Tutorial: Web Scraping in the R Language

Vlad Krotov
Murray State University
31 PUBLICATIONS **174** CITATIONS

Matthew Tennyson
Murray State University
11 PUBLICATIONS **29** CITATIONS

Tutorial: Web Scraping in the R Language

1. Introduction

The data available on the World Wide Web (WWW or simply "the web") today is measured in zettabytes (Cisco Systems, 2017) (1 zettabyte = 10^{21} bytes). This vast volume of data presents researchers and practitioners with a wealth of opportunities for gaining additional insights about individuals, organizations, or even macro-level socio-technical phenomena in real time. Not surprisingly, Information Systems researchers are increasingly turning to the web for data that can be used to address their research questions.

Harnessing the vast data from the web often requires a programmatic approach and a good foundation in various web technologies. Besides vast volume, there are three other common issues associated with accessing and parsing the data available on the web: variety, velocity, and veracity (Goes, 2014; Krotov & Silva, 2018). First, web data comes in a variety of formats that rely on different technological and regulatory standards (Basoglu & White, 2015). Second, this data is characterized by extreme velocity. The data on the web is in a constant state of flux: it is generated in real time and is continuously updated and modified. Another characteristic of web data is veracity (Goes, 2014). Due to the voluntary and often anonymous nature of the interactions on the web, quality and availability of web data are always surrounded with uncertainty. A researcher can never be completely sure if the data he or she needs will be available on the web and whether the data is reliable enough to be used in research (Krotov & Silva, 2018).

Given these issues associated with "big web data", harnessing this data requires a highly customizable, programmatic approach. Many "point and click" web scraping tools have been developed for accessing web data. One of these tools is import.io: a web-based tool for automating the process of web data retrieval and organization. The problem with these tools is that they do not always work in a way that allows the researcher to collect clean and tidy data. These tools are simply not "smart" enough to determine what exact data is needed by a researcher and make "sound" decisions with respect to data pre-processing and organization in ambiguous situations. Moreover, as the web is in a constant state of flux, a "ready-made" tool that works today may not necessarily work tomorrow due to changes in a website architecture, technologies, or simply content. Because of all these difficulties associated with "ready-made" web scraping tools, virtually every research project relying on web data requires a certain degree of web scraping tool modification or customization.

One functional and easily-customizable platform for retrieving and analyzing web data is R - one of the most widely-used programming languages in Data Science (Lander, 2014). R can be used not only for automating web data collection, but also for subjecting this data to a myriad of analysis techniques. Currently, there are more than 12,000 packages (extensions of the R language used for various data analysis techniques - from basic statistics to advanced machine learning and text mining) available for R (CRAN, 2018). This includes packages useful for web crawling and scraping, pre-processing, and organizing data stored on the web in various formats. Given the suitability of R for web data retrieval and analysis, the main goal of this research note is to educate Information Systems researchers and practitioners about the basic web scraping infrastructure in R. This tutorial provides sufficient background information on various web technologies (e.g. HTML, CSS, XML, XPath, etc.), R syntax, and RStudio so that even researchers with minimal knowledge of programming can source data from the web for their research projects.

2. Web Scraping in R: An Overview

Given the volume, variety, velocity, and veracity of Big Data available on the web, collection and organization of this data can hardly be done manually by individual researchers or even large teams of data entry specialists. What is needed to make the data collection and organization fast and error-free is a technological platform that can be used to automate at least some aspects of these processes. This emerging practice of using technology for collecting and organizing data from the web is often referred to as web scraping (Krotov & Tennyson, 2018; Krotov & Silva, 2018).

In this tutorial, web scraping is broadly defined as using technology tools for automatic retrieval and organization of data from the web for the purpose of further analysis of this data (Krotov & Tennyson, 2018; Krotov & Silva, 2018). Web scraping consists of the following main, intertwined phases: website analysis, website crawling, and data organization (see Figure 1) (Krotov & Silva, 2018). Although listed in order, these phases are often intertwined. A researcher has to go back and forth between those phases until a clean, tidy dataset suitable for further analysis is obtained.

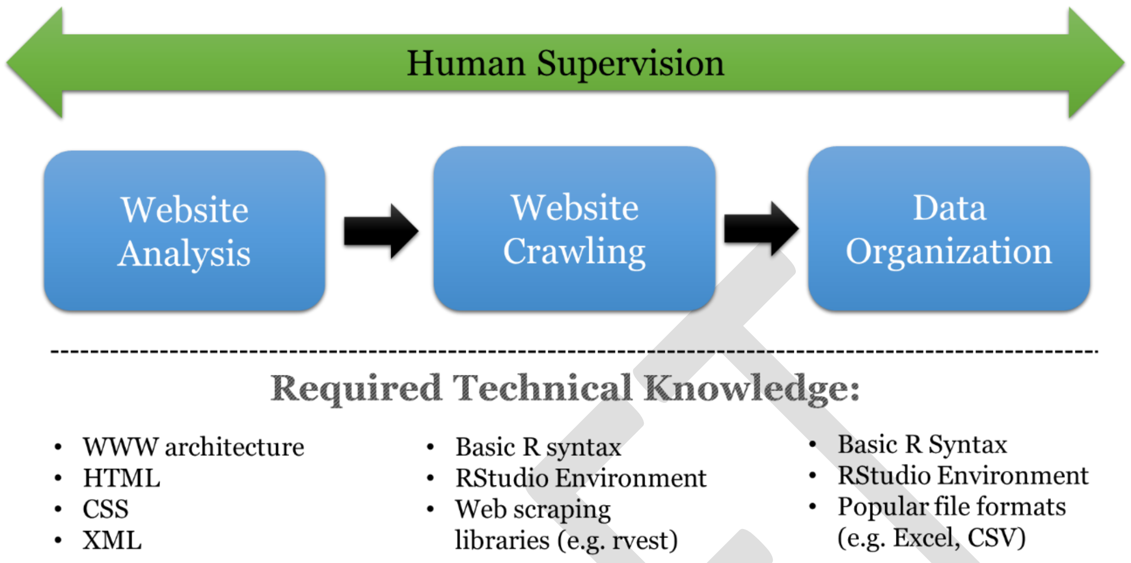


Figure 1. Web Scraping (Adapted from Krotov & Tennyson, 2018; Krotov & Silva, 2018)

The "Website Analysis" phase involves examining the underlying structure of a website or a web repository in order to understand how the needed data is stored at the technical level. This is often done one web page at a time. This analysis requires a basic understanding of the World Wide Web architecture and some of the most commonly used Web technologies used for storing and transmitting data on the web: HTML, CSS, and XML.

The "Web Crawling" phase involves developing and running a script that automatically browses (or "crawls") the web and retrieves the data needed for a research project. These crawling applications (or scripts) are often developed using programming languages such as R or Python. We argue that R is especially suitable for this purpose. This has to do with the overall popularity of R in the Data Science community and availability of various packages for automatic crawling (e.g. the "rvest" package in R) and parsing (e.g. the "jsonlite" package in R) of web data. Furthermore, once data is retrieved using R, it can be subjected to various forms of analysis available in the form of R packages. Thus, R can be used to automate the entire research process – from the time data is acquired to the time visualizations and written reports are produced for a research paper or presentation (the latter can be accomplished with the help of the package called "knitr").

The “Data Organization” phase involves pre-processing and organizing data in a way that enables further analysis. In order to make further analysis of this data easy, the data needs be to be clean and tidy. Data is in a tidy format when each variable comprises a column, each observation of that variable comprises a row, and the table is supplied with intuitive linguistic labels and the necessary metadata (Wickham, 2014b). A dataset is “clean” when each observation is free from redundancies or impurities (e.g. extra white spaces or mark-up tags) that can potentially stand in the way of analyzing the data and arriving to valid conclusions based on this data. This often requires the knowledge of various popular file formats, such as Excel or CSV. Again, R contains various packages and built-in functions for working with a variety of formats. This is another set of features that make R especially suitable for web scraping.

Most of the time, at least some of the processes within these three phases cannot be fully automated and require at least some degree of human involvement or supervision. For example, “ready-made” web scraping tools often select wrong data elements from a web page. This often has to do with poor instructions supplied by the user of such tools or an ambiguous mark-up used to format data. These tools also often fail to save the necessary data elements in the “tidy data” format (Wickham, 2014b), as data cleaning often requires human interpretation of what this data represents. Moreover, numerous networking errors are possible during web crawling (e.g. an unresponsive web server) that require troubleshooting by a human. Finally, things change so fast on the web! A tool that is working now may not work in the future due to changes made to a website. All these problems become more acute for large, complicated web scraping tasks, making these “ready-made” tools practically useless. Thus, at least with the current state of technology, web scraping often cannot be fully automated and requires a “human touch” together with a highly customizable approach. Developing custom tools for web scraping requires a general understanding of the web architecture; a good foundation in a programming environment suitable for web scraping task (e.g. R); and at least basic knowledge of some of the most commonly used mark-up languages on the web, such as HTML, CSS, and XML. Each of these technologies is discussed in the subsequent sections of this tutorial.

3. Web Architecture and Technologies

This section provides a brief overview of web architecture (i.e. how data is stored and transmitted on the web) and various technologies, such as HTML, CSS, XML, and R. This section provides enough technical information for someone wishing to undertake a research project involving scraping data from the web. The reader is encouraged to consult the resources listed in Appendix A for more in-depth treatment of the technologies discussed in this section.

3.1 World Wide Web Architecture

The World Wide Web is a vast collection of documents interconnected via hyperlinks and transferred via the Hypertext Transfer Protocol (HTTP). In the early days of the web, these documents contained text and images. Nowadays, webpages also often contain audio, video, and other forms of media. Some webpages even have code embedded in them or even contain complete software applications. In general, all webpages usually contain hypertext: textual and graphical content interspersed with hyperlinks. Each hyperlink links the current document with another document, which in turn links to other documents, resulting in a literal "worldwide web" of hypertext documents that span the globe.

The web utilizes a “client/server” architecture (see Figure 2). The documents are hosted by web servers. A web server is a software application that continuously runs and waits for requests from clients. When a request for a document is received, it is the web server's responsibility to retrieve the requested document and respond by sending that document to the client that submitted the request. In turn, a client is a software application whose responsibility is to send a request for a particular document to a web server, and when the response is received, to interpret the HTML content of that document. The most common type of web client application is a web browser, with which the reader is undoubtedly familiar. A request is initiated in a browser by the user when a hyperlink is clicked on a page that is already open in the browser, or by typing a URL into the browser's address bar. The mechanism of this request/response exchange between the client and server is dictated by the HTTP protocol. More in-depth sources discussing the World Wide Web, the client/server model, web servers and clients, the HTTP protocol, and related topics can be found in Appendix A.

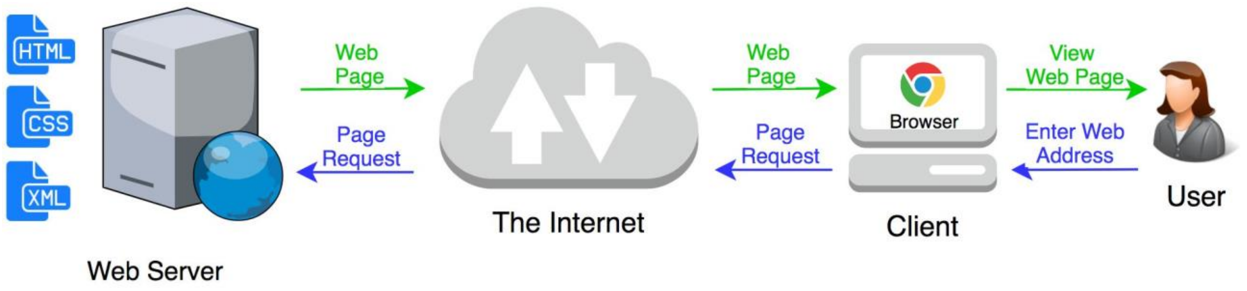


Figure 2. World Wide Web Architecture

Hypertext documents, referred to as "webpages" when accessible via the web, are encoded using the Hypertext Markup Language (HTML). More specifically, the content of a webpage is encoded as HTML. The way the content is presented is often encoded as Cascading Style Sheets (CSS). Data can also be stored and transmitted using Extensible Markup Language (XML). Web scraping requires a basic knowledge of these technologies, which are discussed further in this section.

3.2 HTML Overview

The most important construct in HTML is known as an element, and elements are denoted using tags. There are many types of elements in HTML, each of which is denoted using a corresponding HTML tag. Examples of elements include paragraphs, lists, tables, hyperlinks, images, and other types of content that one might find on a webpage. Tags are represented with angled brackets and they typically come in pairs. An opening tag denotes the beginning of an element, while a closing tag denotes the end of that element. For example, the HTML code below defines (or "marks up") a paragraph in HTML:

```
<p>This is a paragraph.</p>
```

The opening tag <p> denotes the beginning of the paragraph, while the closing tag </p> denotes the end of the paragraph. The text between the opening and closing tags defines the actual content of that paragraph element.

Tags can be embedded inside other tags, which allows for aggregation of content. For example, the code below defines an unordered list that contains three list items:

```
<ul>
  <li>Item #1</li>
  <li>Item #2</li>
  <li>Item #3</li>
```

```
</ul>
```

The opening tag <ul> denotes the beginning of the unordered list, while the closing tag </ul> denotes the end of the list. The list consists of three items, each of which is denoted by a pair of opening and closing tags <li>…</li>. Each of the list items consists of some arbitrary text.

One HTML element that is of special interest to accounting researchers is the table element. HTML tables are often used to display quantitative and qualitative data on the web, including data related to various financial statements. A table element is created by using a pair of <table></table> tags. Table data is added by adding elements between the opening <table> and closing </table> tags. This is done by adding rows (using <tr> and </tr> tags). Each row contains cells that are marked with <td></td> tags. Take a look at the HTML table below:

Table 1. A Simple HTML Table

Row 1 Column 1	Row 1 Column 2
Row 2 Column 1	Row 2 Column 2

The underlying HTML code is provided below:

```
<table border="1">
  <tr>
    <td>Row 1, Column 1</td>
    <td>Row 1, Column 2</td>
  </tr>
  <tr>
    <td>Row 2, Column 1</td>
    <td>Row 2, Column 2</td>
  </tr>
</table>
```

Note that the opening <table> tag contains an attribute with a value (border="1"). This attribute specifies that the table should have a visible border. Attributes are discussed in further detail later in this section.

The HTML standard (https://www.w3.org/TR/html/) dictates that all content inside an HTML document be contained inside a pair of <html> tags. Also, inside the <html> tags, there must exist a pair of <head> tags and a pair of <body> tags. For example, the code below defines an HTML document that contains a head and a body element:

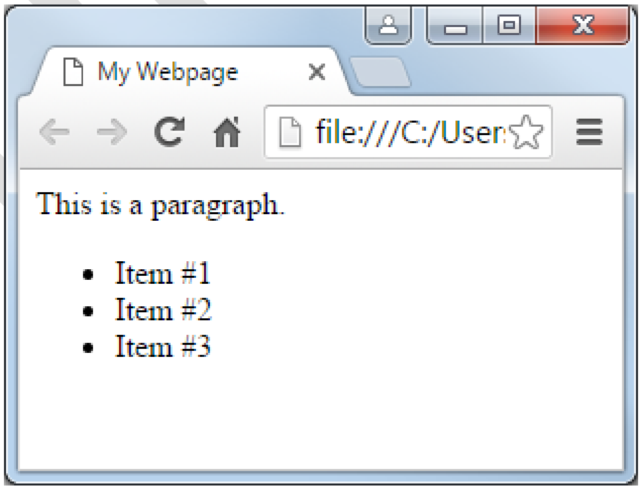
```
<html>
    <head>...</head>
    <body>...</body>
</html>
```

The content of the head and body is omitted for the sake of brevity. Note that every well-formed HTML document will have these three basic elements: html, head, and body.

The <head> of an HTML document consists only of metadata, such as the title of the webpage. This metadata found inside the <head> of the HTML document is not explicitly displayed when it is viewed inside a browser. The <body> of an HTML document, on the other hand, contains of all the webpage's visible content, such as paragraphs, lists, tables, images, and hyperlinks. Here is an example of a simple, complete, and well-formed HTML document:

```
<html>
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <p>This is a paragraph.</p>
    <ul>
      <li>Item #1</li>
      <li>Item #2</li>
      <li>Item #3</li>
    </ul>
  </body>
</html>
```

When viewed inside a browser, this HTML document will appear as shown in Figure 3.



This is a paragraph.

- Item #1
- Item #2
- Item #3

Figure 3. Appearance of HTML as Rendered by a Browser

HTML tags often contain attributes. An attribute provides additional information about an element and is always listed inside the element's opening tag. For example, this HTML code defines a paragraph that is similar to the paragraph seen in the previous example:

```
<p title="foo">This is a paragraph.</p>
```

However, in this paragraph, a title attribute has been defined with "foo" as the attribute's value. Attributes are always defined using this syntax (attribute="value"), where the attribute is followed by an equal sign and the value is contained inside a pair of quotation marks.

The anchor element is denoted using the <a> tag, and it is most often used to represent a hyperlink. When used as a hyperlink, the <a> tag must contain the “href” attribute to signify what document the hyperlink points to. For example, the following HTML code defines a hyperlink that links to the W3C website:

```
<a href="https://www.w3.org/">World Wide Web Consortium</a>
```

The text "World Wide Web Consortium" will be displayed as a clickable hyperlink within the browser window. If the link is clicked by the user, then the browser will submit a request for the "https://www.w3.org/" website. When a response is received, the browser will interpret the HTML and display the new webpage in the browser window.

Another common element that requires the use of attributes is the image element, denoted using the <img> tag. For example, the following code defines an image element:

```
<img src="https://www.w3.org/2008/site/images/icon_sprite.png" />
```

The “src” attribute defines where the image source file can be found. In this case, the image can be found via the web on the W3C website. Note that the <img> tag is quite unusual. The element only contains an opening tag; there is no closing tag. Because of this peculiarity, a slash is placed just before the second angled bracket inside the opening tag of the element. For any element that is denoted using only an opening tag, a slash should be placed inside the opening tag in this manner to denote that no closing tag exists.

3.3 CSS Overview

While HTML is used to define the content of a webpage, Cascading Style Sheets (CSS) are used to define how that content is presented. Things like fonts, colors, margins, background images, among other things, can be defined using CSS. CSS can be embedded into an HTML document in three ways: (1) by storing the CSS code in a separate file, (2) by using the <style> tag inside the <head> of the document, and (3) by assigning a value to the style attribute inside the opening tag of any particular HTML element.

When the CSS code is stored in a separate file, it is pulled into the HTML document using the <link> element inside the <head> of the HTML document, as follows:

```
<head>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

In the <link> element above, the href attribute refers to an external file called style.css, which must contain CSS code. The style defined inside the external file will be applied by the browser to all elements contained in the <body> of that HTML document.

Alternatively, the CSS code can be stored directly inside the HTML document using the <style> element, as follows:

```
<head>
  <style>

    body
    {
      color: red;
      background-color: blue;
    }

  </style>
</head>
```

In the <style> element shown above, the CSS code specifies a style that will be applied to the entire contents of the <body> element inside the HTML document. In this example, the font color will be red and the background color will be blue.

Finally, CSS code can be stored directly inside each individual HTML element within the document using the style attribute, as follows:

```
<p style="color: green; background-color: black;">Some paragraph!</p>
```

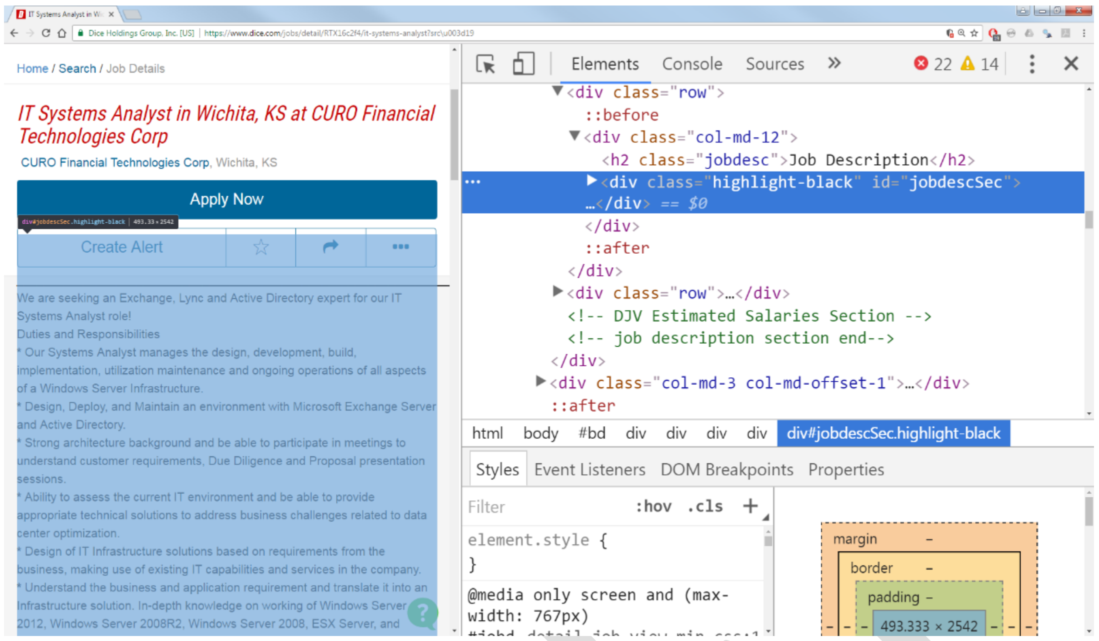
In the paragraph element shown above, the style attribute uses CSS code to specify that this particular paragraph should be displayed with green font and a black background.

Regardless of which of the three strategies is used, the syntax of the CSS code remains the same. CSS code always consists of selectors and respective rules. The rules associated with each selector are contained inside a pair of curly braces:

```
body
{
  color: red;
  background-color: blue;
}

p
{
  color: green;
  background-color: black;
}
```

Understanding of this "selector-rules" principle is useful when retrieving data programmatically from a webpage. For example, once a researcher determines that the needed data is contained within the <p> element, this selector can be passed to an R script (e.g. when using the "rvest" package discussed in subsequent sections) to tell the script to grab data only from the specified selector. One can determine which selector contains the needed data by looking at the source code of a webpage. Several tools can make this task easier. For example, one can use the "Developer Tools" feature of Google Chrome browser to see the source code of a particular page and quickly browse through various HTML and CSS elements (see Figure 4).



Home / Search / Job Details

IT Systems Analyst in Wichita, KS at CURO Financial Technologies Corp

CURO Financial Technologies Corp, Wichita, KS

Apply Now

Create Alert

We are seeking an Exchange, Lync and Active Directory expert for our IT Systems Analyst role!
Duties and Responsibilities
* Our Systems Analyst manages the design, development, build, implementation, utilization maintenance and ongoing operations of all aspects of a Windows Server Infrastructure.
* Design, Deploy, and Maintain an environment with Microsoft Exchange Server and Active Directory.
* Strong architecture background and be able to participate in meetings to understand customer requirements, Due Diligence and Proposal presentation sessions.
* Ability to assess the current IT environment and be able to provide appropriate technical solutions to address business challenges related to data center optimization.
* Design of IT Infrastructure solutions based on requirements from the business, making use of existing IT capabilities and services in the company.
* Understand the business and application requirement and translate it into an Infrastructure solution. In-depth knowledge on working of Windows Server 2012, Windows Server 2008R2, Windows Server 2008, ESX Server, and

```
▼<div class="row">
   ::before
  ▼<div class="col-md-12">
     <h2 class="jobdesc">Job Description</h2>
    ▶<div class="highlight-black" id="jobdescSec">
     …</div> == $0
   </div>
   ::after
 </div>
▶<div class="row">…</div>
 <!-- DJV Estimated Salaries Section -->
 <!-- job description section end-->
</div>
▶<div class="col-md-3 col-md-offset-1">…</div>
::after
```

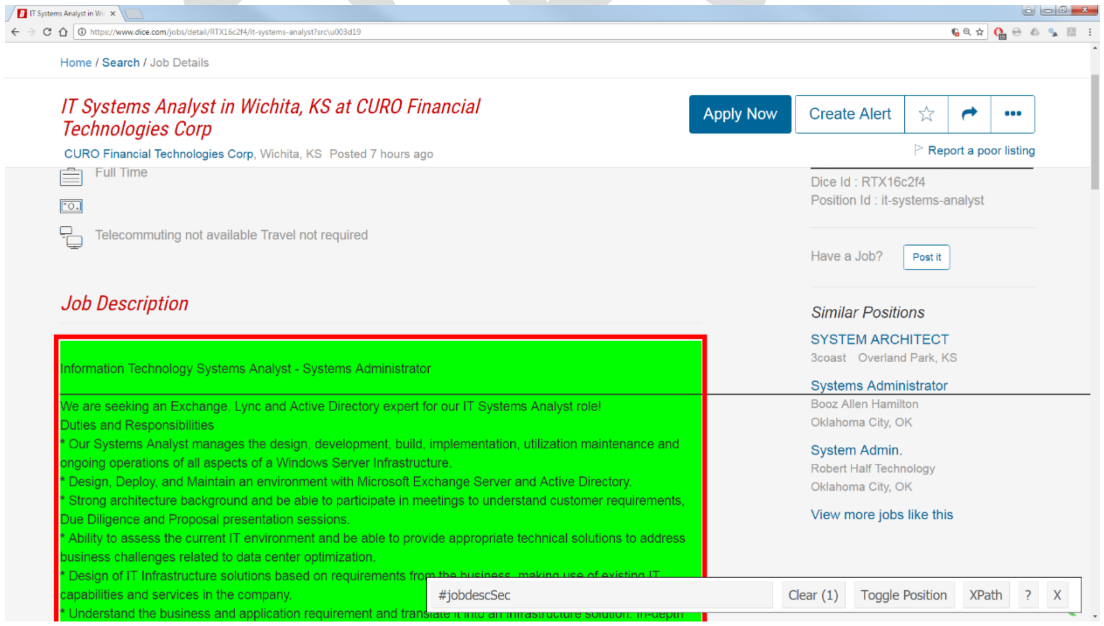
html body #bd div div div div div#jobdescSec.highlight-black

```
element.style {
}
@media only screen and (max-width: 767px)
```

493.333 × 2542

Figure 4. Analyzing CSS Code in Google Chrome

Alternatively, one can use the SelectorGadget add-on to the Google Chrome browser to quickly visualize a page in terms of various CSS selectors, determine which selector needs to be “grabbed”, and determine the XPath address of that specific element to be subsequently used in conjunction with the R script (see Figure 5).



Home / Search / Job Details

IT Systems Analyst in Wichita, KS at CURO Financial Technologies Corp

CURO Financial Technologies Corp, Wichita, KS Posted 7 hours ago

Full Time

Telecommuting not available Travel not required

Dice Id : RTX16c2f4
Position Id : it-systems-analyst

Have a Job? Post it

Job Description

Information Technology Systems Analyst - Systems Administrator

We are seeking an Exchange, Lync and Active Directory expert for our IT Systems Analyst role!
Duties and Responsibilities
* Our Systems Analyst manages the design, development, build, implementation, utilization maintenance and ongoing operations of all aspects of a Windows Server Infrastructure.
* Design, Deploy, and Maintain an environment with Microsoft Exchange Server and Active Directory.
* Strong architecture background and be able to participate in meetings to understand customer requirements, Due Diligence and Proposal presentation sessions.
* Ability to assess the current IT environment and be able to provide appropriate technical solutions to address business challenges related to data center optimization.
* Design of IT Infrastructure solutions based on requirements from the business, making use of existing IT capabilities and services in the company.
* Understand the business and application requirement and translate it into an infrastructure solution. In-depth

Similar Positions
SYSTEM ARCHITECT
3coast Overland Park, KS
Systems Administrator
Booz Allen Hamilton
Oklahoma City, OK
System Admin.
Robert Half Technology
Oklahoma City, OK
View more jobs like this

#jobdescSec Clear (1) Toggle Position XPath ? X

Figure 5. Finding CSS Element Using the SelectorGadget Add-on in Chrome

The tool can also be used in conjunction with XML documents to access specific XML nodes. The essence of XML technology and the structure of an XML document are explained below.

3.4 XML Overview

XML (Extensible Markup Language) is another widely used web technology one often needs to be familiar with in order to perform web scraping. While HTML was designed predominantly to display information on the web, XML was designed to store and transmit data. Thus, XML allows a web developer to separate visual presentation of a webpage from its data. While a number of Internet-based and stand-alone applications rely on XML for data storage and exchange, this markup language was created primarily for storing and transmitting data over the web.

On the surface, XML seems to be quite similar HTML. But there are a number of important differences between XML and HTML (W3Schools, 2017):

- Unlike HTML, which can display text and other media in a browser, XML does not do anything; it only stores data and uses the created data structures to transmit this data.
- XML, unlike HTML, does not have a set of predefined tags. Instead, developers are expected to create their own tags - depending on the data that is stored and transmitted by their web applications.
- Unlike HTML, XML was designed to be readable by both humans and machines. This is achieved by using self-descriptive tags as a part of XML code.

XML documents have a tree-like structure (see Figure 6). As one can see from Figure 6, an XML document starts with a root element. In this particular example, the root element is <bookstore>. Parent elements have children elements, meaning lower-level XML elements that are linked to the parent XML element. In this particular example, a child of <bookstore> is a child element <book>. This relationship can also be explained in the reverse direction: the child element <book> has a parent element <bookstore>. Any element can have a child element and every element, with the exception of the root element, can have a parent. Siblings are children elements on the same level (e.g. element <title> and element <author>).

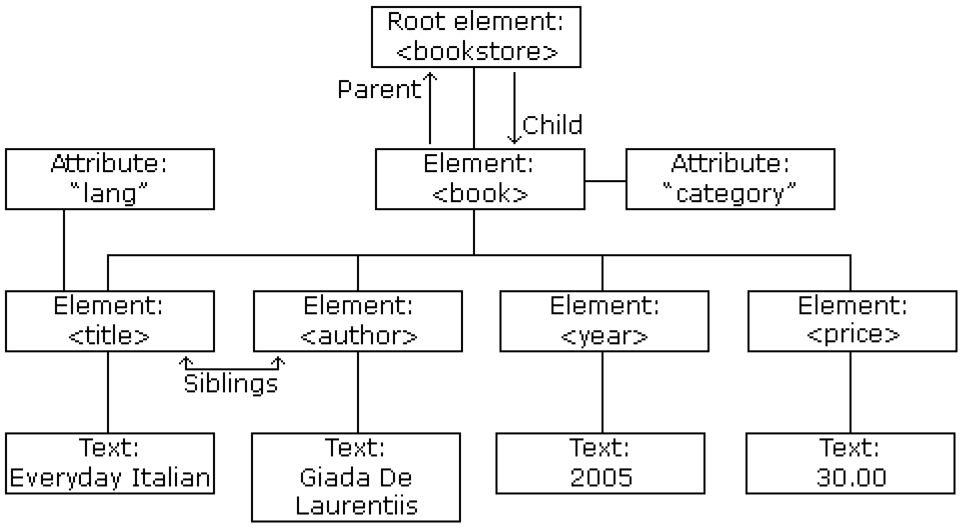


Figure 6. XML Document Structure (Reprinted from W3Schools, 2017)

The structure depicted in Figure 6 can be implemented using the following XML code published by W3Schools (2017):

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J.K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

The XML document above stores information about books. This can be quickly grasped due to the self-explanatory nature of XML tags used in this document. Also note that elements can have text content (e.g. the author element can have "J.K. Rowling" as its text). Also, XML elements can have attributes (e.g. category="children").

These attributes are used to hold data in relation to a specific XML element (e.g. a specific book title, in this case).

4. The R Environment

It is not necessary to have knowledge and experience in R programming in order understand the R scripts for scraping web data discussed in this section. Anyone familiar with any high-level programming language should be able to understand these web scraping examples. However, some basic understanding of the R environment will be helpful for running and, perhaps, customizing these scripts. Of course, some knowledge of R and related web technologies is necessary if one wishes to develop his or her own tools for web scraping. Therefore, this section provides an overview of the R language, the RStudio environment, as well as some packages and other resources available through the Comprehensive R Archive Network (CRAN) website that can be useful for web scraping.

4.1 R Language Overview

R is a programming language used for data manipulations, statistical analysis, text mining, data visualization, and report generation (CRAN, 2018). Just like most programming languages, R includes conditional statements, loops, functions, etc. But unlike a typical programming language, R includes features that make the language especially suited for data analysis. These features include (CRAN, 2018):

- Functions for data retrieval, manipulation, and storage (e.g. functions for reading Excel files)
- Operators for mathematical calculations and other operations using arrays and matrices (most operators in R can be applied to all items in a vector or array)
- Built-in and user-developed collections of tools for basic and intermediate statistical analysis (e.g. descriptive statistics, ANOVA, regression, PLS, etc.)
- Packages implementing popular text mining and machine learning algorithms
- Graphics packages for data analysis and visualization for both on-screen and hard-copy

- The ability to run code written in C, C++, and Fortran (to improve execution speed)
- The ability to interface with code written in Python (another popular language for data science)

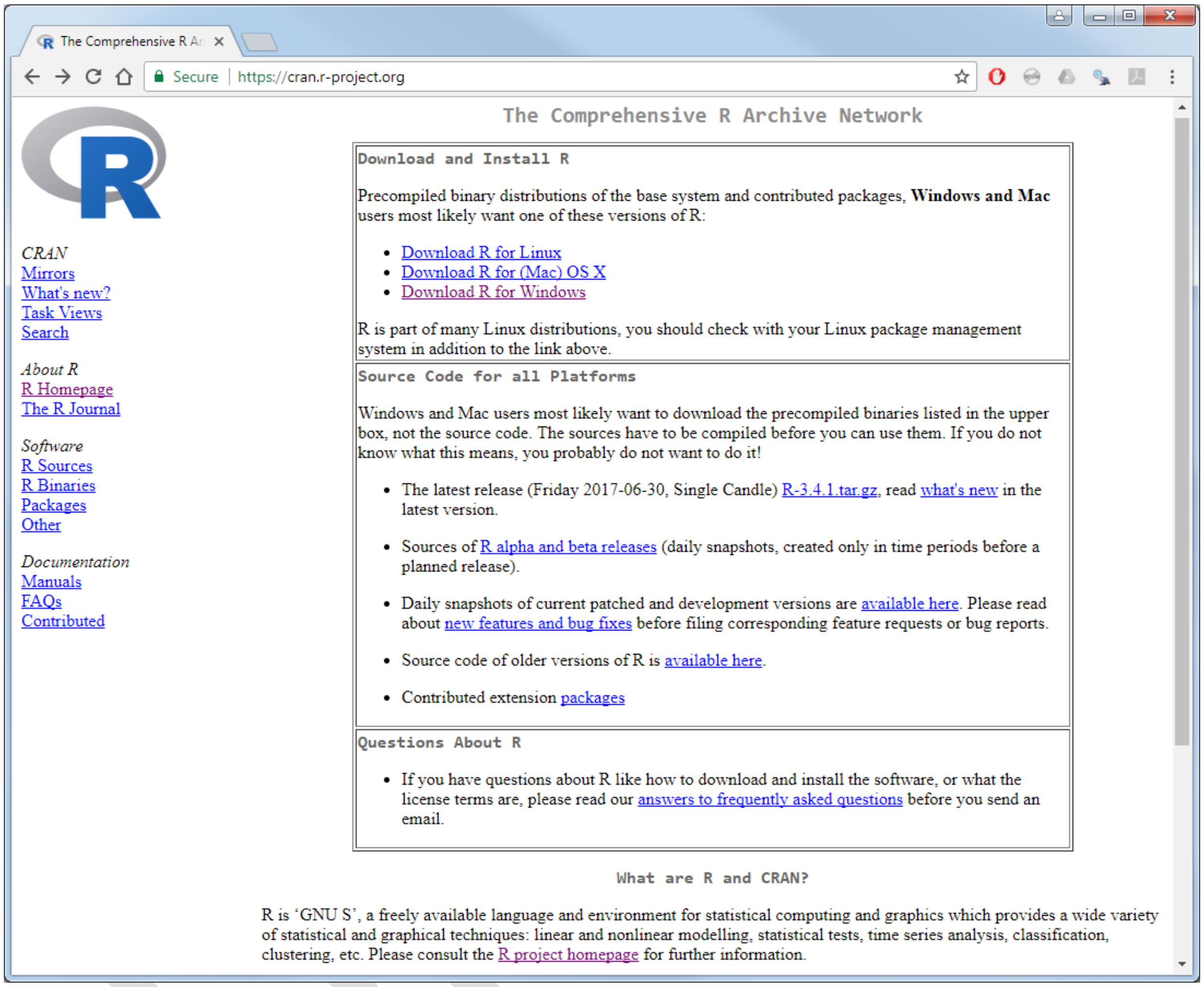
Since its original release in 1992 by Ross Ishaka and Rob Gentleman, two statisticians from the University of Auckland in New Zealand, the R language has grown substantially in terms of its functionality and use (CRAN, 2018). This can be seen by going through the resources provided online by the Comprehensive R Archive Network (CRAN) (https://cran.r-project.org/). The website hosts R project documentation, precompiled binary distributions of the R language for various operating systems, source code, R manuals, and even a scholarly journal called The R Journal.

Of special importance in R are more than 12,000 user-contributed packages available via the CRAN repository. These packages are free R language extensions or add-ons developed by the R community members and "peer reviewed" by the maintainers of CRAN. These packages implement various data manipulation, analysis, and visualization techniques. Using these packages one can perform virtually all known forms of analysis of quantitative and qualitative data, ranging from simple data manipulations (e.g. "dplyr" package) to text mining (e.g. "tm" package) and machine learning (e.g. "FCNN4R" package). Of special importance are graphical packages (e.g. "ggplot2" package) that contain extensive collections of visual elements that allow users to automatically generate professional looking and highly customizable data visualizations. In this tutorial, names of R packages are in double quotes.

With thousands of R packages available, it is not surprising that R is increasingly becoming a de facto language of data science (Lander, 2014; Schutt & O'Neil, 2013). Unlike many of the commercial statistical packages that are geared towards a particular audience (e.g. SPSS is geared towards statistical analysis of quantitative data in business and social sciences), R is used across many industries and scientific disciplines (Lander, 2014) from environmental science to English literature.

4.2 Installing R

To start using R, one needs to install the so-called "R base" first. This is a distributive of the R language itself. The latest version of R base can be downloaded from CRAN. For that, one needs to go to the CRAN website (https://cran.r-project.org/) and click on the link for an appropriate operating system (see Figure 7).



The Comprehensive R Archive Network

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Friday 2017-06-30, Single Candle) R-3.4.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

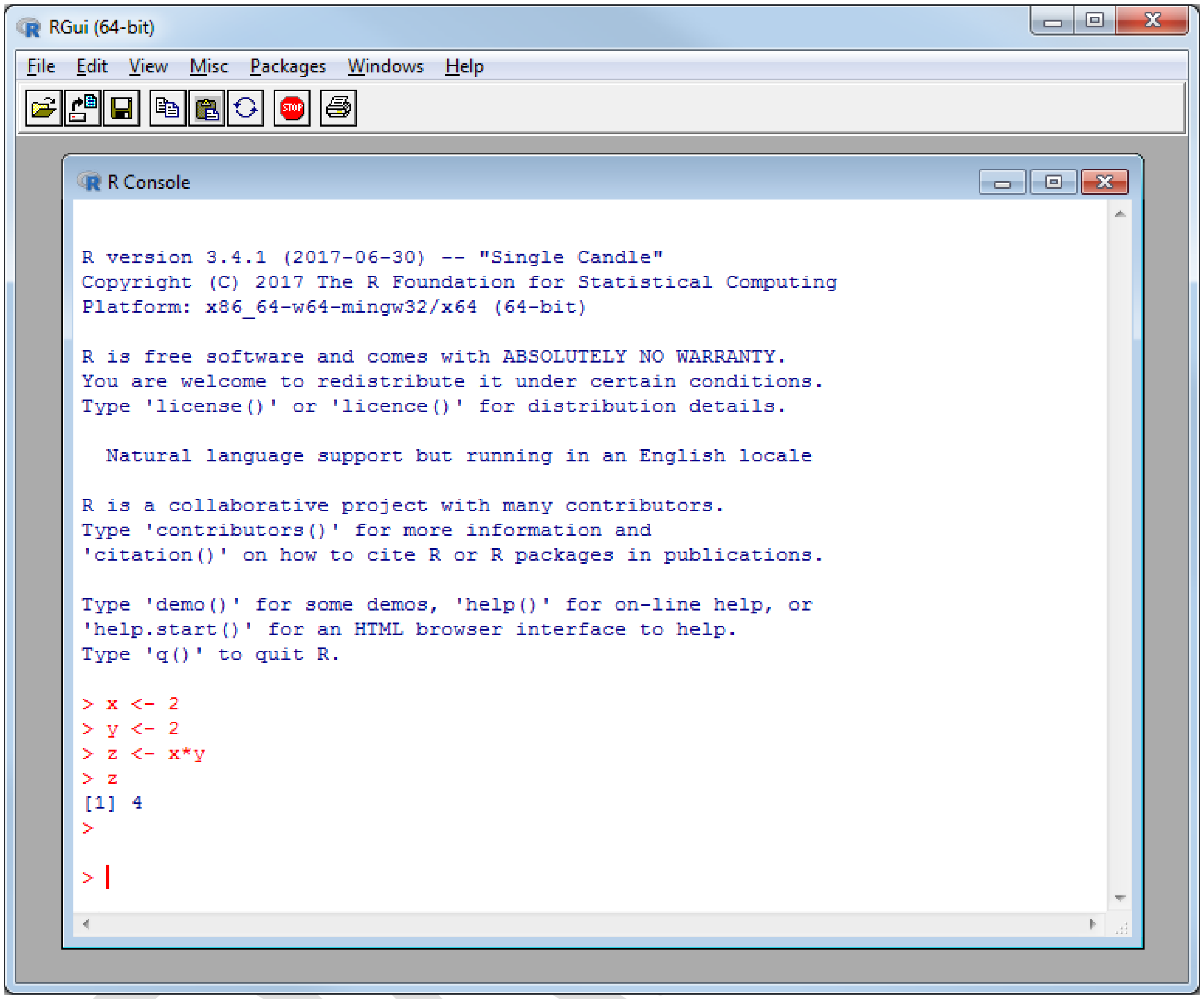
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

Figure 7. Downloading R from the Comprehensive R Archive Network (CRAN)

Once installed, R can be run in a console (see Figure 8). In this mode, each line of R code is entered and interpreted one at a time. The system processes each line of code after the user hits "Enter". A simple example is shown in Figure 8. In this example, the value 2 is stored in two variables, x and y. Those values are then multiplied and the result is stored in the variable z. When the user types the variable "z" by itself, its value "4" is displayed.



```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> x <- 2
> y <- 2
> z <- x*y
> z
[1] 4
>

> 
```

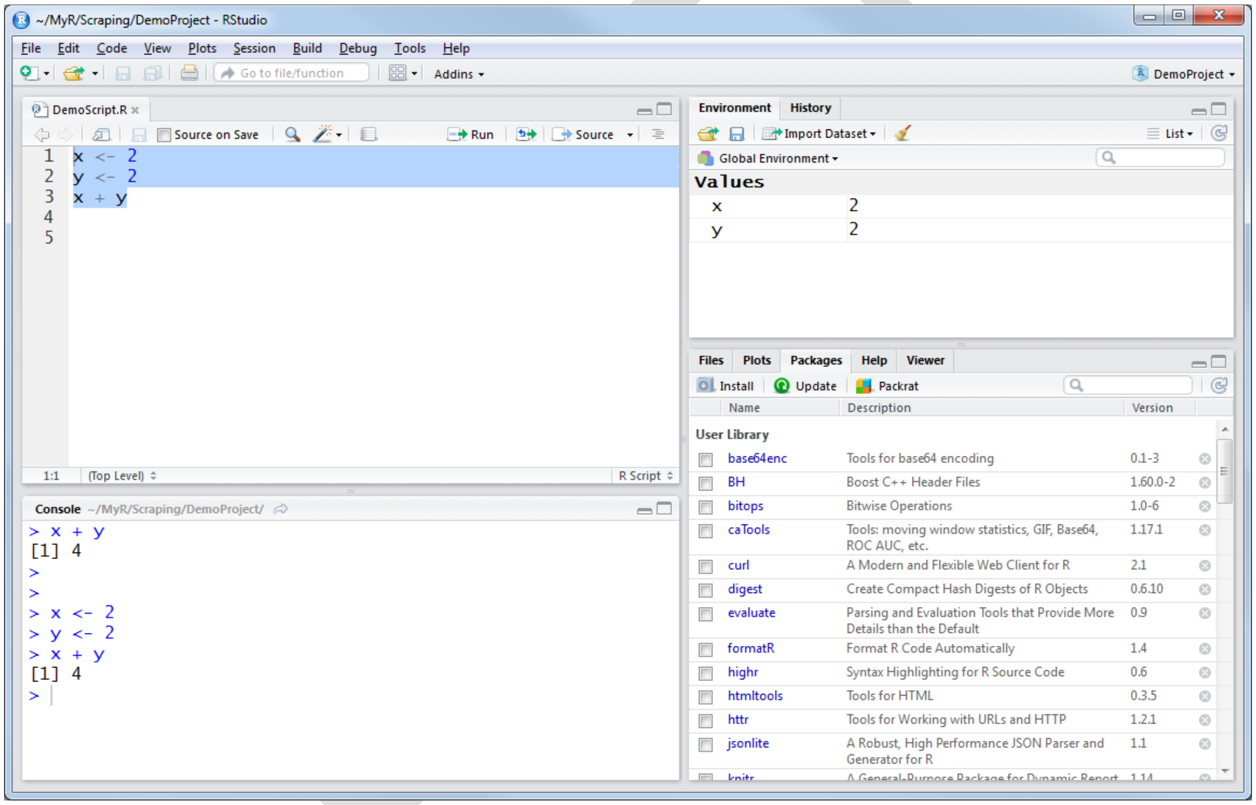
Figure 8. Downloading R from the Comprehensive R Archive Network (CRAN)

4.3 The RStudio Environment

Although R code can be run in a console, most people prefer to install an Integrated Development Environment (IDE) to be used on the top of an R base installation. While there are many IDEs developed for R, RStudio seems to be the most popular one (https://www.rstudio.com/). RStudio includes a free version as well as several enterprise products (e.g. RStudio Commercial Desktop and RStudio Server Pro) that require an annual license fee. The free, desktop version of R usually meets the needs of most individual academic and industry researchers.

There are many advantages associated with using RStudio. RStudio makes the R language easier to use and facilitates development of advanced R scripts. The two essential features that facilitate development in R is a code editor together with code debugging and visualization tools. Numerous add-ons can be installed to work with RStudio interface. For example, Shiny server (available from the RStudio website) provides a framework for building web applications using R without the knowledge of HTML, CSS, and JavaScript. RStudio's Graphical User Interface (GUI) can also be integrated with the "knitr" package (available from CRAN). The "knitr" package can be used in conjunction with the so-called R Markdown language (see http://rmarkdown.rstudio.com/) to generate reports in HTML, PDF, Word, and PowerPoint formats based on the output of data analysis done in R. These reports mesh together R code, user text, and R script output to generate the output.

A typical layout of the RStudio IDE is shown in Figure 9.



```
x <- 2
y <- 2
x + y
```

```
> x + y
[1] 4
>
>
> x <- 2
> y <- 2
> x + y
[1] 4
>
```

Values

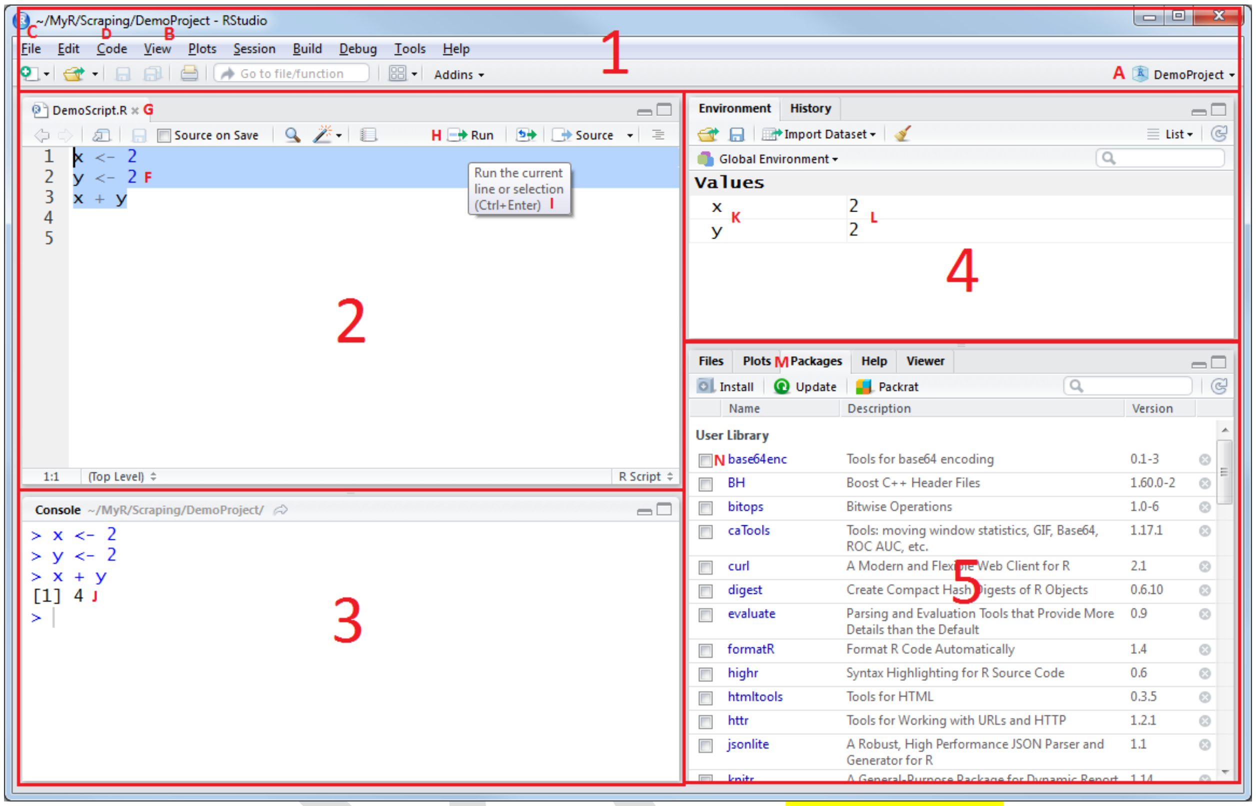
x	2
y	2

Name	Description	Version
User Library		
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.60.0-2
bitops	Bitwise Operations	1.0-6
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
curl	A Modern and Flexible Web Client for R	2.1
digest	Create Compact Hash Digests of R Objects	0.6.10
evaluate	Parsing and Evaluation Tools that Provide More Details than the Default	0.9
formatR	Format R Code Automatically	1.4
highr	Syntax Highlighting for R Source Code	0.6
htmltools	Tools for HTML	0.3.5
httr	Tools for Working with URLs and HTTP	1.2.1
jsonlite	A Robust, High Performance JSON Parser and Generator for R	1.1
knitr	A General-Purpose Package for Dynamic Report	1.14

Figure 9. The RStudio Environment

Note that the layout of RStudio interface depicted in Figure 9 not a default layout that a user sees upon launching a newly installed copy of RStudio. RStudio interface layout can be modified using the options available from the "View" menu item that can be seen at the top of the RStudio window.

The paragraphs below review some of the most essential features of RStudio interface. For the reader convenience, specific areas of the RStudio screen are highlighted in Figure 10 using red color and labeled with numbers. Capital letters are used to reference specific interface areas, like menu items and buttons. The RStudio interface markup and related examples come from (Reference needed).



```
x <- 2
y <- 2
x + y
```

```
> x <- 2
> y <- 2
> x + y
[1] 4
>
```

Values

x	2
y	2

User Library

Name	Description	Version
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.60.0-2
bitops	Bitwise Operations	1.0-6
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
curl	A Modern and Flexible Web Client for R	2.1
digest	Create Compact Hash Digests of R Objects	0.6.10
evaluate	Parsing and Evaluation Tools that Provide More Details than the Default	0.9
formatR	Format R Code Automatically	1.4
highr	Syntax Highlighting for R Source Code	0.6
htmltools	Tools for HTML	0.3.5
httr	Tools for Working with URLs and HTTP	1.2.1
jsonlite	A Robust, High Performance JSON Parser and Generator for R	1.1
knitr	A General-Purpose Package for Dynamic Report	1.14

Figure 10. RStudio GUI Elements with Labels (Reference needed)

As one can see in Figure 10, an R project titled "DemoProject" (see Area 1, Label A) was created in the "DemoProject" folder (see Area 1, Label B). An R project is a collection of various R files (e.g. data files, R source code files) and memory contents (e.g. variables and data structures loaded into memory). R projects are saved by RStudio as files with the extension ".Proj". The project was created by going to File → New Project (see Area 1, Label C) using the main menu of RStudio (see Area 1).

Once the project was created, an R source code file called "DemoScript.R" was added to the project (see Area 2). The file was created by going to "File → New File → R Script" (see Area 1, Label C). The following lines of R code were entered into the opened code window (see Area 2, Label F):

```
x <- 2
y <- 2
x + y
```

Once the code was entered, the script file was saved with the following file name “DemoScript.R” (see Area 2, Label G). The file was saved by going to “File → Save As…” The script was saved into the same folder as the project file: “Demo Project”. RStudio assigns the following extension to files containing R code: “.R” (see Area 2, Label G).

After the lines of R codes were entered, they were executed by highlighting (see Area 2, Label F) the code inside the “DemoScript.R” file (it is highlighted in Figure 10) and pressing the “Run” button at the top of the pane where the code was entered (see Area 2, Label H). The code inside the script can be highlighted using a left mouse button, just like one would highlight text on a web page in a browser. Alternatively, the entire R code in the pane can be highlighted by pressing “Ctrl+A”. Please note that only the highlighted lines of code are executed once the “Run” button is pressed. Thus, a user can chose which lines of code are executed. This feature can be useful for debugging code.

Once the code was executed, the output (see Area 3, Label J) was displayed in the console below the code editor (see Area 3). The environment window (see Area 4) shows the variables used in the script (x and y) (see Area 4, Label K) and their respective values (both contain number 2) (see Area 4, Label L). This means that the variables are now a part of the R environment memory and now can be used and referenced within the R code.

The window in the bottom right corner (see Area 5) shows the R packages installed together with RStudio (see Area 5, Label M). By clicking inside a checkbox (see Area 5, Label N) next to a package, one can load a particular package to be used in a script. Alternatively, the packages can be loaded within R code using “library()” and “require()” commands (this is explained in the next section). Since the simple script used here does not use any of the packages, none of the packages inside the “Packages” tab is checked.

4.4 An Introduction to R Syntax

This tutorial on R syntax assumes that the reader has some familiarity with a programming language. Therefore, this section is not meant to teach the reader programming in general or R programming in particular. Instead, this section presents the most essential elements R syntax that can be helpful for understanding the web scraping code examples presented later. If one wishes to learn more about R language syntax, one should refer to the additional resources listed in Appendix A.

Most of the code examples and related explanations presented in this section and related explanations come from Lander (2014) and (Reference needed). The examples in this section have the following format:

```
> Some R code
Some system message
[1] Output generated by the R code
```

As one can see from the format above, the code examples are copied directly from RStudio code output window (see Area 3 in Figure 10). The line that starts with the "greater than" (">") sign is the code entered by a user. Note that the ">" sign is not a part of the code. This symbol is used by the console to prompt the user to enter a line of code. Each line of code entered in the console (either directly or by using the "Run" button to execute several lines at a time) is marked by ">" also to visually separate R code from its output, which is marked with [1]. Sometimes a "+" character is used by the system to show that a particular line of code is a part of a bigger language construction (e.g. an "if" statement). Sometimes a system message is displayed before an output. System messages are in red fonts. A system message usually contains a warning (e.g. the user is using an outdated version of R base) that did not halt the code execution or an error message (e.g. the code could not be executed due to a typo in a variable name) that haled execution of the code.

Comments in R code are marked with a hash tag (#):

```
> #This is a comment
```

R language does not have reserved symbols for a block of comments (i.e. comments spanning multiple lines). One can "comment out" a section of code by highlighting the code and pressing Ctrl+Shif+C. Alternatively, one can use "Code -> Comment/Uncomment Lines" menu item (see Area 1, Label D).

A web scraping project, just like any data analysis project in R, usually starts with specifying an appropriate directory to which data and files will be stored and from which they will be retrieved by default. There are commands to get information about the current working directory and setting a working directory where the script and file output will be saved and read from. The code below saves the path of current working directory for RStudio into a variable called "wd" and then outputs the content of that variable:

```
> wd <- getwd()
> wd
[1] "C:/Users/Documents/MyR/Scraping/DemoProject"
```

The code below sets "Scraping" as a working directory and then tests the outcome of the operation with the getwd() function:

```
> wd <- "C:/Users/Documents/MyR/Scraping/"
> setwd(wd)
> testwd <- getwd()
> testwd
[1] "C:/Users/Documents/MyR/Scraping"
```

Note that a forward slash "/" is used in the working directory path and not a backslash "\". In Windows, a backslash is used in file or directory paths. The confusion between the two can be a source of errors.

Basic arithmetic operations can be performed in R by entering numerical expressions using the commonly used arithmetic operators. For example:

```
> 2 + 2
[1] 4
> 2 + 2 + 6
[1] 10
> 2*2
[1] 4
> 2*3*5
[1] 30
> 24/2
[1] 12
```

Unlike strongly-typed programming languages such as C++ or Java, R does not require variables to be declared with a particular data type prior to using the variables. For example, variables x and y below are not declared prior to the use. Yet, they can be used right away to store a number and a character string. The data type is automatically assigned to these variables based on the contents of these variables (data types are discussed later in this section).

```
> x <- 5
> y <- "Hello, R!"
> x
[1] 5
> y
[1] "Hello, R!"
```

It is recommended to use the "<-" or "arrow" operator to assign valued to variables. Although one can also use the equal sign "=" in R to assign values to variables, the arrow operator is preferred. Using the arrow operator can potentially eliminate confusion between the assignment operator and the comparison operator "==". For example:

```
> x <- 2
> y <- 2
> x == y
[1] TRUE
```

There are many data types that variables in R can store. The main types are numeric, character, date, and logical (Lander, 2014). In addition to that, variables in R can store R objects, such as functions, graphical plots, or outputs from a statistical analysis. For example, a variable can store output from fitting a linear regression model using the lm() function. One can check a variable's data type using the class() function. Sometimes, determining the data type of a variable is important in debugging and to avoid errors that can otherwise be hard to spot. For example:

```
> x <- 2
> class(x)
[1] "numeric"
> s <- "Some String"
> class(s)
[1] "character"
> t <- TRUE
> f <- FALSE
> class(t)
[1] "logical"
> class(f)
[1] "logical"
> d <- as.Date("1900-01-01")
> class(d)
[1] "Date"
```

To store a sequence of values, a vector can be used. Unlike a variable that can store a single value, a vector can store many values. A vector is similar to an array in other programming languages, such as Visual Basic. In R, a vector is a data structure for storing a sequence of elements of the same data type. For example, the vector called "numvector" consists of six numbers being joined together into one single vector using the "c()" (stands for concatenate) function:

```
> numvector <- c(2,3,4,3,2,19)
> numvector
```

```
[1]  2  3  4  3  2 19
```

Vectors can store not only numbers, but also characters or strings. For example, the vector called "charvector" consists of three words or strings being joined together:

```
> charvector <- c("Paris", "New York", "Tokyo")
> charvector
[1] "Paris"    "New York" "Tokyo"
```

R is a "vectorized language" (Lander, 2014). This means that operators are automatically applied to all elements of a vector; there is no need to explicitly loop through all the elements of a vector (e.g. using a "while" or "for" loop, which are explained in more detail later). In the example below, the "-2" arithmetic operator and the mean() function are applied to all numbers within the vector called "numvector":

```
> numvector <- c(2,3,4,3,2,19)
> numvector
[1]  2  3  4  3  2 19
> numvector - 2
[1]  0  1  2  1  0 17
> mean(numvector)
[1] 5.5
```

R also includes a number of more advanced data structures, such as data frames. Data frames are by far the most useful and most widely-used advanced data structure in R. Data frames are similar to tables in a spreadsheet. Just like a table, a data frame can consist of many columns and rows. Each column within a date frame can be viewed as a vector of a different data type. Thus, a data frame can consist of multiple columns devoted to different data types. This is quite similar to a table in a spreadsheet. Each column in a data frame can have its own name or label.

The code below creates a 3x3 data frame (meaning it has three rows and three columns; rows are listed before columns in R). The three columns in the data frame are "City", "Temperature", and "Condition". The three rows in the data frame contain weather data for three cities: Paris, New York, and Tokyo.

```
> x <- c("Paris", "New York", "Tokyo")
> y <- c(10, -2, 8)
> z <- c("Sunny","Partially Cloudy", "Cloudy")
> weather <- data.frame("City" = x, "Temperature" = y, "Condition" = z)
> weather
      City Temperature        Condition
1    Paris          10            Sunny
2 New York          -2 Partially Cloudy
3    Tokyo           8           Cloudy
```

Data frames are complex objects with many attributes. Many operators can be applied to data frames to retrieve these properties. For example, one can check number of rows and number of columns in a data frame:

```
> nrow(weather)
[1] 3
> ncol(weather)
[1] 3
```

Once can also access individual columns or vectors within a data frame using the "$" notation. For example:

```
> weather$Temperature
[1] 10 -2  8
> weather$Condition
[1] Sunny Partially Cloudy Cloudy
```

Individual values or "cells" within the data frame can also be accessed using the [row, column] notation:

```
> weather[1,3]
[1] Sunny
> weather[3,1]
[1] Tokyo
```

Another popular data structure in R is a list. Lists are used in R to hold arbitrary objects that are not necessarily of the same data type (Lander, 2014). List can store only numbers or only characters or a mix of the two data types. Lists can also contain data frames or a mix of data frames and numbers or characters. Lists can store other lists as well. The main reason lists are discussed here is that lists are often returned by various web scraping functions. These lists can contain various elements of an HTML or XML document, for example.

Lists are created with the help of the "list()" function. Each argument supplied to the list becomes and element of that list. For example, the following code creates a list called "mylist" storing three numbers and displays its content:

```
> mylist <- list(2,4,6)
> mylist
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
```

```
[1] 6
```

Alternatively, all three numbers can be stored as single vector within the list:

```
> mylist <- list(c(2,4,6))
> mylist
[[1]]
[1] 2 4 6
```

A list can store any combination of objects. For example, the list below stores the data frame devoted to weather data discussed earlier, a vector containing three characters, and a vector containing five numbers:

```
> x <- c("Paris", "New York", "Tokyo")
> y <- c(10, -2, 8)
> z <- c("Sunny","Partially Cloudy", "Cloudy")
> weather <- data.frame("City" = x, "Temperature" = y, "Condition" = z)
> mylist <- list(weather, c("a","b","c"), 1:5)
> mylist
[[1]]
      City Temperature        Condition
1    Paris          10            Sunny
2 New York          -2 Partially Cloudy
3    Tokyo           8           Cloudy

[[2]]
[1] "a" "b" "c"

[[3]]
[1] 1 2 3 4 5
```

Individual elements of the list above can be accessed using numbers in square brackets, for example, continuing the above example, one can access the vector of characters from the list called "mylist" as follows:

```
> mylist[2]
[[1]]
[1] "a" "b" "c"
```

In addition to data frames and lists, R supports other complex data structures, such as matrices and arrays. One should refer to the resources listed in Appendix A if one wishes to learn more about other data structures in R. The last part of the section examines such essential mechanisms in R as loops and conditions.

Just like most other programming languages, R includes loops and conditionals. The syntax of these elements in R is similar to that of some other languages. For example, examples of equivalent "for" and "while" loops in R are presented below:

```
> for (index in 1:3)
+ {
+     print(index)
+ }
[1] 1
[1] 2
[1] 3
> index <- 1
> while (index <= 3)
+ {
+     print(index)
+     index <- index + 1
+ }
[1] 1
[1] 2
[1] 3
```

In R one can also loop through numeric or character vectors. For example, this code loops through the vector "numvector" containing numbers:

```
> numvector <- c(1,2,3)
> for (index in numvector)
+ {
+     print(index)
+ }
[1] 1
[1] 2
[1] 3
```

The following code will loop through a vector containing words or names:

```
> charvector <- c("Julie", "Jane", "Janice")
> for (index in charvector)
+ {
+     print(index)
+ }
[1] "Julie"
[1] "Jane"
[1] "Janice"
```

To control the flow of code execution, just like many programming languages, R uses "if statements". A simple, single if statement has the following structure in R:

```
if (test_expression)
{
    statement
}
```

For example, the code below checks whether x is a positive number:

```
> x <- 1
> if (x > 0)
+ {
```

```
+     print("x is positive")
+ }
[1] "x is positive"
```

An “if…else” statement has the following syntax in R:

```
if (test_expression)
{
   statement1
}
else
{
   statement2
}
```

For example, the code below determines whether variable x contains a positive or negative number:

```
> x <- -5
> if(x > 0)
+ {
+     print("x is positive")
+ }
+ else
+ {
+     print("x is negative")
+ }
[1] "x is negative"
```

Once one becomes familiar with the basics of R syntax, one should explore the R packages available on CRAN (https://cran.r-project.org). CRAN is increasingly becoming a “Wikipedia” of data analysis tools. The currently available R packages amount to more than 12,000 and implement virtually all known forms of analysis of quantitative and textual data in various industries and academic fields (Lander, 2014).

An R package listed on CRAN can be installed directly from the repository using the following command:

```
       > install.packages("package_name")
```

For example, the following command should be used to install the package “rvest” that will be covered in next section:

```
> install.packages("rvest")
Installing package into ‘C:/Users/cob.vkrotov/Documents/R/win-library/3.4’
(as ‘lib’ is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.4/rvest_0.3.2.zip'
Content type 'application/zip' length 853613 bytes (833 KB)
downloaded 833 KB
package ‘rvest’ successfully unpacked and MD5 sums checked
```

After the package is installed, it needs to be loaded into the RStudio environment so that it can be accessed. Loading the package requires the following command:

```
> require(rvest)
Loading required package: rvest
Loading required package: xml2
Warning message:
package 'rvest' was built under R version 3.4.4
```

Note the command produces a warning message that the version of R used to build the package does not match the version of R that the user is currently running. This warning is typical and does not usually cause any problems. Once a user installs and loads the "rvest" package using the code discussed above, the package is ready to be used for web scraping tasks.

4.5 Rvest Package

This section contains an overview of the R package called "rvest". Some tables, examples, and related explanations in this section come from Krotov & Tennyson (2018) and Hadley Wickham (2016). There are many features that make "rvest" useful for accessing and parsing data from the web. First, "rvest" contains many functions that can be used for simulating sessions of a web browser. These features come in handy when one needs to browse through many webpages to download the needed data (this is the so-called "web crawling" process). Second, "rvest" contains numerous functions for accessing and parsing data from web documents in HTML, XML, CSS, and JSON formats.

Some of the most essential functions of the rvest package are listed in Table 2. Many other functions are available as a part of "rvest" package (Wickham, 2016). One should refer to the official "rvest" documentation to learn more about the package and its usage (see Wickham, 2016).

Table 2. Some Functions of the rvest Package (Reprinted from Wickham, 2016; Krotov & Tennyson, 2018)

Function Usage and Purpose	**Arguments**

Usage: read_html(x, ..., encoding = "") Purpose: This function reads HTML code of a web page from which data is to be retrieved. Can be used for reading XML as well.	• x: A url, a local path, or a string containing HTML code that needs to be read • …: Additional arguments can be passed to a URL using the GET() metod • encoding: specify encoding of the web page being read
Usage: html_nodes() Purpose: This function is used to select specific elements of a web document. To select these specific elements one can use CSS elements which contain the needed data or use XPath language to specify the "address" of an element of a web page	• x: A document, a node, or a set of nodes from which data is selected • css, xpath: a name of a CSS element or an XPath 1.0 link can be used to select a node
Usage: html_session(url, ...) Purpose: This function allows to start a web browsing session to browse HTML pages for the purpose of collecting data from them.	• url: address of a web page where browsing starts • …: Any additional httr config commands to use throughout session
Usage: html_table(x, header = NA, trim = TRUE, fill = FALSE, dec = ".") Purpose: This function can be used to read HTML tables into data frames (a commonly used data structure in R). Can be especially useful for reading HTML tables containing financial data.	• x: A node, node set or document • header: if NA, then the first row contains data and not column labels • trim: if TRUE, the function will remove leading and trailing whitespace within each cell • fill: If TRUE, automatically fill rows with fewer than the maximum number of columns with NAs • dec: The character used as decimal mark for numbers

4.6 Xml2 Package

In the past, the rvest package was also used to with XML documents using such functions as xml_node(), xml_attr(), xml_attrs(), xml_text() and xml_tag(). Eventually, these XML functions branched out into "xml2" package designed specifically to work with XML files (including with XML files retrieved from the web). The package has numerous functions for working with XML data. Some of the most commonly used functions of this package together with their commonly used arguments are listed in Table 3 below. Additional details about the functions listed in Table 3 (together with many other functions omitted here) are provided in Wickham et al. 2018.

Table 3. Some Functions of the xml2 Package (Reprinted from Wickham et al., 2018)

Function Usage and Purpose	**Arguments**
Usage: read_xml(x, encoding = "", ..., as_html = FALSE, options = "NOBLANKS") Purpose: This function reads XML and HTML files.	• x: a string, a connection, or a raw vector • encoding: specify a default encoding for the document • ...: additional arguments passed on to method • as_html: optionally parse an XML file as if it's HTML • options: set parsing options for the libxml2 parser

Usage: xml_children(x); xml_contents(x); xml_parents(x); xml_siblings(x) Purpose: These functions are used for navigating through an XML document structure. xml_children returns only elements, xml_contents returns all nodes, xml_parents returns all parents up to the root, xml_siblings returns all nodes at the same level,	• x: a document, node, or node set.
Usage: xml_find_all(x, xpath) Purpose: Finds all XML nodes that match a particular XPath expression	• x: a document, node, or node set. • xpath: a string containing an xpath (1.0) expression
Usage: xml_text(x, trim = FALSE); xml_set_text(x, value) Purpose: Used to extract or replace text in an XML document, node, or node set.	• x: a document, node, or node set • trim: If TRUE will trim leading and trailing spaces • value: character vector with replacement text

4.7 Simple Web Scraping Examples

The four examples provide below show how to use the "xml2" and "rvest" packages for accessing data in XML, HTML, and CSS format.

The first example illustrates how "xml2" package is used to extract financial data from an online XML document. The document can be found here:

https://www.sec.gov/Archives/edgar/data/1067983/000119312518238892/brka-20180630.xml

The XML document is a 10-Q statement for Berkshire Hathaway, Inc. posted on EDGAR (an open web database of financial statements of publically listed companies). Technically, the document is an XBRL (eXtensible Business Reporting Language) instance file. XBRL is an extension of XML language used specifically for interexchange of financial reporting data over the web. Since XBRL is based on XML, one can use the common functions of "xml2" package to work with this data. More specifically, the example below extracts the Net Income data reported by Berkshire Hathaway and saves this data into a data frame named Net_Income_Data.

```
#This example requires xml2 R package
require(xml2)

#Parse XML from an online document found at the URL specified below
URL <- "https://www.sec.gov/Archives/edgar/data/1067983/000119312518238892/brka-
20180630.xml"
XML_data <- read_xml(URL)

#Save all nodes related to Net Income or Loss as defined by US GAAP
Nodes <- xml_find_all(XML_data, ".//us-gaap:NetIncomeLoss")

#Convert the node structure into a character vector
Nodes_Vector <- as.character(Nodes)
```

```
#Retreive values for all Net Income or Loss elements and save them into a vetcor
#These are dollar values for each Net Income or Loss item reported in the 10Q
document
Nodes_Values <- xml_text(xml_find_all(XML_data, ".//us-gaap:NetIncomeLoss"))

#Bind the vectors together as columns and convert the structure into a data frame
#The data frame contains two columns and four rows
Net_Income_Data <- data.frame(cbind(Nodes_Vector,Nodes_Values))
```

The resulting data frame (Net_Income_Data) contains various XBRL attributes of each node containing Net Income or Loss data together with the dollar value of Net Income or Loss (in dollars) reported for each of these items. The definitions of each of the four Net Income or Loss items can be explored further by analyzing the values of attributes for each of the nodes saved in the data frame.

The next example illsustrates how the "rvest" package can be used for accessing data from an HTML page available on the web:

https://www.sec.gov/Archives/edgar/data/1067983/000119312516760194/d268144d10q.htm

This HTML page also contains a 10-Q statement posted by Berkshire Hathway on EDGAR, albeit from a different period. This time, the goal is to retrieve Balance Sheet data from the statement. This data is available in the form of an HTML table. The HTML table is a part of the HTML web page containing the entire 10-Q statement by Berkshire Hathaway. The table data is retrieved and saved into a data frame called "balance_sheet_data" using the R code below.

```
#This script requires the rvest package to be installed and activated
require(rvest)

#The variable url contains a URL to the 10-Q document published via EDGAR
url <-
"https://www.sec.gov/Archives/edgar/data/1067983/000119312516760194/d268144d10q.htm"

#read_html() function from rvest package used to read HTML code from page
tenqreport_html <- read_html(url)

#xpath used to retreive HTML code specifically for balance sheet table
balance_sheet_html <- html_nodes(tenqreport_html,
xpath='/html/body/document/type/sequence/filename/description/text/table[7]')

#HTML code from the balance sheet table is obtained
balance_sheet_list <- html_table(balance_sheet_html)

#Balance sheet data is saved from a list into a data frame
balance_sheet_table <- balance_sheet_list[[1]]
```

Once balance sheet data is saved into a data frame (named "balance_sheet_table" in this example), individual values from the balance sheet can be accessed and used in calculations. For example, certain accounting ratios can be calculated. Alternatively, one can match the balance sheet with other data related to the company and available on the web. The rvest package can be used in a similar fashion to obtain quantitative and qualitative data from the same HTML document or other sources on the web.

The code below illustrates how to access top reviews for the iPadPro product listed on Amazon.com. This time, a CSS selector is used to retrieve top reviews for this particular product listed on Amazon. The SelectorGadget tool (discussed earlier) was used to find an XPath for the CSS element containing the top reviews. The resulting variable review_text is a character vector containing 6 reviews. Note that only a portion of one review was displayed to save space.

```
#Require the rvest package
require(rvest)

#Read HTML code for Apple iPad Pro
ipad_page <-
read_html("https://www.amazon.com/gp/product/B01CGXU0GM/ref=s9_dcacsd_dcoop_bw_c_x_17
_w")

#Access top reviews for the product and save them in review_text
review <- html_nodes(ipad_page, xpath='//*[contains(concat( " ", @class, " " ),
concat( " ", "a-expander-partial-collapse-content", " " ))]')
review_text <- html_text(review)

#Display top customer reviews
review_text

## [6] "To fully understand my review, must explain a few things. I come to owning
the iPad Pro 9.7\" via a long lineage of Macs, iPods, iPhones and iPads.
```

The final example is supplied with the "rvest" package (Wickham, 2016). It involves retrieving rating and cast data for "The Lego Movie" found on IMDb website from corresponding HTML or CSS elements. Again, the SelectorGadget can be used to find XPath link associated with each of these elements – this should save time and eliminate the need to be knowledgeable in the particularities of XPath syntax. The "%>%" string in the example below is the so called "pipe" operator used to pass results from one function to another function. Thus, the operator simplified the code by creating a “pipeline” that performs work on data using various functions. The contents of the “rating” and “cast” variables are displayed.

```
 library(rvest)
lego_movie <- read_html("http://www.imdb.com/title/tt1490017/")

rating <- lego_movie %>%
  html_nodes("strong span") %>%
  html_text() %>%
  as.numeric()
rating
#> [1] 7.8

cast <- lego_movie %>%
  html_nodes("#titleCast .itemprop span") %>%
  html_text()
cast
#>  [1] "Will Arnett"     "Elizabeth Banks" "Craig Berry"
#>  [4] "Alison Brie"     "David Burrows"   "Anthony Daniels"
#>  [7] "Charlie Day"     "Amanda Farinos"  "Keith Ferguson"
#> [10] "Will Ferrell"    "Will Forte"      "Dave Franco"
#> [13] "Morgan Freeman"  "Todd Hansen"     "Jonah Hill"
```

As one can see from the examples provided this section, “rvest” is a robust package that can be fine-tuned to automatically scrape data for virtually any Information Systems research project requiring web data. The next section contains a more elaborate example of a web scraping project relying on the “rvest” package.

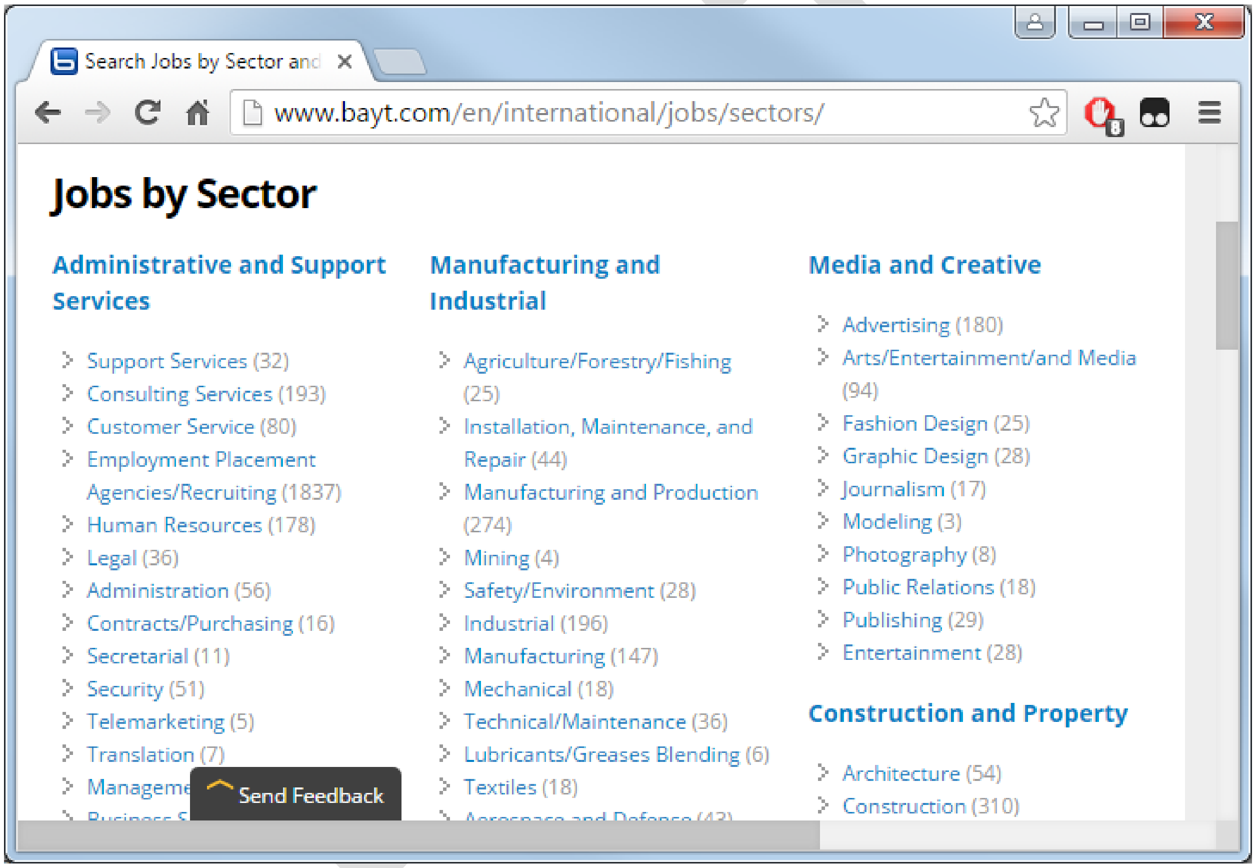
5. A Complex Web Scraping Example

This section contains an example of a web scraping project involving Bayt.com, a leading employment website in the Middle East. At any point in time, the website contains thousands of employment ads from various industries and for a diverse set of roles. The data collected from the website can be used to answer a number of interesting questions about IT competencies together with competencies in other industries and roles. The example is structured in accordance with the three phases of web scraping discussed previously: Website Analysis, Website Crawling, and Data Organization.

Apart from being larger and more complex, there are three features of this more elaborate example that make it different from the examples provided earlier. First, the dataset retrieved is fairly large. Second, retrieving the dataset involves “crawling” the website one page at a time. This is in contrast to the previous examples where data is accessed from a single web page. Third, the data is saved into a file on the computer after the web scraping task is completed.

5.1 Website Analysis

The web scraping project aimed at retrieving data from Bayt.com starts with the examination of the underlying structure of the website. The structure of the Bayt.com site is fairly typical for a job posting site. One way to browse job postings is by sector (http://www.bayt.com/en/international/jobs/sectors/). When viewing the "Jobs by Sector" page, a list of sectors (such as "Technology and Telecom") is displayed, each as a hyperlink (see Figure 11).


Search Jobs by Sector and
www.bayt.com/en/international/jobs/sectors/
Send Feedback

Jobs by Sector

Administrative and Support Services
- Support Services (32)
- Consulting Services (193)
- Customer Service (80)
- Employment Placement Agencies/Recruiting (1837)
- Human Resources (178)
- Legal (36)
- Administration (56)
- Contracts/Purchasing (16)
- Secretarial (11)
- Security (51)
- Telemarketing (5)
- Translation (7)
- Manageme

Manufacturing and Industrial
- Agriculture/Forestry/Fishing (25)
- Installation, Maintenance, and Repair (44)
- Manufacturing and Production (274)
- Mining (4)
- Safety/Environment (28)
- Industrial (196)
- Manufacturing (147)
- Mechanical (18)
- Technical/Maintenance (36)
- Lubricants/Greases Blending (6)
- Textiles (18)

Media and Creative
- Advertising (180)
- Arts/Entertainment/and Media (94)
- Fashion Design (25)
- Graphic Design (28)
- Journalism (17)
- Modeling (3)
- Photography (8)
- Public Relations (18)
- Publishing (29)
- Entertainment (28)

Construction and Property
- Architecture (54)
- Construction (310)

Figure 11. The Bayt.com website when viewing “Jobs by Sector”

Each job sector, therefore, can be accessed using a unique URL. The "Technology and Telecom" and "Banking and Finance" sectors, for example, can be accessed using the following URLs:

https://www.bayt.com/en/international/jobs/sectors/technology-telecom/
https://www.bayt.com/en/international/jobs/sectors/banking-finance/

Note that each URL shares the same base URL, and each is distinguished only by its respective subfolder. This observation will be useful when scraping the website sector by sector later during the Web Crawling phase.

When a particular sector's webpage is viewed, links to the individual job postings are listed, but only in "pages" of twenty at a time. Each page of jobs can be accessed through its own URL, as follows:

https://www.bayt.com/en/international/jobs/sectors/banking-finance/?page=1
https://www.bayt.com/en/international/jobs/sectors/banking-finance/?page=2
https://www.bayt.com/en/international/jobs/sectors/banking-finance/?page=3

On each of these pages, its twenty unique job postings are represented in HTML as a series of "div" elements. Each "div" element contains a hyperlink with the following structure:

```
<a data-js-aid="jobID" href="...">Job Title</a>
```

So, for each job posting, a hyperlink exists that contains all of the information we are scraping. Notice that the element is identified with a distinctive attribute (data-js-aid) that has "jobID" as the value. Therefore, we will need to collect all of the <a> elements that have that distinctive attribute/value pair. Then we will extract the content of the <a> element to get the title, and we will extract the value of the "href" attribute to get the URL.

Also, on each of these pages, there is a <link> element containing the URL to the next page of jobs. The <link> element has the following structure:

```
<link rel="next" href="..." />
```

This will be the element used to determine the URL of the next page of job listings. We will look for a <link> element with a "rel" attribute whose value is "next". Once that element is found, we will use the value of the "href" attribute to determine the URL of the next page of jobs. We will know that we've reached the end of the job listings for the current sector if such an element doesn't exist, at which point we will move on to the next sector of jobs and continue from there.

By thusly evaluating the structure of the website, identifying how the website can be methodically (and programmatically) accessed, identifying which HTML elements contain the desired information, and inspecting the HTML to determine how those elements can be identified, the "Website Analysis" phase is completed.

5.2 Website Crawling

Once the elements that contain the target data have been identified and the respective HTML has been inspected (during the "Website Analysis" phase as discussed in the previous subsection), the data must actually be extracted. In this subsection, we will walk through the script shown in Appendix B step-by-step to demonstrate how the R language can be used to scrape data from the Bayt.com website.

In overview, the script will operate as follows:

1. The necessary library packages will be imported.
2. The working directory will be set.
3. Important input and output variables will be initialized
4. A loop will be set up to iterate through each job sector. At each iteration, one sector will be processed by visiting that sector's webpage.
5. Another loop will be set up to iterate through each page of jobs for that sector. At each iteration, one page of jobs will be processed. A list of the jobs accessible via that page will be collected.
6. Yet another loop will be set up to iterate through that list of jobs. At each iteration, one job posting will be processed. Each job's title and URL title will be extracted. The information will be saved.
7. Finally, once all of the jobs, pages, and sectors have been processed, a file containing all of the collected data will be written.

The remainder of this section will examine and discuss the corresponding lines of the script in detail.

5.2.1 Lines 1-4

```
#Required packages
require(tm)
require(rvest)
require(XML)
```

These lines are used to load packages into the current session, so that their functions can be called. As it was explained earlier, the purpose of the rvest package is "to make it easy to download, then manipulate, both html and xml" documents (Wickham, 2016). The XML package contains functions for parsing XML documents, and is required by the rvest package. Various functions that are contained in these packages are used throughout the scripts, and will be discussed within the context of those portions of the script in which they are used.

5.2.2 Lines 6-19

```
#Set working directory
setwd("C:/Users/.../Research/WebScraping")

#Identify job sectors from which to scrape data
sectors <- c("banking-finance",
             "technology-telecom",
             "media-creative")

#Create frame in which data results will be stored
job_data <- data.frame(Sector=character(0),
                       Title=character(0),
                       URL=character(0))

colnames(job_data) <- c("Sector", "Title", "URL")
```

These lines are used to set up the working environment for the script. The working directory specifies the local directory from which any input files will be read and to which any output files will be written. The directory should be set as desired. The "sectors" vector identifies which job sectors will be queried for job postings. Recall from the "Website Analysis" section that each sector has its own URL such that each URL shares the same base address but is distinguished only by its respective subfolder. The values in this vector represent those subfolders, and so they must match the URL subfolder exactly. The "job_data" frame will be used to store all of the extracted information. Each row of the frame will represent exactly one job posting, while the columns of the frame correspond to the bits of information being collected (i.e., "Sector, "Title", and "URL").

5.2.3 Lines 21-25

```
#For each job sector (each job sector will be processed one at a time)
for (sector in 1:length(sectors))
{
  base_url <- "https://www.bayt.com/en/uae/jobs/sectors/"
  page_url <- paste(base_url, sectors[sector], sep="")
```

These lines are used to loop through each job sector, one at a time. The first line sets up the loop to iterate exactly as many times as there are sectors. The "base_url" variable represents the base URL for all of the job sector websites. That base URL is then concatenated with the name of the current job sector to create the "page_url" variable.

5.2.4 Lines 27-32

```
  #For each page of jobs within the sector
  repeat
  {
    #Read website for the current sector page
    page_html <- read_html(html_session(page_url))
    cat(paste(page_url, "\n")) #display script progress
```

These lines are used to loop through each page of jobs within the current sector. The "html_session" function (from the rvest package) is used to send an HTTP request for the specified webpage and simulate the functionality of a browser. The "read_html" function is used to get the raw HTML code for the webpage associated with that session. The "cat" function is used to simply display output to the console as the script is executing to indicate to the user which page is currently being processed.

5.2.5 Lines 34-36

```
    #Retreive the list of job links
    a_elements <- html_nodes(page_html, "a[data-js-aid=\"jobID\"]")
    cat(paste(length(a_elements), " jobs found\n", sep="")) #display progress
```

These lines are used to collect all of the <a> elements that contain the desired information (job title and URL). Recall from the "Website Analysis" section that each page contains a list of job postings. For each job posting in the list, an <a> element exists, which contains a hyperlink to the respective job posting. Each of these <a> elements is identified with a distinctive "data-js-aid" attribute having "jobID" as the value. The "html_nodes" function is used to retrieve a list of these relevant <a> elements.

5.2.6 Lines 38-44

```
    #For each job within the current page (process one job at a time)
    for (i in 1:length(a_elements))
    {
      job_href <- xml_attr(a_elements[i], "href")
      job_url <- paste("https://www.bayt.com", job_href, sep="")
      job_title <- html_text(a_elements[i], trim=TRUE)
      cat(paste(" ", i, ". ", job_title, "\n", sep="")) #display progress
```

For each of the <a> elements that were collected in the previous step, we must extract the job title and the URL for the respective job posting. The value of the "href" attribute tells us the URL of the job posting, so we use the "xml_attr" function to extract it.

The "href" attribute contains only a relative path, so the Bayt.com domain is prepended to create a complete URL and stored in the "job_url" variable. The content of the <a> element contains the job title, so the "html_text" function is used to extract it, which is stored inside the "job_title" variable. The "cat" function is used to display the progress of the script as it executes by printing the title of the job that was just extracted.

5.2.7 Lines 46-53

```
      #Consolidate all the information about this job into a single dataframe
      job_info <- data.frame("Sector"=sectors[sector],
                             "Title"=job_title,
                             "URL"=job_url)

      #Write the current job info to the frame where all data is stored
      job_data <- rbind(job_data, job_info)
    }
```

At this point in the script, all of the data for the current job posting will have been retrieved. These lines from the script are used to simply write that data to a single-row data frame, and then append that row to the "job_data" frame, where the aggregation of all the data from all the job postings will be stored. The closing brace "}" is used to end the loop that is being used to iterate through each job.

5.2.8 Lines 55-59

```
    #Get the URL for the next page
    next_html <- html_node(page_html, "link[rel=\"next\"]")

    if(length(next_html)==0)
      break
```

Recall from the "Website Analysis" section that there is a <link> element that contains the URL for the page of jobs. It is distinguished by having a "rel" attribute with the value "next", so the "html_node" function is used to extract that element from the HTML, which is stored in the "next_html" variable. If the element is not found, then the variable will have a length of zero, in which case we will break out of the loop. This will end the execution of the loop that is iterating through each page within the sector, so we will then move on to the next sector.

5.2.9 Lines 61-62

```
    next_xml <- xmlParse(next_html, asText=TRUE)
    page_url <- xmlAttrs(xmlRoot(next_xml))["href"]
```

These lines of code will only be reached if we did not break out of the loop during the execution of the previous lines of code, which means that there are still more pages of jobs to process. So, we use the "xmlParse" function to read the HTML as XML, and use the "xmlAttrs" function to extract the value of the "href" attribute, which will contain the URL of the next page of job listings. We update the "page_url" variable with that new URL, and we are then ready for the next iteration of the loop, so that we can process the next page of job listings.

5.3 Data Organization

In the "Data Organization" phase of the web scraping process, the data retrieved during the previous phase is organized into a useful format, such as a spreadsheet. In this case, our data is conveniently collected into a data frame (called "job_data" in the script) that has the exact same structure as a columnar table. That data is written to a comma-separated CSV file in the very last line of the script, which is shown below:

```
#Finally, write the collected data to an output file
write.table(job_data, file="output_data.csv", sep=",", col.names=TRUE,
row.names=FALSE)
```

The CSV file can be opened in a spreadsheet program such as Excel. In this example, no further manipulation of the data is necessary. Our data is clean and tidy. However, in general, if additional manipulation of the data is necessary, it would be performed at this stage. Conditional formatting could be added to the spreadsheet to highlight minimum or maximum values; pivot tables could be added to see different views of the data; and so on.

6. Implications for Researchers

We believe that web scraping using R offers a number of advantages to Information Systems researchers. First, when dealing with Big Data, even simple manipulations with quantitative data or text can be quite tedious and prone to errors if done manually. The R environment can be used to automate a number of simple and also complex data collection and transformation processes and techniques based on complex data transformation heuristics (Krotov & Tennyson, 2018). Second, using R for web scraping and subsequent analysis ensures reproducibility of research (Peng, 2011). Any manual manipulation of data may involve subjective choices and interpretations in relation to what data is retrieved and how it is formatted, pre-processed, and saved. Oftentimes, even simple research involving basic data analysis cannot be replicated by other researchers due to the errors committed at various stages of data gathering and analysis (e.g. see The Economist, 2016). With R, all aspects of data retrieval and manipulation can be unambiguously described and then reproduced by other researchers by running the script used for data collection. Finally, once web data is retrieved using R, it can subjected to virtually all known forms of analysis implemented via more than 12,000 user-generated packages available from CRAN.

7. Conclusion

The World Wide Web is a vast repository of data. Many research questions can be addressed by retrieving and analyzing web data. Unfortunately, web data is often unstructured or semi-structured, based on somewhat loose standards, and generated or updated in real time. Retrieving such data for further analysis requires a programmatic approach. Developing web scraping scripts that automate data collection from the web, regardless of which programming language is used for that, requires a good understanding of web architecture and some of the key web technologies, such as HTML, CSS, and XML. As demonstrated in this tutorial, the R environment is an effective platform for creating and modifying automated tools for retrieving a wide variety of data from the web. We believe that the approach to web scraping outlined in this tutorial is general enough to serve as a good starting point for any IS research project involving web data.

References

Basoglu, K. A., & White, Jr., C. E. (2015). Inline XBRL versus XBRL for SEC reporting. Journal of Emerging Technologies in Accounting, 12(1), 189-199.

Cisco Systems. (2017). Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. Retrieved from: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf

Comprehensive R Archive Network (CRAN). (2018). Retrieved from: https://cran.r-project.org/

Dynamic Web Solutions. (2017). A Conceptual Explanation of the World Wide Web. Retrieved from: http://www.dynamicwebs.com.au/tutorials/explain-web.htm

Goes, P. (2014). Editor's comments: Big data and IS research, MIS Quarterly 38(3), 3-8.

Krotov, V., and Tennyson, M. F. (2018). Scraping financial data from the web using R language. Journal of Emerging Technologies in Accounting.

Krotov, V. and Silva, L. (2018). Legality and ethics of web scraping. The Twenty Fourth Americas Conference on Information Systems.

Lander, J. P. (2014). R for Everyone: Advanced Analytics and Graphics. Boston, MA: Addison-Wesley.

Peng, R. D. (2011). Reproducible research in computational science. Science, 334(6060). 1226-1227.

Schutt, R., & O'Neil, C. (2013). Doing Data Science: Straight Talk from the Frontline. Sebastopol, CA: O'Reilly Media, Inc.

The Economist (2016). Excel errors and science papers. Retrieved from https://www.economist.com/graphic-detail/2016/09/07/excel-errors-and-science-papers

Tutorials Point. (2017). HTTP Tutorial. Retrieved from: http://www.tutorialspoint.com/http/

Wickham, H. (2014a). Advanced R. Boca Raton, FL: CRC Press.

Wickham, H. (2014b). Tidy data. Journal of Statistical Software, 59(10), 1-23.

Wickham, H. (2016). Package 'rvest'. Retrieved from: https://cran.r-project.org/web/packages/rvest/rvest.pdf

Wickham, H., Hester, J. and Ooms, J. (2018). Package 'xml2'. Retrieved from https://cran.r-project.org/web/packages/xml2/xml2.pdf

W3Schools. (2017). Retrieved from: http://www.w3schools.com/

World Wide Web Consortium (W3C). (2017). Retrieved from: https://www.w3.org/

Appendix A: Additional Resources

Web Architecture

The following online tutorials are recommended for those who want to learn more about web architecture:

- Dynamic Web Solutions. 2017. A Conceptual Explanation of the World Wide Web. Available at: http://www.dynamicwebs.com.au/tutorials/explain-web.htm
- Tutorials Point. 2017. HTTP Tutorial. Available at: http://www.tutorialspoint.com/http/

The World Wide Web Consortium (W3C) is the ultimate source on the technologies and specifications related to the web:

- World Wide Web Consortium (W3C). 2017. Available at: https://www.w3.org/

HTML

A free online tutorial from w3schools.com on HTML:

- http://www.w3schools.com/html/default.asp

CSS

A free online tutorial from w3schools.com on CSS:

- http://www.w3schools.com/css/default.asp

XML

A free online tutorial from w3schools.com on XML and a number of related technologies:

- http://www.w3schools.com/xml/xml_exam.asp

In addition to providing a rather thorough treatment of XML, the tutorial also has sections devoted to related technologies, such as XML Namespaces, XML Schema, XPath and XLink

R and RStudio

We recommend the following book to people with basic understanding of computer programming but no previous knowledge of R:

- Lander, J. P. 2014. R for Everyone: Advanced Analytics and Graphics. Boston, MA: Addison-Wesley.

The book contains an excellent introduction into various aspects of R language and contains a manual on installing and using RStudio. Much of the examples found in this note are based on this book.

For those already familiar with R, the following advanced text on R can be recommended:

- Wickham, H. 2014. Advanced R. Boca Raton, FL: CRC Press.

Alternatively, one can access various articles and tutorials on R online:

- https://www.r-bloggers.com/how-to-learn-r-2/

For those wishing to get a practical introduction to data science in R and learn various related technologies and statistical techniques, the following online specialization from John Hopkins University is available in Coursera:

- https://www.coursera.org/specializations/jhu-data-science

Appendix B: Example Web Scraping Script

```
#Required packages
require(tm)
require(rvest)
require(XML)

#Set working directory
setwd("C:/Users/mtennyson/Documents/Research/WebScraping")

#Identify job sectors from which to scrape data
sectors <- c("banking-finance",
             "technology-telecom",
             "media-creative")

#Create frame in which data results will be stored
job_data <- data.frame(Sector=character(0),
                       Title=character(0),
                       URL=character(0))

colnames(job_data) <- c("Sector", "Title", "URL")

#For each job sector (each job sector will be processed one at a time)
for (sector in 1:length(sectors))
{
  base_url <- "https://www.bayt.com/en/uae/jobs/sectors/"
  page_url <- paste(base_url, sectors[sector], sep="")

  #For each page of jobs within the sector
  repeat
  {
    #Read website for the current sector page
    page_html <- read_html(html_session(page_url))
    cat(paste(page_url, "\n")) #display script progress

    #Retreive the list of job links
    a_elements <- html_nodes(page_html, "a[data-js-aid=\"jobID\"]")
    cat(paste(length(a_elements), " jobs found\n", sep="")) #display progress

    #For each job within the current page (process one job at a time)
    for (i in 1:length(a_elements))
    {
      job_href <- xml_attr(a_elements[i], "href")
      job_url <- paste("https://www.bayt.com", job_href, sep="")
      job_title <- html_text(a_elements[i], trim=TRUE)
      cat(paste(" ", i, ". ", job_title, "\n", sep="")) #display progress

      #Consolidate all the information about this job into single frame
      job_info <- data.frame("Sector"=sectors[sector],
                             "Title"=job_title,
                             "URL"=job_url)

      #Write the current job info to the frame where all data is stored
      job_data <- rbind(job_data, job_info)
    }

    #Get the URL for the next page
    next_html <- html_node(page_html, "link[rel=\"next\"]")

    if(length(next_html)==0)
      break

    next_xml <- xmlParse(next_html, asText=TRUE)
    page_url <- xmlAttrs(xmlRoot(next_xml))["href"]
  }
}

#Finally, write the collected data to an output file
write.table(job_data, file="output_data.csv", sep=",", col.names=TRUE,
row.names=FALSE)
```