

Chapter 15

Spreadsheets

Abstract In this chapter, we explore the possibilities for data exchange offered by the Office Open XML (*OOXML*) standard. Many of the office suites have adopted *OOXML* for their spreadsheets, word processing, and presentation tools. We demonstrate the kinds of functionality that can be built using the tools in the *XML* package to interface with *XML*-based spreadsheets from within *R*. Examples include: reading an entire *xlsx* file into an *R* data frame (or list of data frames, one per sheet); extracting and setting cell values in a worksheet; and adding style information on cells, *R* plots to sheets, and *rda* files to the *xlsx* archive. While the focus is on Excel and *xlsx* files, the ideas presented in this chapter can be extended to other spreadsheet applications, e.g., Google Docs and Open Office, and to other office tools, e.g., Word and PowerPoint. The *ROOXML* [19] package provides the basic infrastructure for Microsoft Office, and, for example, *RWordXML* provides facilities for working with word processing files.

15.1 Introduction: A Background in Spreadsheets

Spreadsheets have had a questionable reputation in the field of statistics. Statisticians have published papers warning users about the computational algorithms in spreadsheets [8, 9], e.g., they can produce negative variances, and it is a commonly held opinion that a comma-separated format is preferable to a spreadsheet because it is easy to use with a variety of statistical software languages. For example, in the online manual, *R* Data Import/Export [12], it states that

The most common *R* data import/export question seems to be “how do I read an Excel spreadsheet”. . . . The first piece of advice is to avoid doing so if possible! If you have access to Excel, export the data you want from Excel in tab-delimited or comma-separated form, and use `read.delim()` or `read.csv()` to import it into *R*.

While this is often a reasonable approach, there is potentially much more information in the spreadsheet file format, e.g., data formatted as currency, dates, times, etc., which would be lost in the export. Of course, the situation is often more complex than that and there are many reasons why statisticians may need or even want to use spreadsheets. Moreover, spreadsheets are a widely used format for data storage and exchange as evidenced by the question “how do I read an Excel spreadsheet?”

When the ISO standardized the Office Open XML format [4] for spreadsheets, word processing documents, and presentations, all the major office suites, including Microsoft Office [13], Apple iWork [2], Libre Office [7], KOffice [6], and Open Office [1] incorporated variants of *OOXML*. The widespread adoption of this standard creates a compelling argument for using spreadsheets as an exchange format. In working with *OOXML*, we have found many advantages to this format:

- The **xlsx** file contains meta information about (among other things) the data format. This information can be used to automate the reading of the data, removing the need for us to specify column classes, quotes, comment characters, character encoding, etc. Using this meta information can both simplify the task of reading data from a spreadsheet into *R* and make it more verifiable and reproducible.
- The workbook may have multiple spreadsheets or spreadsheets that are not simple tables, e.g., they can contain ragged arrays or multiple tables. In this case, the simple solution of using the “Save As” feature to create a CSV file requires multiple exports or results in a potential loss of information.
- The *OOXML* format makes it relatively easy to extract data into *R* in a programmatic, repeatable, reproducible fashion. By programmatically extracting the data from the spreadsheet into *R*, we eliminate the possibility of working with data that differ from the spreadsheet. That is, we eliminate the “middle man” and extra steps in the exchange. If the spreadsheet changes, and we have not exported it as CSV since it was changed, then we will be working with the wrong data. Furthermore, we may need to repeat this process multiple times or want to keep a record of what we have done, and hence will want to access the data in the spreadsheet programmatically.
- The GUI-nature of the spreadsheet makes it a useful format for displaying the results of statistical data analysis. For example, the spreadsheet recomputes dependencies when cell values change, and graphical displays can be included directly in the display, as well as interactive controls. We may want to use the spreadsheet as a report format that contains plots, pivot tables, and formulas that recompute cell values and charts when inputs are updated. Additionally, a spreadsheet can provide attractive titles, headers, etc.
- The *OOXML* format also makes it relatively easy to *generate* spreadsheets from *R* in a programmatic, repeatable, reproducible fashion, including generating tables, formulas, and charts. If the data change, then we can repeat this generative process reliably to revise the **xlsx** file.
- The *OOXML* spreadsheet can be treated as a queryable, updateable database. This is very different from thinking of the spreadsheet as something to “throw away” after exporting the data as CSV. For example, we can query the database for hyperlinks, footnotes, and other meta-information.

As data analysts, we want to remain current with the emerging technologies for accessing and presenting quantitative information, and build new tools to access and publish data in these technologies. We have developed several *R* packages that take advantage of the *XML* structure of *OOXML* documents to work with spreadsheets from within *R*. These are **RExcelXML** [18], **ROpenOffice** [20], and **RGoogleDocs** [23].

The focus in this chapter is on the **RExcelXML** package, which provides an *R* interface with **xlsx** documents. However, **ROpenOffice** and **RGoogleDocs** behave similarly; they offer an *R* interface with Open Office and Google Docs documents, respectively, that are based on the same principles developed in **RExcelXML**. These packages offer functionality for *R* users to both query and modify the contents of workbooks and worksheets. The essential theme is that because the spreadsheet is a **zip** archive that contains *XML* structured content, we can manipulate this content from within *R*. Furthermore, the basic approach demonstrated here carries over to word processing documents and presentation files that follow the *OOXML* format. For example, the **RWordXML** package [24] offers facilities for accessing common elements from within *R* of an *OOXML* word processing document. These packages are by no means complete and polished. Instead, they serve as case studies for exploring the power of an *XML*-based data format and how it can be used within *R*.

There are several other *R* packages for working with Excel spreadsheets. These include **XLConnect** [16], **xlsReadWrite** [17], **gdata** [27], **RExcel** [10], **RODBC** [14], **WriteXLS** [15], **dataframes2xls** [25], and **xlsx** [3]. Some of these have more limited scope,

such as `gdata` and `xlsReadWrite`, and most work only with the older `xls` format. The package `xlsx` works with the `xlsx` format. `XLConnect` handles both formats and is very robust. The `XLConnect` and `xlsx` packages take an entirely different approach from `RExcelXML` and are based on a *Java* library for working with `xlsx` files. The approach is an excellent one as it piggy-backs on the development of toolkits in another language so the focus need only be on making the methods in these other toolkits available in *R*. When that happens, a lot of functionality becomes immediately available in *R*, and updates to the toolkit happen as the other toolkit developers revise their software. One downside, however, is that it is not easy to add specialty functionality, such as adding an `rda` file to the archive.

The packages described in this chapter are not as robust as, e.g., `XLConnect`. Our intention here is to demonstrate an alternative *XML*-based approach for working with *OOXML* formatted files. The goal is to provide an example of how an *R* programmer might design a package to handle spreadsheets that leverages knowledge of *OOXML* and generic tools for handling *XML*. We describe the philosophy behind this implementation and the advantages gained.

Finally, we should also mention interactive spreadsheet capabilities with *R*. When we create a spread/work-sheet from within *R*, the results are fixed. It is possible to add code to the worksheet, and workbook generally, which can make calls back to *R* to dynamically compute values within the worksheet using the *R* engine. This is feasible on Windows via *DCOM* (Distributed Component Object Model). With *DCOM*, we can communicate with Microsoft Excel, Word, PowerPoint, and various other applications as they are running. With this connection, we can programmatically query and modify the spreadsheets, documents, presentations directly from within *R* in much the same way that we can here. For example, we can access sheets, cells, and so on. The way we access these is very different, but the document model is very similar.

This chapter includes discussion of both high- and intermediate-level functions that we have created to interface with *OOXML* formatted spreadsheets. The approach we take is to first put in place functionality that handles the vast majority of spreadsheets easily and simply. Next, by delving more deeply into *SpreadsheetML* (the *OOXML* vocabulary for spreadsheets), we see how a few basic tools for accessing *XML* from within *R* can provide the programmer with the flexibility to handle more complex formats. We describe these functions that give easy access to the *XML* spreadsheet structure, e.g., functions to extract and insert values and formulas in cells and to work with styles. Finally, with an understanding of *OOXML*, we use these intermediate-level functions and the generic tools in the `XML` [21] and `Rcompression` [22] packages to produce customized functionality to, e.g., add *R* plots to a spreadsheet, extract meta information from complex spreadsheets, and use the `xlsx` archive to store atypical auxiliary data such as an `rda` file.

15.2 Simple Spreadsheets

Many workbooks consist of a single spreadsheet with a simple rectangular collection of values, where the columns correspond to variables and the rows to records, and there may be a header row at the top of the sheet to indicate column names. When this is the case, we typically want to read these data simply and directly into a data frame, just as we may read a CSV-formatted file into a data frame with `read.csv()`. The function `read.xls()` in `RExcelXML` does this for us. It determines which columns and rows have data in them and the type of data in each column, and it extracts the values in the cells of the spreadsheet into a data frame. We demonstrate how to use it with two examples.

15.2.1 Extracting a Spreadsheet into a Data Frame

Our first example demonstrates how to use `read.xls()` with a spreadsheet that contains only one simple sheet. The first row contains a header column, and the data are arranged in a simple rectangular form.

Example 15-1 Extracting Data from a World Bank Spreadsheet

The World Bank (<http://www.worldbank.org/>) provides financial and technical assistance to developing countries. In 2010, the Bank launched an Open Data Website (<http://data.worldbank.org/>) that provides access to their data, including Excel tables and reports on topics such as GDP, education, health, and the environment. For example, the World Development Report 2011 on Conflict, Security, and Development [29] has an accompanying Excel spreadsheet [28]. This spreadsheet includes 50 years of data relevant to civil war, terrorism, and trafficking for different countries. A subset of rows and columns has been extracted into a smaller spreadsheet and appears in the screenshot in Figure 15.1. We see that it has six columns/variables, each beginning with a variable name. The columns "Year" and "Cwbattleddeaths" are numeric, whereas the others contain character strings.

We read the contents of the spreadsheet into the data frame with the following call to `read.xlsx()`:

```
cwd = read.xlsx("WorldBank/MiniWDR.xlsx", header = TRUE)
```

The names of the variables in the data frame match the values in the first row of the spreadsheet. We confirm this with

```
names(cwd)
```

```
[1] "Countrycode"      "Country"          "Year"
[4] "RegionA"          "RegionB"          "Cwbattleddeaths"
```

Also, we check that the character strings and numeric values in the spreadsheet are stored in *R* as factor and numeric data types, as we might expect:

```
sapply(cwd, class)
```

```
Countrycode      Country          Year
"factor"         "factor"        "numeric"
RegionA          RegionB Cwbattleddeaths
"factor"         "factor"        "numeric"
```

15.2.2 Extracting Multiple Sheets from a Workbook

The `read.xlsx()` function has a few additional arguments to handle somewhat more complex workbooks than the one-table, one-sheet workbook in Example 15-1 (page 504). For example, if a workbook has multiple sheets, we can specify the sheet that we want to read via the `which` parameter. If there are multiple header rows at the top of the spreadsheet, then we can use the `skip` argument to specify the number of rows to ignore. Moreover, we can read multiple sheets into a list of data frames by supplying a vector to `which`. All of the arguments that control the import of a spreadsheet can be

	A	B	C	D	E	F
	Countrycode	Country	Year	RegionA	RegionB	Cwbattledeaths
2	AFG	Afghanistan	1960	SAR	SAR	0
3	AFG	Afghanistan	1961	SAR	SAR	0
4	AFG	Afghanistan	1962	SAR	SAR	0
5	AFG	Afghanistan	1963	SAR	SAR	0
6	AFG	Afghanistan	1964	SAR	SAR	0
7	AFG	Afghanistan	1965	SAR	SAR	0
8	AFG	Afghanistan	1966	SAR	SAR	0
9	AFG	Afghanistan	1967	SAR	SAR	0
10	AFG	Afghanistan	1968	SAR	SAR	0
11	AFG	Afghanistan	1969	SAR	SAR	0
12	AFG	Afghanistan	1970	SAR	SAR	0
13	AFG	Afghanistan	1971	SAR	SAR	0
14	AFG	Afghanistan	1972	SAR	SAR	0
15	AFG	Afghanistan	1973	SAR	SAR	0
16	AFG	Afghanistan	1974	SAR	SAR	0
17	AFG	Afghanistan	1975	SAR	SAR	0
18	AFG	Afghanistan	1976	SAR	SAR	0
19	AFG	Afghanistan	1977	SAR	SAR	0
20	AFG	Afghanistan	1978	SAR	SAR	0
21	AFG	Afghanistan	1979	SAR	SAR	30000
22	AFG	Afghanistan	1980	SAR	SAR	35000
23	AFG	Afghanistan	1981	SAR	SAR	30000
24	AFG	Afghanistan	1982	SAR	SAR	35000

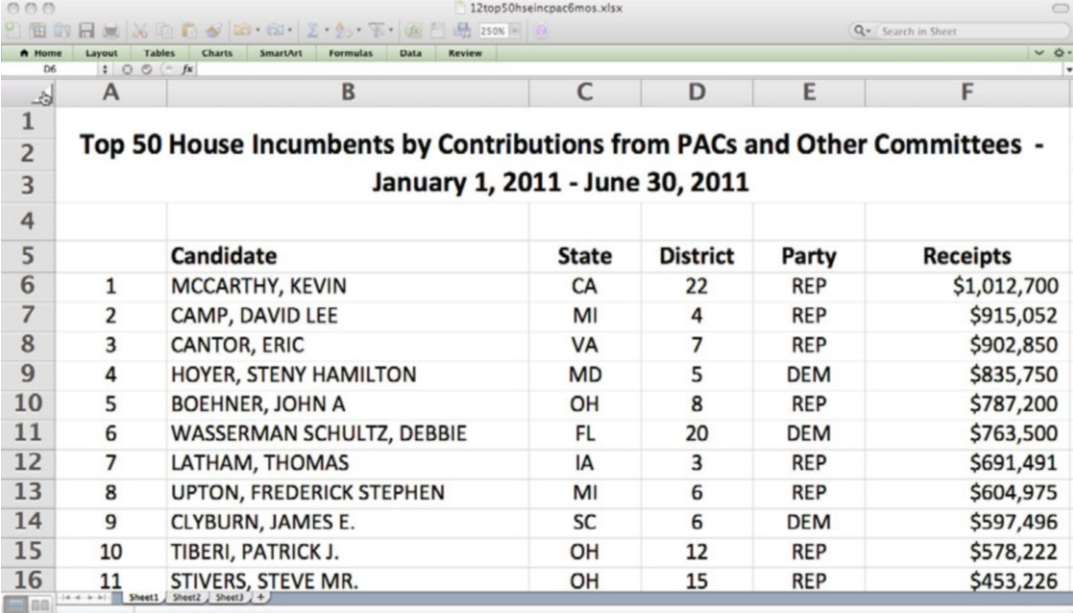
Figure 15.1: World Bank Excel Report on Conflict. This screenshot shows the simple structure of an Excel spreadsheet, which is an extract of the spreadsheet provided by the World Bank Report 2011 on conflict security and development for different countries in different years. The workbook contains only one sheet, which has six columns including the name of the country, year, and the number of deaths due to civil war battles. This information was downloaded from <http://data.worldbank.org/> in September 2011.

passed a vector. This way, we can specify how to handle each sheet independently. We demonstrate how to use these vector arguments in the next example.

Example 15-2 Extracting Federal Exchange Commission (FEC) Data from Multiple Worksheets

The US Federal Election Commission (FEC) is an independent regulatory agency set up to disclose campaign finance information and oversee the public funding of US federal elections. The FEC makes its data publicly available at <http://www.fec.gov/>. One example is its six-month summary, for the first half of 2011, of contributions from Political Action Committees (PAC) and other committees to incumbents running for re-election in the House of Representatives [5]. These data are provided in an Excel workbook (see Figures 15.2 and 15.3), which contains three worksheets. We are primarily interested in the first of these. Although the data in the first sheet are in a rectangular form, the sheet

has a title that occupies the first three rows, the fourth row is empty, and the fifth row contains column headers, which we would like to use as variable names.



	Candidate	State	District	Party	Receipts
1	MCCARTHY, KEVIN	CA	22	REP	\$1,012,700
2	CAMP, DAVID LEE	MI	4	REP	\$915,052
3	CANTOR, ERIC	VA	7	REP	\$902,850
4	HOYER, STENY HAMILTON	MD	5	DEM	\$835,750
5	BOEHNER, JOHN A	OH	8	REP	\$787,200
6	WASSERMAN SCHULTZ, DEBBIE	FL	20	DEM	\$763,500
7	LATHAM, THOMAS	IA	3	REP	\$691,491
8	UPTON, FREDERICK STEPHEN	MI	6	REP	\$604,975
9	CLYBURN, JAMES E.	SC	6	DEM	\$597,496
10	TIBERI, PATRICK J.	OH	12	REP	\$578,222
11	STIVERS, STEVE MR.	OH	15	REP	\$453,226

Figure 15.2: FEC Spreadsheet Report of Campaign Contributions. The screenshot shows the format of the first worksheet in the Excel spreadsheet `12Top50HouseIncumbentByCPAC6Months.xlsx`. Notice that the column names appear in row five, the data begin in row six, and the first column has no column name. The spreadsheet was downloaded from <http://www.fec.gov> in September 2011.

To extract the data, we use the `read.xlsx()` arguments *which*, *skip*, and *header* to specify that we want the data in the first sheet and that we want to skip the first four rows and use the next row as a header. That is,

```
FEC = "FEC/12Top50HouseIncumbentByCPAC6Months.xlsx"
top50 = read.xlsx(FEC, which = 1, skip = 4, header = TRUE)
names(top50)
```

```
[1] "Candidate" "State" "District" "Party" "Receipts"
```

```
dim(top50)
```

```
[1] 50 5
```

The second sheet contains additional information (see Figure 15.3), which we can read into *R* with a second call to `read.xlsx()`. Instead, we demonstrate how to read the data from both worksheets into *R* at once with

```
top50 = read.xlsx(FEC, which = 1:2, header = c(TRUE, FALSE),
                 skip = c(4,0))
```

The return value is a list of two data frames. The second data frame contains the values from "Sheet2". Note that there is no header in this sheet, i.e., the data begin in the first row. This means the variable names in *R* are generic, as would happen with `read.csv()`. We confirm this with

```
names(top50[[2]])

[1] "V1" "V2" "V3"
```

	A	B	C	D
1		8 INCUMBENT	\$2,552,689.07	
2		22 INCUMBENT	\$1,845,236.30	
3		7 INCUMBENT	\$1,327,356.15	
4		26 INCUMBENT	\$1,223,674.86	
5		1 INCUMBENT	\$991,368.18	
6		10 INCUMBENT	\$621,612.12	
7		3 INCUMBENT	\$615,686.00	
8		22 INCUMBENT	\$592,893.01	
9		13 INCUMBENT	\$547,232.62	
10		2 INCUMBENT	\$543,486.85	
11		1 INCUMBENT	\$539,464.00	
12		4 INCUMBENT	\$519,970.00	
13		2 INCUMBENT	\$513,873.03	
14		3 INCUMBENT	\$503,256.00	

Figure 15.3: Second Worksheet in an FEC Workbook. This screen shot shows the contents of the second worksheet of `12Top50HouseIncumbentByCPAC6Months.xlsx`. It has no column names and column C contains currency amounts that contain dollar signs and commas. When we read this sheet into *R* with `read.xlsx()`, each column is extracted into a vector in a data frame and given a generic name, e.g., "V1", and the cell values are converted into the appropriate type, e.g., the currency is converted to numeric.

The `read.xlsx()` function identifies the types of the columns and collapses them to the appropriate type in *R*, e.g., we convert the currency to numeric, not strings with "\$", i.e.,

```
sapply(top50[[2]], class)

      V1      V2      V3
"numeric" "factor" "numeric"
```

Note also that we convert "INCUMBENT" to a factor rather than a string. In addition, if there are blank regions in the data rectangle, then we convert these to missing values.

15.3 Office Open XML

The basic function `read.xlsx()` handles most cases where the data are in a simple rectangular form. How we do this is by extracting information from the *XML* documents within the **xlsx** archive. The `read.xlsx()` function uses the core tools in the **XML** package to extract cell values, find cell formats, etc. When we have more complex spreadsheets that, e.g., have a more flexible format, we can use these same tools to extract content. Rather than converting a worksheet into a data frame all in one step, we provide functions that access various components of the **xlsx** file. These intermediate-level functions offer functionality between the high-level approach of `read.xlsx()` and the basic parsing functions of **XML** (see Chapters 3, 4, and 5). To use these intermediate-level functions, we need to have some knowledge of the **xlsx** archive because these functions are not as all-inclusive as `read.xlsx()`. In this section, we describe the basic structure of the **xlsx** archive and how information is stored in it. This will give us enough understanding to handle more complex extractions and to also create spreadsheets from within *R* that contain, e.g., *R* plots.

15.3.1 The *xlsx* Archive

The Office Open XML (*OOXML*) standard [4, 26] uses a **zip** archive to store the contents of a spreadsheet, word document, or presentation, e.g., PowerPoint or Libre Office. The **zip** files contain data files as well as files that indicate the relationships between the content files. To explore the **zip** archive and the contents of its files, we examine the **xlsx** archive for the simple spreadsheet shown in Figure 15.4.

The documents for this spreadsheet are packaged in a **zip** that contains several directories and about 20 files. There is one file for each worksheet and each image, in addition to files that contain information about styles. Other files contain information that is used to connect the styles and images to the spreadsheet that uses the styles or “contains” the images. We access the files in the archive with the `excelDoc()` function. This function creates an object which allows us to treat the **zip** file as a list in *R* and access the individual files it contains. We call it with:

```
ed = excelDoc("test.xlsx")
```

The `excelDoc()` function relies on the **Rcompression** package to provide general access to the **zip** archive (without writing the files on disk). It provides a convenient interface to the `zipArchive()` function in **Rcompression**. We can examine `ed` to find that our simple spreadsheet consists of 22 files with the following names:

```
names(ed)
```

```
[1] "[Content_Types].xml"
[2] "_rels/.rels"
[3] "xl/_rels/workbook.xml.rels"
[4] "xl/workbook.xml"
[5] "xl/styles.xml"
[6] "xl/theme/theme1.xml"
[7] "xl/worksheets/sheet2.xml"
[8] "xl/worksheets/_rels/sheet1.xml.rels"
[9] "xl/worksheets/_rels/sheet2.xml.rels"
[10] "xl/drawings/_rels/drawing1.xml.rels"
```

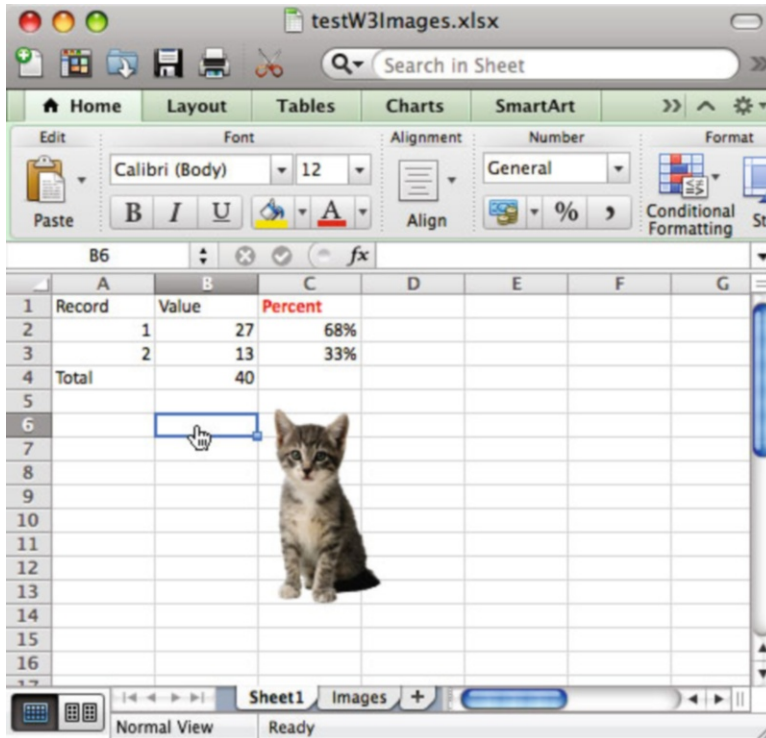



Figure 15.4: Example Spreadsheet. This screenshot shows a small spreadsheet that has a few interesting features. There are two sheets. On the first sheet, the column header for the third column is formatted in a bold-face, red font, and the values in this column are computed using a formula and formatted as a percentage. A PNG file is included on this sheet as well. The second sheet, called "Images", contains two PNG files: a photo and an *R* plot.

```
[11] "xl/drawings/_rels/drawing2.xml.rels"
[12] "xl/worksheets/sheet1.xml"
[13] "docProps/thumbnail.jpeg"
[14] "xl/media/image1.png"
[15] "xl/drawings/drawing2.xml"
[16] "xl/drawings/drawing1.xml"
[17] "xl/media/image2.JPG"
[18] "xl/sharedStrings.xml"
[19] "xl/media/image3.png"
[20] "docProps/core.xml"
[21] "xl/calcChain.xml"
[22] "docProps/app.xml"
```

This structure is quite similar across all *xlsx* files (and is similar to Libre Office, etc.). At its root, the *xlsx* archive contains an XML file called [Content_Types].xml and three directories: *_rels/*, *xl/*, and *docProps/*. Here, we see that the *xl/* directory contains a central workbook file named *workbook.xml* and files for the two worksheets called *worksheets/sheet1.xml* and *worksheets/sheet2.xml*. The core content of the file the user sees is found in this application-

specific directory. The other directories contain information about the data and how they are to appear in the application's interface.

OOXML defines *XML* vocabularies for spreadsheets, as well as word processing documents and presentations. These are *SpreadsheetML*, *WordprocessingML*, and *PresentationML*, respectively. The word processing and presentation archives have similar structures, where the `xl/` directory is replaced by one specific to the document type, e.g., a `word/` directory is used in the word processing **zip** archive. In the following sections, we examine the *SpreadsheetML* vocabulary and some of the auxiliary files in the archive. We see that the **xlsx** document is organized in a distributed manner. The information in the spreadsheet can be stored in a single file, but has been organized for maximum re-use of parts of a document and maximum flexibility in replacing parts, etc. It is confusing at first with the large number of files and level of indirectness, but the structure is quite sensible.

15.3.2 The Workbook

The workbook file called `xl/workbook.xml` keeps track of the worksheets, global settings and other shared components of the workbook. It points to the worksheets via its `<sheets>` node as shown with `xmlRoot(ed[["xl/workbook.xml"]])`:

```
<workbook
  xmlns="http://schemas.../spreadsheetml/2006/main"
  xmlns:r="http://...officeDocument/2006/relationships">
  <fileVersion appName="xl" lastEdited="5"
    lowestEdited="5" rupBuild="21405"/>
  <workbookPr showInkAnnotation="0" autoCompressPictures="0"/>
  <bookViews>
    <workbookView xWindow="0" yWindow="0" windowWidth="25600"
      windowHeight="14840" tabRatio="500" activeTab="1"/>
  </bookViews>
  <sheets>
    <sheet name="Sheet1" sheetId="1" r:id="rId1"/>
    <sheet name="Images" sheetId="2" r:id="rId2"/>
  </sheets>
  <calcPr calcId="140000" concurrentCalc="0"/>
  <extLst>
    <ext xmlns:mx="http://.../mac/excel/2008/main"
      uri="{7523E5D3-25F3-A5E0-1632-64F254C22452}">
      <mx:ArchID Flags="2"/>
    </ext>
  </extLst>
</workbook>
```

We see that neither the worksheet itself nor its file name appears in the workbook. Instead, there is an `r:id` attribute which we use to determine that the first worksheet file name is `xl/worksheets/sheet1.xml`. The information to tie the workbook and worksheet files together is stored in a relationships (i.e., `.rels`) file. This extra layer of indirection can be a bit messy to work with, but it is also very flexible. We describe it in greater detail in Section 15.7 where we provide examples of adding a worksheet to a workbook and an image to a worksheet.

15.3.3 Cells and Worksheets

We next examine the contents of the worksheet *XML* file for the first sheet, i.e., `sheet1.xml`. The root node of this document has eight children with the following names:

```
sheet1 = xmlRoot(ed[["xl/worksheets/sheet1.xml"]])
names(sheet1)
```

```
    dimension      sheetViews      sheetFormatPr    sheetData
"dimension"      "sheetViews"    "sheetFormatPr"  "sheetData"
  pageMargins      pageSetup        drawing           extLst
"pageMargins"    "pageSetup"      "drawing"         "extLst"
```

The `<dimension>` element is:

```
sheet1[["dimension"]]

<dimension ref="A1:C4"/>
```

It has a `ref` attribute that specifies the extent of the rectangular region containing any data in the sheet. The actual cell values are found in the `<sheetData>` node. There is one `<row>` element for each row in the spreadsheet containing content. We examine the first `<row>` node in `<sheetData>` with `sheet1[["sheetData"]][[1]]`. This returns the row with the column headers, "Record", "Value", and "Percent":

```
<row r="1" spans="1:3">
  <c r="A1" t="s"> <v>0</v> </c>
  <c r="B1" t="s"> <v>1</v> </c>
  <c r="C1" s="1" t="s"> <v>2</v> </c>
</row>
```

We see that there is a `<c>` (for cell) element for each nonempty cell in the row. The `<v>` child of `<c>` contains the value of the cell. These are 0, 1, and 2, rather than the strings for the column headers "Record", "Value", and "Percent", as might be expected. The reason for this is that to optimize the use of strings, a single instance of each unique string is stored in a shared strings table for all the worksheets and their cells, and not in the cell's node. This shared table is stored in a separate file called `xl/sharedStrings.xml` in the **xlsx** archive. The cells then reference the string by its index in the shared table. The standard uses zero-based counting so the value of 0 in the A1 cell refers to the first element in the shared strings table. We know that the value refers to a shared string and not the number 0 because the `t` (type) attribute on the parent `<c>` has a value of "s" to denote that the cell contains a string data type. Also, the `s` (style) attribute on the third cell references style information for that cell.

In our example, the shared string table is small, with only four entries, and the first entry contains the string "Record". We confirm this with `ed[["xl/sharedStrings.xml"]]`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sst xmlns="http://schemas.../spreadsheetml/2006/main"
  count="4" uniqueCount="4">
  <si> <t>Record</t> </si>
  <si> <t>Value</t> </si>
  <si> <t>Percent</t> </si>
```

```
<si> <t>Total</t> </si>
</sst>
```

The "Total" string is in the fourth row of the first column of the spreadsheet. The second sheet has no strings, just images so has no entries in this table. If a string is used multiple times, possibly in multiple worksheets, then this approach to storing strings can be very efficient. For example, the spreadsheet in Example 15-1 (page 504) repeats each country name 50 times, once for each year of reporting. Rather than having 50 cells with "Afghanistan" as the value in `<v>`, there are 50 references to the shared string "10". This is very similar to *R*'s factor type.

Lastly, we examine the second row of the worksheet (`sheet1[["sheetData"]][[2]]`) to find:

```
<row r="2" spans="1:3">
  <c r="A2"> <v>1</v> </c>
  <c r="B2"> <v>27</v> </c>
  <c r="C2" s="2"> <f>B2/B4</f> <v>0.6750000000000004</v> </c>
</row>
```

In this row, the value 1 found in the A2 cell corresponds to the number 1 because `<c>` has no `t` attribute so the type defaults to number. The third cell in the row, i.e., C2, has two children, `<f>` and `<v>`. The `<f>` element contains a formula and `<v>` contains the cached value from the last time the formula was calculated. Notice also that this value is 0.675..., rather than 68% as seen in Figure 15.4. The "68%" is a representation of this value. It appears in Excel as 68% because of the formatting applied to it and specified by this style, i.e., the parent `<c>` has a style of "2" that is used to format the cell value.

15.4 Intermediate-Level Functions for Extracting Subsets of a Worksheet

Data can be organized quite flexibly in a spreadsheet, and when the contents of the worksheet are not in a simple tabular form, `read.xlsx()` may give us lots of NAs where there are holes in the table. That is, spreadsheets may be well formatted for people to view and not for data exchange. When this is the case, we may want to work more directly with the `xlsx` archive and the sheet files in order to extract subsets explicitly rather than getting all of the cells in one go. Of course, we can use `read.xlsx()` to extract all of the data from the sheet as one huge data frame and work within *R* to clean it up. However, we may instead want to use the structure of the spreadsheet to extract specific pieces of data.

Another spreadsheet made available by the FEC at <http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx> provides information about PAC contributions in the first half of 2011 (see Figure 15.5). It contains three tables in one sheet. The rows in each table correspond to the source of the contribution, e.g., corporate, labor. The first table provides information about funds received, the second about disbursements, and the third contributions. Blank lines in rows 2, 4, 14, 16, 18, 28, 30, and 32 visually separate the tables from each other and from rows that report column totals for the tables. Notice also that the first table has two columns for each reporting year; one holds the number of receipts and the other the total amount of funds received. The other tables have only one column of information per year.

There are many ways that we may want to organize these data in *R*. We take a simple approach and create separate data frames for each table in the worksheet. When the data are in *R*, we can combine and restructure them into a single data frame more suitable for analysis. Our goal here is

to demonstrate how to work with the Excel file as we extract pieces of a worksheet to create a data frame.

PAC Six-Month Summary (January 1 - June 30, 2011)												
Receipts												
	2001	2003	2005	2007	2009	2011						
Corporate	1,509	\$43,542,145	1,496	\$54,134,395	1,597	\$64,878,716	1,591	\$77,263,727	1,599	\$76,184,307	1,654	\$84,299,493
Labor	309	\$35,473,251	306	\$40,129,076	293	\$47,034,589	271	\$54,850,973	279	\$61,044,226	266	\$63,685,411
Non Connected	909	\$37,469,058	1,016	\$47,647,480	1,243	\$72,541,799	1,079	\$51,333,137	1,176	\$52,128,591	1,206	\$65,293,031
Trade/Member/Health	831	\$33,597,856	865	\$40,033,900	893	\$51,510,871	919	\$65,558,603	955	\$59,016,179	960	\$57,129,827
Cooperative	39	\$926,135	35	\$978,954	36	\$1,725,622	40	\$2,937,207	40	\$2,917,648	36	\$3,019,882
Corp w/o Stock	108	\$2,279,765	92	\$2,252,944	99	\$3,408,505	91	\$3,051,951	100	\$3,155,610	95	\$3,421,511
Leadership PACs							198	\$16,661,051	284	\$19,299,722	351	\$24,556,661
Independent Expenditure-Only								6	\$44,516	112	\$26,585,461	
Total		\$153,288,210		\$185,176,749		\$241,100,102		\$271,656,649		\$273,790,799		\$327,991,277
Disbursements												
	2001	2003	2005	2007	2009	2011						
Corp		\$37,879,047		\$47,418,720		\$63,802,180		\$71,109,926		\$67,678,306		\$75,440,931
Labor		\$29,885,976		\$30,138,575		\$36,290,315		\$40,937,852		\$45,366,040		\$42,101,482
Non Connected		\$28,233,944		\$37,672,126		\$62,173,339		\$45,864,976		\$52,923,017		\$60,569,660
Trade/Member/Health		\$27,541,748		\$31,227,853		\$40,007,764		\$49,185,696		\$43,900,030		\$43,668,622
Cooperative		\$905,169		\$784,688		\$1,097,594		\$2,586,137		\$2,524,108		\$2,728,656
Corp w/o Stock		\$1,942,369		\$1,860,046		\$3,160,958		\$2,576,351		\$3,005,430		\$3,419,235
Leadership PACs								\$14,774,345		\$16,002,225		\$19,204,353
Independent Expenditure-Only												\$6,576,601
Total		\$126,388,253		\$149,102,008		\$206,532,150		\$227,035,283		\$231,399,156		\$253,709,540
Contributions to Candidates, Parties, and Other Committees												
	2001	2003	2005	2007	2009	2011						
Corp		\$28,590,821		\$38,108,007		\$49,401,772		\$57,892,418		\$56,105,184		\$64,899,575
Labor		\$13,183,671		\$13,774,722		\$15,300,906		\$17,071,067		\$18,713,122		\$14,673,935
Non Connected		\$7,130,563		\$9,747,960		\$16,357,230		\$15,937,001		\$17,508,617		\$22,388,156
Trade/Member/Health		\$19,018,205		\$21,918,179		\$28,865,379		\$33,841,637		\$33,843,801		\$34,965,293
Cooperative		\$771,984		\$715,075		\$982,364		\$2,071,698		\$2,090,279		\$2,204,917
Corp w/o Stock		\$1,261,434		\$1,163,071		\$1,787,650		\$1,642,986		\$1,906,206		\$2,409,019
Leadership PACs								\$6,614,011		\$6,024,205		\$6,783,923
Total		\$69,956,678		\$85,427,014		\$112,695,301		\$135,070,818		\$136,191,414		\$148,324,818

Figure 15.5: FEC Worksheet with Multiple Tables. This screenshot shows an Excel spreadsheet published on the Web by the Federal Election Commission. It holds three tables that disclose the source of political contributions by ("Receipts", "Disbursements", and "Contributions to Candidates, Parties and other Committees"). It was downloaded from <http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx> in October, 2011.

15.4.1 The Excel Archive in R

To begin our extraction, we call `excelDoc()` to access the files within the `xlsx` archive:

```
FECExcelDoc = excelDoc("FEC/1pac6mosummary11.xlsx")
```

Although the `ExcelArchive` object is of interest for several tasks, users will typically want to work with a `Workbook` object as this is the entity that contains the worksheets and data.

15.4.2 The Excel Workbook in R

The `workbook()` function creates a different view of the `xlsx` file, i.e., an object that has class `Workbook`. This is an object with two slots: `content`, which is an XML document, and `name`, which is a vector of length 2 giving the name of the `xlsx` file and the name of the XML file that is associated with this workbook. The name is used to remember from whence the content came so that we can save any changes we make to the correct place.

The `workbook()` function takes either the name of the `xlsx` file or the `ExcelArchive` object and returns an object of class `Workbook`. We pass it the archive with

```
wbFEC = workbook(FECEExcelDoc)
```

The `Workbook` class has methods for the `names()` and `[[` functions. The `names()` function returns the names of the worksheets it contains. We see that for our workbook we have three sheets with the typical Excel names:

```
names(wbFEC)

      rId1      rId2      rId3
"Sheet1" "Sheet2" "Sheet3"
```

These are the titles that appear in the tabs along the bottom of the Excel app.

The `[[` method allows you to extract an individual worksheet, either by name or by position. The following are equivalent:

```
ws1 = wbFEC[[1]]
ws1 = wbFEC[["Sheet1"]]
```

The return value is an object of class `Worksheet`.

15.4.3 The Excel Worksheet in R

A `Worksheet` is an object that represents the sheet in the archive. An object of this class describes the sheet in the archive, not its data/contents, i.e., it describes the sheet symbolically. We can think of it like a data frame or matrix. For example, we can ask for its dimensions, e.g.,

```
dim(ws1)

[1] 43 16
```

This tells us that there are 43 rows and 16 columns of active cells in the first worksheet of our workbook. This function ignores the empty columns and rows before and after the main block of data, but it does not ignore the empty columns and rows within the populated region.

In addition to extracting a worksheet with the `[[` operator, we can use `getSheet()` directly to retrieve worksheets as a list of `Worksheet` objects, e.g.,

```
sheets = getSheet(wbFEC, which = 1)
```

In the next example, we will extract the cell values from the middle table (labeled "Disbursements") of the first worksheet. We will perform three extractions to build the data frame. First, we will extract the first column and use these cell values as row names in the data frame.

Then, we extract the data values, and lastly we obtain the row containing the years to use as variable names in our data frame. We use the example to demonstrate several approaches for accessing specific regions in the spreadsheet.

Example 15-3 Extracting a Rectangular Region from an FEC Worksheet

We begin by using the typical Excel specification of a column, e.g., A20:A27, to get the row names for the data frame from the first column of the spreadsheet. We do this with

```
rowNs = ws1["A20:A27"]
head(rowNs)

[1] "Corp"           "Labor"
[3] "Non Connected" "Trade/Member/Health"
[5] "Cooperative"   "Corp w/o Stock"
```

Notice that we need to know the extent of this information (e.g., it is in the first column, rows 20–27) in order to extract it.

We next extract the numeric content of the worksheet. We are interested in the region C20:M27, but we first examine a few of the rows with

```
ws1["c20:J22"]

      C  D      E  F      G  H      I  J
20 37879047 NA 47418720 NA 63802180 NA 71109926 NA
21 29885976 NA 30138575 NA 36290315 NA 40937852 NA
22 28233944 NA 37672126 NA 62173339 NA 45864976 NA
```

We see that columns D, F, H, and J contain NAs as these columns are used only for the first table and we are working with the second table. We can extract only those columns with numbers as follows:

```
disburse = ws1[20:27, seq(3,13, by = 2)]
disburse[ 1:3, 1:4 ]

      C      E      G      I
20 37879047 47418720 63802180 71109926
21 29885976 30138575 36290315 40937852
22 28233944 37672126 62173339 45864976
```

Here we have used *R* terminology to specify the extent, i.e., rows 20:27 and columns 3, 5, 7, etc. That is, the `[` operator has been extended to accept either the spreadsheet cell terminology or *R* terminology for subsetting.

Lastly, we demonstrate how to subset the worksheet by “name” and by logicals. For example, we can extract the row names with

```
ws1[20:27, "A"]
```

and we can extract the years from row 19 with

```
years = ws1[ 19, c(rep(FALSE, 2), rep(c(TRUE, FALSE), 6)) ]
years

      C      E      G      I      K      M
19 2001 2003 2005 2007 2009 2011
```

Now that we have these three pieces, we complete the creation of the data frame for disbursements with

```
row.names(disburse) = rowNs
names(disburse) = paste0("Y", years[1, ])
```

The first row in the spreadsheet in Figure 15.5 has a title that spans multiple columns. Extracting such titles is the topic of the next section.

15.5 Accessing Highly Formatted Spreadsheets

A spreadsheet or workbook can be very effective for displaying data because it allows us to annotate the data with additional information such as explanations of variables, provenance of the data, units for variables, and footnotes about particular values or columns. It also allows us to provide rich headers or titles for columns in each sheet. Often these will be formatted to span multiple columns. One example of a complex, richly formatted spreadsheet is `c07_tabB1.xlsx` that we downloaded from the Census Bureau at <http://blueprod.ssd.census.gov/statab/ccdb/>. Similar data can be found at <http://factfinder.census.gov/>. In the next example, we demonstrate how to read into *R* various parts of the spreadsheet, e.g., footnotes and titles that span multiple columns.

Example 15-4 Working with Detailed Titles and Footnotes in a US Census Spreadsheet

If we view the `c07_tabB1.xlsx` spreadsheet in Excel (or any spreadsheet application), we can see how the different parts deviate from the simple tabular displays in the earlier examples in this chapter (see Figure 15.6). There is a title in cell A1. Cells A3 and A4 contain additional qualifying information. The data are in columns A–Q and in rows 10–3209, inclusive. The values are formatted with commas separating the thousands. The headers span three rows with main titles and subtitles. Additionally, there are footnotes in the document that we want to recover to include in a data description (see Figure 15.7). In this example, we demonstrate how to handle the header information and retrieve the footnotes.

We begin by opening the spreadsheet from within *R* and accessing the sheet called "Main file". Rather than first call `excelDoc()` and then pass the `workbook()` function the `excelArchive` object, we provide `workbook()` with the file name,

```
wb = workbook("cc07_tabB1.xlsx")
sh = wb[[1]]
```

Getting the Header

When we look at the document in a spreadsheet application, we can see that the titles and labels for the columns are above the data and just below the initial title and notes. The titles are recognizable to humans, but hard to recognize algorithmically. We have "Metropolitan area code", "FIPS state and county code" (with a footnote index), "County", "Area, 2000 (square miles)" and "Population". However for the last two of these, the titles span multiple columns and there are subtitles that connect these to the specific columns. For instance, for "Area, 2000" we have Total and Rank. For "Population", we have five columns for four different years—2006, 2005, 2000 and 1990—and an extra column for footnotes associated with the 1990 values. To the right of these, we have columns for "Rank" and "Persons per square mile of land area", again within the general title of "Population", and within these there are subcategories

		Area, 2000 (square miles)		Population						
Metro-politan area code	FIPS state and county code	County	Total	Rank	2006 (July 2005 1)	2000 (April 1 estimates base)	Footnote for 1990 (April 1 estimates base)	1990 (April 1 estimates base)	2006	
	00000	UNITED STATES	3,794,083	(X)	299,398,484	296,507,061	281,424,602		248,790,925	(X)
	01000	ALABAMA	52,419	(X)	4,599,030	4,548,327	4,447,351		4,040,389	(X)
12	33860	01001 Autauga, AL	604	1,724	49,730	48,454	43,671		34,222	959
13	19300	01003 Baldwin, AL	2,027	311	169,162	162,749	143,415		98,280	351
14	21640	01005 Barbour, AL	905	942	28,171	28,291	29,038		25,417	1,480
15	13820	01007 Bibb, AL	626	1,647	21,482	21,454	19,889		16,598	1,744
16	13820	01009 Blount, AL	651	1,572	56,436	55,572	51,034		39,248	879
17		01011 Bullock, AL	626	1,649	10,906	11,011	11,626		11,042	2,373
18		01013 Butler, AL	778	1,218	20,520	20,642	21,399		21,892	1,794
19	11500	01015 Calhoun, AL	612	1,701	112,903	112,242	112,243		116,032	510
20	46740	01017 Chambers, AL	603	1,728	35,176	35,373	36,583		36,876	1,290
21		01019 Cherokee, AL	600	1,742	24,863	24,592	23,988		19,543	1,590
22	13820	01021 Chilton, AL	701	1,436	41,953	41,648	39,593		32,458	1,102
23		01023 Choctaw, AL	921	889	14,656	14,727	15,922		16,018	2,125
24		01025 Clarke, AL	1,253	552	27,248	27,082	27,867		27,240	1,511
25		01027 Clay, AL	606	1,721	13,829	13,920	14,254		13,252	2,178
26		01029 Cleburne, AL	561	1,968	14,700	14,521	14,123		12,730	2,122
27	21460	01031 Coffee, AL	680	1,490	46,027	45,448	43,615		40,240	1,020
28	22520	01033 Colbert, AL	624	1,654	34,766	34,597	34,984		31,666	897
29		01035 Conecuh, AL	853	1,080	13,403	13,227	14,089		14,054	2,221
30	10760	01037 Coosa, AL	666	1,522	11,044	11,133	11,881		11,063	2,365
31		01039 Covington, AL	1,044	720	37,234	36,969	37,631		36,478	1,227
32		01041 Crenshaw, AL	611	1,704	13,719	13,598	13,665		13,635	2,189

Figure 15.6: US Census Spreadsheet of County Population. This screenshot captures the display of c07_tabB1.xlsx in Excel. The original file was downloaded in 2008 as an xls file from <http://blueprod.ssd.census.gov/statab/ccdb/> and converted to a 2007 xlsx format. While this spreadsheet is no longer available on the Web, similar files can be found at <http://factfinder2.census.gov>.

for different years. It is often simpler to specify the names manually, rather than trying to extract them programmatically. We will show here how we can examine the content of the sheet to programmatically determine which cells are merged.

The important part of this computation is to determine which titles span multiple columns. These are “merged cells.”¹ We can find which groups of cells act as one by looking at the `<mergeCells>` node in the worksheet’s XML document in the slot `content`. Then for our sheet,

```
m = getNodeSet(sh@content, "//x:mergeCells", "x")
m[[1]]
```

```
<mergeCells count="14">
  <mergeCell ref="H8:H9"/> <mergeCell ref="J8:J9"/>
  <mergeCell ref="I8:I9"/> <mergeCell ref="D8:D9"/>
  <mergeCell ref="E8:E9"/> <mergeCell ref="C6:C9"/>
  <mergeCell ref="B6:B9"/> <mergeCell ref="K8:N8"/>
  <mergeCell ref="A6:A9"/> <mergeCell ref="O8:Q8"/>
  <mergeCell ref="D6:E7"/> <mergeCell ref="F6:Q7"/>
  <mergeCell ref="F8:F9"/> <mergeCell ref="G8:G9"/>
</mergeCells>
```

¹ To create these in Excel, you highlight the cells and then right click and select "Format cells". Within this, we move to the Alignment tab and click OK.

Note that we need to ensure that there is a `<mergeCells>` node before we subset `m` as not all work sheet documents will have a `<mergeCells>` element.

From the node

```
<mergeCell ref="A6:A9"/>
```

we can see that A6, A7, A8, and A9 are merged together. When we access them in the sheet,

```
sh["A6:A9"]
```

```
[1] "Metro- politan area code" NA
[3] NA                               NA
```

we see that A7–A9 are NA. Similarly, the node

```
<mergeCell ref="D6:E7"/>
```

indicates that the four cells—D6, D7, E6, and E7—are combined; this region in the sheet corresponds to the "Area, 2000 (square miles)" title. The other `<mergeCell>` nodes locate the middle-level titles and lowest-level titles in the spreadsheet.

A possible next step would be to merge the names together to get variable names. One approach is to repeat the values in each of the merged cells and then collapse these down the columns to get the complete names, e.g., F6 becomes "Population.2006.July.1".

Finding the Footnotes

The footnotes are located at the bottom of the document (see Figure 15.7). They are identified by the string "FOOTNOTES" in the first column. To locate them, we first need to find the cell containing that text. We can do this in various ways. One way is to do this directly with R string comparisons. We can get the values of each cell in the first column and find which matches "FOOTNOTES":

```
coll = sh[,1]
start = which(coll == "FOOTNOTES") + 1L
```

We can next find the cell in the column that is blank:

```
end = start + which(is.na(coll[-(1:start)])) [1] - 1L
```

Now, we can get the content of each footnote with

```
fn = coll[start:end]
```

However, several of the footnotes span multiple rows which we want to group together. The pattern is that a new footnote starts with a number; otherwise the text is to be combined with the previous line. Unfortunately, this rule is not quite right. The third footnote has three lines and the third of these starts with the text "1990", which is a year, not the start of a new footnote. We can use a regular expression that looks for one or two digits at the start and this would suffice in this circumstance. For example,

```
i = grepl("^[0-9]{1,2} ", fn)
footnotes = as.character(by(fn, cumsum(i), paste, collapse = " "))
```

But, let's look for a more general and robust approach.

If we have to find the footnotes in a more robust manner that avoids the confusion of the "1900", then we can look at the formatting of the footnotes. To be more specific, we look at the XML content of the cells containing a footnote. These will have the footnote number in a separate (sub-)node from the actual text so we need to find the XML node. Since these cells contain text, and text is stored in a

	A	B	C
3208		56043	Washakie, WY
3209		56045	Weston, WY
3210			
3211			SYMBOLS
3212			NA Not available.
3213			X Not applicable.
3214			Z Less than .05 persons per square mile.
3215			
3216			FOOTNOTES
3217			¹ Federal Information Processing Standards (FIPS) codes
3218			for states and counties.
3219			² Based on 3,141 counties/county equivalents. When counties
3220			share the same rank, the next lower rank is omitted.
3221			³ The Population Estimates base reflects modifications to the
3222			as documented in the Count Question Resolution program and gec
3223			1990 also has adjustment for underenumeration in certain count
3224			⁴ Based on 3,140 counties/county equivalents. When counties
3225			share the same rank, the next lower rank is omitted.
3226			⁵ Persons per square mile was calculated on the basis

Figure 15.7: Footnotes in a US Census Spreadsheet. This screenshot displays the footnotes that appear at the bottom of spreadsheet `c07_tabB1.xlsx` (see Figure 15.6 for a screenshot of the top of the spreadsheet). Some footnotes occupy more than one row, e.g., the third footnote appears in rows 3221 through 3223.

shared string table, we must first get the cell value and then use it to look up the corresponding *XML* node in the collection of shared strings. To do this, we first collect the identifiers for the cells we want

```
ids = sprintf("A%d", start:end)
fnNodes = sh[ids, asNode = TRUE]
```

Then from the nodes in the sheet, we can find the indices of the shared strings

```
idx = as.integer(sapply(fnNodes, function(x)
    xmlValue(x[["v"]])) + 1L)
idx[1]
```

```
368
```

Note that we add 1 to each index since 0-based counting is used in the Excel document. Now we can locate the shared string nodes. To get the value, we index with `idx`. We examine the first one to determine the pattern for which we are looking,

```
ss = getSharedStrings(wb, asNode = TRUE)
ss[idx][1]
```

```
$si
<si>
  <r>
    <t>1</t>
```

```

</r>
<r>
  <rPr>
    <sz val="12"/>
    <rFont val="Courier New"/>
    <family val="3"/>
  </rPr>
  <t xml:space="preserve">
Federal Information Processing Standards...</t>
</r>
</si>

```

We see in this case that there are two `<r>` elements making up the shared string node. The first `<r>` contains a `<t>` element that holds a single number, i.e., the number of the footnote. Basically, any shared string that has two elements is the start of a footnote:

```
which(sapply(ss[idx], xmlSize) > 1)
```

```

si si si si si si si si si si
 1  3  5  8 10 12 13 15 17

```

We can collect the strings for the footnotes in the same manner as before with

```

footnotes = as.character(by(sapply(ss[idx], xmlValue),
                           cumsum(sapply(ss[idx], xmlSize) > 1),
                           paste, collapse = " "))
footnotes = gsub("[0-9] ", "", footnotes)

```

For example, the third footnote that potentially caused problems is

```
footnotes[3]
```

```

[1] "The Population Estimates base reflects ...
for underenumeration in certain counties and cities."

```

15.6 Creating and Updating Spreadsheets

Spreadsheets can be useful for displaying the results from a data analysis or for organizing data for presentation and interactive display. As an example, when preparing an application for permission to raise tuition for our graduate program, we were provided an Excel template in which to insert the requested comparison data. The template standardized the information and its presentation, making the review process more streamlined and consistent. We want to programmatically read this Excel document and add our data to the existing cells in the worksheet. Moreover, we wish to generate a new workbook that contains our report and leave the template untouched as a backup. We may even want to include an **rda** file containing the source data and a script file with the code that was used to create the new **xlsx** archive. That way, we have a self-contained archive for sharing and for future program updates. We demonstrate how to carry out these various actions with the Excel template shown in Figure 15.8.

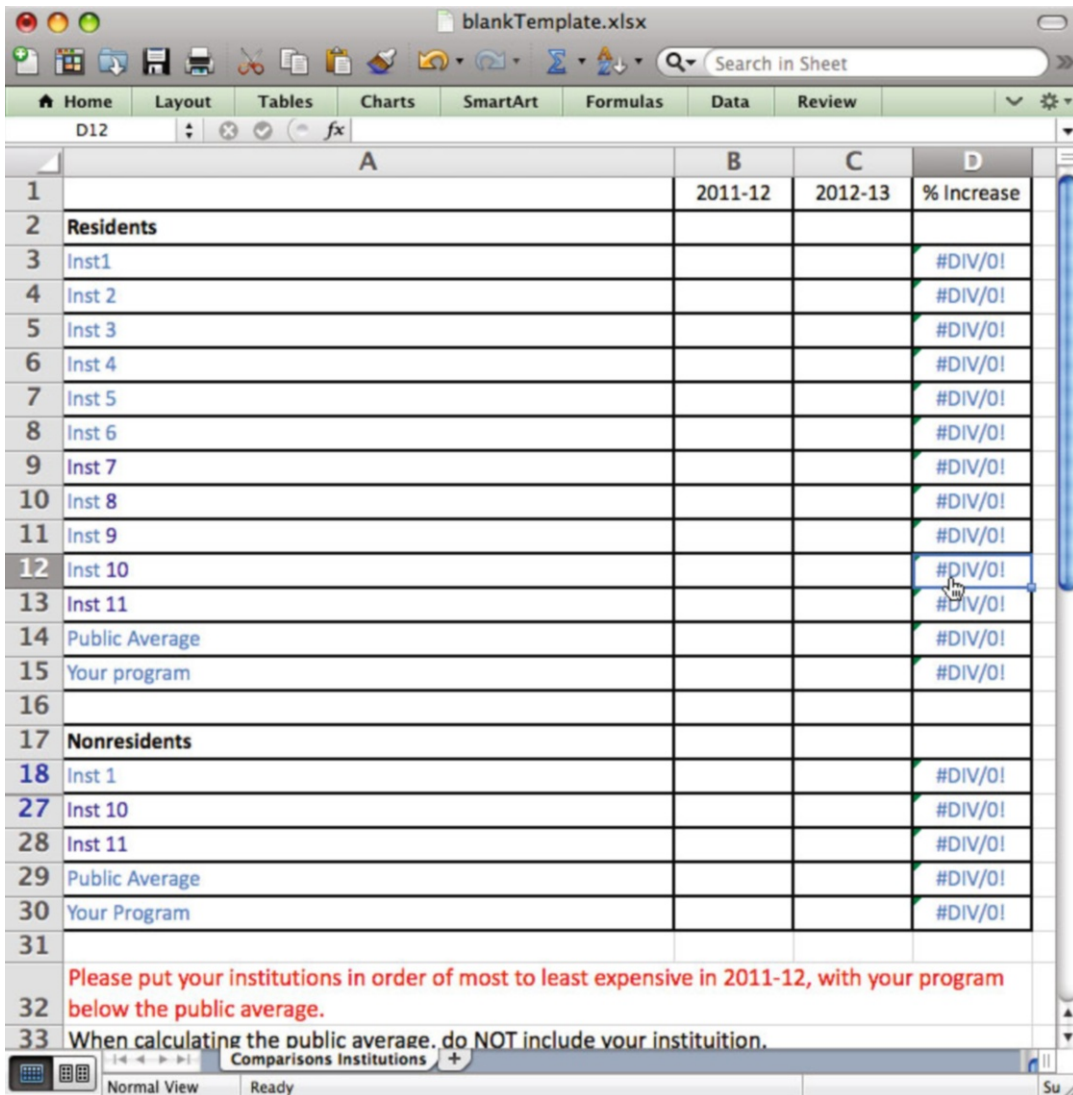


Figure 15.8: Example Report Template in a Spreadsheet. This screenshot shows a spreadsheet called reportTemplate.xlsx to be completed for a report. A copy of this template needs to be filled in with each institution’s name and tuition. We need to add a formula for the average tuition for the public institutions so that the "Public Average" cells will automatically be computed.

15.6.1 Cloning the Excel Document and Entering Cell Values and Formulae

The workbook in Figure 15.8 displays the spreadsheet that we want to fill in by adding data for comparison institutions (stored in a data frame). It is a simple report with four columns, where the rightmost column ("% increase") is calculated by a formula from the cells in columns B and C. We need to fill in the cells in columns A, B, and C. We also want to add formulas to B14 and C14

to compute averages for those rows that correspond to public institutions. (This formula is not part of the original template.)

Briefly, content can be added to an existing worksheet via simple assignments such as `sh[i, j] = "some text"`. The row index `i` can be numeric and the column index can be numeric or the name of a column, e.g., "B", "AC". Alternatively, Excel syntax can be used, e.g., `sh["A3"] = 0`. We can also add more complex objects to the sheet. A vector can be assigned to the worksheet along either a horizontal or vertical axis, e.g., `sh["A2:E2"] = 1:5` sets the cells A2, B2, C2, D2, and E2 to 1, 2, 3, 4, 5, respectively. Similarly, `sh["A2:A6"] = 1:5` sets the cells A2, A3, ..., A6. For a two-dimensional object on the right-hand side of the assignment, a rectangular collection of cells is populated. For example, `sh["C6:E7"] = matrix(1:6, nrow = 3)` sets C6, D6, E6 to 1, 3, 5, respectively and C7, D7, E7 to 2, 4, 6, respectively. When we insert data, we have the choice of updating only the XML document in memory, or we can also add the updated XML document to the `xlsx` archive. This is controlled through the `update` attribute, which takes a logical value indicating whether to update the archive or not.

Example 15-5 Generating an Excel Report from a Template

Rather than change the template document, we create a clone for our actual report and modify that. The function `excelDoc()` creates an Excel document, when the name of the `xlsx` file that we pass to it is not found and the `create` argument is TRUE. In this case, the function creates a new, generic document that has one empty worksheet. However, if we also provide the name of a spreadsheet in the function's `template` argument, then a copy of that spreadsheet is used as the new document. For our report, we clone the report template and then access the worksheet we want to fill in as follows:

```
report = excelDoc("myReport.xlsx", create = TRUE,
                 template = "reportTemplate.xlsx")
reportWB = workbook(report)
sh = getSheet(reportWB, which = 1)[[1]]
```

Next we load the comparison data that will be added to the spreadsheet to create the report:

```
load("comparisons.rda")
head(comparisons)
```

	Institution	Pr	Time	Res11	Res12	NonRes11	NonRes12
1	U Penn	Y	2.000	59503	62478	59503	62478
2	USC	Y	1.500	43871	46065	43871	46065
3	Cornell	Y	1.500	43304	45469	43304	45469
4	CMU	Y	1.500	37185	39044	37185	39044
5	Stanford	Y	1.125	43953	46151	43953	46151
6	Columbia	Y	1.000	44180	46389	44180	46389

In the report, we want to list the institutions in the order of their 2011 resident tuition (in `Res11`). We determine this ordering and pull out Berkeley's values, as follows:

```
comparisons = comparisons[ order(comparisons$Res11,
                                decreasing=TRUE), ]
whichB = which(comparisons$Institution == "Berkeley")
berkeley = comparisons[whichB, ]
comparisons = comparisons[- whichB, ]
```

We begin by filling in column A with institution name,

```
sh[3:13, 1] = comparisons$Institution
sh[15, 1] = berkeley$Institution
```

Next we add the tuition values:

```
sh["B3:B13"] = comparisons$Res11
sh["C3:C13"] = comparisons$Res12
sh["B15:C15"] = berkeley[1, c("Res11", "Res12")]
```

The last step is to add the formulas that compute averages for the public institutions. Different from putting in values, the formula allows the recipient of the spreadsheet to update the data and recalculate.

The function `excelFormula()` takes a formula as a string. Below, we determine the rows in the worksheet to be averaged, and construct the formula as a string,

```
whichPu = which(comparisons$Pr == "N")
formulaStr = paste0("=AVERAGE(", paste("B", (2 + whichPu),
                                         sep = ",", collapse = ","),
                    ")")
```

Now that we have constructed the formula, we call `excelFormula()` to assign it to the appropriate cell in the worksheet:

```
sh["B14"] = excelFormula(formulaStr)
```

We similarly construct the formula for cell C4. The same additions can be made to the nonresident part of the worksheet, which we also do not show here.

The changes that we have made to the spreadsheet have all been made in memory. The actual **xlsx** archive has not been updated because the default assignment is to update the parsed document and to not write the changes to the archive. Our final step then is to update the **zip** file. We call `update()`, passing it the worksheet:

```
update(sh)
```

```
Zip Archive: myReport.xlsx
 [1] "[Content_Types].xml"      "_rels/.rels" ...
 [9] "docProps/core.xml"       "xl/calcChain.xml"
[11] "docProps/app.xml"
```

15.6.2 Working with Styles

In addition to adding content to a worksheet, we can also format the content for display. We can do this for cells or groups of cells and rows and columns by explicitly specifying the appearance of each cell. However, it is best to use an extra layer of abstraction where we assign one or more cells a style that we have separately defined. Then, if we want to update the cell's appearance, we need only make changes to the style definition to have all the associated cells change their appearance. This is one of the benefits of using centralized styles.

We can either define a new style with `createStyle()` or access the styles in the Excel archive with `getStyles()`. Also, `getDocStyles()` allows us to work directly with the styles in the **XML** document,

rather than using the *R* representations that `getStyles()` returns. Once a style has been defined, we can use `setCellStyle()` to assign that style to a cell. We show how in the next example.

Example 15-6 Adding Styles to Cells for an Excel Report

To make it easier to distinguish between public and private universities in the report created in Example 15-5 (page 522), we use different color fonts for the institution name: green for public and blue for private. We create two new styles for this purpose as follows:

```
ft = Font(sz=12L, face="b")
newStPu = createStyle(sh, font = ft, fg = "00FF00")
newStPr = createStyle(sh, font = ft, fg = "0000FF")
```

Next we associate each style with the appropriate rows in the worksheet. The variables `puRows` and `prRows` determine which rows in the worksheet correspond to the public and private institutions. We determined them based on the *R* variable `Pr` in `comparisons`. We assign the two styles to the cells as follows:

```
instPu = paste("A", puRows, sep = "")
instPr = paste("A", prRows, sep = "")
instNames = cells(sh)
setCellStyle(instNames[instPu], newStPu)
setCellStyle(instNames[instPr], newStPr)
```

Since we did not supply the `update` argument when we called `setCellStyle()`, we have taken the default action for this function. This is to update the archive with each modification so there is no need to call `update()` at this point.

15.6.3 Inserting Other Content into the Archive

We have seen that the archive is a collection of files, and we also can store additional files in the archive. The spreadsheet applications will ignore them, but carry them around in the archive for us to use. Take our report for example. We can add to the archive an `rda` file that contains the data frame that was used to produce the report. This addition makes the archive a self-contained document. The data frame contains all the necessary information to fill in and style the cells, so by storing it in the archive, the archive has the data needed to reproduce or update the report.

The `RExcelXML` package uses the `Rcompression` package to access the files within the `xlsx` archive. We can use it for the same purpose, e.g., to access and add an extra file to an archive. In the following example, we demonstrate two ways to do this. The first uses the `excelDoc()` function and `[` available in `RExcelXML`. The second approach directly uses the `zipArchive()` function in `Rcompression`; the function on which the `excelDoc()` relies.

Example 15-7 Adding an rda File to an Excel Archive

We want to add the serialized data frame `comparisons` to our report `myReport.xlsx`, which was created in Example 15-5 (page 522). We begin by opening the archive with the following call to `excelDoc()`:

```
report = excelDoc("myReport.xlsx")
```

Then we serialize the `comparisons` object to a raw vector and add it to `report` with:


```
report[["comparisons.rda"]] = serialize(comparisons, NULL)
```

We can check to see that the **rda** file is indeed in the archive:

```
length(report)
```

```
[1] 12
```

```
names(report)[12]
```

```
[1] "comparisons.rda"
```

Alternatively, we can simply use the `zipArchive()` function in the **Rcompression** package to add the serialized data to the Excel archive with

```
library(Rcompression)
```

```
z = zipArchive("myReport.xlsx")
```

```
z[["comparisons.rda"]] = serialize(comparisons, NULL)
```

Finally, to restore the data frame, we extract it from the archive as a raw vector and use `unserialize()` as shown here:

```
rw = report[["comparisons.rda", mode = "raw"]]
```

```
head(unserialize(rw))
```

	Institution	Pr	Time	Res11	Res12	NonRes11	NonRes12
1	U Penn	Y	2.000	59503	62478	59503	62478
2	USC	Y	1.500	43871	46065	43871	46065
3	Cornell	Y	1.500	43304	45469	43304	45469
4	CMU	Y	1.500	37185	39044	37185	39044
5	Stanford	Y	1.125	43953	46151	43953	46151
6	Columbia	Y	1.000	44180	46389	44180	46389

15.7 Using Relationship and Association Information in the Archive

When we add a worksheet to a workbook or an image or chart to a worksheet, the relationships between files in the archive change. This can be complicated and error-prone so if we want to build functions to make these additions, we need an understanding of the relationships between files in the archive. For example, the `workbook.xml` file keeps track of the worksheets, global settings, and other shared components of the workbook. However, neither the worksheets nor their filenames are embedded in this file. There is an extra layer of indirection used to connect `sheet1.xml` to `workbook1.xml`. A **rels** file in the archive is used to tie the pieces of the workbook together, amongst other relationships. Specifically, `xl/_rels/workbook.xml.rels` contains information to connect the worksheets to the workbook.

We provide two examples in this section demonstrating how to add a worksheet to a workbook and an image to a worksheet. These use different **rels** files in the archive. We use these examples to outline

our approach of leveraging *XML* tools to parse and build *XML* files for applications. The functions `addWorksheet()` and `addImage()` formalize these examples into more general functionality. We note that these functions are not polished; they merely offer a starting point for this functionality. The first example explains the basic approach needed for working directly with *XML* files in the Office Open *XML* archive, in contrast to other approaches that interface with a toolkit from another language, e.g., *Java*, for providing access from within *R* to the contents of a *xlsx* file.

We saw in Section 15.3, that `workbook1.xml` has a `<sheets>` element with two `<sheet>` children, one for each worksheet in the document. Each of these `<sheet>` elements has a `name` attribute that provides the label to display on the tab in the Excel user interface, "Sheet1" and "Images", respectively, and each has a `sheetId` attribute, which is used for ordering the sheets in the display. The `<sheet>` node does not contain the file name that holds the sheet contents, e.g., `sheet1.xml`; instead, the relationship identifier, `r:id` on `<sheet>`, locates this file.

For example, for the sheet labeled "Images", the `r:id` is "rId2". We look in the file called `xl/_rels/workbook.xml.rels` (shown below) for the `<Relationship>` element that has an `Id` attribute value of "rId2". The value of this element's `Target` attribute provides the file name for the sheet. In this case, it is `xl/worksheets/sheet2.xml` (file names are relative to `xl/`).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://.../relationships">
  <Relationship Type="http://...calcChain"
    Id="rId6" Target="calcChain.xml"/>
  <Relationship Type="http://...styles"
    Id="rId4" Target="styles.xml"/>
  <Relationship Type="http://...worksheet"
    Id="rId1" Target="worksheets/sheet1.xml"/>
  <Relationship Type="http://...worksheet"
    Id="rId2" Target="worksheets/sheet2.xml"/>
  <Relationship Type="http://...theme"
    Id="rId3" Target="theme/theme1.xml"/>
  <Relationship Type="http://...sharedStrings"
    Id="rId5" Target="sharedStrings.xml"/>
</Relationships>
```

Figure 15.9 provides a diagram of the layout of the part of the archive that contains the information about the "Images" sheet. In addition to the `rels` and `workbook1.xml` files, the file called `[Content_Types].xml` provides information on the kind of content in each of the files in the archive, and `app.xml` within `docProps/` has information pertaining to the layout of the GUI, e.g., it indicates the spreadsheet has two pages.

Example 15-8 Adding a Worksheet to a Workbook

In this example we demonstrate how to add a worksheet to a spreadsheet, including updating the subsidiary *XML* and `rels` files in the archive. Our basic approach is to use a template worksheet, i.e., a blank `sheet.xml` that has the structure defined by Office Open *XML* for sheets. In addition to adding this worksheet to the archive, we must also update the auxiliary files as shown in Figure 15.9. That is, we follow the steps below.

- Add a `<Relationship>` element to `workbook.xml.rels`. This element has a unique identifier in its `Id` attribute and file name of the worksheet in its `Target` attribute.
- Add a `<sheet>` element to `workbook.xml`, which has an `r:id` that matches the identifier in `Id` on the newly created `<Relationship>` node in the `rels` file.

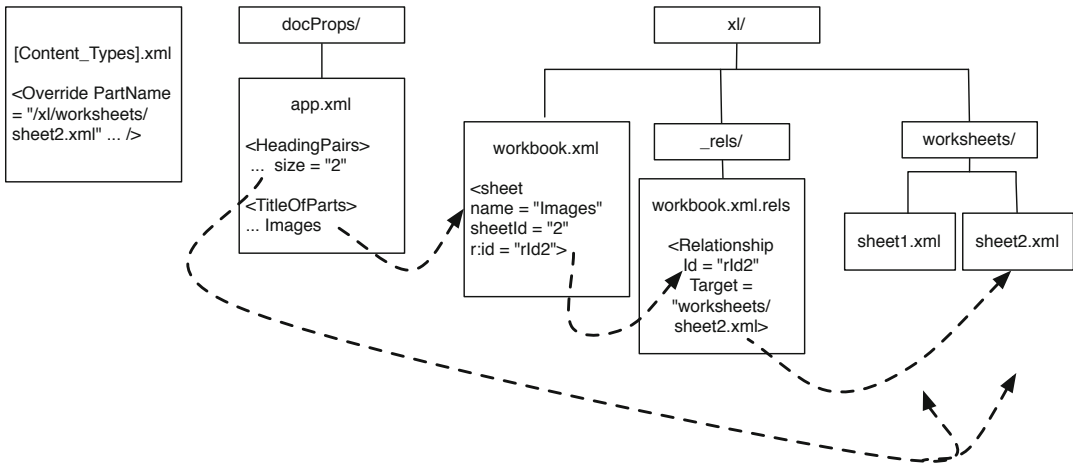


Figure 15.9: Diagram of the Interconnections between Files in an Excel Archive. This diagram shows the connections between various XML and **rels** files in the Excel Archive. For example, rather than `workbook1.xml` containing the names of the worksheet files, e.g., `worksheets/sheet2.xml`, it contains a reference to a `<Relationship>` node in a **rels** file that holds the worksheet file name.

- Update `app.xml` to indicate the new *size* of the workbook and related properties.
- Update `[Content_Types].xml` with information about the content of the new worksheet file.
- Insert a generic worksheet into the archive with the file name given in the *Target* attribute of the new `<Relationship>` element.

We carry out each of these steps here. After accessing the archive, we determine how many sheets are already in the archive. The following code does this and then creates the new sheet's file name based on this information:

```
ed = excelDoc("testAddWS.xlsx")
num = length(grep("xl/worksheets/sheet.*\\.xml", names(ed))) + 1
filename = sprintf("xl/worksheets/sheet%d.xml", num)
```

We must enter this file name into the **rels** file, but first we need to determine a unique identifier to associate with the file name. We extract all `<Relationship>` nodes from the **rels** file and construct a unique identifier based on the number of relationships already present. Here we use tools in XML, e.g., `xpathSApply()` to parse and identify node sets:

```
rels = ed[["xl/_rels/workbook.xml.rels"]]

schemas = "http://schemas.../relationships"
namespace = c(x = schemas)
eids = xpathSApply(rels, "//x:Relationship", xmlGetAttr, "Id",
                  namespaces = namespace)
relId = sprintf("rId%d", length(eids) + 1L)
relId
```

```
[1] "rId7"
```

Now that we have constructed the unique id and file name for the new worksheet, we can add a `<Relationship>` node to the `rels` document with this information. We use `newXMLNode()` in `XML` to do this as follows:

```
newXMLNode (
  "Relationship",
  attrs = c(Id = relId,
            Type = "http://schemas.../worksheet",
            Target = sprintf("worksheets/%s", basename(filename))),
  parent =  xmlRoot(rels))
```

```
<Relationship Id="rId7"
  Type="http://schemas.../worksheet"
  Target="worksheets/sheet3.xml"/>
```

Next, we update the `workbook.xml` file to include a `<sheet>` node that points to the `<Relationship>` element we just added to the `rels` file. For convenience, we again use `num`, the number of sheets, in the sheet title. We do this with

```
name = paste("New Sheet", num)
main = "http://schemas.../spreadsheetml/2006/main"
wbook = ed[["xl/workbook.xml"]]
sheets = getNodeSet(wbook, "//x:sheets", c(x = main))
newXMLNode("sheet",
           attrs = c(name = name, sheetId = num, "r:id" = relId),
           parent = sheets[[1]])
```

```
<sheet name="New Sheet 3" sheetId="3" r:id="rId7"/>
```

The last files that need updating are `app.xml` and `[Content_Types].xml`. We do not provide the code here but instead refer the reader to the `addWorksheet()` function in `RExcelXML` for these final details.

This approach has been generalized and encapsulated into the function `addWorksheet()`. It takes the Excel archive and sheet title as arguments. The following call performs the task of adding a new sheet to a workbook for us:

```
ed = excelDoc("testAddWS.xlsx")
addWorksheet(ed, name="New Sheet")
```

Functions similar to `addWorksheet()` have been developed to add image files and Excel charts to worksheets. As with the relationship between worksheets and workbooks, the `XML` describing the image or chart is not directly contained or referred to in the worksheet. Relationship files are used to connect the image to the sheet that displays it. Figure 15.10 provides a visual representation of the relationships for the image displayed in "Sheet1" of the worksheet shown in Figure 15.4.

These relationships are slightly more complex than those for sheets because they involve two files—the PNG file containing the actual image and an `XML` file called `xl/drawings/drawing1.xml`, which holds information about the placement, size, and shape of the images in the sheet. Following is a snippet of this file:

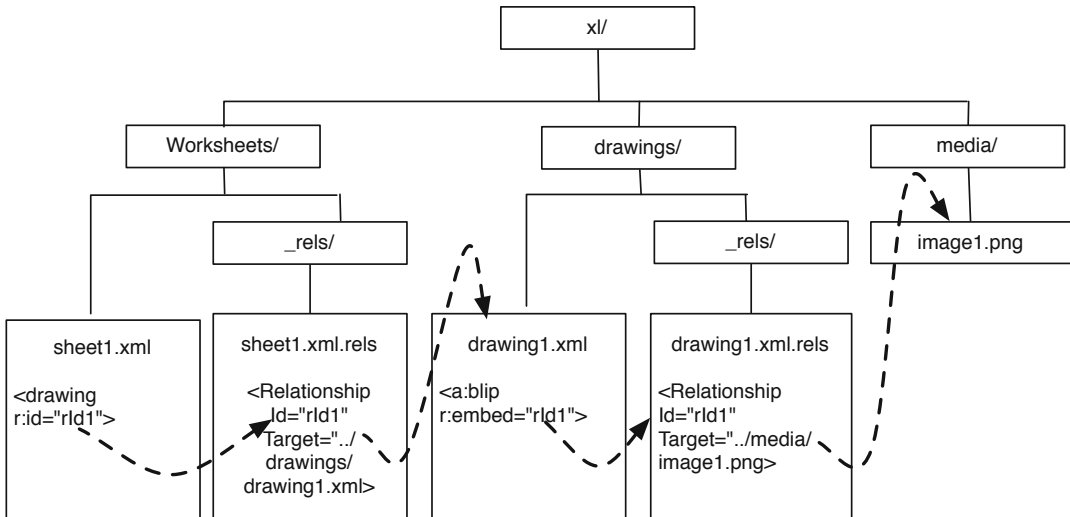


Figure 15.10: Diagram of File Relationships for an Image in a Worksheet. The hierarchy displayed here shows the connections between the files pertaining to the image in the worksheet shown in Figure 15.4. For example, rather than `sheet1.xml` containing the name of the drawing file (`drawings/drawing1.xml`), it contains a reference to a `<Relationship>` that holds the target file name. Similarly, the name of the image file is determined via the relationship in `drawings/_rels/drawing1.xml.rels` that connects `drawing1.xml` to `image1.png`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xdr:wsDr xmlns:xdr="http://...spreadsheetDrawing"
  xmlns:a="http://...drawingml/2006/main">
  <xdr:twoCellAnchor editAs="oneCell">
    <xdr:from>
      <xdr:col>2</xdr:col> <xdr:colOff>114300</xdr:colOff>
      <xdr:row>4</xdr:row> <xdr:rowOff>139700</xdr:rowOff>
    </xdr:from>
    <xdr:to>
      <xdr:col>3</xdr:col> <xdr:colOff>177800</xdr:colOff>
      <xdr:row>13</xdr:row> <xdr:rowOff>12700</xdr:rowOff>
    </xdr:to>
    <xdr:pic>
      ...
      <xdr:blipFill>
        <a:blip xmlns:r="http://...relationships" r:embed="rId1">
          ...
        </a:blip>
      </xdr:blipFill>
    </xdr:pic>
  </xdr:twoCellAnchor>
</xdr:wsDr>
```

The `<twoCellAnchor>` element contains `<from>`, `<to>`, and `<pic>` elements. The `<from>` and `<to>` elements provide the location of the image. These have `<col>`, `<colOff>`, `<row>`, and `<rowOff>` children, where `<col>` and `<row>` are the (zero-based) indices of the column and row. The `<colOff>` and `<rowOff>` elements are in English metric units (EMU) and they are offsets measured from the edge of the specified row or column. (An EMU is 1/360,000 of a centimeter.) The

<pic> element holds the information about the image file. The <blip> node contains a relationship *ID* that leads to the the file name. Again, rather than explicitly containing the file name, this element refers to an identifier, which we look up in the **rels** file associated with this document, i.e., `xl/drawings/_rels/drawing1.xml.rels`, to obtain the name of the image file: `xl/media/image1.png`.

The `addImage()` function in **RExcelXML** handles the creation and editing of these various *XML* files much the same way as we did when we added a sheet to a workbook in Example 15-8 (page 526). We next provide an example of how to use this intermediate-level function.

Example 15-9 Adding an Image to a Worksheet

We continue with the report created in Example 15-5 (page 522) and add an *R* plot of the data to the worksheet. While the format of the table in the report was prescribed by the template, we wanted to add an alternative visual comparison of the tuition values. In this example, we create a dot plot in *R* as a PNG file and add that image to the worksheet (see Figure 15.11).

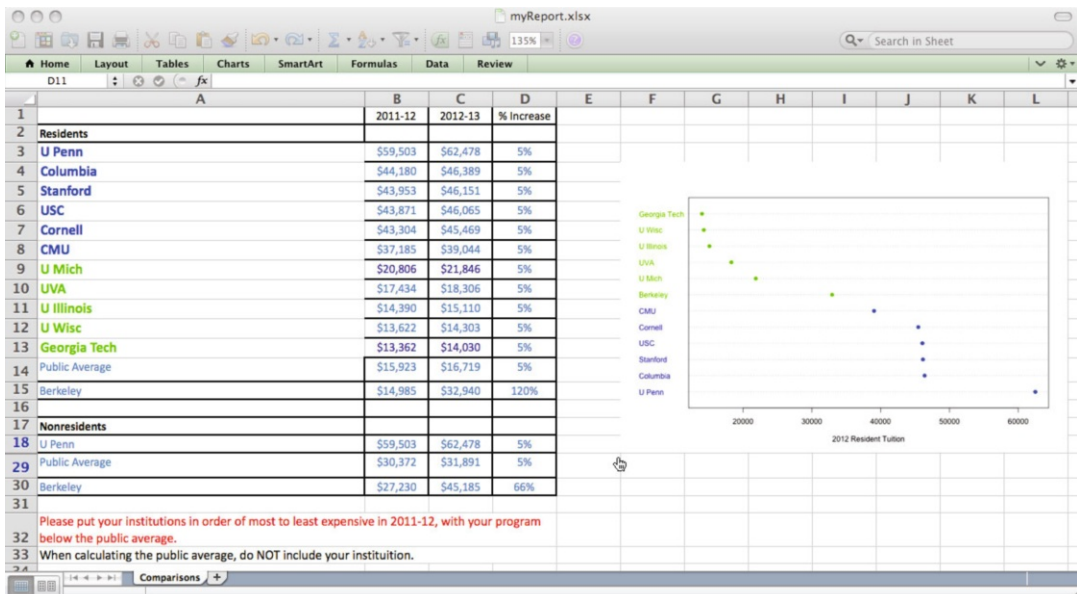


Figure 15.11: Completed Excel Report Template with Dot Plot. This screenshot shows a completed report. Starting from the blank template shown in Figure 15.8, we filled in cells with values from a data frame, added styles to some of the cells, inserted formulae into B14 and C14 to compute averages, and included a dot plot made in *R*. Also, “hidden” in the **xlsx** archive is the **rda** file used to complete the report.

We use the data from `comparisons` to create the bar chart. The colors of the dots match the text color used in the table to distinguish private and public institutions.

```
cI = comparisons[order(comparisons$Res12, decreasing=TRUE), ]
png("TuitionDotChart.png", width=720)
dotchart(cI$Res12, pch=19, col=c("green","blue")[cI$Pr],
         labels = cI$Institution, xlab = "2012 Resident Tuition")
dev.off()
```

The dot plot has been saved as a PNG file, which we now add to the worksheet.

We read the Excel archive and access the spreadsheet with the usual commands:

```
reportWB = workbook("myReport.xlsx")
sh = getSheet(reportWB, 1)[[1]]
```

The `addImage()` function handles all of the details required to add an image to a worksheet. It updates the appropriate `rels` files to make the connection between the worksheet and the image; creates a `drawing.xml` file containing the image-specific information such as where to place the image in the sheet; and inserts the PNG file in the archive. The function takes as input, the worksheet in which to place the image, the name of the image file, and the top left and lower right corners where the image will be placed.

```
addImage(sh, "TuitionDotChart.png", from = c(4, 6), to = c(25, 12))
```

The function also has an `update` argument with a default value of `TRUE` so the archive has been updated with this new information.

15.8 Google Docs and Open Office Spreadsheets

We have primarily focused in this chapter on Excel spreadsheets; however, the same functionality can be developed for other spreadsheet software that follow the Office Open XML standard. The R packages `RGoogleDocs` and `ROpenOffice` are starting points for handling Google Docs and Open Office files, respectively. These packages are not as complete as `RExcelXML`, and the authors invite contributions that will flesh out the functionality. The following example, highlights the similarities and differences in functionality between the `addWorksheet()` functions in `RExcelXML` and `RGoogleDocs`.

Example 15-10 Inserting a Worksheet into a Google Docs Spreadsheet

Google Docs are dynamic documents that are accessible via the Internet. They can be shared with others and edited simultaneously. They need to be accessed with an Internet connection, login, and password. This access can be handled programmatically within R. The following two-step process is described in more detail in Example 10-5. Briefly, we first authenticate by shipping our user id and password with the following call to `getGoogleAuth()`:

```
auth = getGoogleAuth("deb.nolan@gmail.com", "my password", "wise")
```

Note that to access a spreadsheet, we must request “wise” service from Google. Once authenticated, we get the connection with the following call to the `getGoogleDocsConnection()` function:

```
con = getGoogleDocsConnection(auth)
```

Now that we have established a connection, we use the `getDocs()` function to retrieve a list of available documents from the Google Docs account. Given the “wise” constraint, the call below only retrieves the spreadsheets from the account:

```
ssdocs = getDocs(con)
```

Although we needed application-specific code to first access the GoogleDocs spreadsheet, after we obtain that access, the functions to operate on the spreadsheet are essentially identical to those

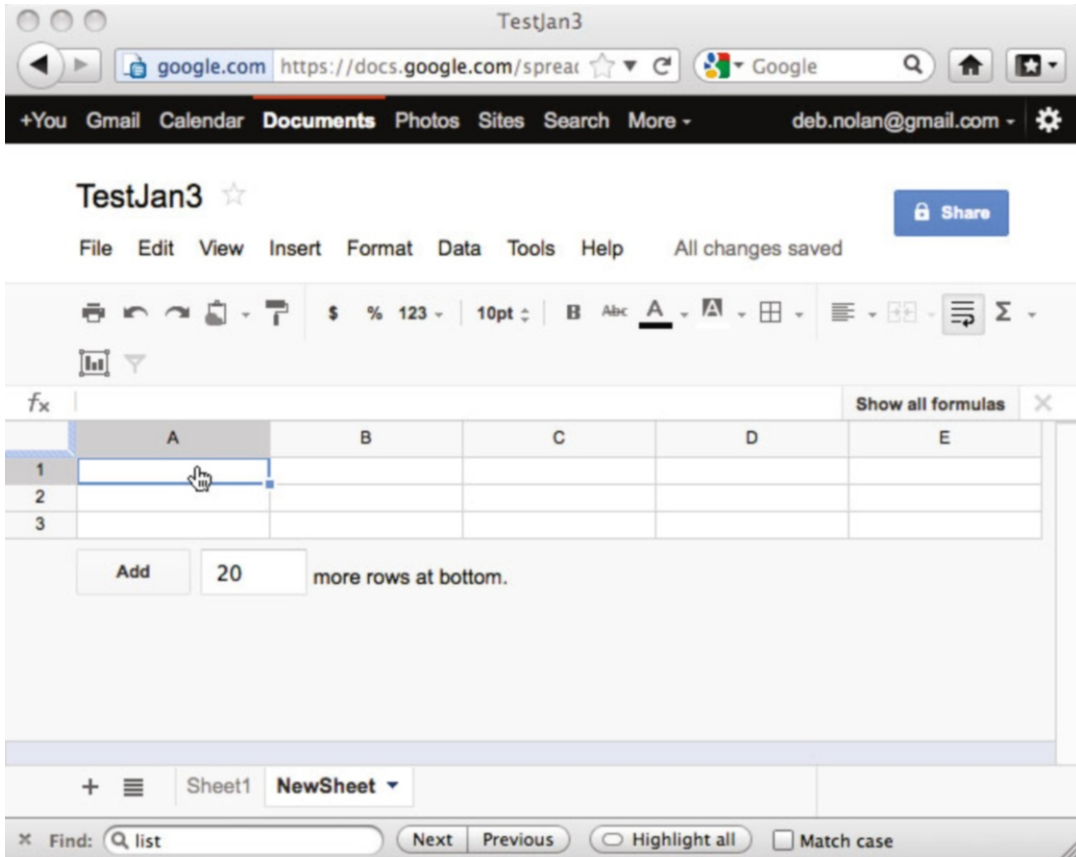


Figure 15.12: Example Google Docs Spreadsheet. This screenshot shows a Google Docs spreadsheet called TestJan3. It has two sheets, "Sheet1" and "NewSheet". The latter was added to the spreadsheet via a call to `addWorksheet()` in the `RGoogleDocs` package.

in `RExcelXML`. For example, we can add a worksheet to a spreadsheet by calling the function `addWorksheet()`, which has a signature very similar to `addWorksheet()` in `RExcelXML`:

```
addWorksheet(ssdocs[[1]], con, title = "NewSheet")
```

One key difference is that we must pass the connection with each of our function calls. The new worksheet is shown in Figure 15.12.

15.9 Possible Enhancements and Extensions

Adding Charts to Worksheets

The XML documents for charts can be reused and shared among spreadsheet, presentation, and word processing applications. Currently, `RExcelXML` has basic functionality to add a bar chart to a spread-

sheet. Similar to `addWorksheet()`, the `addChart()` function uses a template XML document as a starting point. The function adds this template to the Excel archive and handles updating the relationships between files. The `addChart()` function is a proof of concept, and can be extended to include other chart types.

Other OOXML-Formatted Office Suites

As mentioned in Section 15.8, the R packages for handling Google Docs (RGoogleDocs) and Open Office documents (ROpenOffice) are in the preliminary stages of development. The approach described in this chapter with RExcelXML can be extended in a straightforward manner to these other implementations of the Office Open XML standard. The authors invite contributions that will flesh out the functionality of these packages.

Word Processing and Presentation Documents

Again, while the focus in this chapter has been on creating and manipulating Excel documents, Office Open XML includes mark-up for word processing documents (*WordprocessingML*) and presentations (*PresentationML*) in addition to *SpreadsheetML*. The same general approach to using the XML package with XML files carries over to creating and modifying Microsoft Word and presentation files, e.g., RWordXML offers a starting point for creating this interface.

15.10 Summary of Functions in RExcelXML

Understanding the *SpreadsheetML* vocabulary and structure helps us develop functionality that provides easy access to cell values, formula, and styles. These functions offer a convenient means for working with various parts of the `xlsx` archive. This approach enables the programmer to extend and customize the interface to the spreadsheet, workbook, and Excel document. Many of these functions are demonstrated via examples in Sections 15.4 and 15.5. We provide below brief summaries of the functions available in RExcelXML.

`read.xlsx()` Retrieve the contents of the worksheets in an `xlsx` document. The contents are represented as a list of data frames, one for each worksheet. This function extracts the contents for simple worksheets (*which* specifies which sheets in the workbook to extract) and can skip rows (*skip*), convert a header to variable names (*header*), and automatically identify the class of the content of each column.

`excelDoc()` Retrieve the ZIP archive of an `xlsx` document. This function provides access to all of the files within the archive. It can also create (when *create* is TRUE) a new Excel document object from a template (*template*), which can be supplied in the function call.

`workbook()` Create a `Workbook` object from an `excelDoc` or a file name. The `Workbook` provides access to its worksheets through its `[[]` and `[` methods. The worksheets are `Worksheet` objects. See `[[]` and `[` function descriptions below for how to access and set cells in the worksheet.

`getSheet()` Retrieve worksheets from a `Workbook` object or an `ExcelArchive` object. The return value is a list of `Worksheet` objects. The contents of the worksheet can be accessed with `[` and `cells()`.

`cells()` Return a list of all XML cell nodes from a `Worksheet` object. An *XPath* expression can be provided (in the *xquery* parameter) to restrict the set of nodes returned.

`[[]` and `[` Subset worksheets in a workbook and cells in a worksheet. The cells can be subsetted via Excel-style or R-style indexing or a mixture of both, e.g., `sheet["A3:B4"]`, `sheet[3:4, 1:2]` and `sheet[3:4, c("A", "B")]` are all equivalent. The *update* argument controls whether the `zip` archive or the XML file only will be modified, e.g.,

```
sheet["A3:B4", update = FALSE] = 1:4
```

does not update the archive.

update() Update the contents to an **xlsx** file (the **zip** archive) by adding or overwriting existing entries. For efficiency, many of the functions that create and change the contents of a workbook have an *update* argument to control when the archive is updated. This includes, `[[]` and `[[]`.

getSharedStrings() Retrieve as a character vector the shared strings of a workbook or worksheet.

getStyles() Retrieve the shared styles for a workbook or worksheet. The return value will be an *R* representation of the styles.

getDocStyles() Retrieve the *XML* `<styleSheet>` for the `ExcelArchive`. The return value is an `XMLInternalDocument` rather than an *R* representation of the styles.

createStyle() Define a new style for a cell. The foreground color (*fg*), fill color (*fill*), font (*font*), alignment (*halign* and *valign*), and border (*border*), among other styles can be specified.

setCellStyle() Set the style for a cell using an existing style.

excelFormula() Construct from a string an Excel formula that can be added to a cell.

addWorksheet() Add an empty worksheet to an Excel archive with the title provided in *name*.

addImage() Add an image file, e.g., PNG, *PDF*, to a region of a worksheet. Specify the extent of the image with *from* and *to*.

addChart() Construct an Excel chart and add it to a region of a worksheet.

15.11 Further Reading

The official documentation for OOXML is voluminous, but Part 3 of the documentation provides a primer with examples and diagrams [4]. Also, [11] provides a brief overview to the documentation and a discussion of the goals and properties of the standard. Reference [26] offers a brief introduction to the basics of OOXML.

References

- [1] Apache Software Foundation. OpenOffice: The free and open productivity suite; 3.0 New Features. http://www.openoffice.org/dev_docs/features/3.0/, 2011.
- [2] Apple, Inc. Numbers for iOS: Supported file formats. <http://support.apple.com/kb/HT4642>, 2011.
- [3] Adrian Dragulescu. `xlsx`: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files. <http://cran.r-project.org/package=xlsx>, 2011. *R* package version 0.5.0.
- [4] ECMA International. Ecma Office Open XML file formats standard, Part 3: Primer. http://www.ecma-international.org/news/TC45_current_work/TC45_available_docs.htm, 2011.
- [5] Federal Election Commission. Top 50 house incumbents by contributions from PACs and other committees, January 1, 2011 – June 30, 2011. <http://www.fec.gov/press/summaries/2012/PAC/6mth/1pac6mosummary11.xlsx>, 2011.
- [6] KDE e.V. KOffice: Standards-compliant office and productivity applications. <http://userbase.kde.org/KOffice>, 2011.
- [7] LibreOffice; The Document Foundation. Calc: The LibreOffice spreadsheet program. <http://www.libreoffice.org/features/calc/>, 2011.

- [8] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2000 and Excel XP. *Computational Statistics & Data Analysis*, 40:713–721, 2002.
- [9] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52:4570–4578, 2008.
- [10] Eric Neuwirth. **RExcel**: Interface between *R* and Excel. <http://cran.r-project.org/package=RExcel>, 2011. *R* package version 3.2.6.
- [11] Tom Ngo. Office Open XML overview. http://www.ecma-international.org/news/TC45_current_work/OpenXMLWhitePaper.pdf, 2005.
- [12] *R* Core Team. *R Data Import/Export*, 2012. <http://cran.r-project.org/doc/manuals/R-data.html>.
- [13] Frank Rice. Introducing the Office (2007) Open XML file formats. [http://msdn.microsoft.com/en-us/library/aa338205\(v=office.12\).aspx](http://msdn.microsoft.com/en-us/library/aa338205(v=office.12).aspx), 2006.
- [14] Brian Ripley. **RODBC**: ODBC database access. <http://cran.r-project.org/package=RODBC>, 2011. *R* package version 1.3-3.
- [15] Marc Schwartz. **WriteXLS**: Cross-platform *PERL*-based *R* function to create Excel 2003 (XLS) files. <http://cran.r-project.org/package=WriteXLS>, 2011. *R* package version 2.3.0.
- [16] Miria Solutions. **XLConnect**: Manipulate Excel files from *R*. <http://cran.r-project.org/package=XLConnect>, 2011. *R* package version 0.2-3.
- [17] Hans-Peter Suter. **xlsReadWrite**: Natively read and write Excel files. <http://cran.r-project.org/package=xlsReadWrite>, 2011. *R* package version 1.5-4.
- [18] Duncan Temple Lang. **RExcelXML**: Tools for working with Excel XML documents. <http://www.omegahat.org/RExcelXML>, 2011. *R* package version 0.5-0.
- [19] Duncan Temple Lang. **ROOXML**: Simple tools for Open Office XML documents. <http://www.omegahat.org/ROOXML>, 2011.
- [20] Duncan Temple Lang. **ROpenOffice**: Basic reading of Open Office spreadsheets and workbooks. <http://www.omegahat.org/ROpenOffice>, 2011. *R* package version 0.4-1.
- [21] Duncan Temple Lang. **XML**: Tools for parsing and generating XML within *R* and *S-PLUS*. <http://www.omegahat.org/RXML>, 2011. *R* package version 3.4.
- [22] Duncan Temple Lang. **Rcompression**: In-memory decompression for GNU **zip** and bzip2 formats. <http://www.omegahat.org/Rcompression>, 2012. *R* package version 0.94-0.
- [23] Duncan Temple Lang. **RGoogleDocs**: Primitive interface to Google Documents from *R*. <http://www.omegahat.org/RGoogleDocs>, 2012. *R* package version 0.7-0.
- [24] Duncan Temple Lang and Gabriel Becker. **RWordXML**: Tools for Open Office word processing XML documents. <http://www.omegahat.org/RWordXML>, 2010. *R* package version 0.1-0.
- [25] Guido van Steen. **dataframes2xls**: Write data frames to xls files. <http://cran.r-project.org/package=dataframes2xls>, 2011. *R* package version 0.4.5.
- [26] Wouter van Vugt. Open XML: The markup explained. <http://openxmldeveloper.org/blog/b/openxmldeveloper/archive/2007/08/13/1970.aspx>, 2007.
- [27] Gregory Warnes. **gdata**: Various *R* programming tools for data manipulation. <http://cran.r-project.org/package=gdata>, 2011. *R* package version 2.12.0.
- [28] World Bank Group. WDR2011 dataset. <http://databank.worldbank.org/databank/download/WDR2011Dataset.xlsx>, 2011.
- [29] World Bank Group. World development report 2011 on conflict, security and development. <http://data.worldbank.org/data-catalog/wdr2011>, 2011.