Interfacing R with Web Technologies for Data Acquistion and Interactive Visualization

by

Carson Sievert

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Statistics

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The following describes a collection of software interfaces for data acquisiton and visualization. All of these interfaces are freely available as extension packages to the R language and leverage web technologies to achieve accessible, portable, and reproducible workflows. The majority of this work (LDAvis, animint, and plotly) focuses on interactive visualization. These interfaces fall roughly into two categories: (1) domain-specific (LDAvis) and (2) general purpose tools for interactive data visualization (animint and plotly). More specifially, the LDAvis package produces an interactive visualization to aid interpretation of Latent Dirichlet Allocation (LDA) model output. The animint and plotly packages are more general, and build upon principles from the grammar of graphics, but extend those principles in slightly different ways to enable interactivity, such as animation and brushing a scatterplot matrix.

1 Literature Review

1.1 What makes a good software interface?

Unwin and Hofmann (?) discuss the strengths, weaknesses, and differences between using graphical and command-line interfaces for data analysis. Graphical user interfaces (GUIs) can be much more intuitive to use, but at the cost of being less flexible, precise, and repeatable. Unwin and Hofmann argue statistical software should strive to achieve a synergy of two that leverages both of their strengths. That is, a command-line interface when we can precisely describe what we want and a graphical interface for "searching for information and interesting structures without fully specified questions."

Unwin and Hofmann further discuss the different audiences these interfaces attract. Command-line interfaces typically attract "power users" such as applied statisticians and statistical researchers in a university, whereas more casual users of statistical software typically prefer a GUI. In later sections, we discuss GUIs in greater detail within the context of interactive statistical graphics. For now, we briefly discuss some best practices for designing a command-line interface for statistical computing in R.

Before authoring an interface, one should establish the target audience, the class of problems it should address, and loosely define how the interface should actually work. During this process, it may also be helpful to identify your audience as being primarily composed of *software developers* or *data analysts*. Developers are typically more interested in using the interface to develop novel software or incorporating the functionality into a larger scientific computing environment (?). In this case, interactive exploration and troubleshooting is not always a luxury, so robust functionality is of utmost importance. On the other hand, analysts interfaces should work well in an interactive environment since this caters to rapid prototyping of ideas and troubleshooting of errors.

Good developer interfaces often make it easier to implement good analyst interfaces. A great recent example of a good developer interface is the R package **Rcpp**, which provides a seamless interface between R with C++ (**?**). To date, more than 500 R packages use **Rcpp** to make interfaces that are both expressive and efficient, including the highly influential analyst interfaces such as **tidyr** and **dplyr** (**?**); (**?**). These interfaces help analysts focus on the primary task of wrangling data into a form suitable for visualization and statistical modeling, rather than focusing on the implementation details behind how the transformations are performed. (**?**) argues that these interfaces "May have more impact on today's practice of data analysis than many highly-regarded theoretical statistics papers".

Evaluating statistical computing interfaces is certainly a subjective matter since we all have different tastes, different backgrounds, and have different needs. It seems reasonable to evaluate an interface based on its effectiveness and efficiency in aiding a user complete their task, but as (**?**) points out, "There is a tendency to judge software by the most powerful tools they provide (whether with a good interface or not)". As a result, all too often, analysts must spend time gaining the skills of a software developer. Good analyst interfaces often abstract functionality from developer interfaces in a way that allow analysts to focus on their primary task of acquiring/analyzing/modeling/visualizing data, rather than the implementation details. The following focuses on such work with respect to acquiring data from the web and interactive statistical web graphics.

1.2 Acquiring and wrangling web content in R

1.2.1 Interfaces for working with web content

R has a rich history of interfacing with web technologies for accomplishing a variety of tasks such as requesting, manipulating, and creating web content. As an important first step, extending ideas from (**?**), Brian Ripley implemented the connections interface for file-oriented input/output in R (**?**). This interface supports a variety of common transfer protocols (HTTP, HTTPS, FTP), providing access to most files on the web that can be identified with a Uniform Resource Locator (URL). Connection objects are actually external pointers, meaning that, instead of immediately reading the file, they just point to the file, and make no assumptions about the actual contents of the file.

Many functions in the base R distribution for reading data (e.g., `scan`, `read.table`, `read.csv`, etc.) are built on top of connections, and provide additional functionality for parsing well-structured plain-text into basic R data structures (vector, list, data frame, etc.). However, the base R distribution does not provide functionality for parsing common file formats found on the web (e.g., HTML, XML, JSON). In addition, the standard R connection interface provides no support for communicating with web servers beyond a simple HTTP GET request (**?**).

The **RCurl**, **XML**, and **RJSONIO** packages were major contributions that drastically improved our ability to request, manipulate, and create web content from R (**?**). The **RCurl** package provides a suite of high and low level bindings to the C library libcurl, making it possible to transfer files over more network protocols, communicate with web servers (e.g., submit forms, upload files, etc.), process their responses, and handle other details such as redirects and authentication (**?**). The **XML** package provides low-level bindings to the C library libxml2, making it possible to download, parse, manipulate, and create XML (and HTML) (**?**). To make this possible, **XML** also provides some data structures for representing XML in R. The **RJSONIO** package provides a mapping between R objects and JavaScript Object Notation (JSON) (**?**). These packages were heavily used for years, but several newer interfaces have made these tasks easier and more efficient.

The **curl**, **httr**, and **jsonlite** packages are more modern R interfaces for requesting content on the web and interacting with web servers. The **curl** package provides a much simpler interface to libcurl that also supports streaming data (useful for transferring large data), and generally has better performance than **RCurl** (**?**). The **httr** package builds on **curl** and organizes it's functionality around HTTP verbs (GET, POST, etc.) (**?**). Since most web application programming interfaces (APIs) organize their functionality around these same verbs, it is often very easy to write R bindings to web services with **httr**. The **httr** package also builds on **jsonlite** since it provides consistent mappings between R/JSON and most most modern web APIs accept and send messages in JSON format (**?**). These packages have already had a profound impact on the investment required to interface R with web services, which are useful for many things beyond data acquisition. For example, it is now easy to install R packages hosted on the web (**devtools**), perform cloud computing (**analogsea**), and archive/share computational outputs (**dvn**, **rfigshare**, **RAmazonS3**, **googlesheets**, **rdrop2**, etc.).

The **rvest** package builds on **httr** and makes it easy to manipulate content in HTML/XML files (**?**). Using **rvest** in combination with SelectorGadget, it is often possible to extract structured information (e.g., tables, lists, links, etc) from HTML with almost no knowledge/familiarity with web technologies. The **XML2R** package has a similar goal of providing an interface to acquire and manipulate XML content into tabular R data structures without any working knowledge of XML/XSLT/XPath (**?**). As a result, these interfaces reduce the startup costs required for analysts to acquire data from the web.

Packages such as **XML**, **XML2R**, and **rvest** can download and parse the source of web pages, which is *static*, but extracting *dynamic* web content requires additional tools. The R package **rdom** fills this void and makes it easy to render and access the Document Object Model (DOM) using the headless browsing engine phantomjs (**?**). The R package **RSelenium** can also render dynamic web pages and simulate user actions, but its broad scope and heavy software requirements make it harder to use and less reliable compared to **rdom** (**?**). **rdom** is also designed to work seamlessly with **rvest**, so that one may use the `rdom()` function instead of `read_html()` to render, parse, and return the DOM as HTML (instead of just the HTML page source).

Any combination of these R packages may be useful in acquiring data for personal use and/or providing a higher-level interface to specific data source(s) to increase their accessibility. The next section focuses on such interfaces.

1.2.2 Interfaces for acquiring data on the web

The web provides access to the world's largest repository of publicly available information and data. This provides a nice *potential* resource both teaching and practicing applied statistics, but to be practical useful, it often requires a custom interface to make data more accessible. If publishers follow best practices, a custom interface to the data source usually is not needed, but this is rarely the case. Many times structured data is embedded within larger unstructured documents, making it difficult to incorporate into a data analysis workflow. This is especially true of data used to inform downstream web applications, typically in XML and/or JSON format. There are two main ways to make such data more accessible: (1) package, document, and distribute the data itself (2) provide functionality to acquire the data.

If the data source is fairly small, somewhat static, and freely available with an open license, then we can directly provide data via R packaging mechanism. In this case, it is best practice for package authors include scripts used to acquire, transform, and clean the data. This model is especially nice for both teaching and providing examples, since users can easily access data by installing the R package. (**?**) provides a nice section outlining the details of bundling data with R packages.[1]

R packages that just provide functionality to acquire data can be more desirable than bundling it for several reasons. In some cases, it helps avoid legal issues with rehosting copyrighted data. Furthermore, the source code of R packages can always be inspected, so users can verify the cleaning and transformations performed on the data to ensure its integrity, and suggest changes if necessary. They are also versioned, which makes the data acquisition, and thus any downstream analysis, more reproducible and transparent. It is also possible to handle dynamic data with such interfaces, meaning that new data can be acquired without any change to the underlying source code. As explained in Taming PITCHf/x Data with XML2R and pitchRx, this is an important quality of the **pitchRx** R package since new PITCHf/x data is made available on a daily basis.

Perhaps the largest centralized effort in this domain is lead by rOpenSci, a community of R developers that, at the time of writing, maintains more than 50 packages providing access to scientific data ranging from bird sightings, species occurrence, and even text/metadata from academic publications. This provides a tremendous service to researchers who want to spend their time building models and deriving insights from data, rather than learning the programming skills necessary to acquire and clean it.

It's becoming increasingly clear that "meta" packages that standardize the interface to data acquisition/curation in a particular domain would be tremendously useful. However, it is not clear how such interfaces should be designed. The R package **etl** is one step in this

[1]This section is freely available online `http://r-pkgs.had.co.nz/data.html`.

direction and aims to provide a standardized interface for *any* data access package that fits into an Extract-Transform-Load paradigm (**?**). The package provides generic `extract-transform-load` functions, but requires package authors to write custom `extract-transform` methods for the specific data source. In theory, the default `load` method works for any application; as well as other database management operations such as `update` and `clean`.

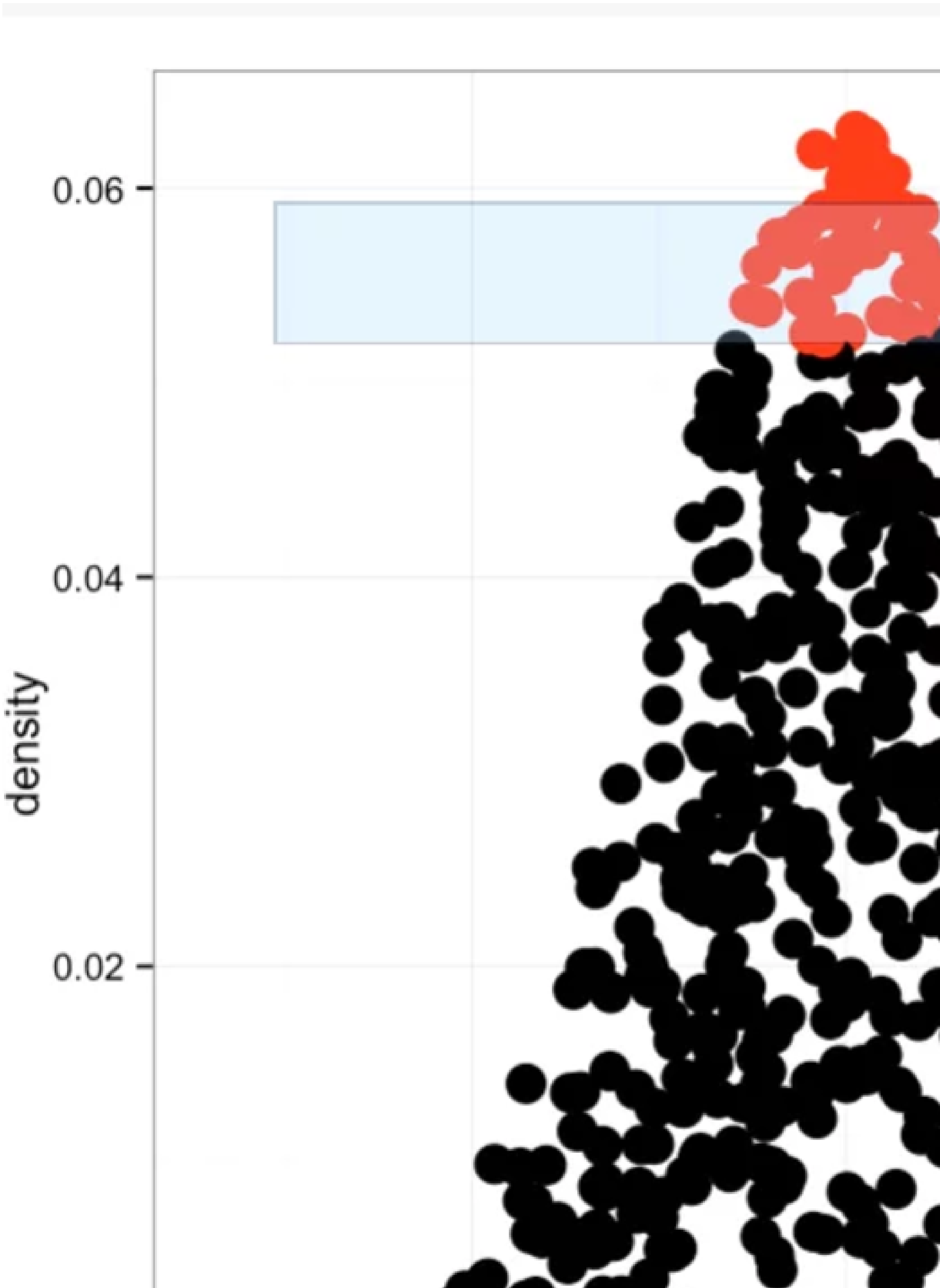
1.3 Dynamic interactive statistical web graphics

1.3.1 Why interactive?

Unlike computer graphics which focuses on representing reality, virtually; data visualization is about garnering abstract relationships between multiple variables from visual representation. The dimensionality of data, the number of variables can be anything, usually more than 3D, which summons a need to get beyond 2D canvasses for display. Technology enables this, enabling the user to see many views, query and link components. As demonstrated in Figure 1.1 using the R package **tourbrush** (**?**), interactive and dynamic graphics allow us to go beyond the constraints of low-dimensional displays to see high-dimensional relationships in data.

Dynamic interactive statistical graphics is useful for descriptive statistics, and also to help build better inferential models. Any statistician is familiar with diagnosing a model by plotting data in the model space (e.g., residual plot, qqplot). This works well for determining if the assumptions of a model are adequate, but rarely suggests that our model neglects important features in the data. To combat this problem, (**?**) suggest that we should plot the model in the data space and use dynamic interactive statistical graphics to do so. Interactive graphics have also proved to be useful for exploratory model analysis, a situation where we have many models to evaluate, compare, and critique (**?**); (**?**); (**?**); (**?**); (**?**). With such power comes responsibility that we can verify that visual discoveries are real, and not due to random chance (**?**); (**?**).

The ASA Section on Statistical Computing and Graphics maintains a video library which captures many useful dynamic interactive statistical graphics techniques. Several videos show how xgobi (predecessor to ggobi), a dynamic interactive statistical graphics system, can be used to reveal high-dimensional relationships and structures that cannot be easily identified


0.06
0.04
0.02
density

using numerical methods alone (?).[2] Another notable video shows how the interactive graphics system mondrian can be used to quickly find interesting patterns in high-dimensional data using exploratory data analysis (EDA) techniques (?).[3] The most recent video shows how dynamic interactive techniques can help interpret a topic model (a statistical mixture model applied to text data) using the R package **LDAvis** (?), which is the first web-based visualization in the library, and is discussed at depth in LDAvis: A method for visualizing and interpreting topics.

In order to be practically useful, interactive statistical graphics must be fast, flexible, accessible, portable, and reproducible. In general, over the last 20-30 years interactive graphics systems were fast and flexible, but were generally not easily accessible, portable, or reproducible. The web browser provides a convenient platform to combat these problems. For example, any visualization created with **LDAvis** can be shared through a Uniform Resource Locator (URL), meaning that anyone with a web browser and an internet connection can view and interact with a visualization. Furthermore, we can link anyone to any possible state of the visualization by encoding selections with a URL fragment identifier. This makes it possible to link readers to an interesting state of a visualization from an external document, while still allowing them to independently explore the same visualization and assess conclusions drawn from it.[4]

1.3.2 Indirect versus direct manipulation

Even within the statistical graphics community, the term *interactive* graphics can mean wildly different things to different people (?). Some early statistical literature on the topic uses interactive in the sense that an interactive command-line prompt allows users to create graphics on-the-fly (?). That is, users enter commands into the command-line prompt, the prompts evaluates the command, and prints the result (known as the read–eval–print loop (REPL)). Modifying a command to generate another variation of a particular result (e.g., to restyle a static plot) can be thought of as a type of interaction that some might call *indirect*

[2] For example, `http://stat-graphics.org/movies/xgobi.html` and `http://stat-graphics.org/movies/grand-tour.html`

[3] `http://stat-graphics.org/movies/tour-de-france.html`

[4] A good example of is `http://cpsievert.github.io/LDAvis/reviews/reviews.html`

manipulation.

Indirect manipulation can be achieved both from the command-line or from a graphical user interface (GUI). Indirect manipulation from the command-line is more flexible since we have complete control over the commands, but it is also more cumbersome since we must translate our thoughts into code. Indirect manipulation via a GUI is more restrictive, but it helps reduces the gulf of execution (i.e., easier to generate desired output) for end-users (**?**). In this sense, a GUI can be useful, even for experienced programmers, when the command-line interface impedes our primary task of deriving insight from data.

In many cases, the gulf of execution can be further reduced through direct manipulation. Roughly speaking, within the context of interactive graphics, direct manipulation occurs whenever we interact with a plot and reveal new information tied to the event. **?** use the terms dynamic graphics and direct manipulation to characterize "plots that respond in real time to an analyst's queries and change dynamically to re-focus, link to information from other sources, and re-organize information." Perhaps the most powerful direct manipulation technique is the paradigm of linked views (**?**), which will be discussed in more detail in a later section.

A simple example to help demonstrate the differences between these interactive techniques would be in an analysis of variance (ANOVA) via multiple boxplots. By default, most plotting libraries sort categories alphabetically, but this is usually not optimal for visual comparison of groups. With a static plotting library such as **ggplot2**, we could indirectly manipulate the default by going back to the command-line, reordering the factor levels of the categorical variables, and regenerate the plot (**?**). This is flexible and precise since we may order the levels by any measure we wish (e.g., Median, Mean, IQR, etc.), but it would be much quicker and easier if we had a GUI with a drop-down menu for most of the reasonable sorting options. In a general purpose interactive graphics system such as mondrian, one can use direct manipulation to directly click and drag on the categories to reorder them, making it quick and easy to compare any two groups of interest (**?**).

1.3.3 Linked views and pipelines

A general purpose interactive statistical graphics system should possess many direct manipulation techniques such as identifying (i.e., mousing over points to reveal labels), focusing (i.e., view size adjustment, pan and zoom), brushing/identifying, etc. However, it is the intricate management of information across multiple views of data in response to user events that is most valuable. Extending ideas from (**?**), (**?**) point out that any visualization system with linked views must implement a data pipeline. That is, a "central commander" must be able to handle interaction(s) with a given view, translate its meaning to the data space, and update any linked view(s) accordingly. In order to do so, the commander must know, and be able to compute, function(s) from data to visual space, as well as from visual space to the data. Implementing a pipeline that is fast, general, and able to handle statistical transformations is incredibly difficult. Unfortunately, literature on the implementation of such pipelines is virtually non-existent, but **?** provides a nice overview of the implementation details in the R package **cranvas** (**?**).

1.3.4 Web graphics

Thanks to the constant evolution and eventual adoption of HTML5 as a web standard, the modern web browser now provides a viable platform for building an interactive statistical graphics systems. HTML5 refers to a collection of technologies, each designed to perform a certain task, that work together in order to present content in a web browser. The Document Object Model (DOM) is a convention for managing all of these technologies to enable *dynamic* and *interactive* web pages. Among these technologies, there are several that are especially relevant for interactive web graphics:

1. HTML: A markup language for structuring and presenting web content.
2. SVG: A markup language for drawing scalable vector graphics.
3. CSS: A language for specifying styling of web content.
4. JavaScript: A language for manipulating web content.

Juggling all of these technologies just to create a simple statistical plot is a tall order.

Thankfully, HTML5 technologies are publicly available, and benefit from thriving community of open source developers and volunteers. In the context of web-based visualization, the most influential contribution is Data Driven Documents (D3), a JavaScript library which provides high-level semantics for binding data to web content (e.g., SVG elements) and orchestrating scene updates/transitions (**?**). D3 is wildly successful because is builds upon web standards, without abstracting them away, which fosters customization and interoperability. However, compared to a statistical graphics environments like R, creating basic charts is complicated, and a large amount of code must be hard-wired to each visualization. Fortunately, there are a number of ways to provide higher-level interfaces to web graphics, and we focus on R interfaces.

1.3.5 Translating R graphics to the web

There are a few ways to simply translate R graphics to a web format, such as SVG. R has built-in support for a SVG graphics device, made available through the `svg()` function, but it can be quite slow, which inspired the new **svglite** package (**?**). The **SVGAnnotation** package provides some functionality to post-process SVG files generated with `svg()` to add some basic interactivity and animation (**?**). The **gridSVG** package is specially designed to translate **grid** graphics (e.g., **ggplot2**, **lattice**, etc.) to SVG, and preserves the naming information of grid objects, making it easier to layer on interactive functionality (**?**). **?** uses **gridSVG** to enable linked brushing between **ggplot2** graphics, but only implements a few chart types. **?** uses **gridSVG** to provide pan and zoom capability to virtually any R graphic.

The **animint** and **plotly** packages take a different approach to translating **ggplot2** graphics to a web format (**?**); (**?**). Instead of translating directly to SVG via **gridSVG**, they extract relevant information from the internal representation of a **ggplot2** graphic[5], store it in JavaScript Object Notation (JSON), and pass the JSON as input to a JavaScript function, which then produces a web based visualization. It is becoming more and more popular to see JavaScript graphing libraries use this design pattern (sometimes referred to as a JSON specification or schema), since it separates out *what* information is contained in the graphic from *how* to actu-

[5]For a visual display of the internal representation used to render a **ggplot2** graph, see my **shiny** app here `https://gallery.shinyapps.io/ggtree`.

ally draw it. This has a number of advantages; for example, **plotly** graphics can be rendered in SVG, or using WebGL (based on HTML5 canvas, not SVG) which allows the browser to render many more graphical marks by leveraging the GPU.

Converting static graphics to web formats such as SVG or canvas not only allows us to embed the graphics into larger HTML documents, but it also allows us to inject basic interactive features at no or little cost to the user. For example, in both **animint** and **plotly**, we provide tool-tips (to obtain data-related information for each graphical mark) and clickable legends that show/hide graphical marks corresponding to the legend entry. In the case of **animint**, we have also extended **ggplot2**'s grammar of graphics implementation to enable animations and categorical linking between plots with relatively small amount of effort by users. This extension is discussed at length in Two new keywords for interactive, animated plot design: clickSelects and showSelected. In the case of **plotly**, we have also enabled animations, highlighting, and linked highlighting (even between non-plotly graphics). These features are discussed at length in plotly for R.

1.3.6 R interfaces for interactive web graphics

Translating existing graphics to a web-based format is useful for quickly breathing new life into existing code, but it is fairly limited in how far we can take it. Assuming the goal is to have a general, yet high-level, interface for creating highly dynamic interactive web graphics from R, we're better off building a new interface designed exactly for this purpose. The first serious attempt in this direction was the R package **rCharts**, whose R interface is heavily inspired by **lattice** (**?**). The most impressive result of **rCharts**'s design is its ability to interface with many different JavaScript charting libraries. However, **rCharts** has little to no support for coordination of dynamic linked views from R.

Another notable interface for creating interactive web graphics from R is **ggvis**, a reworking of ggplot2's grammar of graphics to incorporate interactivity (**?**). Similar to **animint**, **ggvis** encodes plot specific information as JSON, but instead of writing a JavaScript renderer from the ground up, it uses Vega, a popular JSON schema for creating web-based graphics (**?**). This limits the flexibility of **ggvis**, but it also drastically reduces the overhead in maintaining such a software project, allowing the focus to be on building a grammar for expressing interactions from R.

The current version of **ggvis** uses an old version of vega, before a grammar for interactive graphics was added to its JSON schema (**?**). In order to respond to user interactions with vega graphics, **ggvis** has its own custom JavaScript designed specifically for vega. To enable support for coordinated linked views, it exposes the data pipeline to users via the R package **shiny**, a framework for writing web applications in R (**?**). A web application is a website which, when visited by users (aka clients), communicates with a web server. This approach is useful when a website needs to execute code that can not be executed in the web browser (e.g., R code). Figure 1.2 provides a visual demonstration of this model and its relation to the data pipeline necessary for coordinating linked views.

Generally speaking, websites that render entirely client-side are more desirable since they are easier to share, more responsive, and require less computational resources to run[6]. However, the client-server approach can be very useful for dynamically performing statistical computations, a key characteristic of most interactive statistical graphics systems. (**?**) and (**?**) also allow us to execute R code on a web server, and retrieve output via HTTP, but **shiny** is the most heavily used since apps can be written entirely in R using a very powerful, yet approachable, reactive programming framework for handling user events. There are also many convenient shortcuts for creating attractive HTML input forms, making it incredibly easy to go from R script to an web app powered by R that dynamically updates when users alter input values. In other words, **shiny** makes it quick and easy to write web-based GUIs with support for indirect manipulation.

Historically, an advanced understanding of **shiny** and JavaScript was required to implement direct manipulation in a **shiny** app. Recently, **shiny** added support for retrieving information on user events with static R graphics[7], allowing developers to coordinate views in a web app, with no JavaScript involved. This is a powerful tool for R users, but it has its weaknesses. Most

[6]The `http://www.shinyapps.io/` service helps to provide easy access to a **shiny** server (a web server running special shiny software), so that **shiny** apps can be shared via a URL, for example: `https://hadley.shinyapps.io/14-ggvis/linked-brushing.Rmd`

[7]This website shows what information is sent from the client to the server when users interact with plot(s) via mouse event(s) – `http://shiny.rstudio.com/gallery/plot-interaction-basic.html`

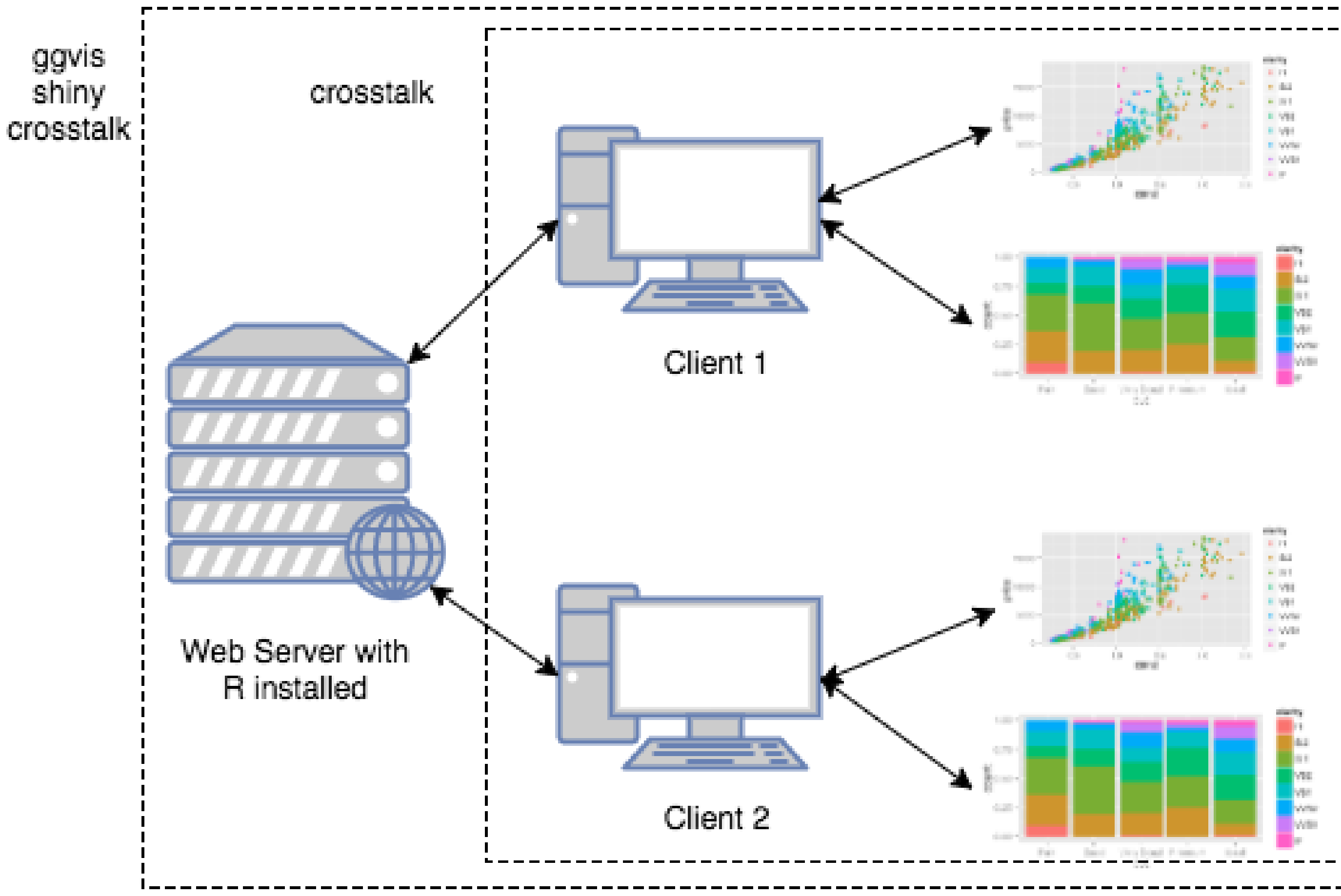


Figure 1.2 A basic visual depiction of the different approaches to implementing a data pipeline for interactive web graphics. The R packages **ggvis** and **shiny** expose the pipeline to users in R, which requires a web server for viewing. The R package **crosstalk** will allow developers to implement and expose the pipeline on both the server and client levels.

importantly, its not clear how to handle interactions when positional scales are categorical (e.g., a bar chart) or how to provide a visual clue that something has been selected.

The touring video in Figure 1.1 purposefully uses **shiny**'s built-in support for brushing to demonstrate the problem with providing a visual clue. This points to the fundamental problem in using non-web-based graphics to implement interactive graphics in a web browser: every time the view updates, the display must be redrawn, resulting in a "glitch" effect. If the plot being brushed used native web graphics (e.g., SVG), it would allow for finer control over how the view updates in response to user interactions and/or dynamic data. On the other hand, since **ggvis** is web-based, and has special client-side functionality, it knows how to smoothly transition from one frame to the next when provided with new data from the **shiny** server, which is crucial for constructing a mental model of the data space. Having richer interfaces for generating web-based interactive graphics from R that can share selections, and handle smooth transitions, would make this, and many other examples, generally better.

Many web-based graphing toolkits have appeared since the advent of **rCharts**, making a single package that interfaces with *every* toolkit infeasible. Some ideas deriving from work on **rCharts**, such as providing the glue to render plots in various contexts (e.g., the R console, shiny apps, and **rmarkdown** documents), have evolved into the R package **htmlwidgets** (**?**). Having built similar bridges for **animint** and **LDAvis**, I personally know and appreciate the amount of time and effort this package saves other package authors.

The **htmlwidgets** framework is not constrained to just graphics, it simply provides a set of conventions for authoring web content from R. Numerous JavaScript data visualization libraries are now made available using this framework, most designed for particular use cases, such as **leaflet** for geo-spatial mapping, **dygraphs** for time-series, and **networkD3** for networks (**?**); (**?**); (**?**).[8] There are also HTML widgets that provide an interface to more general purpose visualization JavaScript libraries such as **plotly**, **rbokeh**, and **rcdimple** (**?**); (**?**); (**?**). Most of these JavaScript libraries provide at least some native support for direct manipulation such as identifying (i.e., mousing over points to reveal labels), focusing (i.e., pan and zoom), and

[8]For more examples and information, see http://www.htmlwidgets.org/ and http://hafen.github.io/htmlwidgetsgallery/

sometimes highlighting (i.e., brushing over points to highlight points in another view). More often than not, the support for dynamic and linked views is lacking, especially if we want to define the linking in R, and produce a standalone HTML document.

The R package **crosstalk** is a new framework for coordinating arbitrary HTML widgets (**?**). It provides both an R and a JavaScript API for querying selections, meaning **crosstalk** powered HTML widgets can work with or without **shiny**, and if implemented carefully by HTML widget authors, provides a means for coordinating multiple HTML widgets without shiny. Generally speaking, **crosstalk** just provides a standard way to set, store, and access selection values in the browser, so the actual logic for updating views based on the selection value(s) is on the HTML widget author, and this part is far from trivial. In a sense, this project is similar to the work of **?**, which provides semantics for "snapping together" arbitrary views that are aware of the relational schema, but does so in a web-based environment, rather than requiring a machine running Windows.

The first HTML widget to leverage **crosstalk** was (**?**), but is limited to linked brushing on scatterplots.[9] Currently, there are a couple other R packages with **crosstalk** support, including **leaflet** and **listviewer**, but **plotly** is the only package which supports a non-identity functions between the data and displays. It also has rich support for interaction types, including mouse hover, click, and multiple types of click+drag selections.

Having HTML widgets that can share selections with each other will be a huge step forward for web-based interactive graphics. With some effort and careful implementation by HTML widget authors, it may be possible to provide sensible defaults for updating views between arbitrary widgets, and users that know some JavaScript will also be able to customize or extend these defaults from R. The **htmlwidget** package provides conventions for this, by allowing one to send arbitrary JavaScript functions from R that execute after the widget has rendered in the browser. The biggest problem in implementing coordinated widgets will be in managing data structures, since each widget will likely have its own data structure for representing a selection. In this case, in order to coordinate them, users may have to embed widgets in a shiny app to access and organize selections. This gives users tremendous control over sharing selections, but

[9] See, for example, `http://rpubs.com/jcheng/crosstalk-demo`

may limit control over smooth transitions between states of a given widget (a key characteristic of dynamic graphics), and increases the amount of complexity involved in sharing their work.

2 Taming PITCHf/x Data with XML2R and pitchRx

Pitch f/x refers a massive, publicly available baseball dataset hosted on the web in XML and JSON format. Since this data is large, increases on a daily basis, and only licensed for individual use, the **pitchRx** package provides a simple interface to download, parse, clean, and transform the data from its source (instead of directly distributing the data). If acquiring large amounts of data, to avoid memory limitations, users may divert incoming data in chunks to a database using any valid R database connection (**?**). It also provides a convenient function to update an existing database with the most recently available data without re-downloading anything.

The **openWAR** package also provides high-level access to Pitch f/x data, but it is currently more limited in the data it can acquire (**?**). It also currently depends on the difficult to install **Sxslt** package, impeding portability (**?**). **openWAR** depends on **Sxslt** to help transform XML files to R data frames via XSL Transformations (XSLT). Without advanced knowledge of XSLT, one must define transformations by hard coding assumptions about the XML format, such as the names of fields of interest. New variables have been added into Pitch f/x several times, and **pitchRx** automatically picks them up, thanks to functionality provided by **XML2R**.

XML2R makes it easy to wrangle relational data stored as a collection of XML files into a list of data frames. Its interface satisfies principles from pure functional programming: the output of each function can be completely determined from the input. The interface is also predictable: each function inputs and outputs a list of observations (an observation is a matrix with one row). It also represents XML content as a list of observations (matrices with one row), allowing each function to operate on native R data structures, making it more intuitive for R programmers to work with compared to the non-native XMLDocumentContent. This new representation is slightly less computationally efficient in some cases, but it has also made it much easier to implement and maintain higher-level interfaces to specific XML data sources, such as **pitchRx** and **bbscrapeR** (**?**).

To see the fully published article “Taming PITCHf/x Data with XML2R and pitchRx”, see http://rjournal.github.io/archive/2014-1/sievert.pdf

3 LDAvis: A method for visualizing and interpreting topics

The R package **LDAvis** creates an interactive web-based visualization of a topic model that has been fit to a corpus of text data using Latent Dirichlet Allocation (LDA). Given the estimated parameters of the topic model, it computes various summary statistics as input to a reusable interactive visualization built with HTML, JavaScript, and D3. The goal is to help users interpret the topics in their LDA topic model, and the interactive visualization is primarily useful for quickly viewing, altering, and tracking changes in rankings of terms for a given topic.

In a topic model, each topic is defined by a probability mass function over each unique term in the corpus. When studying their differences, analysts often look at lists of the top (say 30) terms of a topic ranked by the estimated probability within that given topic. As discussed in the video below and in our paper, this makes it hard to differentiate meaning between topics since words that are likely to appear overall are also likely to appear in a given topic. Instead, we propose ranking terms using a compromise between this probability and lift (probability within topic divided by overall probability). We also conduct a user study which provides evidence that this compromise helps in identifying topics, and propose a sensible starting point for choosing a compromise; but in practice, users will want to adjust this value and understand how rankings are affected. For this reason, it is important that we assist users in their ability to track changes, by using smooth transitions from one ranking to the next.

To read the full paper, see: `http://nlp.stanford.edu/events/illvi2014/papers/sievert-illvi2014.pdf`

4 Two new keywords for interactive, animated plot design: clickSelects and showSelected

This paper explains the clickSelects/showSelected paradigm, implemented in **animint**, which makes it easy to select/query points belonging to arbitrary group(s) and visualize those points in another data space. This differs from the classical linked brushing approach where points must belong to contiguous regions within a subset of the data space.

https://github.com/tdhock/animint-paper/blob/master/HOCKING-animint.pdf

5 Interactive Data Visualization with plotly and shiny

? proposed a taxonomony of interactive data visualization based on three fundamental data analysis tasks: finding Gestalt, posing queries, and making comparisons. The top-level of the taxonomy comes in two parts: *rendering*, or what to show on a plot; and *manipulation*, or what to do with plots.[ˆThe cookbook and advanced manipulation sections of the plotly book] Under the manipulation branch, they propose three branches of manipulation: focusing individual views (for finding Gestalt), linking multiple views (for posing queries), and arranging many views (for making comparisons). Of course, each of the three manipulation branches include a set of techniques for accomplishing a certain task (e.g., within focusing views: controlling aspect ratio, zoom, pan, etc), and they provide a series of examples demonstrating techniques using the XGobi software toolkit (**?**).

This paper explores the taxonomy proposed by **?** in detail, and demonstrates how we can bring these techniques to the web browser via the R packages **plotly** and **shiny** (**?**); (**?**).

5.1 The taxonomy

5.2 Exploring Australian election data

5.3 Exploring pedestrain counts

5.4 Exploring disease outbreaks

- Geographic zoom+pan linked to summary statistics. Fosters all three tasks?
- Explain how

6 References