RevoScaleR Getting Started Guide

The correct bibliographic citation for this manual is as follows: Microsoft Corporation. 2016. *RevoScaleR Getting Started Guide*. Microsoft Corporation, Redmond, WA.

RevoScaleR Getting Started Guide

Microsoft
One Microsoft Way
Redmond WA 98052
U.S.A.

Revised on November 9, 2015

We want our documentation to be useful, and we want it to address your needs. If you have comments on this or any Microsoft R Services document, write to revodoc@microsoft.com.

Table of Contents

1 Overview

This guide is an introduction to ***RevoScaleR***, an R package providing both High Performance Computing (HPC) and High Performance Analytics (HPA) capabilities for R. HPC capabilities allow you to distribute the execution of essentially any R function across cores and nodes, and deliver the results back to the user. HPA adds big data to the challenge. ***RevoScaleR*** provides functions for performing scalable and extremely high performance data management, analysis, and visualization. This guide focuses on these HPA 'big data' capabilities. R, along with many other statistical analysis products, is challenged by problems of capacity and speed. Users cannot perform data analysis because their data is too big to fit into memory, or even if it fits, there is not sufficient memory available to perform analysis. In R this is often a problem because copies of data are frequently made during analysis. Even without a capacity limit, computation may be too slow to be useful. The ***RevoScaleR*** package not only helps to overcome these challenges in R, but surpasses capabilities in other statistics products.

The data manipulation and analysis functions in ***RevoScaleR*** 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 and, 2) to perform computations distributed over several cores, processors, or nodes in a cluster, or 3) to create scalable data analysis routines that can be developed locally with smaller data sets, then deployed to larger data and/or a cluster of computers. These are ideal candidates for ***RevoScaleR*** because ***RevoScaleR*** is based on the concept of operating on chunks of data and using *updating algorithms.*

The ***RevoScaleR*** package also provides an efficient file format for storing data designed for rapid reading of arbitrary rows and columns of data. Functions are provided to import data into this file format before performing analysis. ***RevoScaleR*** analysis functions work directly with this data file format, but also can be used directly with data stored in a text, SPSS, or SAS file or an ODBC connection. Functions are also provided to easily extract a subset of a data file into a data frame in memory for further analysis.

Additional examples of using ***RevoScaleR*** can be found in the following manuals provided with ***RevoScaleR***:

- *RevoScaleR User's Guide* (RevoScaleR_Users_Guide.pdf)
- *RevoScaleR Distributed Computing Guide* [RevoScaleR_Distributed_Computing.pdf; see this guide for HPC examples]
- *RevoScaleR ODBC Data Import Guide* (RevoScaleR_ODBC.pdf)
- *RevoScaleR Getting Started with Hadoop* (RevoScaleR_Hadoop_Getting_Started.pdf)
- *RevoScaleR Getting Started with Teradata* (RevoScaleR_Teradata_Getting_Started.pdf)

The bulk of this guide shows using ***RevoScaleR*** with the ***Revolution R Enterprise R Productivity Environment for Windows***; that component has its own *Getting Started Guide* accessible from its help menu. If you plan to use ***RevoScaleR*** in a Linux environment, the details for loading the package will differ, but the actual R commands used to call ***RevoScaleR*** functions will be identical.

2 Installation

The ***RevoScaleR*** package is installed as part of Microsoft R Services on both Windows and Linux. The package is automatically loaded when you start Microsoft R Services.

3 Running the Examples in the Getting Started Guide

3.1 Downloading and Using 'Big' Sample Data

Most of the examples in this guide can be run using sample data that is included in the RevoScaleR package, but to really get a feel for RevoScaleR features you may want to use larger data sets. You can download these larger data sets online.

The following 'big' datasets are used in this guide:

- *mortDefault*: a set of ten comma-separated files, each of which contains 1,000,000 observations of simulated data on mortgage defaults. These data sets are used in Chapter 8 on analyzing loan defaults. (Windows users should download the zip version, mortDefault.zip, and Linux users mortDefault.tar.gz). They total about 220 MB unpacked.
- *AirOnTime87to12*: an .xdf file containing information on flight arrival and departure details for all commercial flights within the USA, from October 1987 to December 2012. This is a large dataset: there are nearly 150 million records in total. It is used in the examples in Chapter 6 on Analyzing a Large Data Set with RevoScaleR. It is under 4 GB unpacked.
- *AirOnTime7Pct*: A 7% subsample of *AirOnTimeData87to12,* this is an .xdf file containing just the variables used in the examples in this guide. It can be used as an alternative in the examples in Chapter 6 on Analyzing a Large Data Set with RevoScaleR. This subsample contains a little over 10 million rows of data. It is about 100 MB unpacked.

When downloading these files, put them in a directory where you can easily access them. For example, create a directory "C:/MRS/Data" and unpack the files there. When running examples using these files, you will want to specify this location as your `bigDataDir`. For example:

```
bigDataDir <- "C:/MRS/Data"
```

3.2 Creating a New Solution for Your Examples

If you are using the R Productivity Environment, you will probably want to create a new Solution. Then you can then easily store your example R commands in scripts along with data files you create. To do this, select **New Project** from the **File** menu. Your working directory will automatically be reset to the location of the new solution. A new project has a single script `Script.R` by default; to open it, double-click the script name in the **Solution Explorer**.

Each of the commands below can be put into your script. After putting in a new set of commands, you can select and run the added code.

Note that if you want to skip the step of typing in R commands, demo scripts are available containing code very similar to what you see in this guide. These scripts are located in the `demoScripts` subdirectory of your ***RevoScaleR*** installation. On Windows, this is typically:

```
C:\Program Files\Microsoft\MRO-for-RRE\8.0\R-3.2.2\library\RevoScaleR\demoScripts
```

A copy of a demo script can be easily added to your current solution by right-clicking on the project name in the **Solution Explorer**, and choosing **Add**, then **Existing Item**.

4 A Tutorial Introduction to RevoScaleR

This section contains a more detailed introduction to the most important high performance analytics features of ***RevoScaleR,*** focusing on larger data stored in .xdf files. The following tasks are performed***:***

1. Convert text data to the .xdf data file format.
2. Summarize your data.
3. Fit a linear model to the data.
4. Create a new .xdf file, sub-setting the original data set and performing data transformations.
5. Compute crosstabs.
6. Fit a logistic regression model.
7. Compute predicted values.

4.1 Importing Text Data into the .xdf Data File Format

The ***RevoScaleR*** package provides a data file format (.xdf) designed to be very efficient for reading arbitrary rows and columns. To convert a text file into the .xdf data format, use the function `rxImport`.

For example, the *SampleData* folder of the ***RevoScaleR*** package contains a file *AirlineDemoSmall.csv* containing three columns of data: two numeric columns, `ArrDelay` and `CRSDepTime`, and a column of strings, `DayOfWeek`. When we import the data, we want to convert the strings to a categorical or factor variable. The file contains 600,000 rows of data in addition to a first row with variable names. It is a subset of a data set containing information on flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008.

The location of the sample data directory is stored as an option. It is initialized to the location of the *SampleData* directory included in the ***RevoScaleR*** package. You can use the ***`rxGetOption`*** function to retrieve this location:

```
sampleDataDir <- rxGetOption("sampleDataDir")
```

Let's import the text file into an .xdf file named *ADS* in your current working directory. To see this location, enter:

```
getwd()
```

To import the *AirlineDemoSmall.csv* file into the .xdf data file format, use *rxImport* as follows:

```
inputFile <- file.path(sampleDataDir, "AirlineDemoSmall.csv")

airDS <- rxImport(inData = inputFile, outFile = "ADS.xdf",
      missingValueString = "M", stringsAsFactors = TRUE)
```

The input .csv file uses the letter M to represent missing values, rather than the default NA, so we specify this with the *missingValueString* argument[1]. Setting *stringsAsFactors* to *TRUE* will set the levels specification for *DayOfWeek* to the unique strings found in that variable, listed in the order in which they are encountered. Since this order is arbitrary, and can easily vary from data set to data set, it is preferred to explicitly specify the levels in the desired order using the *colInfo* argument. We will also use the argument *overwrite=TRUE*, since we want to replace the file previously imported:

```
colInfo <- list(DayOfWeek = list(type = "factor",
    levels = c("Monday", "Tuesday", "Wednesday", "Thursday",
    "Friday", "Saturday", "Sunday")))

airDS <- rxImport(inData = inputFile, outFile = "ADS.xdf",
      missingValueString = "M", colInfo  = colInfo, overwrite = TRUE)
```

Notice that once we supply the *colInfo* argument, we no longer need to specify *stringsAsFactors*; *DayOfWeek* is our only factor variable.

4.2 Examining Your New Data File

Using the small *airDS* object representing the ADS.xdf file, we can apply some standard R methods to get basic information about the data set:

```
nrow(airDS)
ncol(airDS)
head(airDS)
```

```
> nrow(airDS)
[1] 6e+05
> ncol(airDS)
[1] 3
> head(airDS)
  ArrDelay CRSDepTime DayOfWeek
1        6   9.666666    Monday
2       -8  19.916666    Monday
3       -2  13.750000    Monday
4        1  11.750000    Monday
```

[1] missingValueString is actually an argument to the rxTextToXdf function which is called by rxImport when reading most delimited text files.

```
5       -2   6.416667    Monday
6      -14  13.833333    Monday
```

The *rxGetVarInfo* function provides addition variable information:

```
rxGetVarInfo(airDS)

Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167,
23.9833)
Var 3: DayOfWeek
       7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday
Sunday
```

You can also read an arbitrary chunk of the data set into a data frame for further examination. For example, read 10 rows into a data frame starting with the 100,000th row:

```
myData <- rxReadXdf(airDS, numRows=10, startRow=100000)
myData
```

This code should generate the following output:

```
   ArrDelay CRSDepTime DayOfWeek
1        -2  11.416667  Saturday
2        39   9.916667    Friday
3        NA  10.033334    Monday
4         1  17.000000    Friday
5       -17   9.983334 Wednesday
6         8  21.250000  Saturday
7        -9   6.250000    Friday
8       -11  15.000000    Friday
9         4  20.916666    Sunday
10       -8   6.500000  Thursday
```

Then look to see what the factor levels are for the *DayOfWeek* variable:

```
levels(myData$DayOfWeek)
```

This command should generate the following output:

```
[1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"    "Saturday"
[7] "Sunday"
```

4.3 Summarizing Your Data

Use the *rxSummary* function to obtain descriptive statistics for your .xdf data file. The *rxSummary* function takes a formula as its first argument, and the name of the data set as the second. To get summary statistics for all of the data in your data file, you can alternatively use the *summary* method for the *airDS* object.

```
adsSummary <- rxSummary(~ArrDelay+CRSDepTime+DayOfWeek, data = airDS)
```

or

```
adsSummary <- summary( airDS )

adsSummary
```

Summary statistics will be computed on the variables in the formula, removing missing values for all rows in the included variables[2]:

```
Call:
rxSummary(formula = form, data = object, byTerm = TRUE, reportProgress = 0L)

Summary Statistics Results for: ~ArrDelay + CRSDepTime + DayOfWeek
File name: C:\YourWorkingDir\ADS.xdf
Number of valid observations: 6e+05

 Name       Mean     StdDev    Min        Max        ValidObs MissingObs
 ArrDelay   11.31794 40.688536 -86.000000 1490.00000 582628   17372
 CRSDepTime 13.48227  4.697566   0.016667   23.98333 600000       0

Category Counts for DayOfWeek
Number of categories: 7
Number of valid observations: 6e+05
Number of missing observations: 0

 DayOfWeek Counts
 Monday    97975
 Tuesday   77725
 Wednesday 78875
 Thursday  81304
 Friday    82987
 Saturday  86159
 Sunday    94975
```

Notice that the summary information shows cell counts for categorical variables, and appropriately does not provide summary statistics such as *Mean* and *StdDev*. Also notice that the *Call:* line will show the actual call you entered or the call provided by *summary*, so will appear differently in different circumstances.

You can also compute summary information by one or more categories by using interactions of a numeric variable with a factor variable. For example, to compute summary statistics on Arrival Delay by Day of Week:

```
rxSummary(~ArrDelay:DayOfWeek, data = airDS)
```

The output shows the summary statistics for *ArrDelay* for each day of the week:

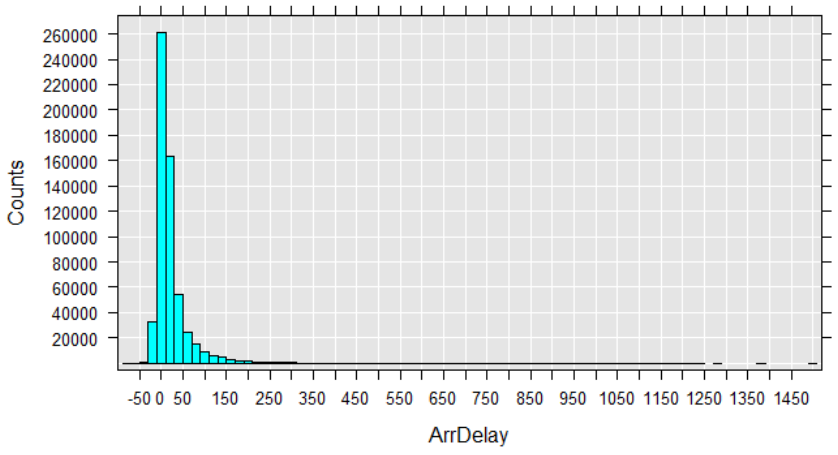
---

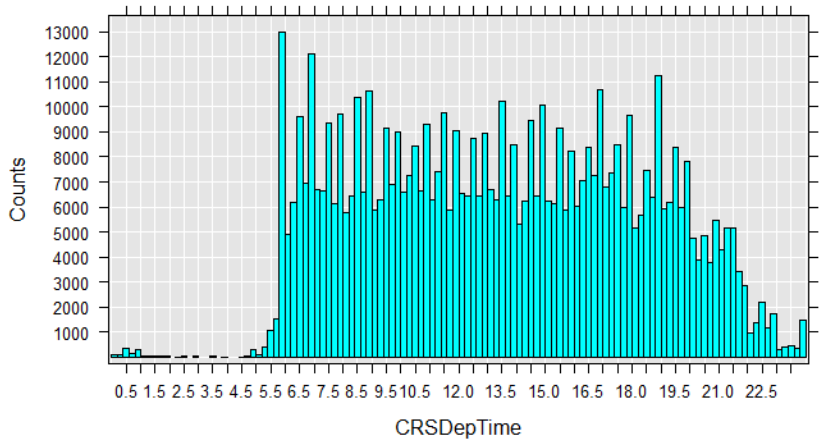
[2] In RevoScaleR 2.0 and later, rxSummary by default computes data summaries term-by-term, and missing values are omitted on a term-by-term basis. In earlier versions, summaries were computed on the complete table after observations with missing elements were omitted.

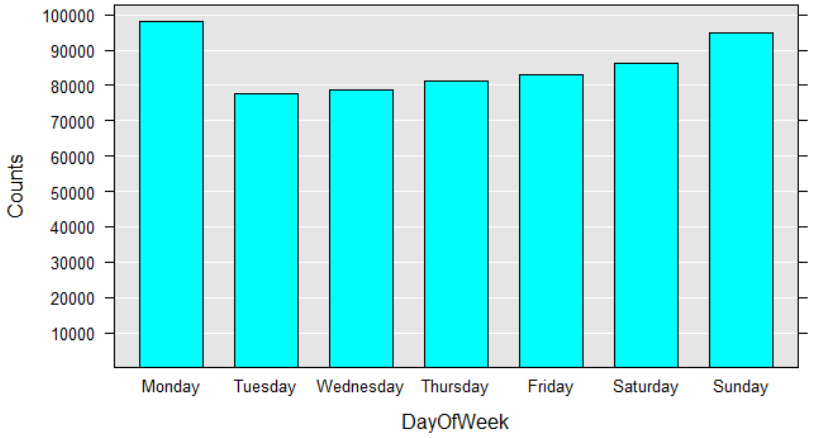
```
Call:
rxSummary(formula = ~ArrDelay:DayOfWeek, data = airDS)

Summary Statistics Results for: ~ArrDelay:DayOfWeek
File name: C:\YourWorkingDir\ADS.xdf
Number of valid observations: 6e+05

 Name                Mean     StdDev   Min Max  ValidObs MissingObs
 ArrDelay:DayOfWeek 11.31794 40.68854 -86 1490 582628   17372

Statistics by category (7 categories):

 Category                           DayOfWeek Means    StdDev  Min Max ValidObs
 ArrDelay for DayOfWeek=Monday      Monday    12.025604 40.02463 -76 1017 95298
 ArrDelay for DayOfWeek=Tuesday     Tuesday   11.293808 43.66269 -70 1143 74011
 ArrDelay for DayOfWeek=Wednesday   Wednesday 10.156539 39.58803 -81 1166 76786
 ArrDelay for DayOfWeek=Thursday    Thursday   8.658007 36.74724 -58 1053 79145
 ArrDelay for DayOfWeek=Friday      Friday    14.804335 41.79260 -78 1490 80142
 ArrDelay for DayOfWeek=Saturday    Saturday  11.875326 45.24540 -73 1370 83851
 ArrDelay for DayOfWeek=Sunday      Sunday    10.331806 37.33348 -86 1202 93395
```

To get a better feel for the data, we can draw histograms for each variable:

```
options("device.ask.default" = T)
rxHistogram(~ArrDelay, data = airDS)
rxHistogram(~CRSDepTime, data = airDS)
rxHistogram(~DayOfWeek, data = airDS)
```

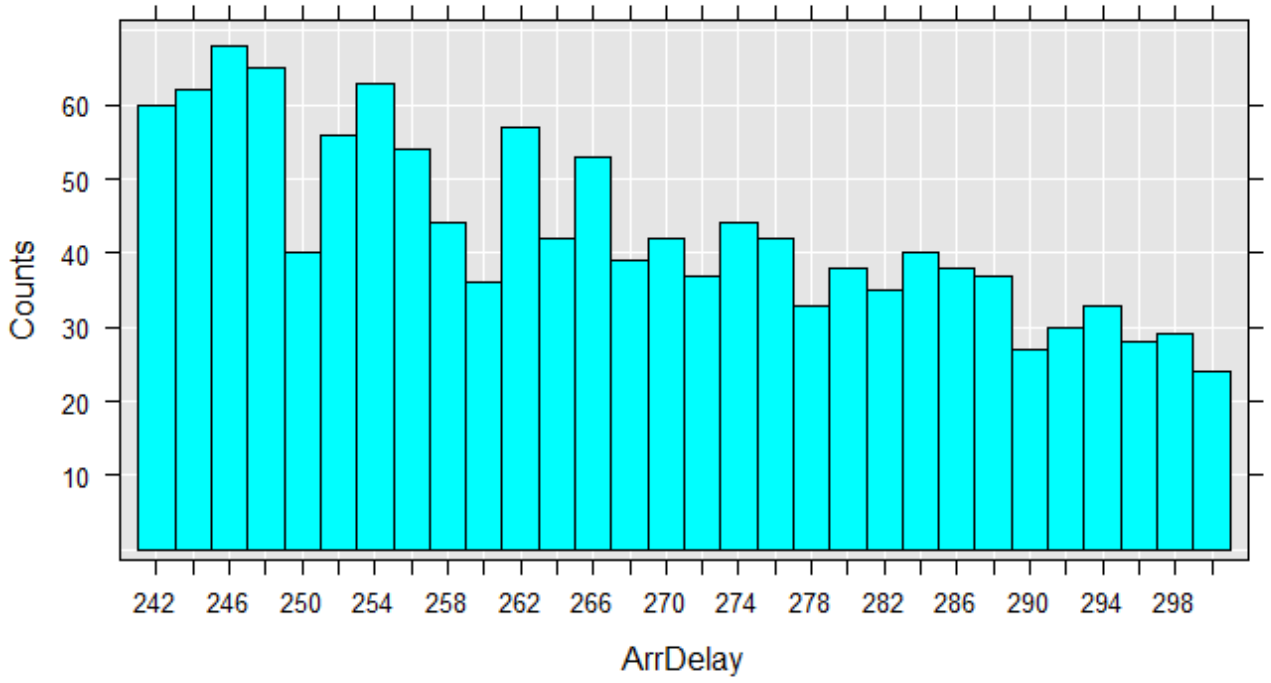






We can also easily extract a subsample of the data file into a data frame in memory. For example, we can look at just the flights that were between 4 and 5 hours late:

```
myData <- rxDataStep(inData = airDS,
      rowSelection = ArrDelay > 240 & ArrDelay <= 300,
      varsToKeep = c("ArrDelay", "DayOfWeek"))
rxHistogram(~ArrDelay, data = myData)
```



4.4 Fitting a Linear Model

4.4.1 Fitting a Simple Model

Use the *rxLinMod* function to fit a linear model using your *ADS.xdf* file. Use a single dependent variable, the factor *DayOfWeek*:

```
arrDelayLm1 <- rxLinMod(ArrDelay ~ DayOfWeek, data = airDS)
summary(arrDelayLm1)
```

The resulting output is:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airDS)

Linear Regression Results for: ArrDelay ~ DayOfWeek
File name: C:\YourWorkingDir\ADS.xdf
Dependent variable(s): ArrDelay
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 582628
Number of missing observations: 17372

Coefficients: (1 not defined because of singularities)
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)          10.3318     0.1330  77.673 2.22e-16 ***
DayOfWeek=Monday      1.6938     0.1872   9.049 2.22e-16 ***
DayOfWeek=Tuesday     0.9620     0.2001   4.809 1.52e-06 ***
```

```
DayOfWeek=Wednesday  -0.1753     0.1980  -0.885    0.376
DayOfWeek=Thursday   -1.6738     0.1964  -8.522 2.22e-16 ***
DayOfWeek=Friday      4.4725     0.1957  22.850 2.22e-16 ***
DayOfWeek=Saturday    1.5435     0.1934   7.981 2.22e-16 ***
DayOfWeek=Sunday     Dropped    Dropped Dropped  Dropped
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 40.65 on 582621 degrees of freedom
Multiple R-squared: 0.001869
Adjusted R-squared: 0.001858
F-statistic: 181.8 on 6 and 582621 DF,  p-value: < 2.2e-16
Condition number: 10.5595
```

4.4.2 Using the *cube* Argument and Plotting Results

If you are using categorical data in your regression, you can use the *cube* argument. If *cube* is set to *TRUE* and the first term of the regression is categorical (a factor or an interaction of factors), the regression is done using a partitioned inverse, which may be faster and use less memory than standard regression computation. Averages/counts of the category "bins" can be computed in addition to standard regression results. The intercept will also be automatically dropped, so that each category level will have an estimated coefficient (unlike the previous example). If *cube* is set to *TRUE* and the first term is not categorical, you get an error message.

Re-estimate the linear model, this time setting *cube* to *TRUE*. Then print a summary of the results:

```
arrDelayLm2 <- rxLinMod(ArrDelay ~ DayOfWeek, data = airDS,
      cube = TRUE)
summary(arrDelayLm2)
```

You should see the following output:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airDS, cube = TRUE)

Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
File name: C:\YourWorkingDir\ADS.xdf
Dependent variable(s): ArrDelay
Total independent variables: 7
Number of valid observations: 582628
Number of missing observations: 17372

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)     | Counts
DayOfWeek=Monday     12.0256     0.1317   91.32 2.22e-16 *** |  95298
DayOfWeek=Tuesday    11.2938     0.1494   75.58 2.22e-16 *** |  74011
DayOfWeek=Wednesday  10.1565     0.1467   69.23 2.22e-16 *** |  76786
DayOfWeek=Thursday    8.6580     0.1445   59.92 2.22e-16 *** |  79145
DayOfWeek=Friday     14.8043     0.1436  103.10 2.22e-16 *** |  80142
DayOfWeek=Saturday   11.8753     0.1404   84.59 2.22e-16 *** |  83851
DayOfWeek=Sunday     10.3318     0.1330   77.67 2.22e-16 *** |  93395
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 40.65 on 582621 degrees of freedom
Multiple R-squared: 0.001869 (as if intercept included)
Adjusted R-squared: 0.001858
F-statistic: 181.8 on 6 and 582621 DF,  p-value: < 2.2e-16
Condition number: 1
```

The coefficient estimates are the mean arrival delay for each day of the week.

When the *cube* argument is set to *TRUE* and the model has only one term as an independent variable, the "*countDF*" component of the results object also contains category means. This data frame can be extracted using the *rxResultsDF* function, and is particularly convenient for plotting.

```
countsDF <- rxResultsDF(arrDelayLm2, type = "counts")
countsDF
```

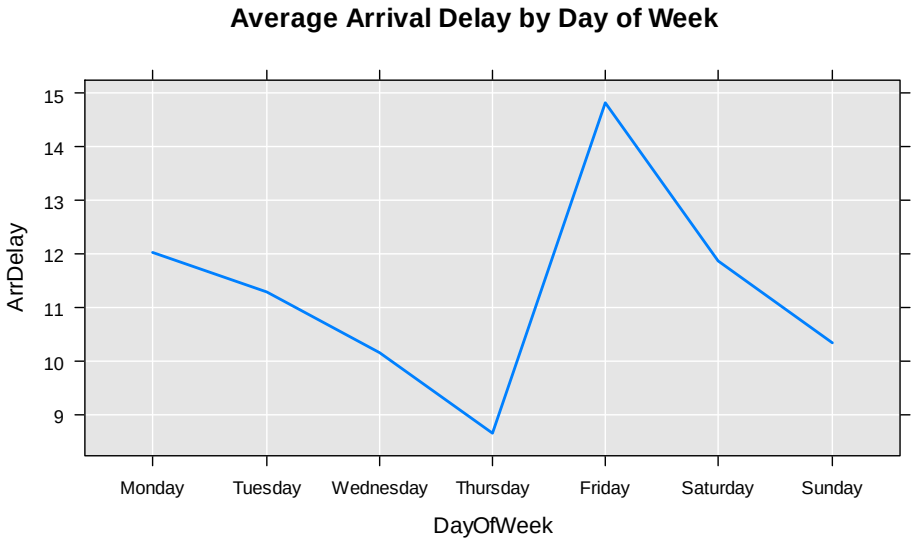
You should see the following output:

```
  DayOfWeek  ArrDelay Counts
1    Monday 12.025604  95298
2   Tuesday 11.293808  74011
3 Wednesday 10.156539  76786
4  Thursday  8.658007  79145
5    Friday 14.804335  80142
6  Saturday 11.875326  83851
7    Sunday 10.331806  93395
```

Now plot the average arrival delay for each day of the week:

```
rxLinePlot(ArrDelay~DayOfWeek, data = countsDF,
    main = "Average Arrival Delay by Day of Week")
```

The following plot is generated, showing the lowest average arrival delay on Thursdays:



4.4.3 A Linear Model with Multiple Independent Variables

We can run a more complex model examining the dependency of arrival delay on both day of week and the departure time. We'll estimate the model using the `F` expression to have the `CRSDepTime` variable interpreted as a categorical or factor variable.[3] By interacting `DayOfWeek` with `F(CRSDepTime)` we are creating a dummy variable for every combination of departure hour and day of the week.

```
arrDelayLm3 <- rxLinMod(ArrDelay ~ DayOfWeek:F(CRSDepTime),
      data = airDS, cube = TRUE)
arrDelayDT <- rxResultsDF(arrDelayLm3, type = "counts")
head(arrDelayDT, 15)
```

The output shows the first fifteen rows of the counts data frame. The variable `CRSDepTime` gives the hour of the departure time and `ArrDelay` shows the average departure delay for that hour for that day of week. `Counts` gives the number of observations that contain that combination of day of week and departure hour.

```
   DayOfWeek CRSDepTime   ArrDelay Counts
1     Monday          0  7.4360902    133
2    Tuesday          0  7.0000000    107
3  Wednesday          0  3.7857143     98
4   Thursday          0  3.0097087    103
5     Friday          0  4.4752475    101
6   Saturday          0  4.5460993    141
```

---

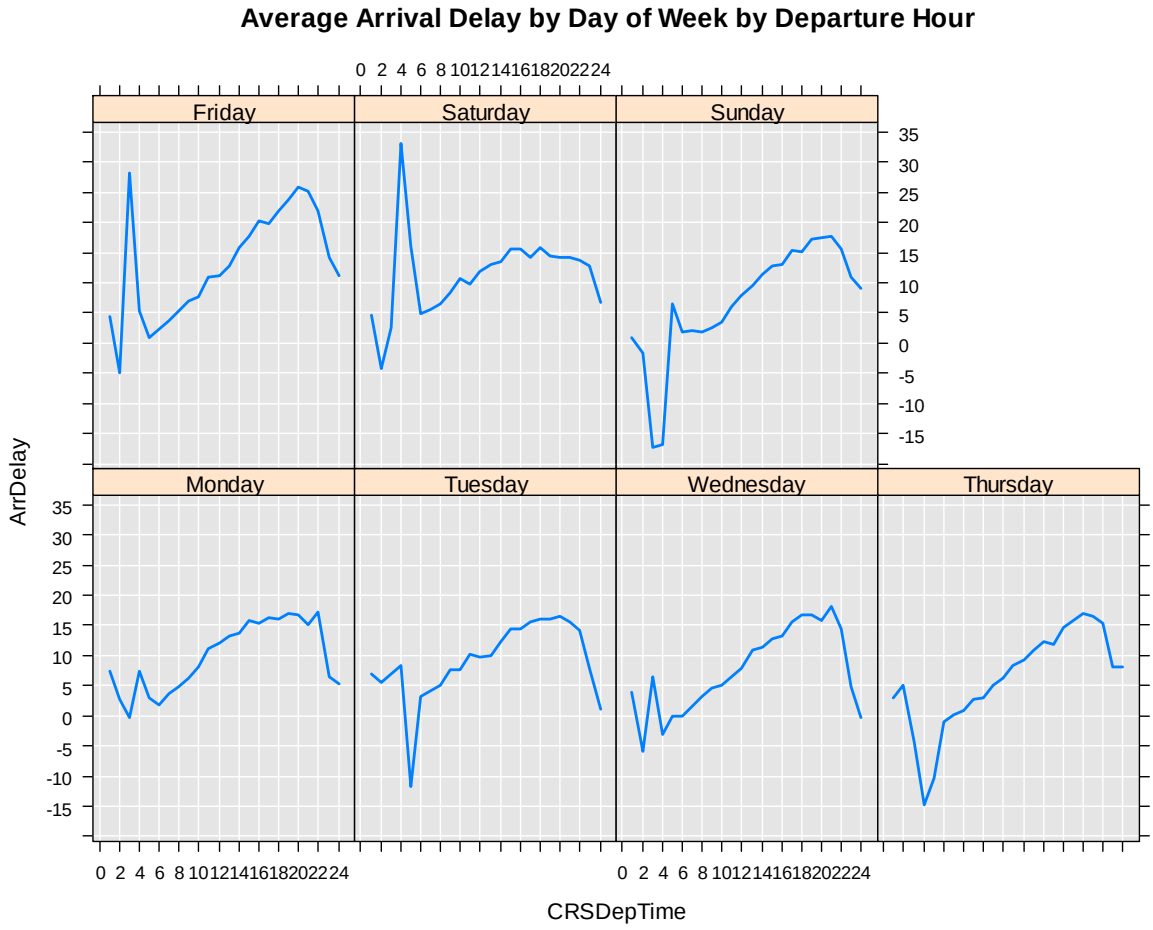
[3] `F()` is not an R function, although it is used as one inside RevoScaleR formulas. It tells RevoScaleR to create a factor by creating one level for each integer in the range `(floor(min(x)), floor(max(x)))` and binning all the observations into the resulting set of levels. See `?rxFormula` for more information.

```
7     Sunday          0  0.9243697    119
8     Monday          1  2.6590909     44
9    Tuesday          1  5.5428571     35
10 Wednesday          1 -5.9696970     33
11  Thursday          1  5.1714286     35
12    Friday          1 -4.9062500     32
13  Saturday          1 -4.2727273     44
14    Sunday          1 -1.7368421     38
15    Monday          2 -0.2000000     15
```

Now plot the results:

```
rxLinePlot( ArrDelay~CRSDepTime|DayOfWeek, data = arrDelayDT,
    title = "Average Arrival Delay by Day of Week by Departure Hour")
```

You should see the following plot:



4.5 Create a Subset of the Data Set and Compute a Crosstab

Create a new data set containing a subset of rows and variables. This is convenient if you intend to do lots of analysis on a subset of a large data set. To do this, we use the `rxDataStep` function with the following arguments:

- *outFile:* the name of the new data set
- *inData:* the name of an .xdf file or an *RxXdfData* object representing the original data set you are subsetting
- *varsToDrop:* a character vector of variables to drop from the new data set, here *CRSDepTime*
- *rowSelection:* keep only the rows where the flight was more than 15 minutes late.

The resulting call is as follows:

```
airLateDS <- rxDataStep(inData = airDS, outFile = "ADS1.xdf",
    varsToDrop = c("CRSDepTime"),
    rowSelection = ArrDelay > 15)
ncol(airLateDS)
nrow(airLateDS)
```

You will see that the new data set has only two variables and has dropped from 600,000 to 148,526 observations.

Compute a crosstab showing the mean arrival delay by day of week.

```
myTab <- rxCrossTabs(ArrDelay~DayOfWeek, data = airLateDS)
summary(myTab, output = "means")
```

The results show that in this data set “late” flights are on average over 10 minutes later on Tuesdays than on Sundays:

```
Call:
rxCrossTabs(formula = ArrDelay ~ DayOfWeek, data = airLateDS)

Cross Tabulation Results for: ArrDelay ~ DayOfWeek
File name: C:\YourWorkingDir\ADS1.xdf
Dependent variable(s): ArrDelay
Number of valid observations: 148526
Number of missing observations: 0
Statistic: means

ArrDelay (means):
             means  means %
Monday    56.94491 13.96327
Tuesday   64.28248 15.76249
Wednesday 60.12724 14.74360
Thursday  55.07093 13.50376
Friday    56.11783 13.76047
Saturday  61.92247 15.18380
Sunday    53.35339 13.08261
Col Mean  57.96692
```

4.6 Creating a New Data Set with Variable Transformations

Now use the *rxDataStep* function to create a new data set containing the variables in *ADS.xdf* plus additional variables created through transformations. Typically additional

variables are created using the *transforms* argument. [See the User's Guide (RevoScaleR_Users_Guide.pdf) for information on doing more complex transformations using a transform function.] Remember that all expressions used in *transforms* must be able to be processed on a chunk of data at a time.

In the example below, three new variables are created. The variable *Late* is a logical variable set to *TRUE* (or *1*) if the flight was more than 15 minutes late in arriving. The variable *DepHour* is an integer variable indicating the hour of the departure. The variable *Night* is also a logical variable, set to *TRUE* (or *1*) if the flight departed between 10 P.M. and 5 A.M.

The *rxDataStep* function will read the existing data set and perform the transformations chunk by chunk, and create a new data set.

```
airExtraDS <- rxDataStep(inData = airDS, outFile="ADS2.xdf",
      transforms=list(
            Late = ArrDelay > 15,
            DepHour = as.integer(CRSDepTime),
            Night = DepHour >= 20 | DepHour <= 5))

rxGetInfo(airExtraDS, getVarInfo=TRUE, numRows=5)
```

You should see the following information in your output:

```
File name: C:\YourWorkingDir\ADS2.xdf
Number of observations: 6e+05
Number of variables: 6
Number of blocks: 2
Compression type: zlib
Variable information:
Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167,
23.9833)
Var 3: DayOfWeek
       7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday
Sunday
Var 4: Late, Type: logical, Low/High: (0, 1)
Var 5: DepHour, Type: integer, Low/High: (0, 23)
Var 6: Night, Type: logical, Low/High: (0, 1)
Data (5 rows starting with row 1):
  ArrDelay CRSDepTime DayOfWeek  Late DepHour Night
1        6   9.666666    Monday FALSE       9 FALSE
2       -8  19.916666    Monday FALSE      19 FALSE
3       -2  13.750000    Monday FALSE      13 FALSE
4        1  11.750000    Monday FALSE      11 FALSE
5       -2   6.416667    Monday FALSE       6 FALSE
```

4.7 Run a Logistic Regression Using the New Data

The function *rxLogit* takes a binary dependent variable. Here we will use the variable *Late*, which is *TRUE* (or *1*) if the plane was more than 15 minutes late arriving. For dependent

variables we will use the *DepHour*, the departure hour, and *Night*, indicating whether or not the flight departed at night.

```
logitObj <- rxLogit(Late~DepHour + Night, data = airExtraDS)
summary(logitObj)
```

You should see the following results:

```
Call:
rxLogit(formula = Late ~ DepHour + Night, data = airExtraDS)

Logistic Regression Results for: Late ~ DepHour + Night
File name: C:\Users\sue\SVN\bigAnalytics\trunk\revoAnalytics\ADS2.xdf
Dependent variable(s): Late
Total independent variables: 3
Number of valid observations: 582628
Number of missing observations: 17372
-2*LogLikelihood: 649607.8613 (Residual deviance on 582625 degrees of
freedom)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.0990076  0.0104460 -200.94 2.22e-16 ***
DepHour      0.0790215  0.0007671  103.01 2.22e-16 ***
Night       -0.3027030  0.0109914  -27.54 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 3.0178
Number of iterations: 4
```

4.8 Computing Predicted Values

You can use the object returned from the call to *rxLogit* in the previous section to compute predicted values. In addition to the model object, we specify the data set on which to compute the predicted values and the data set in which to put the newly computed predicted values. In the call below, we use the same dataset for both. In general, the data set on which to compute the predicted values must be similar to the original data set used to estimate the model in the following ways; it should have the same variable names and types, and factor variables must have the same levels in the same order.

```
predictDS <- rxPredict(modelObject = logitObj, data = airExtraDS,
      outData = airExtraDS)
rxGetInfo(predictDS, getVarInfo=TRUE, numRows=5)
```

You should see the following information:

```
File name: C:\YourWorkingDir\ADS2.xdf
Number of observations: 6e+05
Number of variables: 7
Number of blocks: 2
Compression type: zlib
```

```
Variable information:
Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167,
23.9833)
Var 3: DayOfWeek
       7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday
Sunday
Var 4: Late, Type: logical, Low/High: (0, 1)
Var 5: DepHour, Type: integer, Low/High: (0, 23)
Var 6: Night, Type: logical, Low/High: (0, 1)
Var 7: Late_Pred, Type: numeric, Low/High: (0.0830, 0.3580)
Data (5 rows starting with row 1):
  ArrDelay CRSDepTime DayOfWeek  Late DepHour Night Late_Pred
1        6   9.666666    Monday FALSE       9 FALSE 0.1997569
2       -8  19.916666    Monday FALSE      19 FALSE 0.3548931
3       -2  13.750000    Monday FALSE      13 FALSE 0.2550745
4        1  11.750000    Monday FALSE      11 FALSE 0.2262214
5       -2   6.416667    Monday FALSE       6 FALSE 0.1645331
```

5 Using Code Snippets for RevoScaleR

If you are using the Revolution R Enterprise R Productivity Environment, you can use Code Snippets to increase your productivity with ***RevoScaleR***. Code Snippets provide a "fill-in-the-blanks" approach to script writing and are provided for most ***RevoScaleR*** functions. For example, to create our ADS file using Code Snippets, proceed as follows:

1. Right-click on an empty line in the Script window.
2. Click **Insert Snippet...**.
3. Double-click **RevoScaleR**.
4. Double-click **data.**.
5. Double-click **import to .xdf or data frame**. The Code Snippet is inserted as shown:

```
myData <- rxImport(inData = "myFile.csv", outFile = "myFile.xdf",
    varsToKeep = NULL, varsToDrop = NULL, rowSelection = NULL,
    transforms = NULL, append = "none", overwrite = FALSE,
    numRows = -1, stringsAsFactors = FALSE,
    colClasses = NULL, rowsPerRead = 500000, type = "auto",
    reportProgress = 2)
```

1. Highlighting indicates the current selection. Press TAB to keep *myData* as the returned object and move to the next entry. Notice that *myfile.csv* now shows the blue highlight.
2. Type *file.path(sampleDataDir,"AirlineDemoSmall.csv")* as the *inData* argument, then press TAB again to move to the next entry.
3. Enter *"ADSSnippet"* as the *outFile* name.
4. Press ENTER to exit the Code Snippet.
5. Select the lines of code inserted by the Code Snippet, and press the Run Selection button  .

6 Analyzing a Large Data Set with RevoScaleR

6.1 Getting Set Up to Use Your XDF File

Airline on-time performance data for the years 1987 through 2008 was used in the American Statistical Association Data Expo in 2009 (http://stat-computing.org/dataexpo/2009/). ASA describes the data set as follows:

> *The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12 gigabytes when uncompressed.*

Data by month can be downloaded as comma-delimited text files from the Research and Innovative Technology Administration (RITA) of the Bureau of Transportation Statistics web site (http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236). Data from 1987 through 2012 has been imported into an .xdf file named *AirOnTime87to12.xdf.* This file is available for download online. (Windows users should download *AirOnTime87to12.zip*; Linux users *AirOnTime87to12.tar.gz*.) A much smaller subset containing 7% of the observations for the variables used in this example is also available as *AirOnTime7Pct.xdf*. This subsample contains a little over 10 million rows of data, so has about 60 times the rows of the smaller subsample used in the previous example. More information on this data set can be found in the help file:

```
?AirOnTime87to12
```

To follow the examples in this section at your own computer, you need to download one or both of these data sets.

Specify the correct location of your downloaded data for the `bigDataDir` variable:

```
bigDataDir <- "C:/MRS/Data"

airDataName <- file.path(bigDataDir, "AirOnTime87to12")
```

or

```
airDataName <- file.path(bigDataDir,"AirOnTime7Pct")
```

Create an in-memory RxXdfObject data source object to represent the .xdf data file:

```
bigAirDS <- RxXdfData( airDataName )
```

Use the `rxGetInfo` function to get summary information on the active data set.

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

The full data set has 46 variables and almost 150 million observations.

```
File name: C:\MRS\Data\AirOnTime87to12\AirOnTime87to12.xdf
```

```
Number of observations: 148619655
Number of variables: 46
Number of blocks: 721
Compression type: zlib
Variable information:
Var 1: Year, Type: integer, Low/High: (1987, 2012)
Var 2: Month, Type: integer, Low/High: (1, 12)
Var 3: DayofMonth, Type: integer, Low/High: (1, 31)
Var 4: DayOfWeek
       7 factor levels: Mon Tues Wed Thur Fri Sat Sun
Var 5: FlightDate, Type: Date, Low/High: (1987-10-01, 2012-12-31)
Var 6: UniqueCarrier
       30 factor levels: AA US AS CO DL ... OH F9 YV 9E VX
Var 7: TailNum
       14463 factor levels: 0 N910DE N948DL N901DE N980DL ... N565UW N8326F
N8327A N8328A N8329B
Var 8: FlightNum
       8206 factor levels: 1 2 3 5 6 ... 8546 9221 13 7413 8510
Var 9: OriginAirportID
       374 factor levels: 12478 12892 12173 13830 11298 ... 12335 12389 10466
14802 12888
Var 10: Origin
       373 factor levels: JFK LAX HNL OGG DFW ... IMT ISN AZA SHD LAR
Var 11: OriginState
       53 factor levels: NY CA HI TX MO ... SD DE PR VI TT
Var 12: DestAirportID
       378 factor levels: 12892 12173 12478 13830 11298 ... 11587 12335 12389
10466 14802
Var 13: Dest
       377 factor levels: LAX HNL JFK OGG DFW ... ESC IMT ISN AZA SHD
Var 14: DestState
       53 factor levels: CA HI NY TX MO ... WY DE PR VI TT
Var 15: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0000,
24.0000)
Var 16: DepTime, Type: numeric, Storage: float32, Low/High: (0.0167, 24.0000)
Var 17: DepDelay, Type: integer, Low/High: (-1410, 2601)
Var 18: DepDelayMinutes, Type: integer, Low/High: (0, 2601)
Var 19: DepDel15, Type: logical, Low/High: (0, 1)
Var 20: DepDelayGroups
       15 factor levels: < -15 -15 to -1 0 to 14 15 to 29 30 to 44 ... 120 to
134 135 to 149 150 to 164 165 to 179 >= 180
Var 21: TaxiOut, Type: integer, Low/High: (0, 3905)
Var 22: WheelsOff, Type: numeric, Storage: float32, Low/High: (0.0000,
72.5000)
Var 23: WheelsOn, Type: numeric, Storage: float32, Low/High: (0.0000,
24.0000)
Var 24: TaxiIn, Type: integer, Low/High: (0, 1440)
Var 25: CRSArrTime, Type: numeric, Storage: float32, Low/High: (0.0000,
24.0000)
Var 26: ArrTime, Type: numeric, Storage: float32, Low/High: (0.0167, 24.0000)
Var 27: ArrDelay, Type: integer, Low/High: (-1437, 2598)
Var 28: ArrDelayMinutes, Type: integer, Low/High: (0, 2598)
Var 29: ArrDel15, Type: logical, Low/High: (0, 1)
Var 30: ArrDelayGroups
```

```
       15 factor levels: < -15 -15 to -1 0 to 14 15 to 29 30 to 44 ... 120 to
      134 135 to 149 150 to 164 165 to 179 >= 180
Var 31: Cancelled, Type: logical, Low/High: (0, 1)
Var 32: CancellationCode
       5 factor levels: NA Carrier Weather NAS Security
Var 33: Diverted, Type: logical, Low/High: (0, 1)
Var 34: CRSElapsedTime, Type: integer, Low/High: (-162, 1865)
Var 35: ActualElapsedTime, Type: integer, Low/High: (-719, 1440)
Var 36: AirTime, Type: integer, Low/High: (-2378, 1399)
Var 37: Flights, Type: integer, Low/High: (1, 1)
Var 38: Distance, Type: integer, Low/High: (0, 4983)
Var 39: DistanceGroup
       11 factor levels: < 250 250-499 500-749 750-999 1000-1249 ... 1500-
      1749 1750-1999 2000-2249 2250-2499 >= 2500
Var 40: CarrierDelay, Type: integer, Low/High: (0, 2580)
Var 41: WeatherDelay, Type: integer, Low/High: (0, 1615)
Var 42: NASDelay, Type: integer, Low/High: (-60, 1392)
Var 43: SecurityDelay, Type: integer, Low/High: (0, 782)
Var 44: LateAircraftDelay, Type: integer, Low/High: (0, 1407)
Var 45: MonthsSince198710, Type: integer, Low/High: (0, 302)
Var 46: DaysSince19871001, Type: integer, Low/High: (0, 9223)
```

6.2 Reading a Chunk of Data

Data stored in the xdf file format can be read into a data frame, allowing the user to choose the rows and columns. For example, read 1000 rows beginning at the 100,000th row for the variables *ArrDelay*, *DepDelay*, and *DayOfWeek*.

```
testDF <- rxReadXdf(file = bigAirDS,
      varsToKeep = c("ArrDelay","DepDelay", "DayOfWeek"),
      startRow = 100000, numRows = 1000)
summary(testDF)
```

You should see the following information if *bigAirDS* is the full data set, *AirlineData87to08.xdf*:

```
       ArrDelay          DepDelay       DayOfWeek
 Min.   :-65.000   Min.   : -5.000   Mon :138
 1st Qu.: -9.000   1st Qu.:  0.000   Tues:135
 Median : -2.000   Median :  0.000   Wed :136
 Mean   :  2.297   Mean   :  6.251   Thur:173
 3rd Qu.:  8.000   3rd Qu.:  1.000   Fri :168
 Max.   :262.000   Max.   :315.000   Sat :133
 NA's   :8         NA's   :3         Sun :117
```

And we can easily run a regression using the *lm* function in R using this small data set:

```
lmObj <- lm(ArrDelay~DayOfWeek, data = testDF)
summary(lmObj)
```

It yields the following:

```
Call:
lm(formula = ArrDelay ~ DayOfWeek, data = testDF)

Residuals:
    Min      1Q  Median      3Q     Max
-65.251 -12.131  -3.307   6.749 261.869

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.1314     2.0149   0.065    0.948
DayOfWeekTues   2.1074     2.8654   0.735    0.462
DayOfWeekWed   -1.1981     2.8600  -0.419    0.675
DayOfWeekThur   0.1201     2.7041   0.044    0.965
DayOfWeekFri    0.7377     2.7149   0.272    0.786
DayOfWeekSat   14.2302     2.8876   4.928 9.74e-07 ***
DayOfWeekSun    0.2874     2.9688   0.097    0.923
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 23.58 on 985 degrees of freedom
  (8 observations deleted due to missingness)
Multiple R-squared:  0.03956,    Adjusted R-squared:  0.03371
F-statistic: 6.761 on 6 and 985 DF,  p-value: 4.91e-07
```

But this is only 1/148,619 of the rows contained in the full data set. If we try to read all the rows of these columns, we will likely run into memory problems. For example, on most systems with 8GB or less of RAM, running the commented code below will fail on the full data set with a "cannot allocate vector" error.

```
# testDF <- rxReadXdf(file=dataName, varsToKeep = c("ArrDelay",
#     "DepDelay", "DayOfWeek"))
```

In the next section you will see how you can analyze a data set that is too big to fit into memory by using ***RevoScaleR*** functions.

6.3 Estimating a Linear Model with a Huge Data Set

The RevoScaleR compute engine is designed to work very efficiently with .xdf files, particularly with factor data. When working with larger data sets, the *blocksPerRead* argument is important in controlling how much data is processed in memory at one time. If too many blocks are read in at one time, you may run out of memory. If too few blocks are read in at one time, you may experience slower total processing time. You can experiment with *blocksPerRead* on your system and time the estimation of a linear model as follows:

```
system.time(
      delayArr <- rxLinMod(ArrDelay ~ DayOfWeek, data = bigAirDS,
          cube = TRUE, blocksPerRead = 30)
)
```

The linear model you get will not vary as you change *blocksPerRead*. To see a summary of your results:

```
summary(delayArr)
```

You should see the following results for the full data set:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = bigAirDS, cube = TRUE,
    blocksPerRead = 30)

Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
File name: C:\MRS\Data\AirOnTime87to12\AirOnTime87to12.xdf
Dependent variable(s): ArrDelay
Total independent variables: 7
Number of valid observations: 145576737
Number of missing observations: 3042918

Coefficients:
               Estimate Std. Error t value Pr(>|t|)     |   Counts
DayOfWeek=Mon  6.365682   0.006810   934.7 2.22e-16 *** | 21414640
DayOfWeek=Tues 5.472585   0.006846   799.4 2.22e-16 *** | 21191074
DayOfWeek=Wed  6.538511   0.006832   957.1 2.22e-16 *** | 21280844
DayOfWeek=Thur 8.401000   0.006821  1231.7 2.22e-16 *** | 21349128
DayOfWeek=Fri  8.977519   0.006815  1317.3 2.22e-16 *** | 21386294
DayOfWeek=Sat  3.756762   0.007298   514.7 2.22e-16 *** | 18645919
DayOfWeek=Sun  6.062001   0.006993   866.8 2.22e-16 *** | 20308838
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 31.52 on 145576730 degrees of freedom
Multiple R-squared: 0.002585 (as if intercept included)
Adjusted R-squared: 0.002585
F-statistic: 6.289e+04 on 6 and 145576730 DF,  p-value: < 2.2e-16
Condition number: 1 1
```

Notice that because we specified `cube = TRUE`, we have an estimated coefficient for each day of the week (and not the intercept).

Because the computation is so fast, we can easily expand our analysis. For example, we can compare Arrival Delays with Departure Delays. First, we rerun the linear model, this time using Departure Delay as the dependent variable.

```
delayDep <- rxLinMod(DepDelay ~ DayOfWeek, data = bigAirDS,
      cube = TRUE, blocksPerRead = 30)
```

Next, combine the information from regression output in a data frame for plotting.

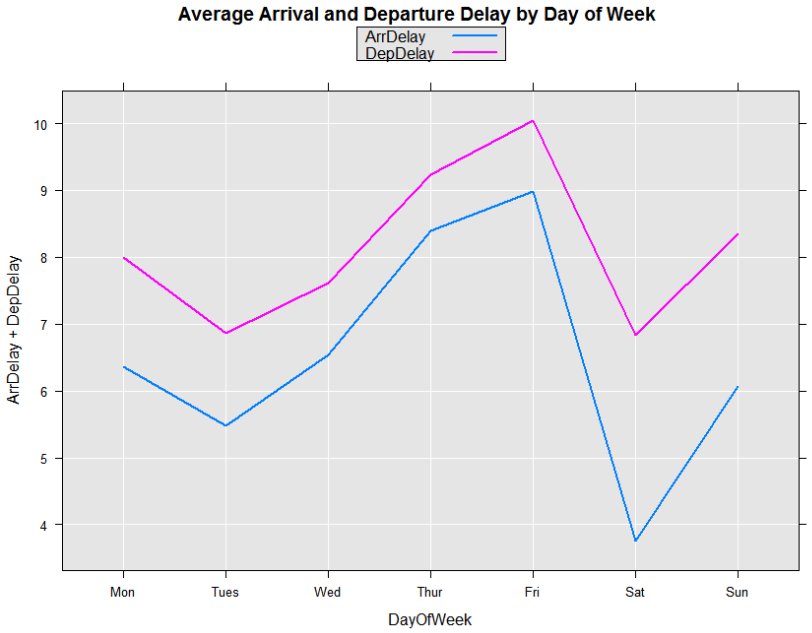
```
cubeResults <- rxResultsDF(delayArr)
cubeResults$DepDelay <- rxResultsDF(delayDep)$DepDelay
```

Then use the following code to plot the results, comparing the Arrival and Departure Delays by the Day of Week

```
rxLinePlot( ArrDelay + DepDelay ~ DayOfWeek, data = cubeResults,
```

```
        title = 'Average Arrival and Departure Delay by Day of Week')
```

You should see the following plot for the full data set:



6.4 Turning Off Progress Reports

By default, ***RevoScaleR*** reports on the progress of the model fitting so that you can see that the computation is proceeding normally. You can specify an integer value from 0 through 3 to specify the level of reporting done; the default is 2. (See help on rxOptions to change the default.) For large model fits, this is usually reassuring. However, if you would like to turn off the progress reports, just use the argument `reportProgress=0`, which turns off reporting. For example, to suppress the progress reports in the estimation of the delayDep rxLinMod object, repeat the call as follows:

```
delayDep <- rxLinMod(DepDelay ~ DayOfWeek, data = bigAirDS,
      cube = TRUE, blocksPerRead = 30, reportProgress = 0)
```

6.5 Handling Larger Linear Models

The data set contains a variable `UniqueCarrier` which contains airline codes for 29 carriers. Because the RevoScaleR Compute Engine handles factor variables so efficiently, we can do a linear regression looking at the Arrival Delay by Carrier. This will take a little longer, of course, than the previous analysis, because we are estimating 29 instead of 7 factor levels.

```
delayCarrier <- rxLinMod(ArrDelay ~ UniqueCarrier,
      data = bigAirDS, cube = TRUE, blocksPerRead = 30)
```

Next, sort the coefficient vector so that the we can see the airlines with the highest and lowest values.

```
dcCoef <- sort(coef(delayCarrier))
```

Next, use the *head* function to show the top 10 with the lowest arrival delays:

```
head(dcCoef, 10)
```

In the full data set the result is:

```
      UniqueCarrier=HA      UniqueCarrier=KH      UniqueCarrier=VX
            -0.175857              1.156923              4.032345
    UniqueCarrier=9E UniqueCarrier=ML (1)      UniqueCarrier=WN
             4.362180              4.747609              5.070878
UniqueCarrier=PA (1)      UniqueCarrier=OO      UniqueCarrier=NW
             5.434067              5.453268              5.467372
    UniqueCarrier=US
             5.897604
```

Then use the *tail* function to show the bottom 10 with the highest arrival delays:

```
tail(dcCoef, 10)
```

With all of the data, you should see:

```
UniqueCarrier=MQ UniqueCarrier=HP UniqueCarrier=OH UniqueCarrier=UA
        7.563716         7.565358         7.801436         7.807771
UniqueCarrier=YV UniqueCarrier=B6 UniqueCarrier=XE UniqueCarrier=PS
        7.931515         8.154433         8.651180         9.261881
UniqueCarrier=EV UniqueCarrier=PI
        9.340071        10.464421
```

Notice that Hawaiian Airlines comes in as the best in on-time arrivals, while United is closer to the bottom of the list. We can see the difference by subtracting the coefficients:

```
sprintf("United's additional delay compared with Hawaiian: %f",
    dcCoef["UniqueCarrier=UA"]-dcCoef["UniqueCarrier=HA"])
```

which results in:

```
[1] "United's additional delay compared with Hawaiian: 7.983628"
```

6.6 Estimating Linear Models with Many Independent Variables

One ambitious question we could ask is to what extent the difference in delays is due to the differences in origins and destinations of the flights. To control for Origin and Destination we would need to add over 700 dummy variables in the full data set to represent the factor levels of *Origin* and *Dest*. The RevoScaleR Compute Engine is particularly efficient at handling this type of problem, so we can in fact run the regression:

```
delayCarrierLoc <- rxLinMod(ArrDelay ~ UniqueCarrier + Origin+Dest,
    data = bigAirDS, cube = TRUE, blocksPerRead = 30)
```

```
dclCoef <- coef(delayCarrierLoc)
sprintf(
"United's additional delay accounting for dep and arr location: %f",
   dclCoef["UniqueCarrier=UA"]- dclCoef["UniqueCarrier=HA"])
paste("Number of coefficients estimated: ", length(!is.na(dclCoef)))
```

Accounting for origin and destination reduces the airline-specific difference in average delay:

```
[1] "United's additional delay accounting for dep and arr location: 2.045264"
[1] "Number of coefficients estimated:  780"
```

We can continue in this process, next adding variables to take account of the hour of day that the flight departed. Using the `F` function around the `CRSDepTime` will break the variable into factor categories, with one level for each hour of the day.

```
delayCarrierLocHour <- rxLinMod(ArrDelay ~
      UniqueCarrier + Origin + Dest + F(CRSDepTime),
      data = bigAirDS, cube = TRUE, blocksPerRead = 30)
dclhCoef <- coef(delayCarrierLocHour)
```

Now we can summarize all of our results:

```
sprintf("United's additional delay compared with Hawaiian: %f",
    dcCoef["UniqueCarrier=UA"]-dcCoef["UniqueCarrier=HA"])
paste("Number of coefficients estimated: ", length(!is.na(dcCoef)))
sprintf(
   "United's additional delay accounting for dep and arr location: %f",
   dclCoef["UniqueCarrier=UA"]- dclCoef["UniqueCarrier=HA"])
paste("Number of coefficients estimated: ", length(!is.na(dclCoef)))
sprintf(
   "United's additional delay accounting for location and time: %f",
   dclhCoef["UniqueCarrier=UA"]-dclhCoef["UniqueCarrier=HA"])
paste("Number of coefficients estimated: ", length(!is.na(dclhCoef)))
```

For the full data set, the results look like:

```
[1] "United's additional delay compared with Hawaiian: 7.983628"
[1] "Number of coefficients estimated:  30"
[1] "United's additional delay accounting for dep and arr location: 2.045264"
[1] "Number of coefficients estimated:  780"
[1] "United's additional delay accounting for location and time: 0.837038"
[1] "Number of coefficients estimated:  805"
```

6.7 Predicting Airline Delay

Using the estimated regression coefficients, write a function to estimate the predicted delay given the carrier, the airport of origin, the destination airport, and the departure time (0 – 24 hours):

```
expectedDelay <- function( carrier = "AA", origin = "SEA",
    dest = "SFO", deptime = "9")
```

```
{
    coeffNames <- c(
    sprintf("UniqueCarrier=%s", carrier),
    sprintf("Origin=%s", origin),
    sprintf("Dest=%s", dest),
    sprintf("F_CRSDepTime=%s", deptime))
    return (sum(dclhCoef[coeffNames]))
}
```

We can use this function to compare, for example, the expected delay for trips going from Seattle to New York, or Newark, or Honolulu:

```
# Go to JFK (New York) from Seattle at 5 in the afternoon on United
expectedDelay("AA", "SEA", "JFK", "17")
# Go to Newark from Seattle at 5 in the afternoon on United
expectedDelay("UA", "SEA", "EWR", "17")
# Or go to Honolulu from Seattle at 7 am on Hawaiian
expectedDelay("HA", "SEA", "HNL", "7")
```

The three expected delays are calculated (using the full data set) as:

```
[1] 12.45853
[1] 17.02454
[1] 1.647433
```

6.8 Computing a Large Scale Regression Using a Compute Cluster

Up to now, all of our examples have assumed you are running your computations on a single computer, which might have multiple computational cores. Many users with large data to analyze, however, have access to compute clusters that work together to provide greater computing power. With RevoScaleR, you can easily connect to an HPC Server or Hadoop cluster and distribute your computation among the various nodes of the cluster. Here we will show a quick example of using an HPC Server cluster consisting of a head node and one or more compute nodes. For more examples, see the *RevoScaleR Distributed Computing Guide* (RevoScaleR_Distributed_Computing.pdf).

To make the connection to an HPC Server cluster, you need to know the following pieces of information about your cluster (all of which can be obtained from your system administrator):

- The name of the cluster's head node.
- The name of the network share directory created for use by Microsoft R Services, and the subdirectory of that network share created for your use.
- The path to the Microsoft R Services bin\x64 directory.
- The name of the data directory created to hold .xdf files on each of the nodes.

Once you have this information, you can create your distributed compute context object by calling RxHpcServer, substituting in the information for your setup in as appropriate:

```
myCluster <- RxHpcServer(
    headNode="cluster-head2",
    shareDir= paste("AllShare\\", Sys.getenv("USERNAME"), sep="")
    revoPath="C:\\Program Files\\Microsoft\\MRO-for-RRE\\8.0\\R-3.2.2\\bin\\x64\\",
```

```
    dataPath="C:\\data")
```

Here *headNode* should be the name of the cluster's head node, *shareDir* should be your subdirectory of the network share directory, *revoPath* is the path to the Microsoft R Services bin\x64 directory, and *dataPath* is the path to the data directory on each node containing copies of the .xdf files you will be using. You can then make the cluster connection object active by using *rxSetComputContext*:

```
rxSetComputeContext( myCluster)
```

With your compute context set to the HPC cluster, all of the RevoScaleR data analysis functions (i.e., rxSummary, rxCube, rxCrossTabs, rxLinMod, rxLogit, rxGlm, rxCovCor (and related functions), rxDTree, rxDForest, rxBTrees, rxNaiveBayes, and rxKmeans) will automatically distribute computations across the nodes of the cluster. It is assumed that copies of the data are in the same path on each node. (Other options are discussed in the RevoScaleR Distributed Computing Guide.)

Now you can re-run the large scale regression from Section 6.6, this time just specifying the name of the data file without the path:

```
dataFile <- "AirOnTime87to12.xdf"

delayCarrierLocDist <- rxLinMod(ArrDelay ~ UniqueCarrier+Origin+Dest,
    data = dataFile, cube = TRUE, blocksPerRead = 30)
```

The computations are automatically distributed over all the nodes of the cluster.

To reset the compute context to your local machine, use:

```
rxSetComputeContext("local")
```

7 An Example Using the 2000 U. S. Census

7.1 Workers from Three States from the 5% IPUMS Sample

The built-in data set *CensusWorkers.xdf* provides a subsample of the 2000 5% IPUMS U.S. Census data. It contains information on individuals from 20 to 65 years old in the states of Connecticut, Indiana, and Washington who worked during 2000. First, let's learn a little about the sample data set:

```
sampleDataDir <- rxGetOption("sampleDataDir")
dataFile <- file.path( sampleDataDir, "CensusWorkers")
rxGetInfo(dataFile, getVarInfo=TRUE, numRows=3)
```

The results show that the data set has 6 variables and a little over 300,000 observations.

```
File name: …SampleData\CensusWorkers.xdf
Number of observations: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Variable information:
Var 1: age, Age
       Type: integer, Low/High: (20, 65)
Var 2: incwage, Wage and salary income
       Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
       2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
       Type: integer, Low/High: (21, 52)
Var 6: state
       3 factor levels: Connecticut Indiana Washington
Data (3 rows starting with row 1):
  age incwage perwt    sex wkswork1   state
1  23   22000    39 Female       45 Indiana
2  25   18000    27 Female       52 Indiana
3  43   12000    24 Female       52 Indiana
```

7.2 Plotting the Weighted Counts of Males and Females by Age

Use `rxCube` to compute and plot the Age/Sex distribution by year of age for the data set. This data set has probability weights associated with each observation in the variable `perwt`. We will use those weights in our calculations. As seen above, `age` is an integer variable. We would like to have it treated as a categorical or factor variable in the calculation, so we will use the `F()` function when using `age` in the formula:

```
ageSex <- rxCube(~F(age):sex, data=dataFile, pweights="perwt")
```

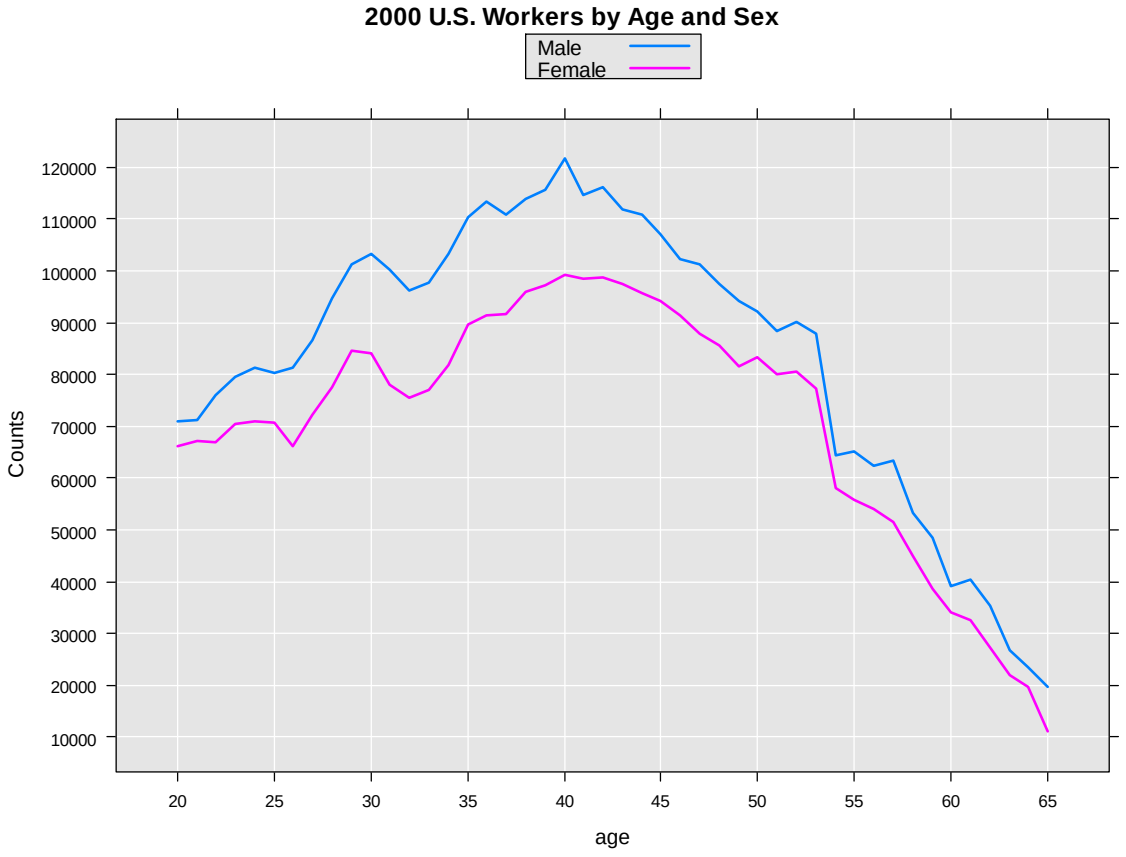
To extract a data frame containing the counts for every combination of age and sex, use *rxResultsDF*:

```
ageSexDF <- rxResultsDF(ageSex)
```

Now plot the number of males and females by age.

```
rxLinePlot(Counts~age, groups=sex, data=ageSexDF,
      title="2000 U.S. Workers by Age and Sex")
```

The resulting graph is shown below:



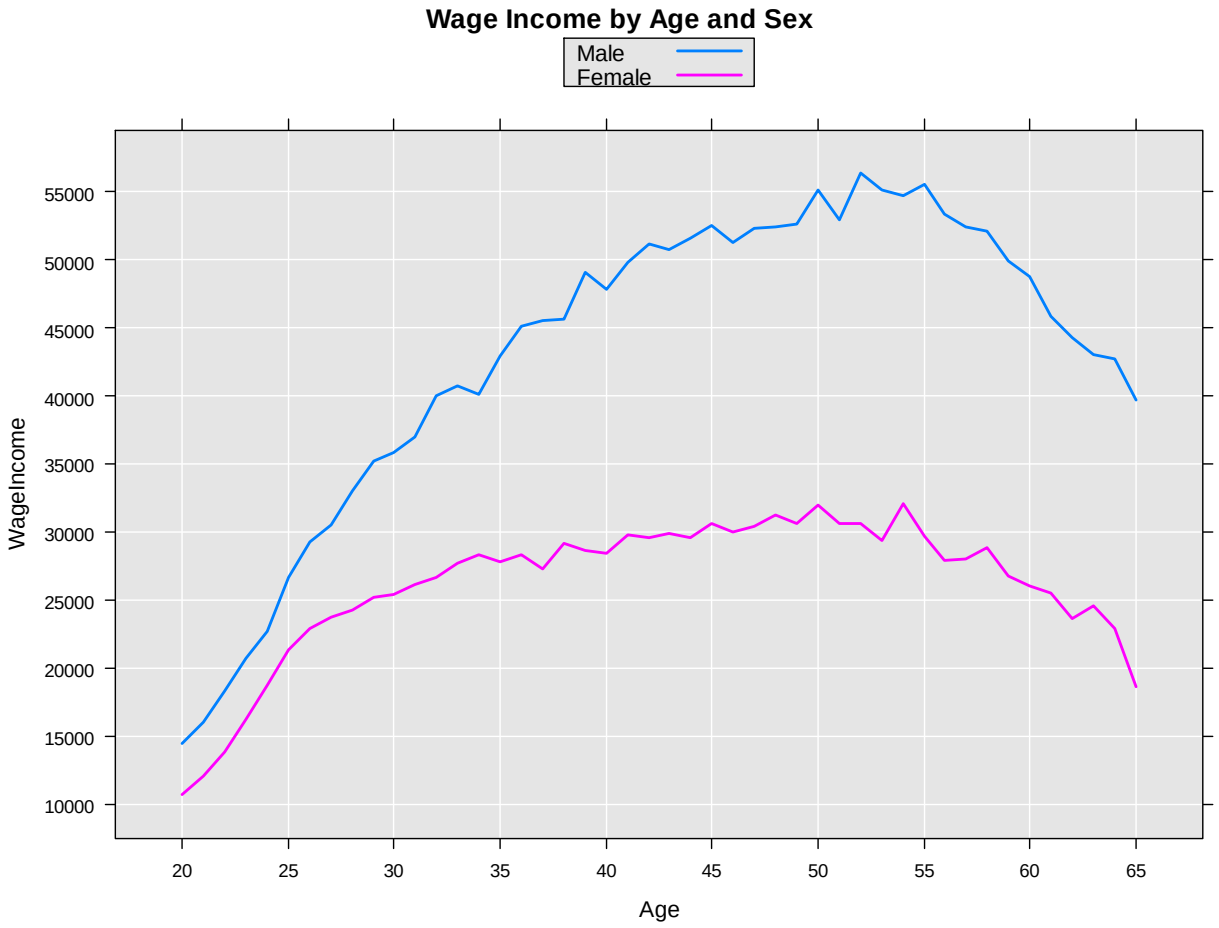
The graph shows us that for the population in the data set, there are more men than women at every age.

7.3 Looking at Wage Income by Age and Sex

Next, run a regression looking at the relationship between age, sex, and wage income and plot the results:

```
wageAgeSexLm  <- rxLinMod(incwage~F(age):sex, data=dataFile,
      pweights="perwt", cube=TRUE)
wageAgeSex <- rxResultsDF(wageAgeSexLm)
colnames(wageAgeSex) <- c("Age","Sex","WageIncome","Counts" )
rxLinePlot(WageIncome~Age, data=wageAgeSex, groups=Sex,
      title="Wage Income by Age and Sex")
```

The resulting graph shows that, in this data set, wage income is higher for males than females at every age:

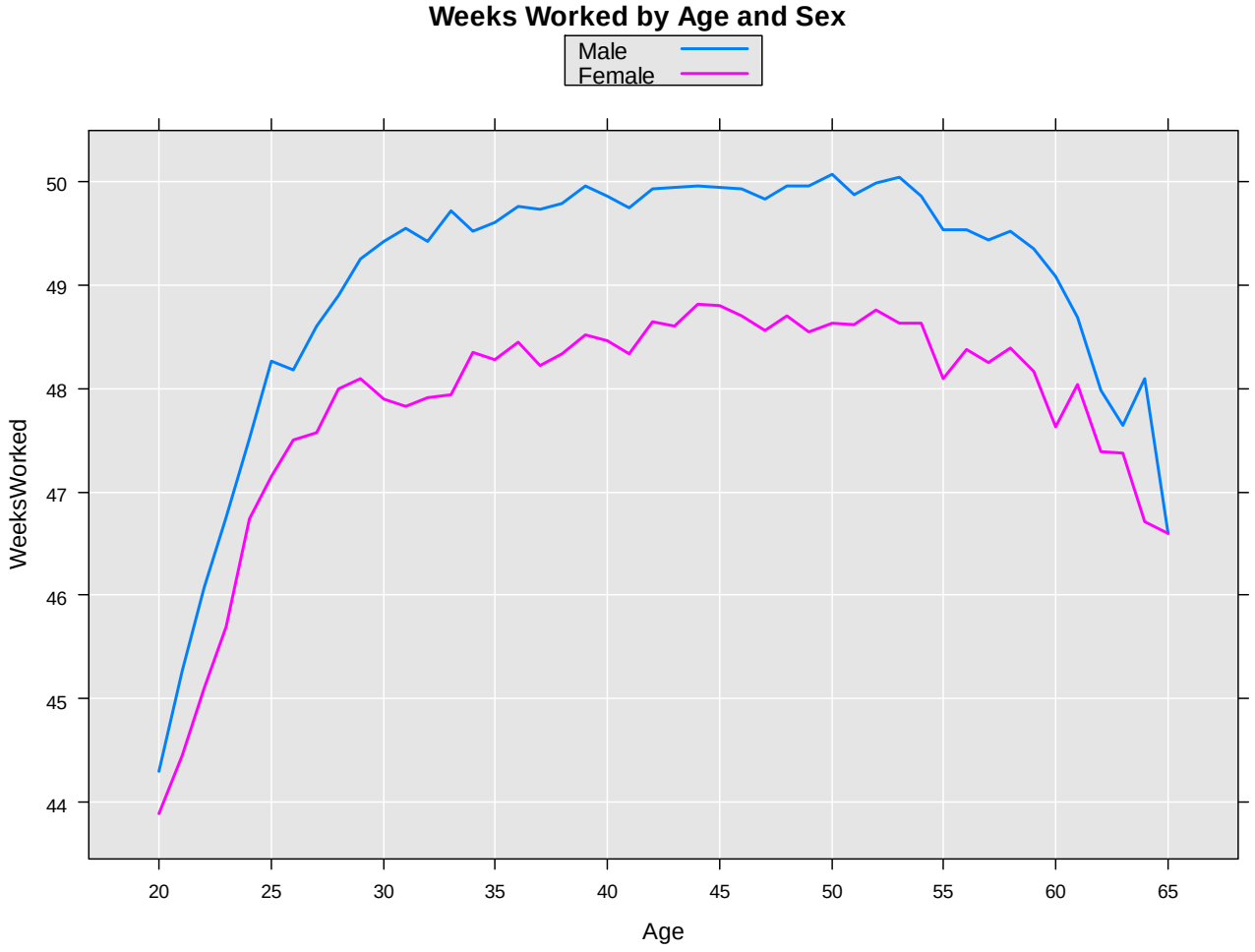


7.4 Looking at Weeks Worked by Age and Sex

We can further explore the data by looking at the relationship between weeks worked and age and sex, following a similar process.

```
workAgeSexLm <- rxLinMod(wkswork1 ~ F(age):sex, data=dataFile,
      pweights="perwt", cube=TRUE)
workAgeSex <- rxResultsDF(workAgeSexLm)
colnames(workAgeSex) <- c("Age","Sex","WeeksWorked","Counts" )
rxLinePlot( WeeksWorked~Age, groups=Sex, data=workAgeSex,
      main="Weeks Worked by Age and Sex")
```

This resulting graph shows that it is also the case that on average females work fewer weeks per year at every age during the working years:



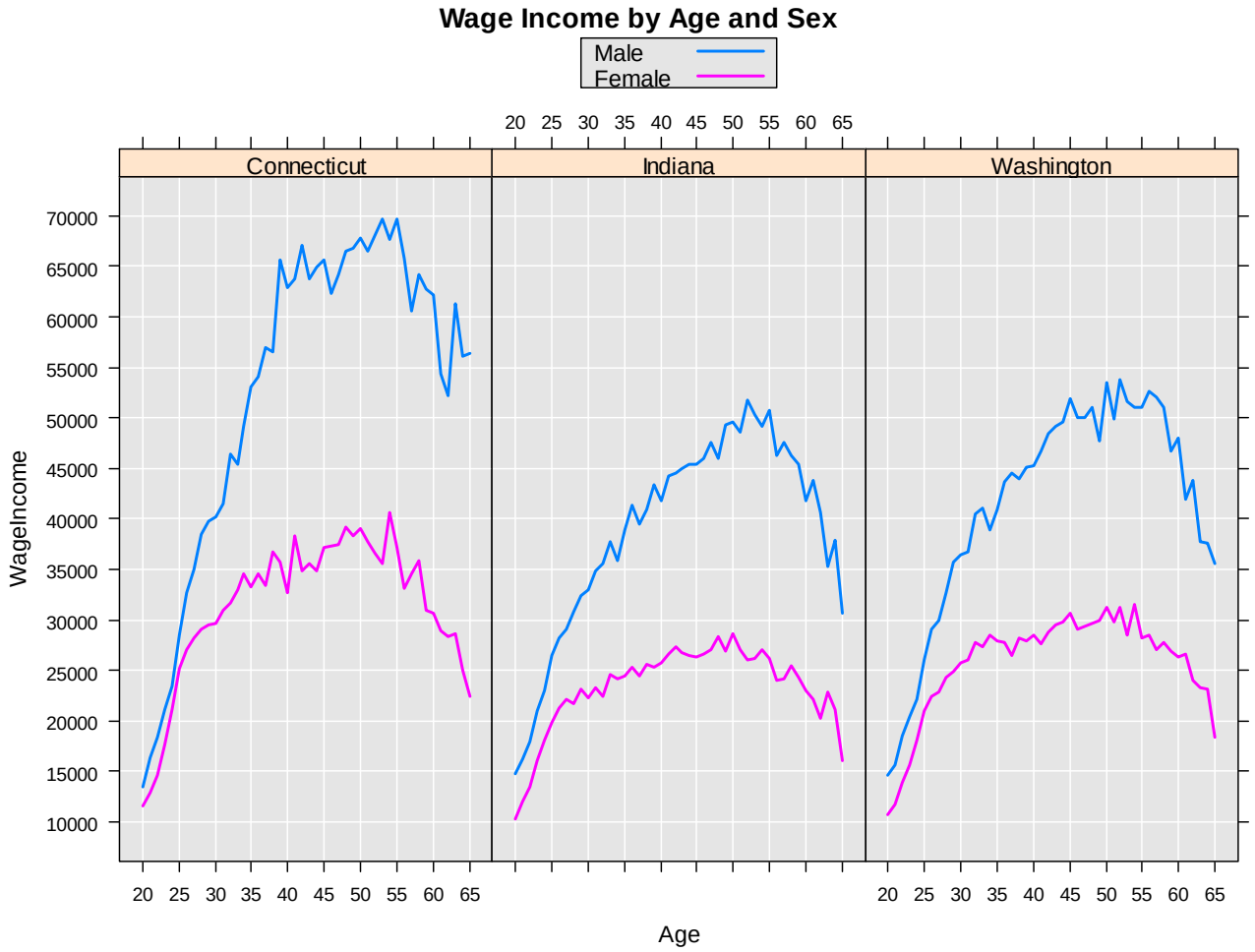
7.5 Looking at Wage Income by Age, Sex, and State

Last, we can drill down by state. For example, let's look at the distribution of wage income by sex for each of the three states. First, do the computations as in the earlier examples:

```
wageAgeSexStateLm  <- rxLinMod(incwage~F(age):sex:state, data=dataFile,
      pweights="perwt", cube=TRUE)
wageAgeSexState <- rxResultsDF(wageAgeSexStateLm)
colnames(wageAgeSexState) <-
      c("Age","Sex","State", "WageIncome","Counts" )
```

Now draw the plot:

```
rxLinePlot(WageIncome~Age|State, groups=Sex, data=wageAgeSexState,
      layout=c(3,1), main="Wage Income by Age and Sex")
```



7.6 Subsetting an .xdf File into a Data Frame

One advantage of storing data in .xdf file is that it is very fast to access a subsample of rows and columns. If not too large, the subset of data can be easily read into a data frame and analyzed using any of the numerous functions available for data frames in R. For example, suppose that we want to perform further analysis on workers in Connecticut who are 50 years old or under:

```
ctDataFrame <- rxDataStep (inData=dataFile,
      rowSelection = (state == "Connecticut") & (age <= 50))
nrow(ctDataFrame)
head(ctDataFrame, 5)
```

The resulting data frame has only 60,755 observations, so can be used quite easily in R for analysis. For example, use the standard R *summary* function to compute descriptive statistics:

```
summary(ctDataFrame)
```

This should give the result:

```
      age           incwage           perwt            sex
 Min.   :20.00   Min.   :     0   Min.   :  2.00   Male  :31926
 1st Qu.:30.00   1st Qu.: 17500   1st Qu.: 16.00   Female:28829
 Median :37.00   Median : 32000   Median : 19.00
 Mean   :36.57   Mean   : 42601   Mean   : 20.30
 3rd Qu.:43.00   3rd Qu.: 50000   3rd Qu.: 25.00
 Max.   :50.00   Max.   :354000   Max.   :152.00
    wkswork1             state
 Min.   :21.00   Connecticut:60755
 1st Qu.:50.00   Indiana    :    0
 Median :52.00   Washington :    0
 Mean   :49.07
 3rd Qu.:52.00
```

8 An Example Analyzing Loan Defaults

This example uses simulated data at the individual level to analyze loan defaults. Suppose that data has been collected every year for 10 years on mortgage holders, and is available in a comma delimited data set for each year. The data contains 5 variables:

- *default* – a 0/1 binary variable indicating whether or not the mortgage holder defaulted on the loan
- *creditScore* – a credit rating
- *yearsEmploy* – the number of years the mortgage holder has been employed at their current job
- *ccDebt* – the amount of credit card debt
- *houseAge* – the age (in years) of the house
- *year* – the year the data was collected

Small versions of the data sets are included with the ***RevoScaleR*** package:

mortDefaultSmall2000.csv
mortDefaultSmall2001.csv
mortDefaultSmall2002.csv
mortDefaultSmall2003.csv
mortDefaultSmall2004.csv
mortDefaultSmall2005.csv
mortDefaultSmall2006.csv
mortDefaultSmall2007.csv
mortDefaultSmall2008.csv
mortDefaultSmall2009.csv

Each file contains 10,000 rows, for a total of 100,000 observations.

Alternatively, you can download a large version of these data sets online, (again, Windows users should download the zip version, mortDefault.zip, and Linux users mortDefault.tar.gz). Each download contains ten files, each of which contains 1,000,000 observations for a total of 10 million:

mortDefault2000.csv
mortDefault2001.csv
mortDefault2002.csv
mortDefault2003.csv
mortDefault2004.csv
mortDefault2005.csv
mortDefault2006.csv
mortDefault2007.csv
mortDefault2008.csv
mortDefault2009.csv

8.1 Importing a Set of Comma Delimited Files

The first step is to import the data into an XDF file format for analysis. First, specify which set of source files you will be using. For the smaller, included data sets, enter:

```
sampleDataDir <- rxGetOption("sampleDataDir")
mortCsvDataName <- file.path(sampleDataDir, "mortDefaultSmall")
```

If you have downloaded the large files, enter the location of your downloaded data:

```
bigDataDir <- "C:/MRS/Data"
mortCsvDataName <- file.path(bigDataDir, "mortDefault", "mortDefault")
```

Next, specify the name of the .xdf file you will create in your working directory (use *"mortDefault"* instead of *"mortDefaultSmall"* if you are using the large data sets):

```
mortXdfFileName <- "mortDefaultSmall.xdf"
```

or

```
mortXdfFileName <- "mortDefault.xdf"
```

8.1.1 Importing a set of files in a loop using *append*

Use the *rxImport* function to import the data. When the first data file is read, a new XDF file is created using the *dataFileName* specified above. Subsequent data files are appended to that XDF file. Within the loop, the name of the imported data file is created.

```
append <- "none"
for (i in 2000:2009)
{
    importFile <- paste(mortCsvDataName, i, ".csv", sep="")
    mortDS <- rxImport(importFile, mortXdfFileName,
        append=append)
    append <- "rows"
}
```

If you have previously imported the data and created the .xdf data source, you can create an .xdf data source representing the file us RxXdfData:

```
mortDS <- RxXdfData( mortXdfFileName )
```

To get a basic summary of the data set and show the first 5 rows enter:

```
rxGetInfo(mortDS, numRows=5)
```

The output should look like the following if you are using the large data files:

```
File name: C:\YourWorkingDir\mortDefault.xdf
```

```
Number of observations: 1e+07
Number of variables: 6
Number of blocks: 20
Compression type: zlib
Data (5 rows starting with row 1):
  creditScore houseAge yearsEmploy ccDebt year default
1         615       10           5   2818 2000       0
2         780       34           5   3575 2000       0
3         735       12           1   3184 2000       0
4         713       15           5   6236 2000       0
5         689       10           5   6817 2000       0
```

8.2 Computing Summary Statistics

Use the *rxSummary* function to compute summary statistics for the variables in the data set, setting the *blocksPerRead* to *2*.

```
rxSummary(~., data = mortDS, blocksPerRead = 2)
```

The following output is returned (for the large data set):

```
Call:
rxSummary(formula = ~., data = mortDS, blocksPerRead = 2)

Summary Statistics Results for: ~.
File name: C:\YourWorkingDir\mortDefault.xdf
Number of valid observations: 1e+07
Number of missing observations: 0

 Name        Mean         StdDev       Min  Max   ValidObs MissingObs
 creditScore  700.0382475 5.000443e+01  432   955 1e+07    0
 houseAge      20.0007868 7.646592e+00    0    40 1e+07    0
 yearsEmploy    5.0042771 2.009815e+00    0    15 1e+07    0
 ccDebt      5003.6681809 1.988664e+03    0 15566 1e+07    0
 year        2004.5000000 2.872281e+00 2000  2009 1e+07    0
 default        0.0049555 7.022068e-02    0     1 1e+07    0
```

8.3 Computing a Logistic Regression

Using the binary *default* variable as the dependent variable, estimate a logistic regression using *year*, *creditScore*, *yearsEmploy*, and *ccDebt* as independent variables. Year is an integer value, so that if we include it "as is" in the regression, we would get an estimate of a single coefficient for it indicating the trend in mortgage defaults. Instead we can treat *year* as a categorical or factor variable by using the *F* function. Then we will get a separate coefficient estimated for each year (except the last), telling us which years have higher default rates - controlling for the other variables in the regression. The logistic regression is specified as follows:

```
logitObj <- rxLogit(default~F(year) + creditScore +
       yearsEmploy + ccDebt,
      data = mortDS, blocksPerRead = 2,
      reportProgress = 1)
```

```
summary(logitObj)
```

You will see timings for each iteration and the final results printed. The results for the large data set are:

```
Call:
rxLogit(formula = default ~ F(year) + creditScore + yearsEmploy +
    ccDebt, data = mortDS, blocksPerRead = 2, reportProgress = 1)

Logistic Regression Results for: default ~ F(year) + creditScore +
yearsEmploy + ccDebt
File name: C:\YourWorkingDir\mortDefault.xdf
Dependent variable(s): default
Total independent variables: 14 (Including number dropped: 1)
Number of valid observations: 1e+07
Number of missing observations: 0
-2*LogLikelihood: 300023.0364 (Residual deviance on 9999987 degrees of
freedom)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -7.294e+00  7.773e-02  -93.84 2.22e-16 ***
F_year=2000 -3.996e+00  3.428e-02 -116.57 2.22e-16 ***
F_year=2001 -2.722e+00  2.148e-02 -126.71 2.22e-16 ***
F_year=2002 -3.865e+00  3.236e-02 -119.43 2.22e-16 ***
F_year=2003 -4.182e+00  3.661e-02 -114.24 2.22e-16 ***
F_year=2004 -4.721e+00  4.625e-02 -102.09 2.22e-16 ***
F_year=2005 -4.903e+00  4.930e-02  -99.45 2.22e-16 ***
F_year=2006 -4.442e+00  4.098e-02 -108.39 2.22e-16 ***
F_year=2007 -3.209e+00  2.546e-02 -126.02 2.22e-16 ***
F_year=2008 -7.228e-01  1.240e-02  -58.28 2.22e-16 ***
F_year=2009    Dropped    Dropped Dropped  Dropped
creditScore -6.761e-03  1.062e-04  -63.67 2.22e-16 ***
yearsEmploy -2.736e-01  2.698e-03 -101.41 2.22e-16 ***
ccDebt       1.365e-03  3.922e-06  348.02 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 6.685
Number of iterations: 10
```

8.4 Computing a Logistic Regression with Many Parameters

If you are using the large *mortDefault* data set, you can continue and estimate a logistic regression with many parameters.

One of the variables in the data set is *houseAge*. As with *year*, we have no reason to believe that the logistic expression is linear with respect to the age of the house. We can look at how the age of the house is related to the default rate by including *houseAge* in the formula as a categorical variable, again using the *F* function. A coefficient will be estimated for 40 of the 41 values of *houseAge*.

```
system.time(
logitObj <- rxLogit(default  ~ F(houseAge) + F(year)+
    creditScore + yearsEmploy + ccDebt,
    data = mortDS, blocksPerRead = 2, reportProgress = 1))
summary(logitObj)
```

The results of the estimation are:

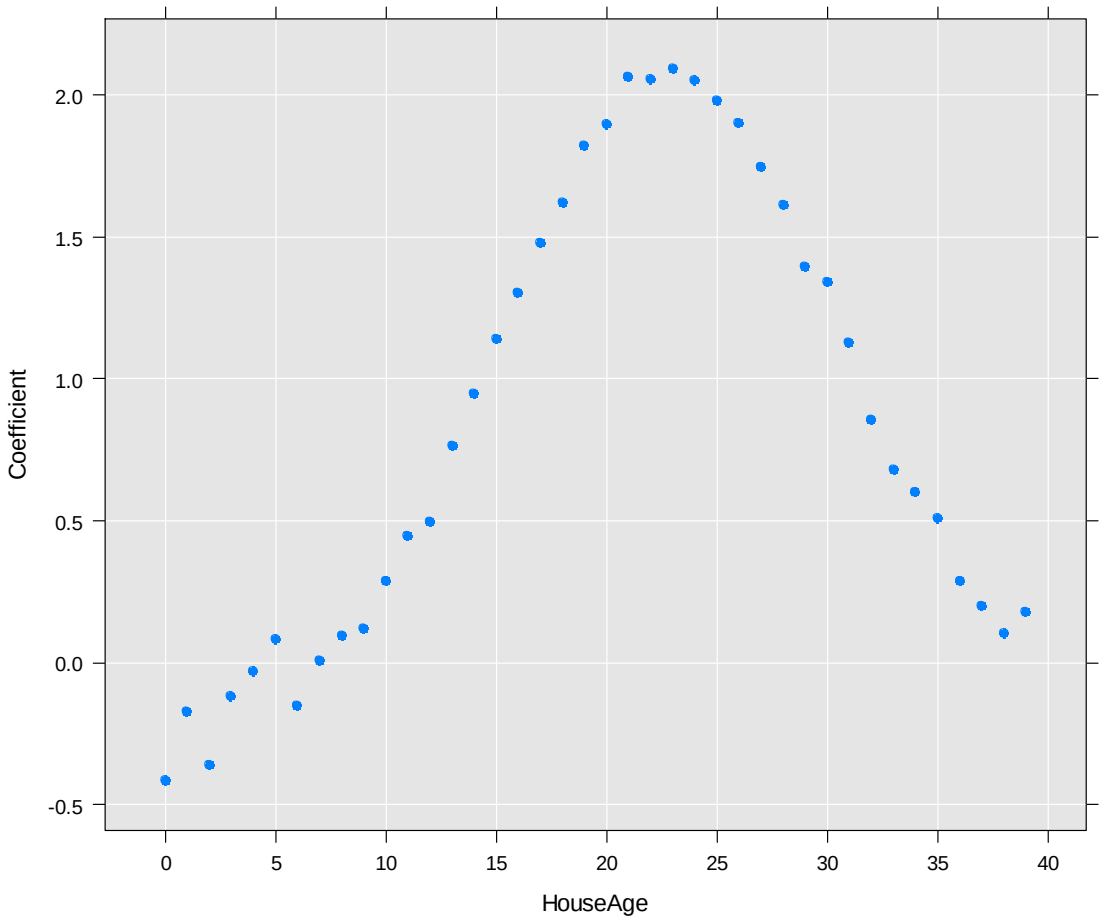
```
Call:
rxLogit(formula = default ~ F(houseAge) + F(year) + creditScore +
    yearsEmploy + ccDebt, data = dataFileName, blocksPerRead = 2,
    reportProgress = 1)

Logistic Regression Results for: default ~ F(houseAge) + F(year) +
    creditScore + yearsEmploy + ccDebt
File name: mortDefault.xdf
Dependent variable(s): default
Total independent variables: 55 (Including number dropped: 2)
Number of valid observations: 1e+07
Number of missing observations: 0
-2*LogLikelihood: 287165.5811 (Residual deviance on 9999947 degrees of
freedom)

Coefficients:
                Estimate Std. Error  z value Pr(>|z|)
(Intercept)   -8.889e+00  2.043e-01  -43.515 2.22e-16 ***
F_houseAge=0  -4.160e-01  2.799e-01   -1.486 0.137262
F_houseAge=1  -1.712e-01  2.526e-01   -0.678 0.497815
F_houseAge=2  -3.593e-01  2.433e-01   -1.477 0.139775
F_houseAge=3  -1.183e-01  2.274e-01   -0.520 0.602812
F_houseAge=4  -2.874e-02  2.177e-01   -0.132 0.895000
F_houseAge=5   8.370e-02  2.106e-01    0.397 0.691025
F_houseAge=6  -1.524e-01  2.088e-01   -0.730 0.465538
F_houseAge=7   5.902e-03  2.033e-01    0.029 0.976837
F_houseAge=8   9.319e-02  2.000e-01    0.466 0.641225
F_houseAge=9   1.180e-01  1.977e-01    0.597 0.550485
F_houseAge=10  2.850e-01  1.954e-01    1.459 0.144683
F_houseAge=11  4.472e-01  1.936e-01    2.310 0.020911 *
F_houseAge=12  4.951e-01  1.929e-01    2.567 0.010266 *
F_houseAge=13  7.642e-01  1.918e-01    3.985 6.75e-05 ***
F_houseAge=14  9.484e-01  1.911e-01    4.963 6.93e-07 ***
F_houseAge=15  1.141e+00  1.905e-01    5.991 2.09e-09 ***
F_houseAge=16  1.305e+00  1.902e-01    6.863 6.76e-12 ***
F_houseAge=17  1.481e+00  1.899e-01    7.797 2.22e-16 ***
F_houseAge=18  1.622e+00  1.897e-01    8.552 2.22e-16 ***
F_houseAge=19  1.821e+00  1.895e-01    9.610 2.22e-16 ***
F_houseAge=20  1.895e+00  1.893e-01   10.012 2.22e-16 ***
F_houseAge=21  2.063e+00  1.893e-01   10.893 2.22e-16 ***
F_houseAge=22  2.056e+00  1.894e-01   10.859 2.22e-16 ***
F_houseAge=23  2.094e+00  1.894e-01   11.056 2.22e-16 ***
F_houseAge=24  2.052e+00  1.895e-01   10.831 2.22e-16 ***
F_houseAge=25  1.980e+00  1.896e-01   10.440 2.22e-16 ***
F_houseAge=26  1.901e+00  1.898e-01   10.014 2.22e-16 ***
F_houseAge=27  1.748e+00  1.901e-01    9.193 2.22e-16 ***
```

```
F_houseAge=28  1.613e+00  1.906e-01    8.466 2.22e-16 ***
F_houseAge=29  1.397e+00  1.913e-01    7.304 2.22e-16 ***
F_houseAge=30  1.340e+00  1.919e-01    6.987 2.82e-12 ***
F_houseAge=31  1.127e+00  1.930e-01    5.840 5.23e-09 ***
F_houseAge=32  8.557e-01  1.951e-01    4.386 1.15e-05 ***
F_houseAge=33  6.801e-01  1.976e-01    3.442 0.000576 ***
F_houseAge=34  6.015e-01  2.002e-01    3.004 0.002666 **
F_houseAge=35  5.077e-01  2.050e-01    2.477 0.013239 *
F_houseAge=36  2.856e-01  2.098e-01    1.361 0.173455
F_houseAge=37  1.988e-01  2.192e-01    0.907 0.364336
F_houseAge=38  1.050e-01  2.311e-01    0.454 0.649508
F_houseAge=39  1.778e-01  2.422e-01    0.734 0.462974
F_houseAge=40    Dropped    Dropped  Dropped  Dropped
F_year=2000   -4.111e+00  3.482e-02 -118.075 2.22e-16 ***
F_year=2001   -2.802e+00  2.185e-02 -128.261 2.22e-16 ***
F_year=2002   -3.972e+00  3.282e-02 -121.025 2.22e-16 ***
F_year=2003   -4.307e+00  3.711e-02 -116.064 2.22e-16 ***
F_year=2004   -4.852e+00  4.676e-02 -103.770 2.22e-16 ***
F_year=2005   -5.019e+00  4.973e-02 -100.921 2.22e-16 ***
F_year=2006   -4.563e+00  4.152e-02 -109.901 2.22e-16 ***
F_year=2007   -3.303e+00  2.587e-02 -127.649 2.22e-16 ***
F_year=2008   -7.461e-01  1.261e-02  -59.171 2.22e-16 ***
F_year=2009      Dropped    Dropped  Dropped  Dropped
creditScore   -6.987e-03  1.079e-04  -64.747 2.22e-16 ***
yearsEmploy   -2.821e-01  2.746e-03 -102.729 2.22e-16 ***
ccDebt         1.406e-03  4.092e-06  343.586 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 5254.541
Number of iterations: 10
```

We can extract the coefficients from *logitObj* for the *houseAge* variables and plot them:

```
cc <- coef(logitObj)
df <- data.frame(Coefficient=cc[2:41], HouseAge=0:39)
rxLinePlot(Coefficient~HouseAge,data=df, type="p")
```

The resulting plot shows that the age of the house is associated with a higher default rate in the middle of the range than for younger and older houses.


Coefficient
2.0
1.5
1.0
0.5
0.0
-0.5
0
5
10
15
20
25
30
35
40
HouseAge

8.5 Compute the Probability of Default

Using the *rxPredict* function, we can compute the probability of default for given characteristics using our estimated logistic regression model as input. First create some vectors containing values of interest for each of the independent variables:

```
creditScore <- c(300, 700)
yearsEmploy <- c( 2, 8)
ccDebt <- c(5000, 10000)
year <- c(2008, 2009)
houseAge <- c(5, 20)
```

Now create a data frame with combinations of those values:

```
predictDF <- data.frame(
      creditScore = rep(creditScore, times = 16),
      yearsEmploy = rep(rep(yearsEmploy, each = 2), times = 8),
      ccDebt      = rep(rep(ccDebt, each = 4), times = 4),
      year        = rep(rep(year, each = 8), times = 2),
      houseAge    = rep(houseAge, each = 16))
```

Using the *rxPredict* function, we can compute the predicted probability of default for each of the variable combinations. The predicted values will be added to the *outData* data set, if it already exists:

```
predictDF <- rxPredict(modelObject = logitObj, data = predictDF,
    outData = predictDF)
predictDF[order(predictDF$default_Pred, decreasing = TRUE),]
```

The results should be printed to your console, with the highest default rate at the top:

```
   creditScore yearsEmploy ccDebt year houseAge default_Pred
29         300           2  10000 2009       20 9.879328e-01
21         300           2  10000 2008       20 9.748884e-01
31         300           8  10000 2009       20 9.377608e-01
13         300           2  10000 2009        5 9.304742e-01
23         300           8  10000 2008       20 8.772219e-01
5          300           2  10000 2008        5 8.638767e-01
30         700           2  10000 2009       20 8.334778e-01
15         300           8  10000 2009        5 7.112344e-01
22         700           2  10000 2008       20 7.035689e-01
7          300           8  10000 2008        5 5.387369e-01
32         700           8  10000 2009       20 4.794787e-01
14         700           2  10000 2009        5 4.500066e-01
24         700           8  10000 2008       20 3.040133e-01
6          700           2  10000 2008        5 2.795344e-01
16         700           8  10000 2009        5 1.308739e-01
25         300           2   5000 2009       20 6.756520e-02
8          700           8  10000 2008        5 6.664645e-02
17         300           2   5000 2008       20 3.321951e-02
27         300           8   5000 2009       20 1.316013e-02
9          300           2   5000 2009        5 1.170657e-02
```

```
19          300           8   5000 2008       20 6.284003e-03
1           300           2   5000 2008        5 5.585628e-03
26          700           2   5000 2009       20 4.410500e-03
11          300           8   5000 2009        5 2.175239e-03
18          700           2   5000 2008       20 2.096317e-03
3           300           8   5000 2008        5 1.032677e-03
28          700           8   5000 2009       20 8.146337e-04
10          700           2   5000 2009        5 7.236564e-04
20          700           8   5000 2008       20 3.864641e-04
2           700           2   5000 2008        5 3.432878e-04
12          700           8   5000 2009        5 1.332594e-04
4           700           8   5000 2008        5 6.319589e-05
```

9 Writing Your Own Chunking Algorithms

All of the main analysis functions in ***RevoScaleR*** (`rxSummary`, `rxLinMod`, `rxLogit`, `rxGlm`, `rxCube`, `rxCrossTabs`, `rxCovCor`, `rxKmeans`, `rxDTree`, `rxBTrees`, `rxNaiveBayes`, and `rxDForest`) use *chunking* or *external memory algorithms*, that is, they analyze each chunk of data separately, combining intermediate results. When all the data has been processed, final results can be calculated and the analysis is complete. Because all of the data does not need to be in memory at one time, you can analyze huge data sets with this type of algorithm.

You can create your own chunking algorithms, using the `rxDataStep` function to automatically chunk through your data set, and for each chunk using arbitrary R functions to process your data. In this section, we will show a simple chunking algorithm for tabulating data implemented using `rxDataStep`. (If you actually have huge data sets to tabulate, use the `rxCrossTabs` or `rxCube` functions built into ***RevoScaleR***; this example is meant for instructional purposes only.)

(The Microsoft package RevoPemaR provides another, more systematic way to create R applications using your own parallel external memory algorithms. See the RevoPemaR Getting Started Guide.)

All updating algorithms can be broken down into four main tasks:

- Initialization: declare and initialize variables needed in the computation
- ProcessData: for each block, perform calculations on the data in the block.
- UpdateResults: combine all of the results of the ProcessData step.
- ProcessResults: when results from all blocks have been combined, do any final computations.

In this example, no initialization is required.

The Process Data step is performed within a `transformFunc` that is called by `rxDataStep` for each chunk of data. In this case we begin with a simple call to the `table` function after converting the chunk to a data frame. The results are then converted back to a data frame with a single row, which will be appended to a data set on disk. So, the call to `rxDataStep` reads in the data chunk-by-chunk and creates a new summary data set where each row represents the "intermediate results" of a chunk.

The AggregateResults function shown below combines the UpdateResults and ProcessResults tasks. The summary data set is simply read into memory and the columns are summed.

To try this out, create a new script *chunkTable.R* with the following contents:

```
chunkTable <- function(inDataSource, iroDataSource, varsToKeep = NULL,
     blocksPerRead = 1 )
{
      ProcessChunk <- function( dataList)
      {
          # Process Data
          chunkTable <- table(as.data.frame(dataList))
          # Convert table to data frame with single row
          varNames <- names(chunkTable)
          varValues <- as.vector(chunkTable)
          dim(varValues) <- c(1, length(varNames))
          chunkDF <- as.data.frame(varValues)
          names(chunkDF) <- varNames
          # Return the data frame
          return( chunkDF )
      }

      rxDataStep( inData = inDataSource, outFile = iroDataSource,
            varsToKeep = varsToKeep,
            blocksPerRead = blocksPerRead,
            transformFunc = ProcessChunk,
            reportProgress = 0, overwrite = TRUE)

       AggregateResults <- function()
       {
           iroResults <- rxDataStep(iroDataSource)
           return(colSums(iroResults))
       }

      return(AggregateResults())
}
```

To test the function, use the sample data *AirlineDemoSmall.xdf* file with a local compute context. We'll call our new `chunkTable` function, processing 1 block at a time so we can take a look at the intermediate results:

```
inDataSource <- file.path(rxGetOption("sampleDataDir"),
    "AirlineDemoSmall.xdf")
iroDataSource <- "iroFile.xdf"
chunkOut <- chunkTable(inDataSource = inDataSource,
    iroDataSource = iroDataSource, varsToKeep="DayOfWeek")
chunkOut
```

You will see the following results:

```
  Monday   Tuesday Wednesday  Thursday    Friday  Saturday    Sunday
   97975     77725     78875     81304     82987     86159     94975
```

To see the intermediate results, we can read the data set into memory:

```
rxDataStep(iroDataSource)
```

```
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
1  33137   27267     27942    28141  28184    25646  29683
2  32407   25607     25915    26106  26211    29950  33804
3  32431   24851     25018    27057  28592    30563  31488
```

And to delete the intermediate results file:

```
file.remove(iroDataSource)
```