Introdução ao R – 2011
Leitura de dados do BCB

Alexandre Rademaker

6 de Fevereiro de 2011

1 Bibliotecas

A principal biblioteca que usaremos é a `SSOAP`, disponível no site `http://omegahat.org/R`. Na realidade, encontrei um pequeno bug na biblioteca que corrigi e diponibilizei para o autor. Segundo o autor, a biblioteca já foi corrigida. Quem tiver problemas, também pode recorrer a versão que disponibilizei em `https://github.com/arademaker/SSOAP`.

A instalação da biblioteca `SSOAP` pode ser feita com o seguinte comando.

```
> install.packages("SSOAP", repos = "http://www.omegahat.org/R",
     type = "source")
```

Para instalar a minha versão, o primeiro passo é fazer o download em formato `tar.gz` do GitHub e depois executar:

```
> install.packages("SSOAP.tar.gz", repos = NULL, type = "source")
```

Obviamente, adicionando o caminho para o arquivo `SSOAP.tar.gz` se este não estiver no mesmo diretório retornado pelo comando `getwd()`.

Também precisaremos de outras duas bibliotecas. A `XML` será usada para ler arquivos XML e a `RCurl` para baixar arquivos de um dado endereço da internet.

```
> library(SSOAP)
> library(XML)
> library(RCurl)
```

2 O arquivo de serviços

Nosso primeiro objetivo é recuperar o arquivo `WSDL` que contém as definições das funções disponibilizadas pelo site do BCB para recuperação de séries temporais. O BCB disponibiliza seus serviços em um endereço `https` cujo certificado está expirado. Para contornar este problema, nossa requisição terá um argumento adicional que faz a requisição a uma url `https` ignorar a verificação do certificado digital.

```
> wsdl <- getURL("https://www3.bcb.gov.br/sgspub/JSP/sgsgeral/FachadaWSSGS.wsdl",
     ssl.verifypeer = FALSE)
```

A variável `wsdl` contém agora um texto, o conteúdo do arquivo `FachadaWSSGS.wsdl` baixado do site do BCB. Este conteúdo é um arquivo XML, cuja estrutura deve ser carregada em memória para poder ser inspecionada. Fazemos isso com a função `xmlInternalTreeParse` do pacote `XML`.

```
> doc <- xmlInternalTreeParse(wsdl)
```

Com a estrutura do arquivo corretamente carregada, usamos a função `processWSDL` do pacote `SSOAP` para reconhecer as funções disponibilizadas pelo BCB e criar funções em R que poderão ser usadas como qualquer outra função que tivessemos programado em R.

```
> bcbdef <- processWSDL(doc)
> ff <- genSOAPClientInterface(def = bcbdef)
```

A variável `ff` contém agora uma estrutura de dados bastante complexa, com todas as definições de funções do site do BCB e tipos de dados. Não irei descrever aqui toda a estrutura. No decorrer do texto ficará claro seu uso.

3 Recuperando Dados

Para facilitar o desenvolvimento, selecionei um pequeno conjunto de códigos de séries temporais do site do BCB para trabalhar. Obviamente, este processo foi manual, visitando o site do BCB, consultando as séries disponíveis e anotando seus códigos para depois criar o vetor:

```
> codigos <- c(4606, 4607, 4608, 4609, 4610, 4611, 4612, 4613,
     4614, 4615, 4616)
> data.ini <- "01/01/1998"
> data.fim <- "01/01/2011"
```

O passo principal agora é recuperar as séries temporais desejadas em um dado período. A função que usaremos é a função `getValoresSeriesXML` que estava definida no arquivo `WSDL` que baixamos e foi gerada em R pelo comando `processWSDL`.

```
> xmlstr <- ff@functions$getValoresSeriesXML(codigos, data.ini,
     data.fim, ssl.verifypeer = FALSE)
> doc <- xmlInternalTreeParse(xmlstr)
```

Finalmente o processamento dos dados recebidos. Desejamos percorrer o conteúdo do arquivo XML [1] recebido procurando pelas tags `SERIE` e transformando todo o conteúdo dentro destas em um data frame. É exatamente isto que o trecho de código a seguir faz e retorna uma lista de data frames na variável `series`. A função `xpathApply` recebe um documento, uma string XPATH [2] e uma função para ser aplicada a cada nó selecionado pela expressão XPATH aplicada ao documento.

```
> series <- xpathApply(doc, "//SERIE", function(s) {
     id <- xmlGetAttr(s, "ID")
     s1 <- xmlSApply(s, function(x) xmlSApply(x, xmlValue))
     s1 <- t(s1)
     dimnames(s1) <- list(NULL, dimnames(s1)[[2]])
     df <- as.data.frame(s1, stringsAsFactors = FALSE)
     df$SERIE <- id
     df
 })
```

A lista de data frames é então transformada em um único data frame com a aplicação da função `rbind` sucessivas vezes. Isto é feito com o comando `Reduce`.

```
> df <- Reduce(rbind, series)
```

Por último, realizamos uma limpeza nos dados, transformando a coluna de datas em tipo `Date`, a coluna dos valores em tipo numérico e a coluna com os identificadores de séries em `Factor`. As colunas antigas são então removidas se a variável `remove.old` estiver com valor `TRUE`, vide a seguir.

```
> df$data <- as.Date(sapply(strsplit(df$DATA, "/"), function(x) paste(c(x[2:1],
     1), collapse = "-")), "%Y-%m-%d")
> df$valor <- as.numeric(df$VALOR)
> df$serie <- factor(df$SERIE)
> if (remove.old) {
     df$BLOQUEADO <- NULL
     df$SERIE <- NULL
     df$DATA <- NULL
     df$VALOR <- NULL
 }
```

Estes trechos de código acima podem ser unidos no corpo de uma função quer receberá um vetor de códigos, datas de início e fim e a variável `remove.old`. A função retorna o data frame resultante.

[1] `http://en.wikipedia.org/wiki/XML`
[2] `http://en.wikipedia.org/wiki/XPath`

```
> getSeries <- function(codigos, data.ini = "01/01/1998", data.fim = "01/01/2011",
     remove.old = TRUE) {
     xmlstr <- ff@functions$getValoresSeriesXML(codigos, data.ini,
         data.fim, ssl.verifypeer = FALSE)
     doc <- xmlInternalTreeParse(xmlstr)
     series <- xpathApply(doc, "//SERIE", function(s) {
         id <- xmlGetAttr(s, "ID")
         s1 <- xmlSApply(s, function(x) xmlSApply(x, xmlValue))
         s1 <- t(s1)
         dimnames(s1) <- list(NULL, dimnames(s1)[[2]])
         df <- as.data.frame(s1, stringsAsFactors = FALSE)
         df$SERIE <- id
         df
     })
     df <- Reduce(rbind, series)
     df$data <- as.Date(sapply(strsplit(df$DATA, "/"), function(x) paste(c(x[2:1],
         1), collapse = "-")), "%Y-%m-%d")
     df$valor <- as.numeric(df$VALOR)
     df$serie <- factor(df$SERIE)
     if (remove.old) {
         df$BLOQUEADO <- NULL
         df$SERIE <- NULL
         df$DATA <- NULL
         df$VALOR <- NULL
     }
     df
 }
```

4 Agregando os dados

Um típico uso dos dados seria agrega-los por ano. Temos duas formas de fazer isso. Vamos começar usando a função `aggregate`. O primeiro passo é, usando a função criada na seção anterior, recuperar os dados das séries desejadas. Vamos usar os valores default para o intervalo de datas.

```
> df <- getSeries(codigos)
```

Vejamos as primeiras linhas do data frame retornado.

```
> head(df)
```

```
        data   valor serie
1 1998-01-01 3358.52  4606
2 1998-02-01 3347.94  4606
3 1998-03-01 3305.03  4606
4 1998-04-01 4181.86  4606
5 1998-05-01 3497.74  4606
6 1998-06-01 3969.04  4606
```

Para agregar os dados por ano, precisamos extrair o componente ano das datas. Isto pode ser feito com a função `cut`. Na realidade, trata-se da função `cut.Date` que foi selecionada pelo R pelo fato da entrada passada para a `cut` serem do tipo `Date`.

```
> tmp <- df
> tmp$year <- cut(tmp$data, "year")
```

Agora podemos agregar os valor por cada combinação de ano e série. Para cada subconjunto de valores iremos calcular a média.

```
> tmp <- aggregate(valor ~ year + serie, data = tmp, FUN = mean)
> head(tmp)
```

```
        year serie    valor
1 1998-01-01  4606 4178.458
2 1999-01-01  4606 4084.008
3 2000-01-01  4606 3787.253
4 2001-01-01  4606 3937.753
5 2002-01-01  4606 3495.687
6 2003-01-01  4606 8408.036
```

Uma outra forma de agregar os dados seria usando a biblioteca `sqldf`. Com ela, podemos manipular data frames como tabelas de um banco de dados relacional. Não sou particularmente um fã desta abordagem, mas é bom saber que existe a alternativa. Observe ainda como a string SQL poderia ser facilmente construída em tempo de execução. Obviamente, neste caso, ela poderia ter sido passada como uma única string.

```
> library(sqldf)
> colunas <- c(maximum = "max(valor)", minimum = "min(valor)",
     soma = "sum(valor)", serie = "serie")
> mysql <- paste("select", paste(apply(cbind(colunas, names(colunas)),
     1, paste, collapse = " "), collapse = ", "), "from df group by serie")
> sqldf(mysql)
```

```
    maximum   minimum       soma serie
1  16264.11  -5847.65 1158524.33  4606
2  16588.86 -10758.28 1147691.16  4607
3  20666.40 -17256.56  -39308.11  4608
4  15214.86  -1090.21  526257.96  4609
5  12222.19  -1090.96  442019.34  4610
6   2992.67     -4.63   84238.65  4611
7   4500.03  -2412.88   20009.95  4612
8   1803.98  -1533.79   -6427.77  4613
9   2912.63  -1025.66   37306.75  4614
10    69.52   -277.47    2398.55  4615
11 19535.67   4986.20 1409155.94  4616
```

5 Visualizando os dados

Para plotar os dados, vamos usar a biblioteca `ggplot2` [3].

```
> library(ggplot2)
```

Não entrarei nos detalhes do uso desta biblioteca, mas os comandos abaixo dão uma boa idéia do que pode ser feito. Observe que os dados são agregados pela função `scale_x_date`. Os códigos das séries servem para definir as cores, ajudando assim a diferenciar as séries (representadas por linhas). O resultado da impressão da variável `p` está na figura 1.

```
> p <- qplot(data, valor, data = df, colour = serie, geom = "line") +
     scale_x_date(format = "%Y") + opts(axis.text.x = theme_text(angle = 90,
     hjust = 0))
```

O mesmo gráfico poderia ainda ser produzido por outras funções do pacote ggplot2. Na realidade, a função qplot é bastante limitada quando comparada com a ggplot.

```
> p1 <- ggplot(df, aes(x = data, y = valor, group = serie, colour = serie)) +
     geom_point() + geom_line() + theme_bw() + opts(panel.grid.major = theme_blank()) +
     xlab(NULL) + ylab(NULL) + opts(axis.text.x = theme_text(angle = 90,
     hjust = 0))
```

[3] `http://had.co.nz/ggplot2/`

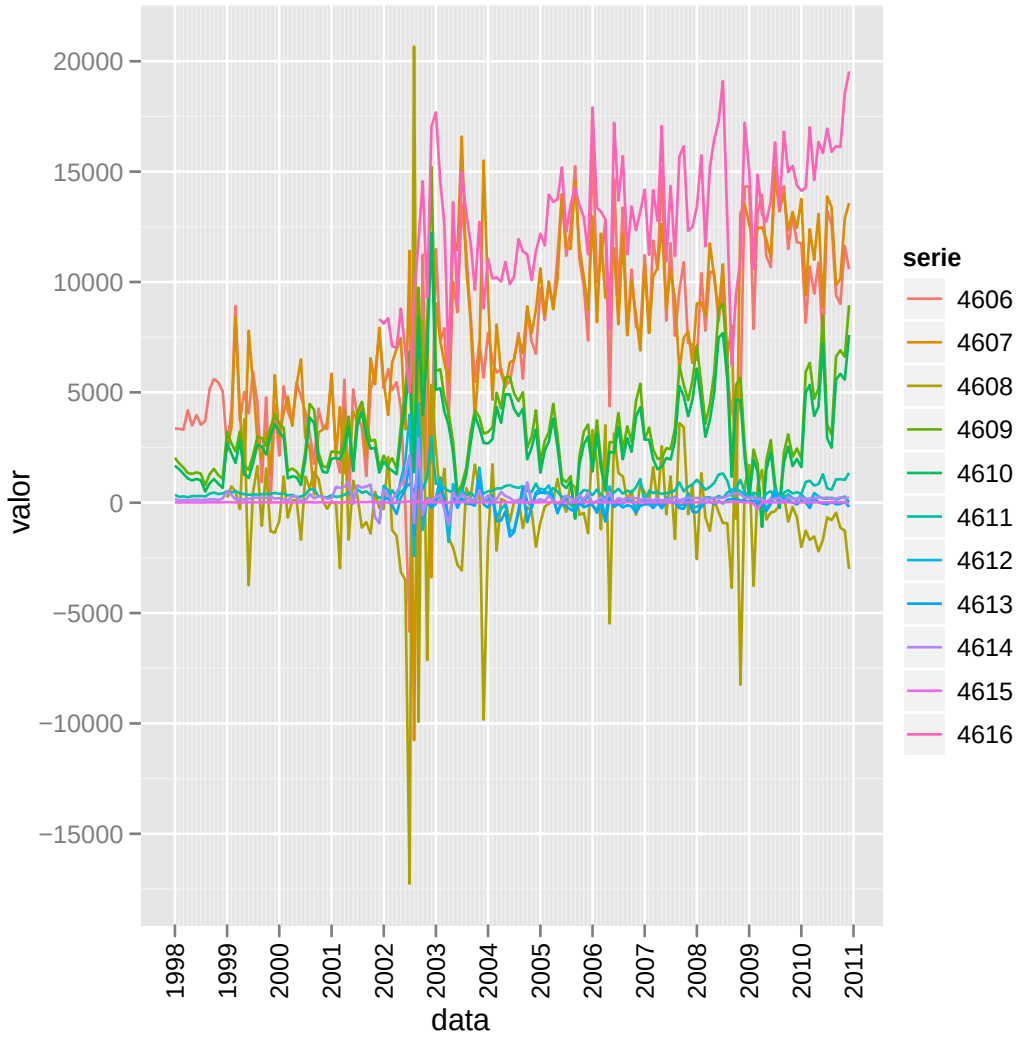

Figura 1: Dados por ano e por série

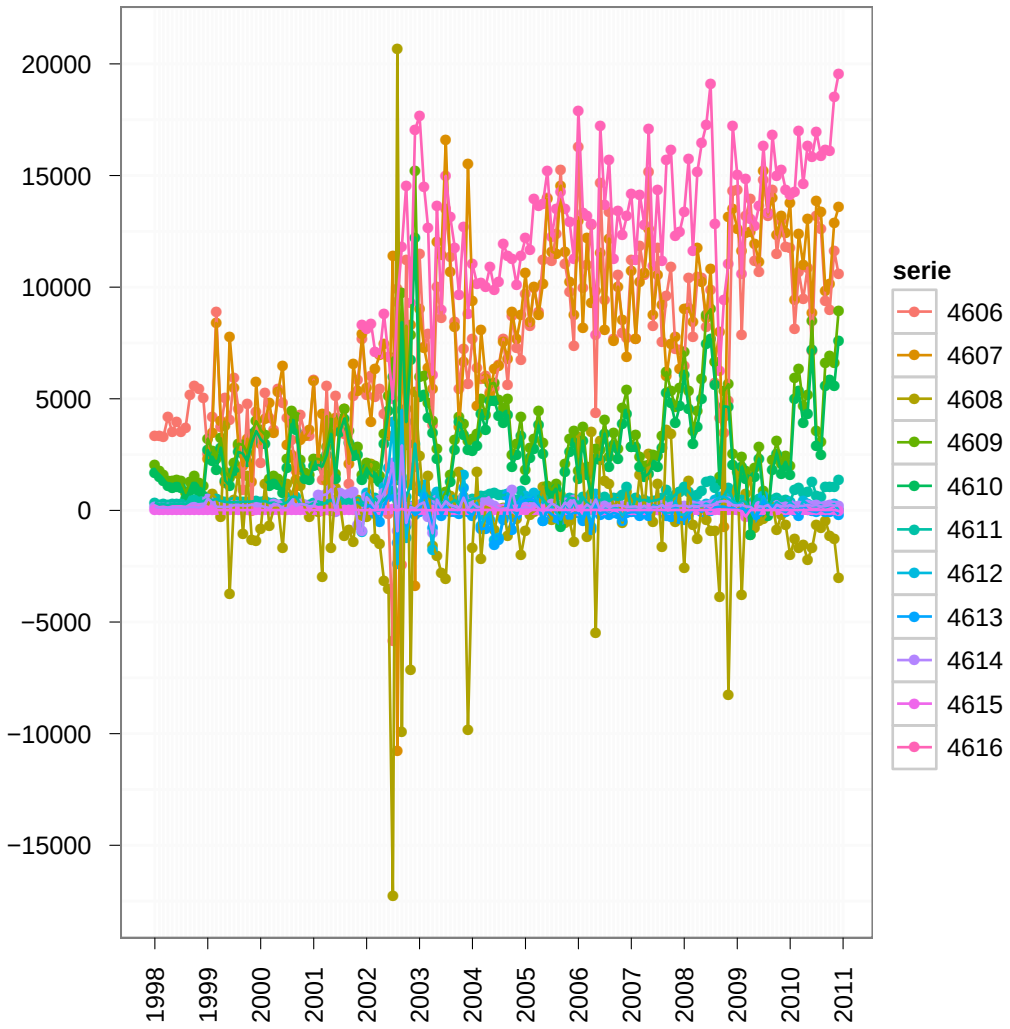

Figura 2: Dados por ano e por série: alternativa 1