dplyr with XDFs i.e. using dplyr verbs on large datasets

In each case we will use the ScaleR library of big data functions that comes with Microsoft R Server. Therefore, each solution is *scalable* i.e. will work beyond memory (RAM) limits.

This lab does not cover any predictive modelling, as we save that for another module. Instead, we provide an in-depth study of **rxDataStep** for data transformation and pre-processing.

Dropping unhelpful columns

It is often the case that once we have joined many datasets together to form a heavily de-normalised table, that many of the columns may need to be dropped from the analysis due to:

- Large number of missing values
- Too many factor levels

In addition, it can be the case that categorical variables are read into R as characters by mistake and need to converted into factors.

In this section, we outline some ScaleR functions that can help to drop unhelpful columns such as those outlined above.

We are using data from the KDD Cup 2009 based on CRM data (http://www.kdd.org/kdd-cup/view/kdd-cup-2009/Data). The dataset is provided in the R project associated with this document (orange_test_small.data). The first 190 columns of the data are numerical and the last 40 columns are categorical.

1. Open the data_cleaning R project file in Visual Studio (or RStudio), which is contained in the same folder as this lab document.

2. The R code that we describe below can be found in the **01_data_cleaning.R** file. As we are describing the code, execute it in Visual Studio (or RStudio) – feel free to experiment with the code.

3. Firstly, we convert the text data into XDF format:

   ```
   txt <- RxTextData("orange_small_test.data", delimiter = "\t")
   xdf <- RxXdfData("orange.xdf")
   rxDataStep(txt,xdf, overwrite = TRUE)
   ```

4. Next, we inspect the structure of the data and find that the last 40 columns have been read in as character strings rather than factors (categorical):

   ```
   rxGetVarInfo(xdf)
   ```

5. To convert the last 40 columns into factors we use the rxFactors function and re-look at the structure of the XDF object:

   ```
   factorCols <- paste("Var", 191:230, sep = "")
   xdfFactors <- RxXdfData("orange_factors.xdf")
   rxFactors(xdf, factorCols, outFile=xdfFactors, overwrite = TRUE)
   ```

   In the above code, we create another XDF object called xdfFactors which is the output file for the results of rxFactors function. A parameter to the rxFactors function is a vector containing the variable names to convert to factor

6. Now, we need to determine which columns have a large number of missing values. To do this we can utilize the rxSummary function, which calculates the number of valid

observations and missing observations. We can then create a vector containing the variables names where the proportion of missing values is (say) greater than 50%:

```
summData <- rxSummary( ~ ., xdfFactors)$sDataFrame
summData
missingThreshold <- 0.5
summData$missProp <- summData$MissingObs/dim(xdf)[1]
missingColumns <- summData[summData$missProp > missingThreshold,
1]
```

N.B. We can use the open source dim function on an XDF to get the dimension of the XDF file (i.e. number of rows and columns).

7. Next, we determine which factors have too many levels. To do this we use rxGetVarInfo as this contains the level information. The good thing about using rxGetVarInfo is that it is computationally cheap. In the code below we store into lotsOfLevels the names of columns that have more than 50 levels:

```
myStr <- rxGetVarInfo(xdfFactors)
levelThreshold <- 50
nLevels <- unlist(lapply(myStr, function(x) { length(x$levels)}))
ix <- which(nLevels > levelThreshold)
lotsOfLevels <- names(nLevels)[ix]
lotsOfLevels
```

8. We do not need to write out a brand new XDF with just the columns we want to keep for analysis. Instead, we can redefine out XDF object and remove the offending variables:

```
xdfToUse <- RxXdfData("orange_factors.xdf", varsToDrop =
unique(c(missingColumns, lotsOfLevels)))
rxGetVarInfo(xdfToUse)
```

Notice that our XDF object now only uses 58 columns (rather than 230) for analysis.

N.B. It is important to understand here that the underlying XDF file still contains all 230 variables, but our object is skipping over the columns defined in varsToDrop (i.e. the columns with a large number of missing values and the columns with a large number of factor levels).

Normalizing big data

1. In Visual Studio (or RStudio) open **02_data_cleaning.R**. Execute the code as we describe the important snippets below. We will firstly, normalize the data the correct way before looking at the wrong way!

2. Firstly, we set up an XDF data object:

   ```
   mortgages <- RxXdfData("mortgages.xdf")
   head(mortgages)
   ```

   N.B. we can use head/tail functions with XDF objects.

3. On big datasets where we are working on chunks (or blocks) of data rather than the whole data file, we need to compute the 'global' min and max of the dataset. If we computed the min and max for each chunk, we would not end up with a correctly normalized dataset. Thankfully, an XDF stores the low and high value for each variable:

   ```
   minScore <- rxGetVarInfo(mortgages)[["creditScore"]][["low"]]
   maxScore <- rxGetVarInfo(mortgages)[["creditScore"]][["high"]]
   ```

4. Next, we create a function that will compute the normalized scores for each chunk using the global min and max.

   ```
   # transform function to normalize
   simply_normalize <- function(lst) {
       lst[["normCreditScore"]] <- as.numeric(
       (lst[["creditScore"]] - minCreditScore) / (maxCreditScore -
   minCreditScore))
       lst[["moreThanHalf"]] <- lst[["normCreditScore"]] > 0.5
       return(lst)
   }
   ```

5. We run the rxDataStep defining the transformFunc to be simply_normalize:

   ```
   rxDataStep(inData = mortgages,
              outFile = mortgages,
              transformFunc = simply_normalize,
              transformObjects = list(minCreditScore = minScore,
              maxCreditScore = maxScore),
              append = "cols",
              overwrite = TRUE)
   ```

   Note: the transformsObjects parameter takes on a list of variables that will be made available to transform function (simply_normalize) i.e. minCreditScore and maxCreditScore.

6. Check the new variable to make sure it is in the range [0,1]:

   ```
   rxGetVarInfo(data = mortgages)
   ```

7. The wrong method for doing normalization on big data would be the following where the min and max are defined in the transform function applied to each chunk of data:

   ```
   wrong_normalization <- function(lst) {
       minCreditScore_bad <- min(lst[["creditScore"]], na.rm =
   TRUE)
   ```

```
    maxCreditScore_bad <- max(lst[["creditScore"]], na.rm =
TRUE)
    lst[["normCreditScore2"]] <- (lst[["creditScore"]] -
minCreditScore_bad) /
    (maxCreditScore_bad - minCreditScore_bad)
    lst[["normDiff"]] <- lst[["normCreditScore2"]] -
lst[["normCreditScore"]]
    return(lst)
}

rxDataStep(inData = mortgages,
           outFile = mortgages,
           transformFunc = wrong_normalization,
           overwrite = TRUE)

rxGetVarInfo(mortgages)
```

Exercises

1. In the transform function (i.e. simply_normalize) print the first 20 rows of each chunk – or better still – run the script using the debugger and inspect the objects:
    a. lst
    b. .rxChunkNum
    c. .rxIsTestChunk
2. What do you notice about the first pass through the transform function that rxDataStep makes?
3. Look at the help file for rxTransform function – what use cases would you use .rxGet and .rxSet?
4. Can you write a transform function that does z-score scaling (i.e. (column-columnMean)/columnStdDev)?
5. Can you make your transform function normalize many columns?
6. Can you write a transform function to impute missing values with the mean?

Cross Chunk Communication

Four additional variables providing information on the RevoScaleR processing are available for use in your transform functions:

• .rxStartRow: The row number from the original data that was read as the first row of the current chunk.

• .rxChunkNum: The current chunk being processed.

• .rxReadFileName: The name of the .xdf file currently being read.

• .rxIsTestChunk: Whether the chunk being processed is being processed as a test sample of data.

These are particularly useful if you need to access additional rows of data when processing a chunk. This is the case, for example, when you want to include lagged data in a calculation. The following example is a transformation function "generator" to compute a simple moving average. By creating this "function within a function" we are able to easily pass arguments into a transformation function. This one takes as arguments the number of days (or time units) for the moving average, the name of the variable that will be used to compute the moving average, and the name of the new variable to create. The transformation function computes the number of lagged observations to read in, then reads in that data to create a long data vector with the lags. The simple moving average calculations are performed and put into a new variable. The additional lagged rows are removed from the original data vector before returning from the function.

Take a look at **03_data_cleaning.R** in the R project folder that is included with this document. There are a couple of things to note:

1. We can get prior values by inspecting the XDF as it is being built

2. A transform function can be used in other ScaleR functions and not just rxDataStep e.g. in the example code, rxLinePlot uses the transform function, but equally a modelling function like rxDForest (decision forest) can also leverage a transform function.

How Transforms Are Evaluated

User-defined transforms and transform functions are evaluated in essentially the same way. An evaluation environment is constructed from the base environment together with the utils, stats, and methods packages (you can modify this basic list by setting the rxOption transformPackages), packages specified with the transformPackages argument, and any specified transformObjects, and the closure of the transform function. (User-defined transforms are combined into a single, "hidden" transform function for evaluation.) The transform function is then evaluated in this evaluation environment. Functions that are in packages that are not part of the evaluation environment can be used, but must be prefixed by their package name and two or three colons, depending on whether the function is exported.

Transformations to Avoid

Transform functions are very powerful, and we recommend their use. However, there are four types of transformation that should be avoided:
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, this is because RevoScaleR works on chunks of data, and sampling with replacement chunk by chunk is not equivalent to sampling with replacement from the full data set.

If you change the length of one variable, you will get errors from subsequent analysis functions that not all variables are the same length. If you try to change the length of all variables (essentially, performing some sort of row selection), you need to pass all of your data through the transform function, and this can be very inefficient. We therefore recommend the row selection procedures described in previous chapters. If you create a factor within a transformation function, you may get unexpected results because all of the data is not in memory at one time. It is recommended that if creating a factor within a transformation function, you always explicitly set the values and labels. For example,

```
dataList$xfac <- as.factor(dataList$x, levels = c(1, 2,3), labels =
c("One", "Two", "Three"))
```

Cleaning strings (precursor to text mining)

In this section we cover some basic pre-processing for text mining i.e. converting text to lower case, removing stop words etc. Text mining is a use case that can be scaled using Microsoft R Server as there are no cross-chunk dependencies.

1. Open **04_data_cleaning.R** and execute the code as we describe it below.

2. Firstly, we need to load the tm package (text mining). If you do not already have this in your environment that use the usual install.packages("tm") to download and install the package.

```
library(tm)
```

3. Next we need to scan in some text. For this example, we scan in the book *The Adventures of Sherlock Holmes* (by Arthur Conan Doyle) which is included in the R Project that comes with this lab.

```
conanDoyle <- "ConanDoyle.txt"
textFile <- scan(conanDoyle, what="character", sep="\n")
str(textFile)
```

4. The book has some pre-text and post-text that we do not require, therefore we find where the actual book starts and finishes using:

```
limits <- local({
  limitTerms <- c("START OF THIS PROJECT GUTENBERG EBOOK", "END 
OF THIS PROJECT GUTENBERG EBOOK")
  sapply(limitTerms, grep, textFile, USE.NAMES=FALSE)  + c(1, -
1)
})
Limits
```

5. We place the relevant part of the text (i.e. the main story) into a data.frame:

```
textFile <- textFile[limits[1]:limits[2]]
```

6. Next, we create a transform function that – for each chunk of data – will convert the text to lower case, remove punctuation, remove numbers and remove stop words.

```
tmTransform <- function(dataList) {
  x <- dataList$text
  x <- tolower(x)
  x <- removePunctuation(x)
  x <- removeNumbers(x)
  x <- removeWords(x, stopwords("english"))
  dataList$text <- x
  return(dataList)
}
```

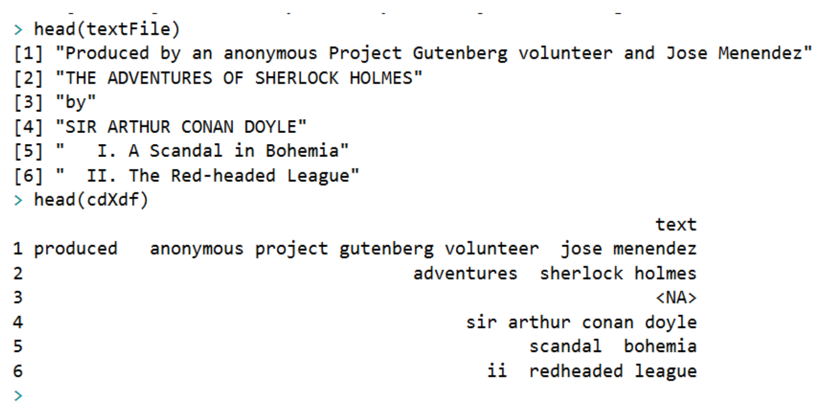
7. We create an XDF object for the output of the transformation (i.e. the cleaned text) to go:

```
cdXdf <- RxXdfData("ConanDoyle.xdf")
```

8. We run the transformation function – on each chunk of data (1000 rows in this case) – using rxDataStep. Notice, that we need to tell rxDataStep that our transform function uses the tm package – we do this using transformPackages.

```
rxDataStep(inData = data.frame(text=textFile,
stringsAsFactors=FALSE),
           outFile = cdXdf,
           overwrite=TRUE,
           transformFunc = tmTransform,
           transformPackages = "tm",
           rowsPerRead=1000)
```

9. Finally, we compare the datasets to make sure that our data cleansing has worked as expected:



```
> head(textFile)
[1] "Produced by an anonymous Project Gutenberg volunteer and Jose Menendez"
[2] "THE ADVENTURES OF SHERLOCK HOLMES"
[3] "by"
[4] "SIR ARTHUR CONAN DOYLE"
[5] "   I. A Scandal in Bohemia"
[6] "  II. The Red-headed League"
> head(cdXdf)
                                                             text
1 produced   anonymous project gutenberg volunteer  jose menendez
2                                     adventures  sherlock holmes
3                                                            <NA>
4                                          sir arthur conan doyle
5                                                scandal  bohemia
6                                            ii  redheaded league
>
```

Exercises

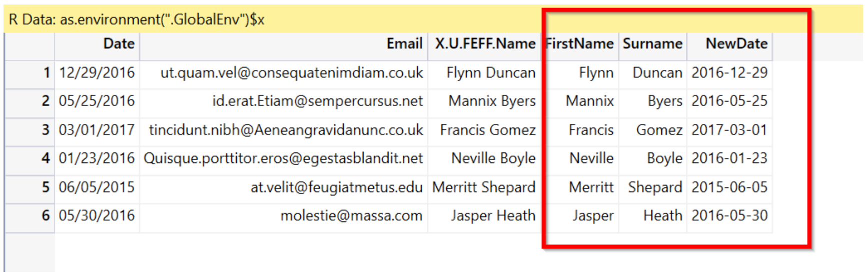
1. In the R Project that comes with this documentation is a text file called TextFileForExercise. A sample of the file is given below (note the delimiter is a pipe):

```
Name|Date|Email
Flynn Duncan|12/29/2016|ut.quam.vel@consequatenimdiam.co.uk
Mannix Byers|05/25/2016|id.erat.Etiam@sempercursus.net
Francis Gomez|03/01/2017|tincidunt.nibh@Aeneangravidanunc.co.uk
Neville Boyle|01/23/2016|Quisque.porttitor.eros@egestasblandit.net
```

The names are in “FirstName Surname” format, dates are given in character strings as “mm/dd/yyyy” format. The dates need to be converted to dates in R (i.e. as.Date function). Also the names need to be split into First Name and Surname. Create a transform function to run through an rxDataStep that creates the following 3 new variables and puts the output into an XDF:

a. FirstName
b. Surname
c. NewDate (in an R date format)

An example of what the new output should be is given below:



R Data: as.environment(".GlobalEnv")$x

	Date	Email	X.U.FEFF.Name	FirstName	Surname	NewDate
1	12/29/2016	ut.quam.vel@consequatenimdiam.co.uk	Flynn Duncan	Flynn	Duncan	2016-12-29
2	05/25/2016	id.erat.Etiam@sempercursus.net	Mannix Byers	Mannix	Byers	2016-05-25
3	03/01/2017	tincidunt.nibh@Aeneangravidanunc.co.uk	Francis Gomez	Francis	Gomez	2017-03-01
4	01/23/2016	Quisque.porttitor.eros@egestasblandit.net	Neville Boyle	Neville	Boyle	2016-01-23
5	06/05/2015	at.velit@feugiatmetus.edu	Merritt Shepard	Merritt	Shepard	2015-06-05
6	05/30/2016	molestie@massa.com	Jasper Heath	Jasper	Heath	2016-05-30

Using dplyr with XDFs

The dplyr package is a popular toolkit for data transformation and manipulation. Since its release dplyr has become a hot topic in the R community, for the way in which it streamlines and simplifies many common data manipulation tasks.

Out of the box, dplyr supports data frames, data tables (from the data.table package), and the following SQL databases: MySQL/MariaDB, SQLite, and PostgreSQL. However, a feature of dplyr is that it's *extensible*: by writing a specific backend, you can make it work with many other kinds of data sources. For example the development version of the RSQLServer package implements a dplyr backend for Microsoft SQL Server.

The dplyrXdf package implements such a backend for the xdf file format, a technology supplied as part of Revolution R Enterprise. All of the data transformation and modelling functions provided with RRE support xdf files, which allow you to break R's memory barrier: by storing the data on disk, rather than in memory, they make it possible to work with multi-gigabyte or terabyte-sized datasets.

dplyrXdf brings the benefits of dplyr to xdf files, including support for pipeline notation, all major verbs, and the ability to incorporate xdfs into dplyr pipelines. It also provides some additional benefits which are more specific to working with xdfs:

- The RevoScaleR functions require keeping track of where your data is saved. In some situations, writing a function's output to the same file as its input is allowed, while in others, it causes problems. You can often end up with many different version of the data scattered around your filesystem, introducing reproducibility problems and making it difficult to keep track of changes. dplyrXdf abstracts this task of file management away, so that you can focus on the data itself.

- Related to the above, the source xdf to a dplyrXdf pipeline is never modified. This provides a measure of security, so that even if there are bugs in your code (maybe you meant to use a mutate rather than a transmute), the original data is safe.

- Consistency of interface: functions like rxCube and rxSummary use formulas in different ways, because they are designed to do slightly different things. Similarly, many RevoScaleR functions use factors but don't automatically create those factors; or they require handholding when trying to combine factor with non-factor data. With dplyrXdf, you don't have to remember which formula syntax goes with which function, or create factors yourself. If you *do* have to create factors, it provides a new verb (factorise) to streamline this as well.

- The verbs in dplyrXdf all read from xdf files and write to xdf files. The data is thus never read entirely into memory, so a dplyrXdf pipeline will work with datasets that are arbitrarily large.

1. Open **05_data_cleaning.R** in the R Project that comes with this document. Execute the code in the R files as we go through and discuss each code block below.

   Install all the required packages and load them into your R environment:

```
install.packages("devtools")
install.packages("nycflights13")
install.packages(c("magrittr", "DBI",
"assertthat","tibble"))
devtools::install_github("RevolutionAnalytics/dplyrXdf
")
library(dplyrXdf)
library(nycflights13)
library(magrittr)
```

2. Inspect and write out some US flight data to XDF:

We want to do the following with this data:

- Select rows the first half of the 2013 year
- Compute the distance each flight travelled in km
- Compute the delay
- Convert the Airline carrier into a factor variable
- Get the mean delay and total km by each carrier
- Sort the results by decreasing mean delay

3. Using ScaleR to achieve the above we would write the following R code:

```
flights_rx1 <- rxDataStep(flightsXdf, outFile = "flights_rx1.xdf",
                          rowSelection = month <= 6 & year == 2013,
                          overwrite = TRUE)

# variable transformations
flights_rx2 <- rxDataStep(flights_rx1, outFile = "flights_rx2.xdf",
                          transforms = list(dist_km = distance * 1.6093,
                                            delay = (arr_delay + dep_delay) / 2),
                          overwrite = TRUE)

# convert carrier into a factor variable (or rxSummary will complain)
flights_rx3 <- rxFactors(flights_rx2, factorInfo = "carrier",
                         outFile = "flights_rx3.xdf", overwrite = TRUE)

# use rxSummary to get the summary table(s) (could also use rxCube twice)
flights_rx4 <- rxSummary( ~ delay:carrier + dist_km:carrier, data = flights_rx3,
                         summaryStats = c("mean", "sum"))

# extract the desired tables from the rxSummary output
flights_rx4_1 <- flights_rx4$categorical[[1]][c("carrier", "Means")]
names(flights_rx4_1)[2] <- "mean_delay"

flights_rx4_2 <- flights_rx4$categorical[[2]][c("carrier", "Sum")]
names(flights_rx4_2)[2] <- "sum_dist"

# merge the tables together
flights_rx5 <- merge(flights_rx4_1, flights_rx4_2, by = "carrier", all = TRUE)

# sort the results
flights_rx5 <- flights_rx5[order(flights_rx5$mean_delay, decreasing = TRUE),]

head(flights_rx5)
```

4. Using dplyrXdf we can achieve the same using the following pipeline:

```
flightsSmry <- flightsXdf %>%
               filter(month <= 6, year == 2013) %>%
               mutate(dist_km = distance * 1.6093, delay = (arr_delay + dep_delay) / 2) %>%
               group_by(carrier) %>%
               summarise(mean_delay = mean(delay), sum_dist = sum(dist_km)) %>%
               arrange(desc(mean_delay))

head(flightsSmry)
```

Even with this very straightforward example, dplyrXdf hides the complexity of calling RevoScaleR functions while retaining their power. In particular, note the following:

- There is no need to keep track of input and output file locations: the verbs in the dplyrXdf pipeline will automatically create files and reuse them as needed. Files that are no longer used will be deleted, so there won't be multiple orphaned files cluttering up your hard disk.

- The summarise verb is much simpler to work with than the RevoScaleR rxSummary function. It doesn't require scanning through a list of output objects to find the information you're after, and it accepts grouping variables of any type (numeric, character or factor).

- The pipeline notation makes it clear at a glance what is the sequence of operations being carried out. This is one of the major benefits of dplyr, and is now also available for those working with xdf files.

Single-table verbs
dplyrXdf supports all the basic dplyr single-table verbs:

- filter and select to choose rows and columns
- mutate and transmute to do data transformation
- group_by to define groups
- summarise and do to carry out computations on grouped data
- arrange to sort by variables
- rename to rename columns
- distinct to drop duplicates

Under the hood, they work by translating your pipeline into calls to the base RevoScaleR functions for working with xdf files: for example, mutate calls rxDataStep to compute transformations; arrange calls rxSort, and so on.

Most of these verbs work exactly as they do in dplyr. Thus if you know how to use dplyr, then you also know how to use the bulk of dplyrXdf.

Two-table verbs

dplyrXdf supports the main table-join verbs from dplyr: left_join, right_join, inner_join and full_join. The syntax is the same as for the dplyr versions, including joining on non-matching column names. The underlying implementation usesrxMerge with the appropriate arguments for each type of join.

For example, one of the joins in the dplyr two-table verbs vignette joins the flights table with the airports table, using the columns dest (in flights) and faa (in airports). The same code in dplyr also works in dplyrXdf:

```
airportsXdf <- rxDataFrameToXdf(airports, "airports.xdf",
overwrite=TRUE)
flightsJoin <- left_join(
    flightsXdf %>% select(year:day, hour, origin, dest,
tailnum, carrier),
    airportsXdf,
    by=c("dest"="faa"))
head(flightsJoin)
```

Non-xdf and non-local data sources

Despite the name, dplyrXdf supports all file data sources defined by RevoScaleR, not just xdf files. This includes delimited text (RxTextData), SAS datasets (RxSasData) and SPSS datasets (RxSpssData). If you pass one of these data sources to a dplyrXdf pipeline, it will import the data to an xdf file first before executing the rest of the pipeline.

Terms of Use