Chapter 1

Introduction to XML

In this chapter we explore a variety of different ways in which we as scientists can deploy XML, the eXtensible Markup Language. We start by considering its use as a way to store structured information and exchange it between different applications, either on the same machine or over the network. We introduce facilities in the S language for reading and writing XML quite easily and allowing people to create their own languages or dialects for storing and exchanging information. We also consider serialization of objects within different systems, e.g. S, SAS, Omegahat, Matlab in a common format which allows them to be easily shared by others. And finally, we look at how we can use the XML facilities in R to implement SOAP (Simple Object Access Protocol) clients and servers.

Statisticians often need to use different software tools in their data analysis, and by far and away the most common solution to sharing data between applications has been to exchange data via ASCII or text files. One application will write out its view of some data; the other application will then read this file and construct its internal version of this dataset and go about its business. If we want to make the results of computations in this application available to the original application or any other, we write the results to another ASCII file and they are there to be picked up by any other application. We don't reserve this style of communication for dynamic interaction between two applications running simultaneously. We exchange datasets in the same way, using agreed upon formats such as white-space delimited or comma-separated data files. Since most of the datasets over the years are rectangular tables or arrays, the ASCII format is relatively natural. Each line corresponds to a record, and each element within a line corresponds to a variable and is the value for that variable.

The benefits of using tabular ASCII files are easy to see.

- We can edit them in simple text editors.
- There is a natural connection between the visual layout in the file and the way we think about the dataset when computing on it.
- It is quite simple to both write and read data in this format.

But what happens as data become more complex? How do we represent *missing values* in our data? A common approach is to choose a number that is not in any of our datasets and to have our software treat such a value in a special way. And so we see datasets with numbers like '999'. Of course, the software cannot be used to read datasets that actually have '999' as an actual value since it will mark these as missing.

Another problem arises if we have *repeated measurements* within records and there are a different number of measurements for each record. The data are no longer rectangular and we will have to modify our software to be able to read and write these files. Not only that, how will we tell the software that there are 6 consecutive values for this record that are to be collected together into a single array, but for the next record there are only 3? In other words, how do we separate the different values within a record and associate them with a given variable. In the simple table version, each variable in each record had only a single value. If we have an arbitrary number of values per variable, a simple way to indicate this in the file is to put in another variable that tells the software how many values to expect. Suddenly, the simplicity of the tabular ASCII file is disappearing and the software in all the different applications has to be changed.

Repeated measurements are a simple example of a general problem. Even within the context of a rectangular array, what if some of the *entries are objects* rather than simple values? The least we should be able to do with good data analysis software is to write out a dataset and read it back in without losing information. This is often called *round-tripping*. If we want to be able to read these objects, just as for the repeated measurements, we have to be able to identify that the next k values go to make up the object. So, again, we are seeing that we have to add meta-information to the dataset so that our software can be certain to read it back unambiguously.

In the past, people have distributed a dataset as a collection of files. One file contains the actual values and the others contain information such as the variable names, descriptions of the variables, information about the origins of the data and how they were collected, and so on. These additional files are

sometimes called the *codebook*. XGobi, an interactive graphical visualization system, used a format that consisted of several files to represent the data (records and variables), variable names, the color of the records, the glyph for the record, and so on. One of the major problems with these multiple file approaches is that it is very easy to edit one file and forget to update the others. For example, if we remove a record in the dataset and not in the color specification, the two files no longer match. At best, we would get an error message about this. Alternatively, we just get the wrong values for the different records. Essentially, having the data in different files reduces the advantage of being able to edit it in a natural fashion. The visual similarity of the data and the conceptual model is no longer there.

Not only does the multiple-file approach make editing harder, but it also makes it harder to distribute. Nowadays we use HTTP or FTP from within applications to download files. There syntax for specifying a collection of files is less natural than giving a single URL. And this becomes an issue as we combine datasets together.

We have focussed on rectangular data arrays since these are probably the most common form of data that statisticians experience. Many of the same problems arise with other formats, and indeed can be worse. A popular format for name-value pairs is the properties file format given either as

```
name: value
```

or

```
name=value
```

With this format, one has to be careful about how white space at the beginning and end of the value is handled. And long values that extend onto second and third lines can be tricky. Obviously this format is limited to specific types of data. Like many other formats, it is hard to extend it to include different types of information. Again, if we want to do something as easy as combine multiple sets of name-value pairs, there is no easy way to separate the sets within the file.

There is a theme inherent in this list of problems associated with representing ASCII data in simple files. The basic impediment is that there is no way to add meta-information to the data. Ideally, we would like to be able to include information such as what the missing value identifier is; identify objects as a collection of (potentially named) values; include multiple separate but related datasets in a single file. We would like the software to be able to read the meta-information if it is there, and yet not require it

to be there and ignore it if the software doesn't understand it. The ability to add meta-information to representations of data would make us more expressive. We could differentiate between a real number that has an integer value and an integer, i.e. the different between 1 and 1.0 that often gets lost when software reads ASCII data. We could refer to one piece of the data in another part of the data, i.e. cross-reference the elements, to indicate that if one piece is updated, the other pieces should reflect this. Without meta-information or markup for ASCII data, we are quite limited in what we can express.

The general markup language, XML – the eXtensible Markup Language, allows us to mark up ASCII data with meta-information. Just being able to add meta-information via markup allows us to solve several of the problems mentioned above. But we have two other issues to face when looking for a markup mechanism. Firstly, if we are the only people using it, aren't we going to restrict our ability to exchange data with other communities? Since this exchange is becoming increasingly important, that would be a very bad thing. But again, we are fortunate because XML is quickly emerging as an extremely popular and widespread general data format. It is used to represent genetic information, geographical maps, output from databases, protocol for remote procedure calls (RPC), authoring technical articles and books, and so on. Word processors, spreadsheets, and relational databases now provide options to save their contents as XML.

XML's popularity answers the second question that we should ask when considering using XML to represent data: What is the cost of switching to XML? A new format requires us to rewrite software. This involves retesting our software, etc. which is a very time consuming task. So if we don't get any tools to help us write this new software, the cost of switching to XML may well be excessive. And again, the good news is that since all these other communities are actively using XML, they are also providing extensive collections of tools for working with XML. And we can incorporate those into our environments and get the benefits of XML relatively transparently.

So what are the drawbacks to XML? It does solve all of the problems we have identified in the previous paragraphs. But, like all pieces of software, it is not a silver bullet that solves all problems and introduces no new ones. Rather it gives us more options (since we can always use non-marked up data, even as parts within XML documents), and is a step in our evolution. It raises higher-level problems which we will then have to try to solve while people use XML. One of the strengths of XML is that it is structured to make it easy for a computer to read. However, it leads to very verbose content and it is quite hard for us humans to read. And that means it is hard to

edit directly. And so we lose the simplicity afforded by raw ASCII data, which means we need tools that allow us to visualize and manipulate XML content to create inputs. Or in other words, we need software that reads and writes the XML and insulates us from the details. As we will see, this is not a simple problem in general because XML is itself extremely general. However, each community will gradually develop tools for viewing its own types of data and hopefully these will be shared when possible. Even now, some general tools for editing XML are emerging and can be customized to our needs.

Now that we have motivated the value of XML, we will go into a little more detail. We will first give a more precise description of XML and its parts. Next, we will discuss the *XML* package for R and S-Plus which allow us to both read and write XML directly from within the S language. We will illustrate this package by looking at some real examples of XML for reading and writing data from other applications. Finally we will also discuss some advanced uses of XML for communicating with other applications via SOAP (Simple Object Access Protocol).

1.1 What is XML?

The eXtensible Markup Language provides a standard for the semantic management of data. It is a formal meta-language facility for defining a markup language. The basic unit in an XML file is an entity or chunk that contains content and markup. The content is the actual information such as 6. The markup describes the content, in this case the markup is the name **CYL** for the cylinders variable in the dataset (see the example in Section 0.2.1). More generally, markup consists of tags, attributes, comments, and processing instructions for the content. The tag marks the beginning and end of a piece of content. That is, content must be surrounded by a start tag and a corresponding end tag.

A tag has a name and possibly other pieces of information describing the element's content. In a start tag, the name and any additional information are surrounded by the < and > characters. Similarly, an end tag consists of the tag name (it must match the tag name in the start tag) surrounded by < / and >. For example, the following XML entity

```
<CYL> 6 </CYL>
```

is a **CYL** element with content 6. It is possible to have empty tags, i.e. tags with no content,

```
<CYL></CYL>
```

In this case the start and end tags can be combined into one tag as follows, `<CYL/>`.

Attributes provide additional information about the content. For example, the **dim** tag below has a *size* attribute which has a value of 2. (See Section 0.2.2 for the related example).

```
<dim size="2">
```

Attributes are specified via name-value pairs. The syntax rules are provided below.

1.1.1 XML syntax

For XML to be *well-formed* it must obey the following syntax rules.

- XML is case sensitive so start and end tag names must match exactly. For example, the following start and end tags have a mismatch in case and so are not well-formed,

  ```
  <CARS>
  </Cars>
  ```

  The end-tag name needs to have all capital letters, i.e. `</CARS>`, in order for it to match the start tag.

- No spaces are allowed between the $<$ and the tag name.

- Tag names must begin with an alpha character, and contain only alphanumeric characters.

- An element must have an open and closing tag unless it is empty.

- An empty element that does not have a closing tag must be of the form $< \ldots / >$. For example, `<nan/>`.

- Tags must nest properly. That is, when one element contains another element then the start and end tags of the inner element must be between the start and end tags of the parent element. For example,

  ```
  <CARS>
     <CYL> 6 </CYL>
  </CARS>
  ```

Here the **CYL** tag is nested within the **CARS** tag. Note the use of indentation makes it easier to see the nesting.

- All attribute values must appear in quotes in a `name = "value"` format.

  ```
  <dim size="2"/>
  ```

  This example shows an empty tag with a *size* attribute of "2".

- Isolated markup characters are not allowed in text. However, they may be specified via entity references. For example, the < is specified by the entity reference `<` and the > symbol is `>`.

In addition to element tags, XML has markup for comments, which is information not shown to the user; processing instructions, which is similar to code meant for the processor; and character data that is not to be processed but simply passed straight through to the user. Comments must appear between <!−− and −− >. For example,

```
<!--  This is a comment which is so long
      that it apears on three lines of the
      document before it ends with two - followed by >. -->
```

It is possible to include in a document character data that is not processed and so the > is ignored. The character data must appear between <![CDATA[and]]>.

```
<![CDATA[  This is character data that can have any special
    character in it such as < or > or & and not have to worry
    about it being interpreted as a special character by the
    processor. ]]>
```

Finally, processing instructions must appear between <? and ?>. For example, an XML document must start with the processing instruction that identifies it as an xml document and provides the XML version number as an attribute,

```
<?xml version = "1.0" ?>
```

1.1.2 Valid XML

The rules provided in the previous section are just syntax rules for insuring that an XML document is well-formed. But we typically want to have documents that are more than well-formed; we want to include application specific structure in the markup. For example, with geographic data we may want tags for locations, x and y coordinates, city names, etc. Tags for these entities are specified through a set of Document Type Definitions (DTD for short) or schema. With a DTD we can: provide the name of a valid element; limit the content of an element to character data, specific other elements, or to be empty; and specify the attributes that are required or allowed in the tag.

Well-formed XML obeys XML syntax rules described in the previous section, but *valid* XML, in addition to being well-formed, obeys a specified DTD. The DTD may appear within the document itself or be provided via reference,

```
<!DOCTYPE dataset SYSTEM "../DataSetByRecord.dtd">
```

Here we specify a DTD to use via a document type declaration. The **dataset** gives the root element name that the DTD will be applied to in the verification process.

At times we may want to use more than one DTD. We can do this through name spaces. That is, each DTD is given a name and the DTD name is prepended to the appropriate tag names and attribute names. For example, the **object** element below lists three name spaces, `r`, `c`, and `bioc`, and the URL's where their respective DTDs can be found.

```
<object xmlns:r="http://www.r-project.org"
xmlns:c="http://www.c.org"
xmlns:bioc="http://www.bioconductor.org"
type="R-pop-environment" hidden="true">
```

We specify which name space a tag uses within the **object** element as follows:

```
<r:code>
x <- rep(23, 2)
</r:code>
<c:code>
x+
</c:code>
```

A parser has the job of reading the XML, checking it for errors, and passing it on to the intended application. If no DTD or schema is provided, the parser simply checks that the XML is well-formed. If a DTD is provided then the parser also determines whether the XML is valid, i.e. that the tags, attributes, and content meet the specifications found in the DTD, before passing it on to the application. Models for parsing XML are described in greater detail in sections **??**.

1.1.3 XHTML

Some readers will have thought of HTML when we mentioned markup and meta-information. After all, what is the difference between HTML and XML? We can add meta information to HTML documents using the `META` tag, but this is not exactly what we mean by meta-information. A little thought and familiarity with HTML will quickly bring us to problems. HTML has a fixed set of markup elements, e.g. `H1`, `H2`, `a`, `img`, `B`, and so on. It doesn't even have a `NUMBER` or `REAL` markup for representing real numbers.

Hopefully it is evident at this point that an important role of XML is to separate out information (content) from the structure and format. The markup provides the structure of the content, and the the format determines how the content is to be rendered for viewing by the user. As a simple example, an array of numbers that corresponds to the miles per gallon of various makes of cars may be provided via XML as follows:

```
<array name="MPG" size="7" type="numeric">
  <e>21.0</e> <e>21.0</e> <e>22.8</e> <e>21.4</e>
  <e>18.7</e> <e>18.1</e> <e>14.3</e>
</array>
```

The content consists of the set of values 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, and 14.3. The structure provided via the markup tells us that the content forms an array of numbers of length 7, and the array is named MPG.

HTML does not make the same division between content, structure, and format. Many tags describe how to format content, but provide no information about the type of content. For example, the HTML tags `B` for bold face, `br` for line break, and `hr` for horizontal rule are all instructions for the visual rendering of content.

HTML has been extended to XHTML by requiring all tags to be lower case, all elements to be properly closed with end tags, and attribute values to appear between quotes. Although these rules mean that we can require

XHTML documents to be well-formed and valid, XHTML is not up to the job of describing complex structures such as factors, data frames, S objects, and so on. Clearly XHTML is lacking in this regard. With XML we can define much richer application-specific markup.

To drive home the difference between format and content, consider the numbers in the above example: 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, and 14.3. They can be represented as a text list, (21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3), as a summary statistic: 19.6, as a stem-and-leaf plot,

```
14 | 3
16 |
18 | 17
20 | 004
22 | 8
```

or in a graphic such as a histogram. The conversion of the content into one of these formats occurs when the XML is processed for viewing. The eXtensible Stylesheet Language (XSL) contains instructions for formatting the XML. (XSL is itself an XML document that provides a template describing how to view the document.) The XML document together with the XSL stylesheet are processed by an XSLT processor to create the data display. The content remains the same as shown in the XML document, but with a processor the format is changed for the data view. For further references on XSLT processors XML display see ...

1.2 XML examples

Data exchange between different software tools typically poses a problem because different applications use their own, proprietary, undocumented formats for data storage. The use of a well-defined, common exchange format can help solve this problem. We explore here two proposals for XML-based data exchange formats. The first is supported by SAS, and the second by R, Matlab, and Octave.

1.2.1 SAS

The SAS XML library allows you to export an XML document from a SAS dataset and to import an XML document into a SAS dataset. The XML document below gives an example of XML data that can be read into SAS.

```
<?xml version="1.0"?>
```

```
<LIBRARY>
   <CARS>
         <ID> Mazda RX4 </ID>
         <MPG> 21.0 </MPG>
         <CYL> 6 </CYL>
         <HP> 110 </HP>
         <AM> 1 </AM>
   </CARS>
   <CARS>
         <ID> Datsun 710 </ID>
         <MPG> 22.8 </MPG>
         <CYL> 4 </CYL>
         <HP> 93 </HP>
         <AM> 1 </AM>
   </CARS>
   <CARS>
         <ID> Valiant </ID>
         <MPG> 18.1 </MPG>
         <CYL> 6 </CYL>
         <HP> 105 </HP>
         <AM> 0 </AM>
   </CARS>
</LIBRARY>
```

The root element in the XML document is denoted by the **LIBRARY** tag. The tag name for the second-level element represents the SAS dataset name, in this case `CARS`. Each **CARS** element translates into a record in the SAS dataset. We see from the figure that this dataset has 3 records, one for each of the 3 occurrences of the **CARS** elements.

The variables in the dataset correspond to the elements nested within the CARS element. The tag names for these elements translate into the variable names. That is, ID, MPG, HP, and AM, are the names for the variables pertaining to the identification, miles per gallon, horse power, and automatic/manual transmission information for the cars. The content of these elements gives us the value for the variables in that record. The first record has the value "Mazda RX4" for ID, 21.0 for MPG, etc., the second record has the value "Datsun 710" for ID, 22.8 for MPG, etc.

Note that this input format handles only rectangular datasets and that type information, units, and missing values are not specified. The XML data are read in to SAS and shown below

```
libname test xml 'C:\My Documents\test\cars.xml';
proc print data = test.cars;
run;

ID          MPG              CYL            HP              AM

Mazda RX4      21.0 6 110 1
Datsun 710      22.8 4 93 1
Valiant       18.1 6 105 0
```

SAS 9.1.3 provides alternatives via XMLMap to include type information for variables, handle ragged arrays, and to specify a translation of other XML formats into the one provided here (see `fttp://support.sas.com`).

1.2.2 StatDataML

Meyer, Leisch, Hothorn, and Hornik, propose a data exchange format for statistical data, called StatDataML. The R package *StatDataML* provides an implementation of this data exchange format.

An example of XML that obeys the StatDataML requirements for importing data from XML into R, Matlab, Octave appears below. The content is the same as that shown in the example in Section 0.2.1:

ID	MPG	CYL	HP	AM
Mazda RX4	21.0	6	110	1
Datsun 710	22.8	4	93	1
Valiant	18.1	6	105	0

However, the XML document is much more verbose as it includes more structure. For example, the variables are typed. We specify in the XML elements **type**, **categorical**, and **level** that the variable *AM* is categorical with the level 0 indicating automatic and level 1 manual transmission. The StatDataML format is general enough to describe complex S objects such as lists of arbitrary content, which means that it must be more verbose than a data exchange format that assumes a rectangular array. The data in this example have the simple rectangular shape, and so can be stored in a data frame in R. The data frame is a special case of the list which forms a rectangular shape where the columns can be arbitrary types. Here, our columns are a mixture of character, numeric and categorical data types. If we had relied on the inherent structure of the data frame in describing the strucutre of the data then the XML would be made more compact by eliminating the **dimension** and **dim** tags within each **Array** tag would not be necessary.

```
<?xml version="1.0"?>
<!DOCTYPE StatDataML PUBLIC "StatDataML.dtd">
<StatDataML xmlns="http://www.ci.tuwein.ac.at/StatDataML">
  <description>
    <title> Cars </title>
    <comment> A subset of the mtcars data from R </title>
  </description>

  <dataset>
    <list>
      <dimension>
        <dim size = "5">
          <e>ID</e> <e>MPG</e> <e>CYL</e> <e>HP</e> <e>AM</e>
        </dim>
      </dimension>
      <listdata>
        <array>
  <dimension> <dim size="3"/> </dimension>
  <type> <character/> </type>
          <data>
            <e> Mazda RX4 </e> <e> Datsun 710 </e> <e> Valiant </e>
          </data>
        </array>
        <array>
  <dimension> <dim size="3"/> </dimension>
  <type> <numeric/> </type>
          <data>
            <e> 21.0 </e> <e> 22.8 </e> <e> 18.1 </e>
          </data>
        </array>
        <array>
  <dimension> <dim size="3"/> </dimension>
  <type> <integer/> </type>
          <data>
            <e> 6 </e> <e> 4 </e> <e> 6 </e>
          </data>
        </array>
        <array>
  <dimension> <dim size="3"/> </dimension>
  <type> <numeric/> </type>
          <data>
            <e> 110 </e> <e> 93 </e> <e> 105 </e>
          </data>
        </array>
        <array>
```

```
  <dimension> <dim size="3"/> </dimension>
  <type>
    <categorical mode="unordered">
      <label code="0">Automatic</label>
      <label code="1">Manual</label>
    </categorical>
          </type>
          <data>
            <e> 1 </e> <e> 1 </e> <e> 0 </e>
          </data>
        </array>
      </listdata>
    </list>
  </dataset>

</StatDataML>
```

We provide a brief description of the elements of this StatDataML document.

- The name of the document containing the DTD is provided in the **DOCUMENTTYPE** element.
- The root element of the document is **StatDataML**. It contains two elements, the **description** element and the **dataset** element.
- The **description** element contains information pertaining to the source of the data.
- The **dataset** element contains the data along with all the markup describing it.
- The data may be a simple array or a list. In our case, we have a list object and so use the **list** element. Notice that it contains 5 **array** elements, one for each column of the data frame.
- The first **dimension** tag provides information about the number and names of arrays in the dataframe. We see that there are 5 columns, and their names are provided via the **e** tag.
- The other **dimension** tags appear inside the **array** elements and provide information about the length of the array.
- Information about the data type is provided in the array element via the **type** entity. Notice that this tag has no content, i.e. it is empty in all cases because the data type is provided via a **numeric**, **character**, or **categorical** tag.
- Finally, the **data** entity provides the values of the variables. Rather than prividing this information one record at a time, it is provided one variable at a time. This approach is more condusive to the list data structure.