Interfacing R with Web Technologies for Interactive Statistical Graphics and Computing with Data

Carson Sievert

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This thesis would not be possible without many people. First and foremost, special thanks to my major professor Heike Hofmann. I would not made it to this point without such a warm and friendly mentor who was often more confident in my abilities than I was of my own. Thank you for always supporting me no matter how many things I had going on to distract me from research. I aspire to inherit the same empathy and support that you show to your students on a daily basis.

Another special thanks goes to Di Cook. One day during the fourth year of my PhD, Di took me to lunch, told me she was transferring to Monash University in Australia, and invited me to join her. Of course, I said yes, and when I arrived, I immediately felt welcomed and a part of the group – all thanks to Di. She strategically assigned me to assist her students with their thesis projects, run R workshops for the university, and most importantly, work on my tennis game. Through this "work" I met so many amazing people and had many memorable experiences. Those 6 months gave me a new perspective on life in general and I am forever grateful for being blessed enough to take the opportunity.

Thank you to many of Heike and Di's former students who came before me (just to name a few: Hadley Wickham, Michael Lawrence, Yihui Xie, Xiaoyue Cheng, Barrett Schloerke, Susan VanderPlas). Your work has not only inspired and enabled my work, but it has also enabled an entire community of people working with data to do amazing things. Without this strong history and community at Iowa State, I would not have had the courage or the vision to follow such a "non-traditional" research path. I hope the

University continues to value this type of work as it teaches students skills that are in high demand and generally improves the way data-driven research is performed.

Thank you to all my collaborators, especially Toby Dylan Hocking. Toby and Susan VanderPlas laid the initial framework for **animint** – which I first worked on as a Google Summer of Code student under Toby's guidance. Toby later went on write the initial version of the `ggplotly`() function in **plotly**, borrowing a lot of ideas from **animint**. As Toby became busy with other things, he introduced me to the plotly team, and eventually handed over the reigns on the project, which has helped to financially support the last year or so of grad school.

Thank you also to the plotly team, and in particular, the software engineers who work on the open source project plotly.js. My work has benefited greatly from your responsiveness to my questions, feature requests, and bug reports. I have a great amount of respect for the work that you do, and I hope this project keeps improving at its current break neck pace.

Finally, thank you to my family for their encouragement and keeping me grounded throughout this experience. Thank you to my father for the initial encouragement to pursue a PhD, conversations surrounding work-life balance, and also pushing me to "graduate before I'm 40". Thank you to my mother for her unconditional love, endless care, and predenting to understand what I do for a living. Thank you to my brothers for providing me with shelter, beer, and Twins tickets. Thank you all for your willingness to drop everything to help me at any given moment. I can't say I've always been as willing, as I have been selfish with my time during my PhD, but I hope to change that after graduation.

1 Problem statement

> "[The web] has helped broaden the focus of statistics from the modeling stage to all stages of data science: finding relevant data, accessing data, reading and transforming data, visualizing the data in rich ways, modeling, and presenting the results and conclusions with compelling, interactive displays." - (Nolan and Temple Lang 2014)

The web enables broad distribution and presentation of applied statistics products and research. Partaking often requires a non-trivial understanding of web technologies, unless a custom interface is designed for the particular task. The CRAN task views on *open data* (Jaime Ashander, n.d.) and *web services* (Thomas Leeper, n.d.) document such interfaces for the R language, the world's leading open source data science software (R Core Team 2015). This large community effort helps R users make their work easily accessible, portable, and interactive.

R has a long history of serving as an interface to computational facilities for the use of people doing data analysis and statistics research. In fact, the motivation behind the birth of R's predecessor, S, was to provide a direct, consistent, and interactive interface to the best computational facilities already available in languages such as FORTRAN and C (Becker and Chambers 1978). This empowers users to focus on the primary goal of statistical modeling and data analysis problems, rather than the computational implementation details. By providing more and better interfaces to web services, we can continue to empower R users in a similar way, by making it easier to acquire and/or share data, create interactive web graphics and reports, distribute research products to

a large audience in a portable way, and more generally, take advantage of modern web services.

Portability prevents the broad dissemination of statistical computing research, especially interactive statistical graphics. Interactive graphics software traditionally depend on software toolkits like GTK+ or openGL that provide widgets for making interface elements, and also event loops for catching user input. These toolkits need to be installed locally on a user's computer, across various platforms, which adds to installation complexity, impeding portability. Modern web browsers with HTML5 support are now ubiquitous, and provide a cross-platform solution for sharing interactive statistical graphics. However, interfacing web-based visualizations with statistical analysis software remains difficult, and still requires juggling many languages and technologies. By providing better interfaces for creating web-based interactive statistical graphics, we can make them more accessible, and therefore make it easier to share statistical research to a wider audience. This research addresses this gap.

2 Overview

2.1 What makes a good statistical software interface?

2.1.1 Synergy between interfaces

Roughly speaking, there are two broad categories of software interfaces: graphical user interfaces (GUIs) and programming interfaces. Within the domain of statistical computing and data analysis, Unwin and Hofmann (2009) explores the strengths and weaknesses of each category, and argues that an effective combination of both is required in order to use statistical software to its full potential. Their main argument is that, since programming interfaces are precise and repeatable, they are preferable when we can describe exactly what we want, but a GUI is better when: "Searching for information and interesting structures without fully specified questions."

Unwin and Hofmann (2009) further discuss the different audiences these interfaces tend to attract. Programming interfaces attract power users who need flexibility, such as applied statisticians and statistical researchers in a university, whereas more casual users of statistical software prefer GUIs since they help hide implementation details and allow the focus to be on data analysis. GUIs are still certainly useful for power users in their own work, especially when performing data analysis tasks that are more exploratory (data first, hypothesis second) than confirmatory (hypothesis first, data second) in nature. At the end of the day, all software interfaces are fundamentally *user* interfaces, and the interface which enables one to do their work most effectively should be preferable.

Unfortunately, as Unwin and Hofmann (2009) says, "There is a tendency to judge software by the most powerful tools they provide (whether its a good interface or not), rather than by whether they do the simple things well". This echoes similar thinking found in the Unix philosophy, a well-known set of software engineering principles which derived from work at Bell Laboratories creating the Unix operating system (Doug McIlroy 1978); (Raymond 2003). The Unix philosophy primarily values interfaces that each do one simple thing, do it well, and most importantly, **work well with each other**. It is all too common that we evaluate interfaces in isolation, but as Friedman and Wand (2008) writes: "No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system".

The next section discusses work that brings this philosophy towards statistical *programming* interfaces, but it can also be useful to apply this philosophy towards GUI design. The concept of a GUI can be made much more broad than some may realize. For example, most would not think to consider graphical elements of a plot to be elements of a GUI; but for good reason, this is a key feature in interactive graphics software. In other words, interactive graphics could themselves be considered a GUI, which can (in theory) be embedded inside a larger GUI system. For this reason, interactive graphics software should strive to work well with other software so users can combine their relative strengths.

Buja et al. (1991) first described direct manipulation (of graphical elements) in multiple linked plots to perform data base queries and visually reveal high-dimensional structure in real-time. Cook, Buja, and Swayne (2007) argues this framework is preferable to posing data base queries dynamically via a menus, as described by Ahlberg, Williamson, and Shneiderman (1991), and goes on to state that "Multiple linked views are the optimal framework for posing queries about data". Unwin and Hofmann (2009) agrees with this perspective, and more generally state that: "The emphasis with GUIs should be on direct

manipulation systems and not on menu (indirect manipulation) systems." Ideally, one would be able to directly manipulate views linked between different interactive statistical graphics software systems, but traditionally this has not been possible.

As with any GUI, any interactive graphics system has its own special set of strengths and limitations that power users are bound to run up against. This is especially true of systems that are entirely GUI-based, as forcing every possible option into a GUI generally reduces its ease of use. The classic way to resolve the problem is to provide a *programming* interface designed to enhance the GUI itself. A great example is the R package **rggobi** which provides a programming interface to the GUI-based interactive statistical graphics software GGobi – a descendant of the X-Windows based system XGobi (Cook and Swayne 2007); (Wickham et al. 2008); (Swayne, Cook, and Buja 1998). This interface allows users to leverage both the strengths of R (statistical modeling, data manipulation, etc) and GGobi (interactive and dynamic graphics). Although GGobi allows one to link multiple views of high-dimensional data in numerous windows and/or dynamic displays, this extension does not enable linked views between GGobi and another graphics rendering system.

As with any GUI, any interactive graphics system requires a reactive programming framework which handles scene updates when certain user events occur. To minimize computations and optimize efficacy in scene updates, these programming frameworks are typically based on the Model-View-Controller (MVC) paradigm. In this paradigm, the model contains all the data and logic necessary to generate view(s). The controller can be thought of as a component on top of a view that listens for certain events, and feeds those events to the model so that the model can update view(s) accordingly. In interactive *statistical* graphics, controllers can trigger the model to perform statistical computations, and Andreas Buja and McDonald (1988) first proposed some general stages that the model (aka. "data pipeline") should possess.

Wickham et al. (2010) and Xie, Hofmann, and Cheng (2014) provide a comprehensive overview of data pipeline implementation approaches in notable interactive statistical graphics software. All of the systems considered assume that the software runs as a desktop application on the user's machine. Furthermore, they all take a rather monolithic view on the framework used to link and render graphics. As a result, the only way to allow users to extend their interface is to plug custom methods into certain stages of the pipeline (Peter Sutherland and Cook 2000); (Lawrence 2002), but this fails to address other limitations in the overall system. From a user perspective, its much more powerful to be able to combine relative strengths of each system (e.g., imagine a combination of GGobi for continous data and MANET for categorical/missing data (Unwin A. 1996)). Moreover, it is difficult to share graphics produced with these systems many of these systems have strong software requirements

Most of the fundamental issues with traditional approaches to interactive statistical graphics software can be addressed using a web-based approach. First off, since the web browser is now ubiquitous, web-based visualization are easy to run, deploy, and share. Taking a web-based approach also opens the possibility of linking views with (a huge number of!) other interactive web graphics systems. This broadens the scope of what is possible when extending an interactive graphics systems by combining multiple systems. However, in a web-based approach, special attention should be taken to where computations occur since its much more preferrable to have the system operate entirely in a web browser, rather than making calls to an external web server (to execute R commands, for instance).

As discussed further in linking views without shiny, it is now possible to create a standalone web pages with linked views from R within or even between two different interactive graphics systems. North and Shneiderman (1999) describes a similar framework for linking views on a Windows platform, but provides no examples beyond simple fil-

ter/subset operations. Usually what distinguishes interactive *statistical* graphics from interactive graphics is the ability to perform statistical computations on dynamically changing data (i.e., the data pertaining to the selection). Compared to something like R, the language of the web browser (JavaScript) has very limited resources for doing statistical computing, so interactive web graphics need to interface with other languages in some way to become more statistical.

Numerous R packages which create interactive web graphics have some native support[1] for focusing and linking multiple plots, but most are lacking strong native support for the data pipeline necessary to perform statistical computations in real-time within the browser (Hocking, VanderPlas, and Sievert 2015); (Sievert et al. 2016); (Hafen and team 2015). This is partly by design as JavaScript (the language of the web) has poor support for statistical computing algorithms and requiring an external R process introduces a significant amount of complexity required to view and share visualizations. There are some promising JavaScript projects that are attempting to provide the statistical resources necessary to build pipelines entirely within the browser, but they are still fairly limited (Lab 2016a); (Lab 2016b); (Bååth 2016).

The data pipeline computes transform(s)/statistic(s) given different input data which represents user selection(s). If the number of selection states is relatively small, it may be possible to precompute every possible output (in R), and push the results to the browser, creating a standalone web page. On the other hand, if the number of selection states is large, a client-server model (i.e., a web application) is more appropriate. In a web application, the web browser (client) requests computations to be performed on a web server which then returns some output. Figure **??** shows a visual depiction of this difference (standalone web page vs web application) in a linked views environment.

[1] Native support here implies that all computations can be performed entirely inside the web browser, without any external calls to a separate process (e.g. R).

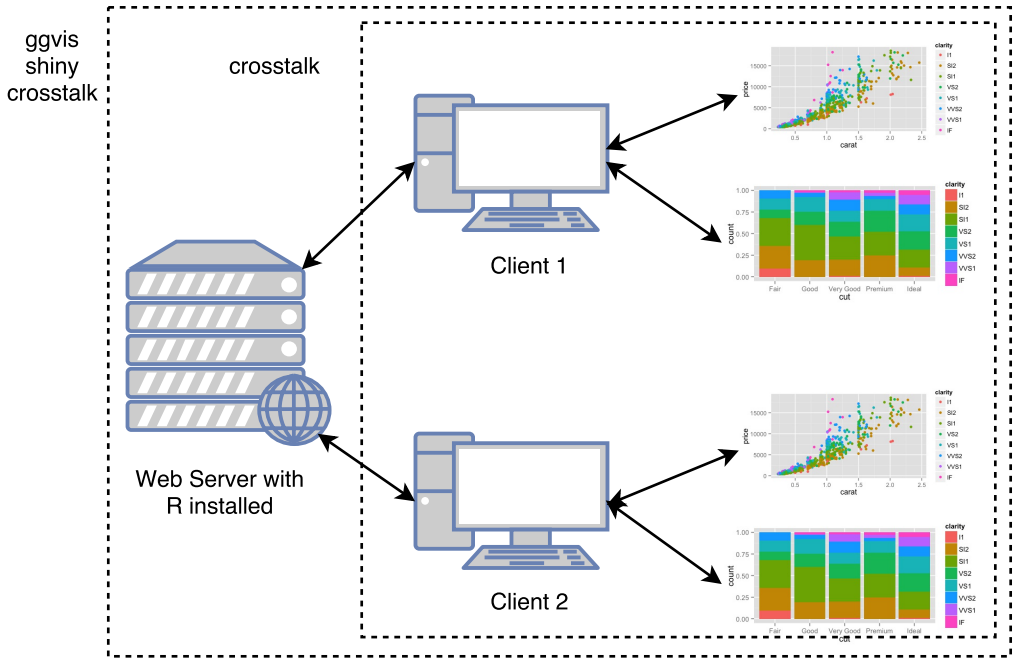


Figure 2.1 A basic visual depiction of linked views in a standalone web page (A) versus a client-server model (B). In some cases (A), linked views can be resolved within a web browser, which generally leads to a better user experience. In other cases (B), updating views may require calls to a web server running special software.

There are a number of ways to request R computations from a web browser (Ooms 2014b); (Urbanek and Horner 2015); (Trestle Technology 2016), but the R package **shiny** is by far the most popular approach since authors can create web applications entirely within R (without HTML/JavaScript knowlegde) via an approachable reactive programming framework (Chang et al. 2015).[2] Going forward, there is no doubt the reactive programming framework that **shiny** provides will be useful for implementing the data pipeline necessary for creating web-based interactive statistical graphics.

At some rudimentary level, a programming interface like **shiny** provides a way to design a graphical interface around programming interface(s). This type of software not only

[2]There are also many convenient shortcuts for creating attractive HTML input forms, making it incredibly easy to go from R script to a nice looking web app powered by R that dynamically updates when users alter input values (thereby enabling indirect manipulation).

enables people to share their work with others in a user friendly fashion, but it can also make their own work more efficient. In order to be efficient, the time invested to create the GUI must be less than the amount of time it saves by automating the programming task(s). And to empower this type of efficiency, it helps tremendously to have programming interfaces that work well together.

2.1.2 Synergy among programming interfaces

A typical data analysis workflow involves many different tasks, from data acquistion, import, wrangling, visualization, modeling, and reporting. Wickham and Grolemund (2016) points out the non-linear iteration between wrangling, visualization, and modeling required to derive insights from data. Often times, each stage requires one or more programming interface, and switching from one interface to another can be frustrating since different interfaces often have different philosophies. In some cases, the frustation involved from transitioning from one interface to another is unavoidable, but in most cases, working with a collection of interfaces that share the same underlying principles helps alleviate friction. This is the motivation behind the R package **tidyverse** which bundles numerous interfaces for doing fundamental data analysis tasks based largely on the tidy data framework (Wickham 2016b); (Wickham 2014b).

While tidy tools provide a nice cognitive framework for many common data analysis tasks, sometimes it's necessary to work with messy (i.e., non-rectangular) data structures. In fact, Wickham and Grolemund (2016) argues that 80% of data analysis tasks can be solved with tidy tools while the remaining 20% requires other tools. Probably the most common non-tidy data structure in R is the nested list which is the most flexible and reliable way to represent web-based data structures, such as JSON and XML, in R.

2.2 Acquiring and wrangling web content in R

2.2.1 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 (Chambers 1999), Brian Ripley implemented the connections interface for file-oriented input/output in R (Ripley 2001). 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 (Lang 2006).

The **RCurl**, **XML**, and **RJSONIO** packages were major contributions that drastically improved our ability to request, manipulate, and create web content from R (Nolan and Temple Lang 2014). 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 (Temple Lang 2014a). The **XML** package provides low-level bindings to the C library libxml2, making

it possible to download, parse, manipulate, and create XML (and HTML) (Lang 2013). 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) (Temple Lang 2014b). 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** (Ooms 2015). The **httr** package builds on **curl** and organizes it's functionality around HTTP verbs (GET, POST, etc.) (Wickham 2015a). 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 (Ooms 2014a). 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 (Wickham 2015c). 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

(Sievert 2014b). As a result, these interfaces reduce the start-up 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 (Sievert 2015a). 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** (Harrison 2014). **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.

2.2.2 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. Wickham (2015b) provides a nice section outlining the details of bundling data with R packages.[3]

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 re-hosting 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

[3]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 (Baumer and Sievert 2016). 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`.

2.3 Interactive statistical web graphics

2.3.1 Why interactive graphics?

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, allowing one to link multiple low-dimensional displays in meaningful ways to reveal high-dimensional structure. As demonstrated in Figure 2.2 using the R package **tourbrush** (Sievert 2015b), interactive and dynamic statistical graphics allow us to go beyond the constraints of low-dimensional displays to perceive high-dimensional relationships in data.

Interactive statistical graphics are a useful tool for descriptive statistics, as well as for building 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, Wickham, Cook, and Hofmann (2015) suggest to plot the model in the data space and use dynamic interactive statistical graphics to do so. Interactive graphics have also proved to be useful

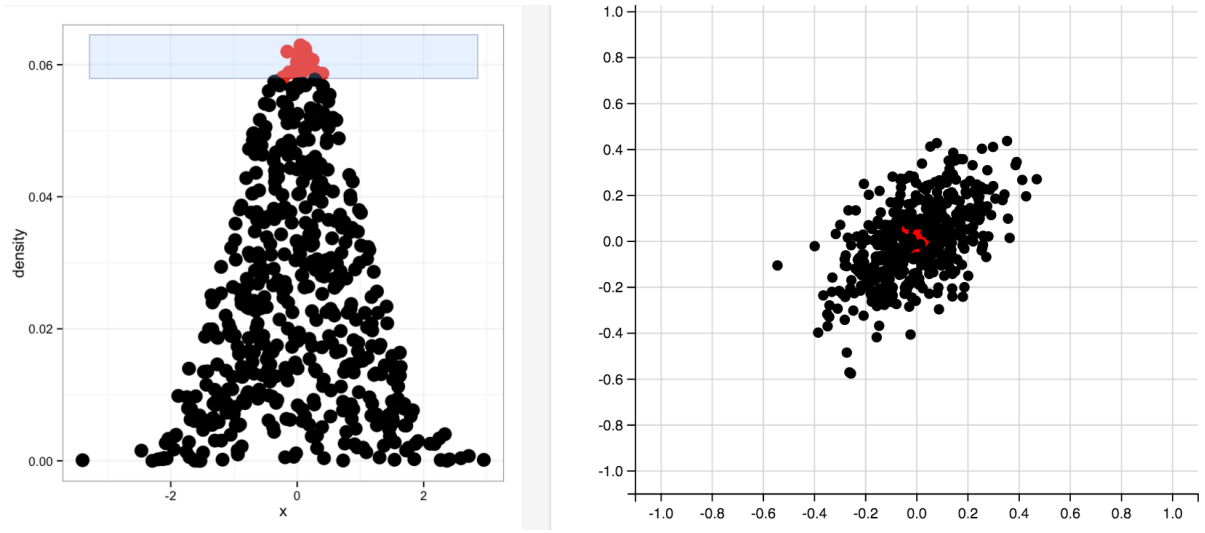


Figure 2.2 A video demonstration of interactive and dynamic techniques for visualizing high-dimensional relationships in data using the R package **tourbrush**. You can view this movie online at https://vimeo.com/148050343.

for exploratory model analysis, a situation where we have many models to evaluate, compare, and critique (Unwin, Volinsky, and Winkler 2003); (Urbanek 2004); (Ripley 2004); (Unwin 2006); (Wickham 2007). With such power comes responsibility that we can verify that visual discoveries are real, and not due to random chance (Buja et al. 2009); (Majumder, Hofmann, and Cook 2013).

Even within the statistical graphics community, the term *interactive* graphics can mean wildly different things to different people (Swayne and Klinke 1999). Some early statistical literature on the topic uses interactive in the sense that a command-line prompt allows users to create graphics on-the-fly (R. A. Becker 1984). That is, users enter commands into a 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 manipulation*.

Indirect manipulation can be achieved via a GUI or a programming (i.e., command-line) interface. 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 (Hutchins, Hollan, and Norman 1985). In this sense, a GUI can be useful, even for experienced programmers, when the command-line interface impedes the 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 direct interaction with plotting elements refocuses or reveals new information tied to the event. Cook and Swayne (2007) 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."

Although this thesis focuses on linked views, direct manipulation encompasses many useful techniques that could be used, for example, to re-focus a particular view. A simple example would be directly manipulating the ordering of boxplots to enhance graphical analysis of variance (ANOVA). 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 (Wickham 2009a). 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 (Theus and Urbanek 2008).

The ASA Section on Statistical Computing and Graphics maintains a video library which captures many useful 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 using numerical methods alone (Swayne, Cook, and Buja 1998).[4] 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 (Theus and Urbanek 2008).[5] 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** (Sievert and Shirley 2014), 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

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

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

a visualization from an external document, while still allowing them to independently explore the same visualization and assess conclusions drawn from it.[6]

2.3.2 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 (Heer 2011). 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 customized for each visualization. As a result, web graphics are widely used for

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

presentation graphics (visualization type is known), but are not (yet!) practically useful for exploratory graphics (visualization type is not known).

There are a number of projects attempting to make interactive web graphics more practically useful for ad-hoc data analysis tasks. For statisticians and other people working with data, this means the interface should look and feel like other graphing interfaces in R where many of the rendering details are abstracted away allowing the user to focus on data analysis. The next section discusses some approaches that provide a direct translation of R graphics to a web-based format – requiring essentially no additional effort by existing R users to take advantage of them. The section afterwards discusses other approaches which design a brand-new R interface for creating web graphics.

2.3.3 Translating R graphics to the web

There are a few ways to 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 (Wickham et al. 2016). The `grid.export()` function from the **gridSVG** package also provides an SVG device, designed specifically for **grid** graphics (e.g., **ggplot2**, **lattice**, etc.).[7] It adds the ability to retain structured information about grid objects in the SVG output, making it possible to layer on interactive functionality (Potter and Murrell 2012). The **rvg** package is a similar project, but implements its own set of graphics devices to support SVG output as well as proprietary XML-based formats (e.g., Word, Powerpoint, XLSX) (Gohel 2016b).

A number of projects attempt to layer interactivity on top of a SVG output generated from R code an SVG graphics device. The **SVGAnnotation** package was the first attempt at post-processing SVG files (created via `svg()`) to add some basic interactivity

[7]The **gridGraphics** package makes it possible to draw **base** graphics as **grid** graphics – meaning that **gridSVG** can (indirectly) convert any R graphic.

including: tooltips, animation, and even linked brushing via mouse hover (Nolan and Temple Lang 2012).[8] The **ggiraph** package is a more modern approach using a similar idea. It uses the **rvg** package to generate SVG output via a custom graphics device; but focuses specifically on extending **ggplot2**'s interface, and currently has no semantics for linking plots (Gohel 2016a). There are also a number of notable projects that layer interactivity on top of SVG output provided by **gridSVG**'s graphics device, including **vdmR** which enables (very limited support for) linked brushing between **ggplot2** graphics and **svgPanZoom** which adds zooming and panning (Fujino 2015); (Riutta et. al. and Russell 2015). Translating R graphics at this level is a fundamentally limited approach, however, because it loses information about the raw data and its mapping to visual space.

The **animint** and **plotly** packages take a different approach in translating **ggplot2** graphics to a web format (Hocking, VanderPlas, and Sievert 2015); (Sievert et al. 2016). Instead of translating directly to SVG via **gridSVG**, they extract relevant information from the internal representation of a **ggplot2** graphic[9], store it in JavaScript Object Notation (JSON), and pass the JSON as an input to a JavaScript function, which then produces a web based visualization. This design pattern is popular among modern web-based graphing libraries, since it separates out *what* information is contained in the graphic from *how* to actually 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. It also has the advantage of more extensible from R since novel features can be enabled by adding to and/or modifying the underlying data structure (instead of writing ad-hoc JavaScript code to modify the DOM).

[8] Unfortunately, this package now serves as a proof of concept as most examples are now broken, and no one has contributed to the project in 3 years.

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

Translating existing R graphics to a web-based format is useful for quickly enabling some basic interactivity, but an extension of the underlying graphics interface may be required to enable more advanced features (e.g. linked views). For example, in both **animint** and **plotly**, we automatically provide tooltips (which reveal more information about each graphical mark on mouse hover) and clickable legends that show/hide graphical marks corresponding to the legend entry. The **animint** package extends **ggplot2**'s grammar of graphics implementation to enable animations and linked views with relatively small amount of effort required by those familiar with **ggplot2**. This extension is discussed at length in the chapter Extending ggplot2's grammar of graphics implementation for linked and dynamic graphics on the web. The **plotly** package supersedes **animint** to support a larger taxonomy of interaction types (e.g., hover, click, click+drag), interaction modes (e.g., dynamically controllable persistent brush), chart types (e.g., 3D surfaces), and even provides a consistent interface for enabling animation/linked-views across graphs created via **ggplot2** or its own custom (non-ggplot2) interface. These features are discussed in plotly for R.

2.3.4 Interfacing with interactive web graphics

Although it is more onerous for users to learn a new interface, there are a number of advantages to designing a new R interface (that is independent of any translation) to a web graphics system. For one, the translation may require assumptions about internal workings of another system, making it vulnerable to changes in that system. Moreover, a new interface may be designed to take advantage of *all* the features available in the web graphics system. In some cases, the custom interface can even be used to provide an elegant way to extend the functionality of a translation mechanism, as described in Extending `ggplotly()`.

An early attempt to design an R interface for general purpose interactive web graphics was the R package **rCharts** whose R interface is heavily inspired by **lattice** (Vaidyanathan 2013); (Sarkar 2008). The most innovative part of **rCharts** was its ability to interface with many different JavaScript graphing libraries using a single R interface. As the number of JavaScript graphing libraries began to explode, it became obvious this was not a sustainable model, as the R package must bundle each JavaScript library that it supports. However, a lot of the infrastructure, 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** (Vaidyanathan et al. 2015). 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, and most are designed for particular use cases, such as **leaflet** for geo-spatial mapping, **dygraphs** for time-series, and **networkD3** for networks (Cheng and Xie 2015); (Vanderkam and Allaire 2015); (Gandrud, Allaire, and Russell 2015)[10]. There are also HTML widgets that provide an interface to more general purpose visualization JavaScript libraries such as **plotly** and **rbokeh** (Sievert et al. 2016); (Hafen and team 2015).

Many **htmlwidgets** 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 sometimes linking multiple views (i.e., animation, brushing over points to highlight points in another view, etc). In some cases, this interactivity is handled entirely by the underlying JavaScript library, but **htmlwidgets** authors may also provide an R interface to custom JavaScript based logic to extend the interface's functionality. The R package

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

plotly uses this approach to enable features not found in plotly.js' JSON specification, such as linked views. To understand how it works, it helps to know about Model-View-Controller (MVC) paradigm, and the different ways to approach MVC in a web-based environment.

2.3.5 Multiple MVC paradigms

The Model-View-Controller (MVC) paradigm is a popular programming technique for designing graphical user interfaces with a minimal amount of dependencies which helps increase responsiveness. Responsiveness is absolutely crucial for interactive graphics since it greatly impacts our ability to make observations, draw generalizations, and generate hypotheses (Z. L. A. J. Heer 2014). In a MVC paradigm, the model contains all the data and logic necessary to generate view(s). The controller can be thought of as a component on top of a view that listens for certain events, and feeds those events to the model so that the model can update view(s) accordingly. As Figure 2.3 shows, modern R interfaces to interactive web graphics should have multiple model(s) that reside in different places (depending the event type).

For relatively simple interactions, like identification (scenario A/C) and resizing (scenario B) in Figure 2.3, the model should reside within the web browser. Many JavaScript libraries have a native MVC paradigm for manipulating a single graph, like drawing tooltips on mouse hover (scenario A), but there may be other useful manipulation events that the paradigm doesn't support. This could be result of the fact that the JavaScript library simply does not know about the R interface. For example, resizing should work consistently across various contexts (whether viewing in RStudio, **shiny** apps, **rmarkdown** documents, etc). Thankfully the **htmlwidgets** package provides a JavaScript library that triggers a `resize()` event whenever the window size changes within any of

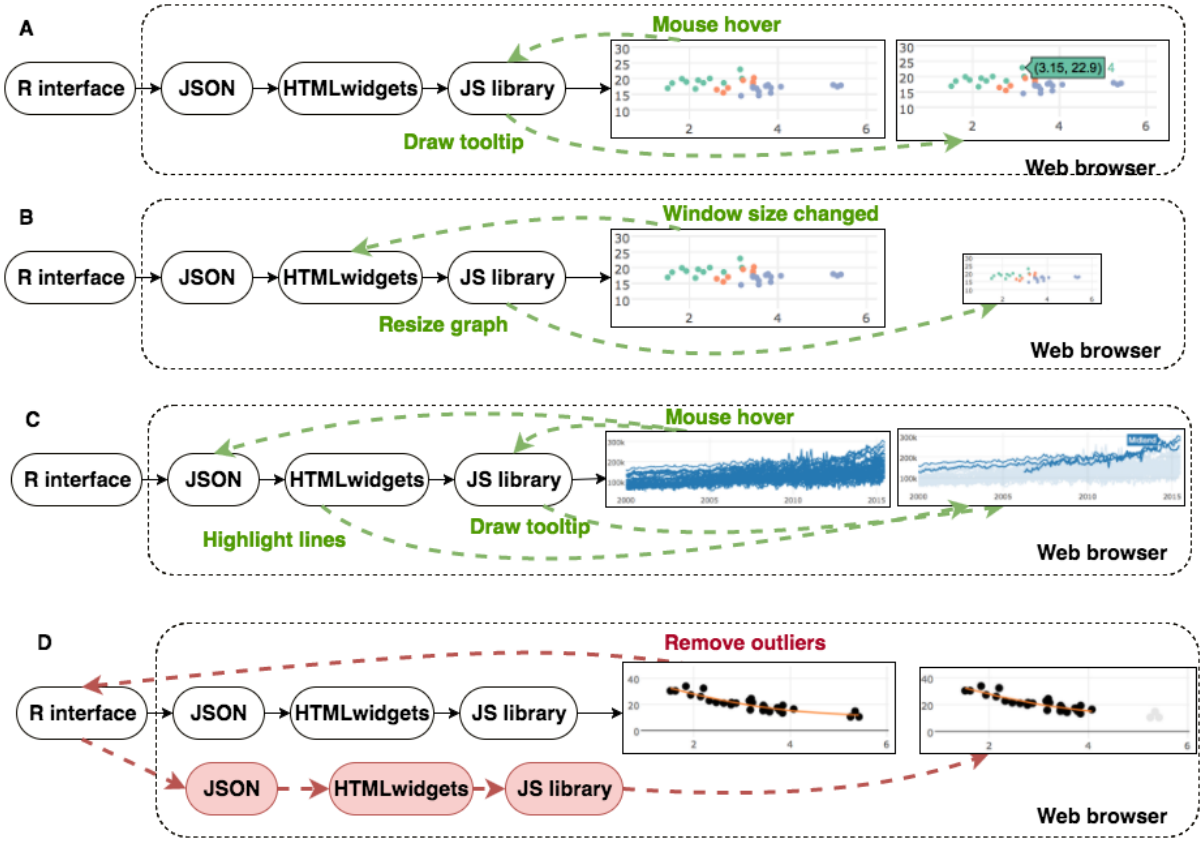


Figure 2.3 Four different MVC paradigms. In all the scenarios, the graph acts as the controller, but the model (i.e., the data and logic which updates the view) exists in different places. In Scenario A, a mouse hover event manipulates the model within the underlying JavaScript library. In Scenario B, a window resizing manipulates the model within the HTMLwidget.resize() method, defined by the widget author. In Scenario C, a mouse hover event manipulates the model within the underlying JavaScript library *and* a model defined by both the user (in R) and the widget author (in JavaScript). In Scenario D, removing outliers from the raw data may require R code to be executed.

these contexts – allowing the widget author to focus on the resizing model rather defining all the possible controllers.

For more complicated interactions, the model may have to be defined in R by the user, even when manipulating a single graph. Even when the model is defined in R, that does not necessarily imply the controller requires a hook back to re-execute R code. For example, scenario C in Figure 2.3 details a situation where the R user has specified that lines should be highlighted upon hover from the R interface, which is not natively supported by the underlying JavaScript graphing library, but can be implemented by the widget author by providing a framework which translates a model from R to JSON (+ custom JavaScript). In any case, the widget author should take special care not to trigger a full redraw of the plot, and do the minimal amount of work necessary to update the view. For example, in scenario C, the lines are highlighted by simply reducing the opacity of the non-selected lines. In fact, all of the examples in the section linking views without shiny follow this paradigm (scenario C).

A callback to re-execute R code is only necessary if the model requires information that is not already contained in the JSON object and JavaScript logic. So, for scenario D in Figure 2.3, it is theoretically possible to precompute linear models for every possible view and let the browser handle the MVC paradigm. However, this approach does not scale well, so it is often more practical to have the controller trigger callbacks to R. To provide access to a controller in **shiny**, htmlwidget authors need to leverage the JavaScript function `Shiny.onInputChange()` to inform the **shiny** server (i.e., the model) about a certain event. Here is a hypothetical and over-simplified example where `view` is an object representing a particular graph/view which has an `on()` method that triggers the execution of a function whenever a `"mouse_hover"` on this view happens.

```
view.on("mouse_hover", function(d) {
  Shiny.onInputChange("mouse_hover", d);
})
```

Users may then subscribe to the event in a **shiny** app by accessing the event in the server logic (a R function which allows one to access `input` values, manipulate them, and generate output). Assuming the data attached the event (`d`) simply contains the `id` property defined in `scatterplot()`, this is one way you could modify the color of a point by hovering on it.

```
function(input, output) {
  x <- 1:10
  y <- rnorm(10)
  output$view <- renderWidget({
    isSelected <- x == input$mouse_hover
    scatterplot(x, y, id = x, color = isSelected)
  })
}
```

As previously mentioned, this example is over-simplified for the purpose of demonstrating the basic approach to subscribing a controller in **shiny**. A proper implementation should also scope input values by view, so users can distinguish events from different views. The section on Targeting views gives an example of why scoping is important.

Recently, **shiny** used this approach to allows users to add controllers on top of static R graphics[11], effectively allowing developers to coordinate views in a web app, with no JavaScript involved. Although this is a useful tool, it is fundamentally limited, since updates to a static image will always require a full redraw. More specifically, within the

[11] 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

MVC paradigm context, static images require the entire model (the logic for updating the view) to reside on the server and very rarely can images be sent to the client fast enough to appear instantaneous.

The touring video in Figure 2.2 helps point out why these limitations matter. Notice that when a brush event occurs on the left hand panel (a static graphic), the entire view must be flushed and repopulated to provide a visual clue for the selection, resulting in a "glitch" effect. If that view was web-based, it would be possible to provide a visual clue without the glitch effect. This glitching effect would seriously limit the usefulness of the visualization if the right hand panel of Figure 2.2 also had to be flushed for every animation frame. Fortunately, that view is web-based, and can smoothly transition from one frame to the next when it receives new data from the R server, thanks to its hybrid MVC paradigm.

2.3.6 Hybrid MVC for one interface

Without a doubt, the MVC paradigm depicted in scenario D of Figure 2.3 provides a powerful foundation for interactive statistical graphics. In fact, if interactive web graphics are to replicate and extend classical approaches to the subject, they should follow this paradigm to leverage the statistical facilities that R provides, but *only* when it is necessary, since requiring the browser to communicate with an external R process is costly on multiple levels. That being said, assuming the additional infrastructure is required, it can be very useful to share semantics between R and JSON/JavaScript so that the browser can receive dynamically changing data from an R server and be smart and efficient when updating the view.

A great example of a hybrid MVC can be found in the R package **ggvis**, a reworking of ggplot2's grammar of graphics to incorporate interactivity (Chang and Wickham 2015). In **ggvis**, graphs are created with either an ordinary data frame or a special *reactive* data

frame (i.e., a reactive expression in **shiny** that outputs a data frame). The JavaScript model is aware of these special reactive data frames, and there mapping to visual space, so that when the JavaScript model receives new data, it can avoid a full redraw and only modify the visual properties for values that have changed. This not only enables updates from one view to the next to appear instaneous to the user, it also opens up the possibility of smoothly transitioning specific objects from one view to the next (i.e., interpolating between views). Smooth transitions help facilitate object constancy (i.e., tracking an object from one view to the next) which can improve graphical perception (Heer and Robertson 2007).

Another nice example of a hybrid MVC can be found in the R package **leaflet**, an interface to the popular JavaScript library for interactive maps. Although its hybrid MVC requires the map to be embedded within a **shiny** app, it allows one to modify graphical markings laid on top of a map without redrawing the map itself.

A hybrid MVC does not have to rely on **shiny**, and in fact, it can be used link multiple views without requiring a full redraw or R code to be re-executed. Furthermore, it can be implemented in such a way that is arguably easier to express from R compared to a full-blown **shiny** app. Figure 2.4 gives a high-level overview of how this works in the **plotly** package. Scenario B shows a situation where a brush event manipulates a model defined in a **shiny** app (via the `event_data()` function) which requires a full redraw. Scenario A shows a situation where a brush event directly manipulates a linked view via a **plotly**'s hybrid MVC model; and in this case, the JavaScript function `Plotly.restyle()` is used to simply modify the color the selected points in both views and reduce the opacity of non-selected points.

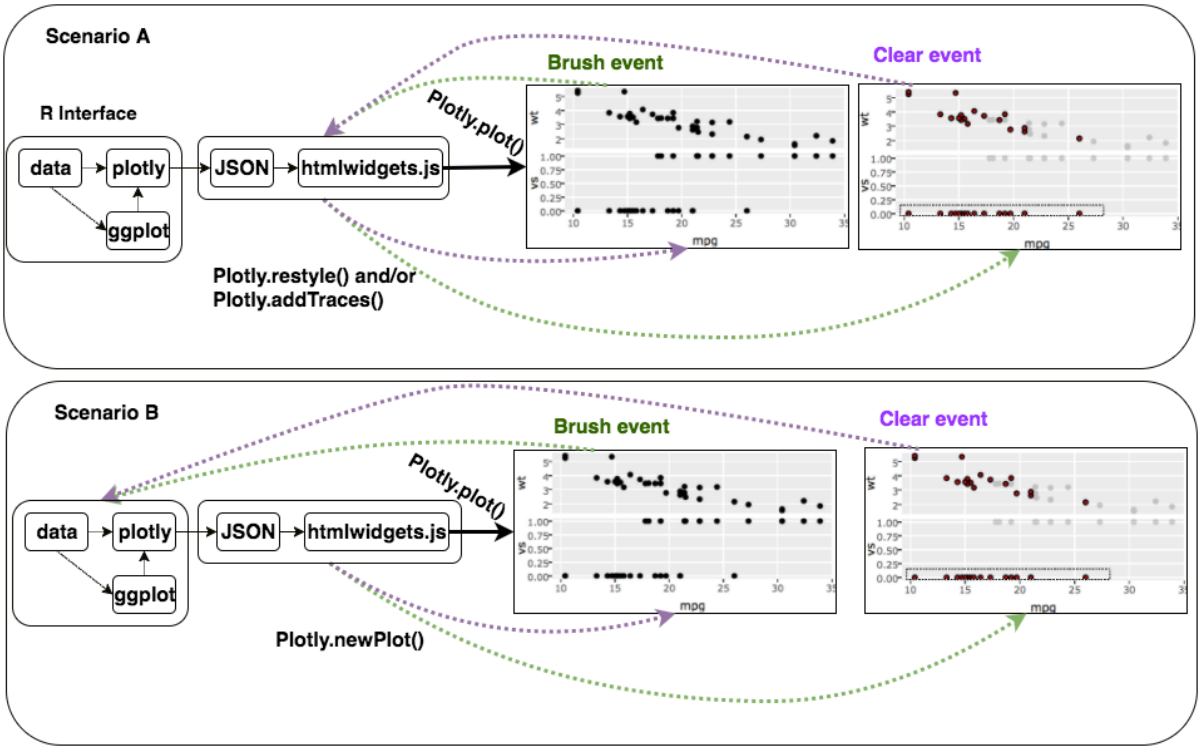


Figure 2.4 Linking views in plotly with shiny (scenario B) versus without shiny (scenario A).

2.3.7 MVC for multiple interfaces

As mentioned in Interfacing with interactive web graphics, there is now a diverse collection of R packages that interface with JavaScript graphing libraries. Since these interfaces generate web-based output, we may start to envision a hybrid MVC paradigm for linking views between different interfaces *without* a client-server infrastructure like **shiny**. For convenience, I will refer to this paradigm as a *centralized* pipeline, which is depicted on the right-hand side of Figure 2.5. Since centralized pipelines cannot dynamically re-execute R code, they are somewhat limited in their ability to dynamically transform raw data compared to data pipelines discussed by Andreas Buja and McDonald (1988) and Lawrence (2002). Figure 2.5 attempts to show this limitation in direct comparison to the GGobi pipeline, but also highlight the benefit of enabling linked views between arbitrary systems (and easily sharing them). As it turns out, a centralized pipeline is sufficient for

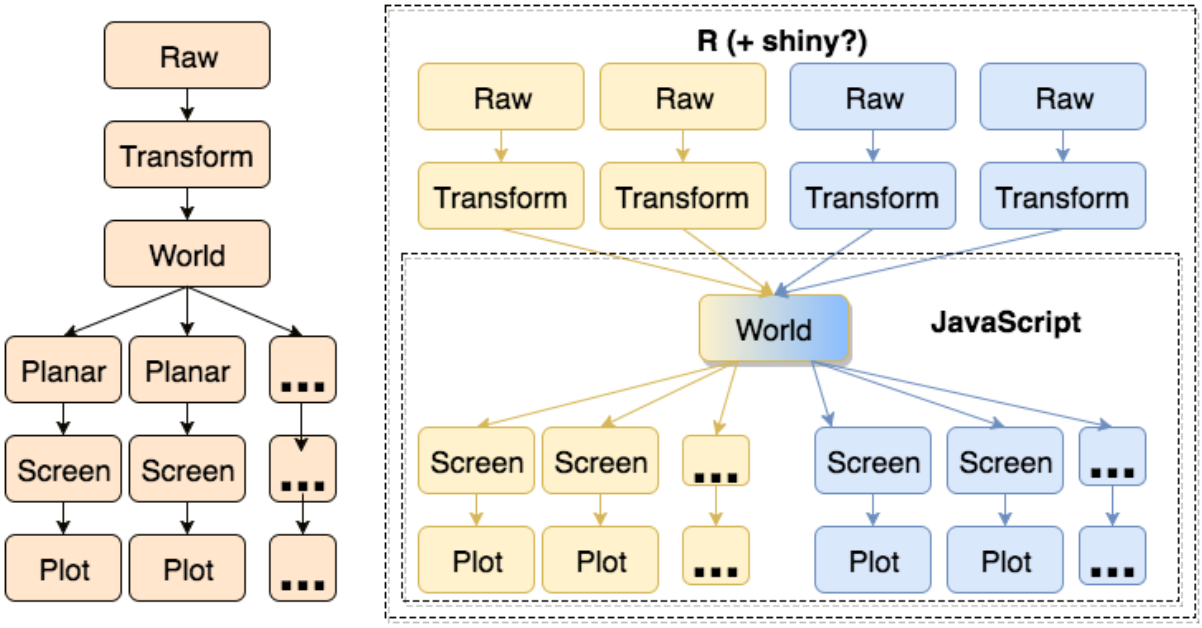


Figure 2.5 The GGobi pipeline, as described by Lawrence (2002), in comparison to a centralized pipeline. The GGobi pipeline is shown in peach color while the centralized pipeline is in both yellow and blue to point out multiple interfaces can be linked in a centralized pipeline.

creating a fairly large class of useful interactive graphics, and thanks to tools like **shiny**, users may still opt into a server-client infrastructure, if need be.

In a centralized pipeline, the first thing that is needed is a way to declare a link between the views from R. There are multiple ways to approach linking, but Figure 2.6 defines a link between views using a primary key relationship (in the data sets used to create each view), analogous to an approach first described by Buja et al. (1991). A primary key relationship is more flexible than a unique key in the sense that the column used to define the key can have multiple rows with the same value. This flexibility allows m-to-n linking (e.g. select m rows in one data set and n in another) to happen. For example, in Figure 2.6, a brush selects three cities, which is tied to 30 about rows in the data behind the parallel coordinate plot, but only 3 rows in the data behind the map. Assuming this primary key is translated to JSON, the client has *access to* enough information to update

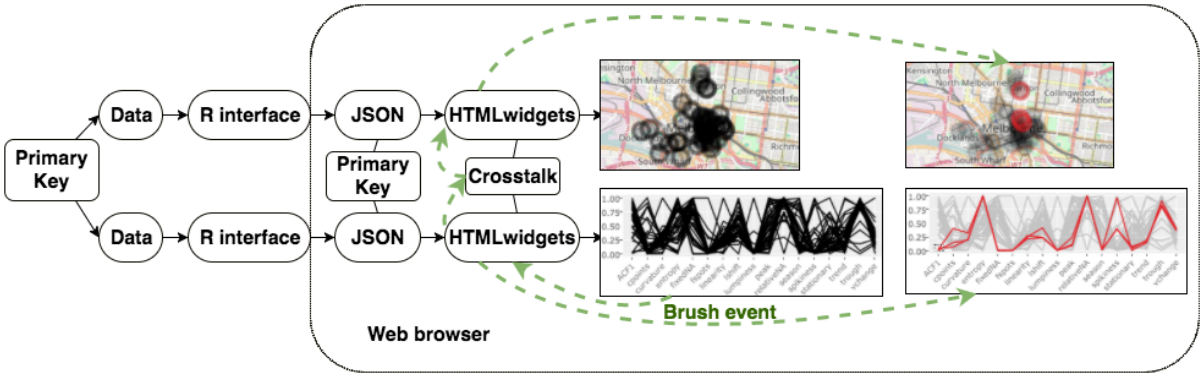


Figure 2.6 The MVC design for linking multiple htmlwidgets with crosstalk

the views – but the model (i.e., the logic for updating view(s)) has to be implemented separately by each interface (in JavaScript).

To enable linked views between multiple R interfaces, as in Figure 2.6, it helps to have standards for both defining a primary key from R (done by the user) and to set/access primary key selections in JavaScript (done by the developer). The new R package **crosstalk** proposes a `SharedData` class in R that allows one to add this primary key attribute to data frames (Cheng 2015a). It also provides a JavaScript library for setting, storing, and accessing selection values in the browser, but leaves the actual logic for updating views to widget authors. In a sense, this project is similar to the work of North and Shneiderman (1999), which provided standards for "snapping together" arbitrary views that are aware of a relational database schema, but **crosstalk** does so in a web-based environment, rather than requiring a machine running Windows.

The first HTML widget to leverage **crosstalk** was Cheng (2015b), but is currently limited to linked scatterplot brushing.[12] Currently, there are a couple other R packages with **crosstalk** support, including **leaflet** and **DT** (Xie 2015), but **plotly** is the only package with some support for the data pipeline, different selection modes (transient vs persistent selection), and different interaction types (e.g., mouse hover, click, and multiple types of click+drag selections). The pedestrians case study in plotly for R provides an example

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

of linking views between **leaflet** and **plotly** via **crosstalk** to quickly pose queries about high-dimensional data. The section linking views without shiny provides many examples designed to teach the reader how to use **plotly** and **crosstalk** to link views in numerous scenarios.

3 Scope

Explain your contributions to each project

https://github.com/ropensci/plotly/graphs/contributors

4 Taming PITCHf/x Data with XML2R and pitchRx

This chapter is a paper published in *The R Journal* (Sievert 2014b). I am the sole author of the paper which is avaliable online here https://journal.r-project.org/archive/2014-1/sievert.pdf

The formatting of paper has been modified to make for consistent typesetting across the thesis.

ABSTRACT

XML2R is a framework that reduces the effort required to transform XML content into tables in a way that preserves parent to child relationships. **pitchRx** applies **XML2R**'s grammar for XML manipulation to Major League Baseball Advanced Media (MLBAM)'s Gameday data. With **pitchRx**, one can easily obtain and store Gameday data in a remote database. The Gameday website hosts a wealth of XML data, but perhaps most interesting is PITCHf/x. Among other things, PITCHf/x data can be used to recreate a baseball's flight path from a pitcher's hand to home plate. With **pitchRx**, one can easily create animations and interactive 3D scatterplots of the baseball's flight path. PITCHf/x data is also commonly used to generate a static plot of baseball locations at the moment they cross home plate. These plots, sometimes called *strike-zone plots*, can also refer to a plot of event probabilities over the same region. **pitchRx** provides an easy and robust way to generate strike-zone plots using the **ggplot2** package.

4.1 Introduction

4.1.1 What is PITCHf/x?

PITCHf/x is a general term for a system that generates a series of 3D measurements of a baseball's path from a pitcher's hand to home plate (Alt and White 2008). [1] In an attempt to estimate the location of the ball at any time point, a quadratic regression model with nine parameters (defined by the equations of motion for constant linear

[1] A *pitcher* throws a ball to the opposing *batter*, who stands besides home plate and tries to hit the ball into the field of play.

acceleration) is fit to each pitch. Studies with access to the actual measurements suggest that this model is quite reasonable – especially for non-knuckleball pitches (Nathan 2008). That is, the fitted path often provides a reasonable estimate (within a couple of inches) of the actual locations. Unfortunately, only the parameter estimates are made available to the public. The website that provides these estimates is maintained by MLBAM and hosts a wealth of other baseball related data used to inform MLB's Gameday webcast service in near real time.

4.1.2 Why is PITCHf/x important?

On the business side of baseball, using statistical analysis to scout and evaluate players has become mainstream. When PITCHf/x was first introduced, DiMeo (2007) proclaimed it as,

> "The new technology that will change statistical analysis [of baseball] forever."

PITCHf/x has yet to fully deliver this claim, partially due to the difficulty in accessing and deriving insight from the large amount of complex information. By providing better tools to collect and visualize this data, **pitchRx** makes PITCHf/x analysis more accessible to the general public.

4.1.3 PITCHf/x applications

PITCHf/x data is and can be used for many different projects. It can also complement other baseball data sources, which poses an interesting database management problem. Statistical analysis of PITCHf/x data and baseball in general has become so popular that it has helped expose statistical ideas and practice to the general public. If you have witnessed television broadcasts of MLB games, you know one obvious application of PITCHf/x is locating pitches in the strike-zone as well as recreating flight trajecto-

ries, tracking pitch speed, etc. Some on-going statistical research related to PITCHf/x includes: classifying pitch types, predicting pitch sequences, and clustering pitchers with similar tendencies (Pane et al. 2013).

4.1.4 Contributions of pitchRx and XML2R

The **pitchRx** package has two main focuses (Sievert 2014a). The first focus is to provide easy access to Gameday data. Not only is **pitchRx** helpful for collecting this data in bulk, but it has served as a helpful teaching and research aide (http://baseballwithr.wordpress.com/ is one such example). Methods for collecting Gameday data existed prior to **pitchRx**; however, these methods are not easily extensible and require juggling technologies that may not be familiar or accessible (Fast 2007). Moreover, these working environments are less desirable than R for data analysis and visualization. Since **pitchRx** is built upon **XML2R**'s united framework, it can be easily modified and/or extended (Sievert 2014c). For this same reason, **pitchRx** serves as a model for building customized XML data collection tools with **XML2R**.

The other main focus of **pitchRx** is to simplify the process creating popular PITCHf/x graphics. Strike-zone plots and animations made via **pitchRx** utilize the extensible **ggplot2** framework as well as various customized options (Wickham 2009a). **ggplot2** is a convenient framework for generating strike-zone plots primarily because of its facet schema which allows one to make visual comparisons across any combination of discrete variable(s). Interactive 3D scatterplots are based on the **rgl** package and useful for gaining a new perspective on flight trajectories (Adler, Murdoch, and others 2016).

4.2 Getting familiar with Gameday

Gameday data is hosted and made available for free thanks to MLBAM via http://gd2.mlb.com/components/game/mlb/. [2] From this website, one can obtain many different types of data besides PITCHf/x. For example, one can obtain everything from structured media metadata to insider tweets. In fact, this website's purpose is to serve data to various http://mlb.com web pages and applications. As a result, some data is redundant and the format may not be optimal for statistical analysis. For these reasons, the `scrape` function is focused on retrieving data that is useful for PITCHf/x analysis and providing it in a convenient format for data analysis.

Navigating through the MLBAM website can be overwhelming, but it helps to recognize that a homepage exists for nearly every day and every game. For example, http://gd2.mlb.com/components/game/mlb/year_2011/month_02/day_26/ displays numerous hyperlinks to various files specific to February 26th, 2011. On this page is a hyperlink to miniscoreboard.xml which contains information on every game played on that date. This page also has numerous hyperlinks to game specific pages. For example, gid_2011_02_26_phimlb_nyamlb_1/ points to the homepage for that day's game between the NY Yankees and Philadelphia Phillies. On this page is a hyperlink to the players.xml file which contains information about the players, umpires, and coaches (positions, names, batting average, etc.) coming into that game.

Starting from a particular game's homepage and clicking on the inning/ directory, we *should* see another page with links to the inning_all.xml file and the inning_hit.xml file. If it is available, the `inning_all.xml` file contains the PITCHf/x data for that game. It's important to note that this file won't exist for some games, because some games are played in venues that do not have a working PITCHf/x system in place. This is

[2] Please be respectful of this service and store any information after you extract it instead of repeatedly querying the website. Before using any content from this website, please also read the copyright.

especially true for preseason games and games played prior to the 2008 season when the PITCHf/x system became widely adopted. [3] The `inning_hit.xml` files have manually recorded spatial coordinates of where a home run landed or where the baseball made initial contact with a defender after it was hit into play.

The relationship between these XML files and the tables returned by `scrape` is presented in Table 4.1. The `pitch` table is extracted from files whose name ends in `inning_all.xml`. This is the only table returned by `scrape` that contains data on the pitch-by-pitch level. The `atbat`, `runner`, `action` and `hip` tables from this same file are commonly labeled somewhat ambiguously as play-by-play data. The `player`, `coach`, and `umpire` tables are extracted from `players.xml` and are classified as game-by-game since there is one record per person per game. Figure 4.1 shows how these tables can be connected to one another in a database setting. The direction of the arrows represent a one to possibly many relationship. For example, at least one pitch is thrown for each *at bat* (that is, each bout between pitcher and batter) and there are numerous at bats within each game.

In a rough sense, one can relate tables returned by `scrape` back to XML nodes in the XML files. For convenience, some information in certain XML nodes are combined into one table. For example, information gleaned from the 'top', 'bottom', and 'inning' XML nodes within `inning_all.xml` are included as `inning` and `inning_side` fields in the `pitch`, `po`, `atbat`, `runner`, and `action` tables. This helps reduce the burden of merging many tables together in order to have inning information on the play-by-play and/or pitch-by-pitch level. Other information is simply ignored simply because it is redundant. For example, the 'game' node within the `players.xml` file contains information that can be recovered from the `game` table extracted from the `miniscoreboard.xml` file. If the reader wants a more detailed explanation of fields in these tables, Marchi and Albert (2013) provide nice overview.

[3] In this case, `scrape` will print "failed to load HTTP resource" in the R console (after the relevant file name) to indicate that no data was available.

Table 4.1 Structure of PITCHf/x and related Gameday data sources accessible to ‘scrape()‘

Source file	Information	XML nodes	Tables Returned
miniscoreboard.xml	game-by-game	games, game game_media, media	game, media
players.xml	game-by-game	game, team, player, coach, umpire	player, coach, umpire
inning_all.xml	play-by-play, pitch-by-pitch	game, inning, bottom, top, atbat, po, pitch, runner action	atbat, po, pitch, runner, action
inning_hit.xml	play-by-play	hitchart, hip	hip

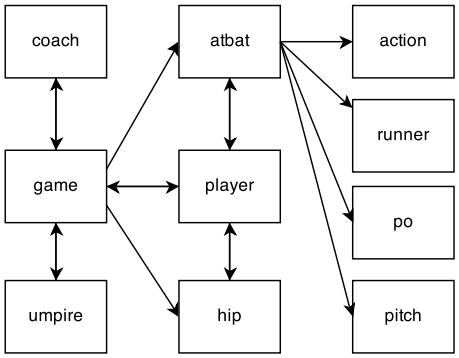


Figure 4.1 Table relations between Gameday data accessible via `scrape()`. The direction of the arrows indicate a one to possibly many relationship.

4.3 Introducing XML2R

XML2R adds to the CRAN Task View on Web Technologies and Services by focusing on the transformation of XML content into a collection of tables. Compared to a lower-level API like the **XML** package, it can significantly reduce the amount of coding and cognitive effort required to perform such a task. In contrast to most higher-level APIs, it does not make assumptions about the XML structure or its source. Although **XML2R** works on any structure, performance and user experience are enhanced if the content has an inherent relational structure. **XML2R**'s novel approach to XML data collection breaks down the transformation process into a few simple steps and allows the user to decide how to apply those steps.

The next few sections demonstrate how **pitchRx** leverages **XML2R** in order to produce a collection of tables from `inning_all.xml` files. A similar approach is used by `pitchRx::scrape` to construct tables from the other Gameday files in Table **??**. In fact, **XML2R** has also been used in the R package bbscrapeR which collects data from nba.com and wnba.com.

4.3.1 Constructing file names

Sometimes the most frustrating part of obtaining data in bulk off of the web is finding the proper collection of URLs or file names of interest. Since files on the Gameday website are fairly well organized, the `makeUrls` function can be used to construct `urls` that point to every game's homepage within a window of dates.

```
urls <- pitchRx::makeUrls(start = "2011-06-01", end = "2011-06-01")
sub("http://gd2.mlb.com/components/game/mlb/", "", head(urls))
#> [1] "year_2011/month_06/day_01/gid_2011_06_01_anamlb_kcamlb_1"
#> [2] "year_2011/month_06/day_01/gid_2011_06_01_balmlb_seamlb_1"
```

```
#> [3] "year_2011/month_06/day_01/gid_2011_06_01_chamlb_bosmlb_1"
#> [4] "year_2011/month_06/day_01/gid_2011_06_01_clemlb_tormlb_1"
#> [5] "year_2011/month_06/day_01/gid_2011_06_01_colmlb_lanmlb_1"
#> [6] "year_2011/month_06/day_01/gid_2011_06_01_flomlb_arimlb_1"
```

4.3.2 Extracting observations

Once we have a collection of XML `files`, the next step is to parse the content into a list of *observations*. An observation is technically defined as a matrix with one row and some number of columns. The columns are defined by XML attributes and the XML value (if any) for a particular XML lineage. The name of each observation tracks the XML hierarchy so observations can be grouped together in a sensible fashion at a later point.

```
library(XML2R)
files <- paste0(urls, "/inning/inning_all.xml")
obs <- XML2Obs(files, url.map = TRUE, quiet = TRUE)
table(names(obs))
#>
#>                                 game
#>                                   15
#>                           game//inning
#>                                  137
#>          game//inning//bottom//action
#>                                  116
#>           game//inning//bottom//atbat
#>                                  532
```

```
#>  game//inning//bottom//atbat//pitch
#>                                1978
#>     game//inning//bottom//atbat//po
#>                                  61
#> game//inning//bottom//atbat//runner
#>                                 373
#>           game//inning//top//action
#>                                 150
#>            game//inning//top//atbat
#>                                 593
#>     game//inning//top//atbat//pitch
#>                                2183
#>        game//inning//top//atbat//po
#>                                  75
#>    game//inning//top//atbat//runner
#>                                 509
#>                             url_map
#>                                   1
```

This output tells us that 247 pitches were thrown in the bottom inning and 278 were thrown in the top inning on June 1st, 2011. Also, there are 12 different levels of observations. The list element named `url_map` is not considered an observation and was included since `url.map = TRUE`. This helps avoid repeating long file names in the `url_key` column which tracks the mapping between observations and file names.

```
obs[1]
#> $`game//inning//top//atbat//pitch`
#>      des              des_es           id  type tfs
```

```
#> [1,] "Called Strike" "Strike cantado" "3" "S"  "201107"
#>      tfs_zulu               x        y        event_num
#> [1,] "2011-06-01T20:11:07Z" "103.00" "149.38" "3"
#>      sv_id           play_guid start_speed end_speed sz_top sz_bot
#> [1,] "110601_151108" ""        "94.0"      "86.1"    "2.85" "1.36"
#>      pfx_x   pfx_z  px       pz      x0       y0     z0      vx0
#> [1,] "-8.12" "11.0" "-0.143" "2.376" "-2.435" "50.0" "5.831" "9.058"
#>      vy0        vz0      ax        ay       az        break_y
#> [1,] "-137.334" "-7.288" "-15.446" "31.474" "-11.175" "23.8"
#>      break_angle break_length pitch_type type_confidence zone nasty
#> [1,] "46.3"      "4.0"        "FF"       ".865"          "2"  "46"
#>      spin_dir  spin_rate  cc mt url_key
#> [1,] "216.336" "2753.789" "" "" "url1"
```

4.3.3 Renaming observations

Before grouping observations into a collection tables based on their names, one may want to `re_name` observations. Observations with names `'game//inning//bottom//atbat'` and `'game//inning//top//atbat'` should be included in the same table since they share XML attributes (in other words, the observations share variables).

```
tmp <- re_name(obs, equiv = c("game//inning//top//atbat",
  "game//inning//bottom//atbat"), diff.name = "inning_side")
```

By passing these names to the `equiv` argument, `re_name` determines the difference in the naming scheme and suppresses that difference. In other words, observation names that match the `equiv` argument will be renamed to `'game//inning//atbat'`. The information removed from the name is not lost; however, as a new column is appended

to the end of each relevant observation. For example, notice how the `inning_side` column contains the part of the name we just removed:

```
tmp[grep("game//inning//atbat", names(tmp))][1:2]
#> $`game//inning//atbat`
#>      num b   s   o   start_tfs start_tfs_zulu         batter   stand
#> [1,] "1" "3" "2" "0" "201034"  "2011-06-01T20:10:34Z" "430947" "L"
#>      b_height pitcher  p_throws
#> [1,] "5-10"   "462956" "R"
#>      des
#> [1,] "Erick Aybar singles on a line drive to center fielder Melky Cabrera.  "
#>      des_es
#> [1,] "Erick Aybar pega sencillo con línea a jardinero central Melky Cabrera.  "
#>      event_num event    event_es   home_team_runs away_team_runs
#> [1,] "12"      "Single" "Sencillo" "0"            "0"
#>      url_key inning_side
#> [1,] "url1"  "top"
#>
#> $`game//inning//atbat`
#>      num b   s   o   start_tfs start_tfs_zulu         batter   stand
#> [1,] "2" "2" "3" "1" "201412"  "2011-06-01T20:14:12Z" "110029" "L"
#>      b_height pitcher  p_throws
#> [1,] "6-0"    "462956" "R"
#>      des
#> [1,] "Bobby Abreu called out on strikes.  "
#>      des_es                                 event_num event
#> [1,] "Bobby Abreu se poncha sin tirarle.  " "24"      "Strikeout"
#>      event_es home_team_runs away_team_runs url_key inning_side
```

```
#> [1,] "Ponche" "0"              "0"              "url1"  "top"
```

For similar reasons, other observation name pairs are renamed in a similar fashion.

```
tmp <- re_name(tmp, equiv = c("game//inning//top//atbat//runner",
  "game//inning//bottom//atbat//runner"), diff.name = "inning_side")
tmp <- re_name(tmp, equiv = c("game//inning//top//action",
  "game//inning//bottom//action"), diff.name = "inning_side")
tmp <- re_name(tmp, equiv = c("game//inning//top//atbat//po",
  "game//inning//bottom//atbat//po"), diff.name = "inning_side")
obs2 <- re_name(tmp, equiv = c("game//inning//top//atbat//pitch",
  "game//inning//bottom//atbat//pitch"), diff.name = "inning_side")
table(names(obs2))
#>
#>                          game                 game//inning
#>                            15                          137
#>         game//inning//action          game//inning//atbat
#>                           266                         1125
#>  game//inning//atbat//pitch      game//inning//atbat//po
#>                          4161                          136
#> game//inning//atbat//runner                      url_map
#>                           882                            1
```

4.3.4 Linking observations

After all that renaming, we now have 7 different levels of observations. Let's examine the first three observations on the `game//inning` level:

```
obs2[grep("^game//inning$", names(obs2))][1:3]
#> $`game//inning`
#>      num away_team home_team next url_key
#> [1,] "1" "ana"     "kca"     "Y"  "url1"
#>
#> $`game//inning`
#>      num away_team home_team next url_key
#> [1,] "2" "ana"     "kca"     "Y"  "url1"
#>
#> $`game//inning`
#>      num away_team home_team next url_key
#> [1,] "3" "ana"     "kca"     "Y"  "url1"
```

Before grouping observations into tables, it is usually important preserve the parent-to-child relationships in the XML lineage. For example, one may want to map a particular pitch back to the inning in which it was thrown. Using the `add_key` function, the relevant value of `num` for `game//inning` observations can be `recycled` to its XML descendants.

```
obswkey <- add_key(obs2, parent = "game//inning", recycle = "num", key.name = "inning
```

As it turns out, the `away_team` and `home_team` columns are redundant as this information is embedded in the `url` column. Thus, there is only one other informative attribute on this level which is `next`. By recycling this value among its descendants, we remove any need to retain a `game//inning` table.

```
obswkey <- add_key(obswkey, parent = "game//inning", recycle = "next")
```

It is also imperative that we can link a `pitch`, `runner`, or `po` back to a particular `atbat`. This can be done as follows:

```
obswkey <- add_key(obswkey, parent = "game//inning//atbat", recycle = "num")
```

4.3.5 Collapsing observations

Finally, we are in a position to pool together observations that have a common name. The `collapse_obs` function achieves this by row binding observations with the same name together and returning a list of matrices. Note that `collapse_obs` does not require that observations from the same level to have the same set of variables in order to be bound into a common table. In the case where variables are missing, `NA`s will be inserted as values.

```
tables <- collapse_obs(obswkey)
#As mentioned before, we do not need the 'inning' table
tables <- tables[!grepl("^game//inning$", names(tables))]
table.names <- c("game", "action", "atbat", "pitch", "po", "runner")
tables <- setNames(tables, table.names)
head(tables[["runner"]])
#>      id       start end  event                event_num url_key
#> [1,] "430947" ""    "1B" "Single"             "12"      "url1"
#> [2,] "430947" "1B"  "2B" "Stolen Base 2B"     "19"      "url1"
#> [3,] "430947" "2B"  "3B" "Groundout"          "30"      "url1"
#> [4,] "430947" "3B"  ""   "Groundout"          "36"      "url1"
#> [5,] "543333" ""    "1B" "Single"             "58"      "url1"
#> [6,] "543333" "1B"  ""   "Pickoff Attempt 1B" "69"      "url1"
#>      inning_side inning next num score rbi earned
#> [1,] "top"       "1"    "Y"  "1" NA    NA  NA
#> [2,] "top"       "1"    "Y"  "2" NA    NA  NA
```

```
#> [3,] "top"       "1"    "Y"  "3" NA    NA  NA
#> [4,] "top"       "1"    "Y"  "4" NA    NA  NA
#> [5,] "bottom"    "1"    "Y"  "7" NA    NA  NA
#> [6,] "bottom"    "1"    "Y"  "8" NA    NA  NA
```

4.4 Collecting Gameday data with pitchRx

The main scraping function in **pitchRx**, `scrape`, can be used to easily obtain data from the files listed in Table **??**. In fact, any combination of these files can be queried using the `suffix` argument. In the example below, the `start` and `end` arguments are also used so that all available file types for June 1st, 2011 are queried.

```
library(pitchRx)
files <- c("inning/inning_all.xml", "inning/inning_hit.xml",
  "miniscoreboard.xml", "players.xml")
dat <- scrape(start = "2011-06-01", end = "2011-06-01", suffix = files)
```

The `game.ids` option can be used instead of `start` and `end` to obtain an equivalent `dat` object. This option can be useful if the user wants to query specific games rather than all games played over a particular time span. When using this `game.ids` option, the built-in `gids` object, is quite convenient.

```
data(gids, package = "pitchRx")
gids11 <- gids[grep("2011_06_01", gids)]
head(gids11)
#> [1] "gid_2011_06_01_anamlb_kcamlb_1" "gid_2011_06_01_balmlb_seamlb_1"
#> [3] "gid_2011_06_01_chamlb_bosmlb_1" "gid_2011_06_01_clemlb_tormlb_1"
#> [5] "gid_2011_06_01_colmlb_lanmlb_1" "gid_2011_06_01_flomlb_arimlb_1"
```

```
dat <- scrape(game.ids = gids11, suffix = files)
```

The object `dat` is a list of data frames containing all data available for June 1st, 2011 using `scrape`. The list names match the table names provided in Table **??**. For example, `dat$atbat` is data frame with every at bat on June 1st, 2011 and `dat$pitch` has information related to the outcome of each pitch (including PITCHf/x parameters). The `object.size` of `dat` is nearly 300MB. Multiplying this number by 100 days exceeds the memory of most machines. Thus, if a large amount of data is required, the user should exploit the R database interface (R Special Interest Group on Databases 2013).

4.5 Storing and querying Gameday data

Since PITCHf/x data can easily exhaust memory, one should consider establishing a database instance before using `scrape`. By passing a database connection to the `connect` argument, `scrape` will try to create (and/or append to existing) tables using that connection. If the connection fails for some reason, tables will be written as csv files in the current working directory. The benefits of using the `connect` argument includes improved memory management which can greatly reduce run time. `connect` will support a MySQL connection, but creating a SQLite database is quite easy with **dplyr** (Wickham and Francois 2014).

```
library(dplyr)
db <- src_sqlite("GamedayDB.sqlite3", create = TRUE)
# Collect and store all PITCHf/x data from 2008 to now
scrape(start = "2008-01-01", end = Sys.Date(),
  suffix = "inning/inning_all.xml", connect = db$con)
```

In the later sections, animations of four-seam and cut fastballs thrown by Mariano Rivera and Phil Hughes during the 2011 season are created. In order to obtain the data for those animations, one could query `db` which now has PITCHf/x data from 2008 to date. This query requires criteria on: the `pitcher_name` field (in the `atbat` table), the `pitch_type` field (in the `pitch` table), and the `date` field (in both tables). To reduce the time required to search those records, one should create an index on each of these three fields.

```
library(DBI)
dbSendQuery(db$con, "CREATE INDEX pitcher_index ON atbat(pitcher_name)")
dbSendQuery(db$con, "CREATE INDEX type_index ON pitch(pitch_type)")
dbSendQuery(db$con, "CREATE INDEX date_atbat ON atbat(date)")
```

As a part of our query, we'll have to join the `atbat` table together with the `pitch` table. For this task, the `gameday_link` and `num` fields are helpful since together they provide a way to match pitches with at bats. For this reason, a multi-column index on the `gameday_link` and `num` fields will further reduce run time of the query.

```
dbSendQuery(db$con, 'CREATE INDEX pitch_join ON pitch(gameday_link, num)')
dbSendQuery(db$con, 'CREATE INDEX atbat_join ON atbat(gameday_link, num)')
```

Although the query itself could be expressed entirely in SQL, **dplyr**'s grammar for data manipulation (which is database agnostic) can help to simplify the task. In this case, `at.bat` is a tabular *representation* of the remote `atbat` table restricted to cases where Rivera or Hughes was the pitcher. That is, `at.bat` does not contain the actual data, but it does contain the information necessary to retrieve it from the database.

```
at.bat <- tbl(db, "atbat") %>%
  filter(pitcher_name %in% c("Mariano Rivera", "Phil Hughes"))
```

Similarly, `fbs` is a tabular representation of the `pitch` table restricted to four-seam (FF) and cut fastballs (FC).

```
fbs <- tbl(db, "pitch") %>%
  filter(pitch_type %in% c("FF", "FC"))
```

An `inner_join` of these two filtered tables returns a tabular representation of all four-seam and cut fastballs thrown by Rivera and Hughes. Before `collect` actually performs the database query and brings the relevant data into the R session, another restriction is added so that only pitches from 2011 are included.

```
pitches <- inner_join(fbs, at.bat) %>%
  filter(date >= "2011_01_01" & date <= "2012_01_01") %>%
  collect()
```

4.6 Visualizing PITCHf/x

4.6.1 Strike-zone plots and umpire bias

Amongst the most common PITCHf/x graphics are strike-zone plots. Such a plot has two axes and the coordinates represent the location of baseballs as they cross home plate. The term strike-zone plot can refer to either *density* or *probabilistic* plots. Density plots are useful for exploring what *actually* occurred, but probabilistic plots can help address much more interesting questions using statistical inference. Although probabilistic plots can be used to visually track any event probability across the strike-zone, their most popular use is for addressing umpire bias in a strike versus ball decision (Green and Daniels 2014). The probabilistic plots section demonstrates how **pitchRx** simplifies the process behind creating such plots via a case study on the impact of home field advantage on umpire decisions.

In the world of sports, it is a common belief that umpires (or referees) have a tendency to favor the home team. PITCHf/x provides a unique opportunity to add to this dis-

cussion by modeling the probability of a called strike at home games versus away games. Specifically, conditioned upon the umpire making a decision at a specific location in the strike-zone, if the probability that a home pitcher receives a called strike is higher than the probability that an away pitcher receives a called strike, then there is evidence to support umpire bias towards a home pitcher.

There are many different possible outcomes of each pitch, but we can condition on the umpire making a decision by limiting to the following two cases. A *called strike* is an outcome of a pitch where the batter does not swing and the umpire declares the pitch a strike (which is a favorable outcome for the pitcher). A *ball* is another outcome where the batter does not swing and the umpire declares the pitch a ball (which is a favorable outcome for the batter). All `decisions` made between 2008 and 2013 can be obtained from `db` with the following query using **dplyr**.

```
# First, add an index on the pitch description to speed up run-time
dbSendQuery(db$con, "CREATE INDEX des_index ON pitch(des)")

pitch <- tbl(db, "pitch") %>%
  filter(des %in% c("Called Strike", "Ball")) %>%
  # Keep pitch location, descriptions
  select(px, pz, des, gameday_link, num) %>%
  # 0-1 indicator of strike/ball
  mutate(strike = as.numeric(des == "Called Strike"))

atbat <- tbl(db, "atbat") %>%
  # Select variables to be used later as covariates in probabilistic models
  select(b_height, p_throws, stand, inning_side, date, gameday_link, num)
```

```
decisions <- inner_join(pitch, atbat) %>%
  filter(date <= "2014_01_01") %>%
  collect()
```

4.6.1.1 Density plots

The `decisions` data frame contains data on over 2.5 million pitches thrown from 2008 to 2013. About a third of them are called strikes and two-thirds balls. Figure 4.2 shows the density of all called strikes. Clearly, most called strikes occur on the outer region of the strike-zone. Many factors could contribute to this phenomenon; which we will not investigate here.

```
# strikeFX uses the stand variable to calculate strike-zones
# Here is a slick way to create better facet titles without changing data values
relabel <- function(variable, value) {
  value <- sub("^R$", "Right-Handed Batter", value)
  sub("^L$", "Left-Handed Batter", value)
}
strikes <- subset(decisions, strike == 1)
strikeFX(strikes, geom = "raster", layer = facet_grid(. ~ stand, labeller = relabel))
```

Figure 4.2 shows one static rectangle (or strike-zone) per plot automatically generated by `strikeFX`. The definition of the strike-zone is notoriously ambiguous. As a result, the boundaries of the strike-zone may be noticeably different in some situations. However, we can achieve a fairly accurate representation of strike-zones using a rectangle defined by batters' average height and stance (Fast 2011). As Figure 4.4 reinforces, batter stance makes an important difference since the strike-zone seems to be horizontally shifted away

Figure 4.2 Density of called strikes for right-handed batters and left-handed batters (from 2008 to 2013).

from the batter. The batter's height is also important since the strike-zone is classically defined as approximately between the batter's knees and armpits.

Figure 4.2 has is one strike-zone per plot since the `layer` option contains a **ggplot2** argument that facets according to batter stance. Facet layers are a powerful tool for analyzing PITCHf/x data because they help produce quick and insightful comparisons. In addition to using the `layer` option, one can add layers to a graphic returned by `strikeFX` using **ggplot2** arithmetic. It is also worth pointing out that Figure 4.2 could have been created without introducing the `strikes` data frame by using the `density1` and `density2` options.

```
strikeFX(decisions, geom = "raster", density1 = list(des = "Called Strike"),
  density2 = list(des = "Called Strike")) + facet_grid(. ~ stand, labeller = relabel)
```

In general, when `density1` and `density2` are identical, the result is equivalent to subsetting the data frame appropriately beforehand. More importantly, by specifying *different* values for `density1` and `density2`, differenced densities are easily generated. In this case, a grid of density estimates for `density2` are subtracted from the corresponding grid of density estimates for `density1`. Note that the default `NULL` value for either density option infers that the entire data set defines the relevant density. Thus, if `density2` was `NULL` (when `density1 = list(des = 'Called Strike')`), we would obtain the density of called strikes minus the density of *both* called strikes and balls. In Figure 4.3, we define `density1` as called strikes and define `density2` as balls. As expected, we see positive density values (in blue) inside the strike-zone and negative density values (in red) outside of the strike-zone.

```
strikeFX(decisions, geom = "raster", density1 = list(des = "Called Strike"),
  density2 = list(des = "Ball"), layer = facet_grid(. ~ stand, labeller = relabel))
```

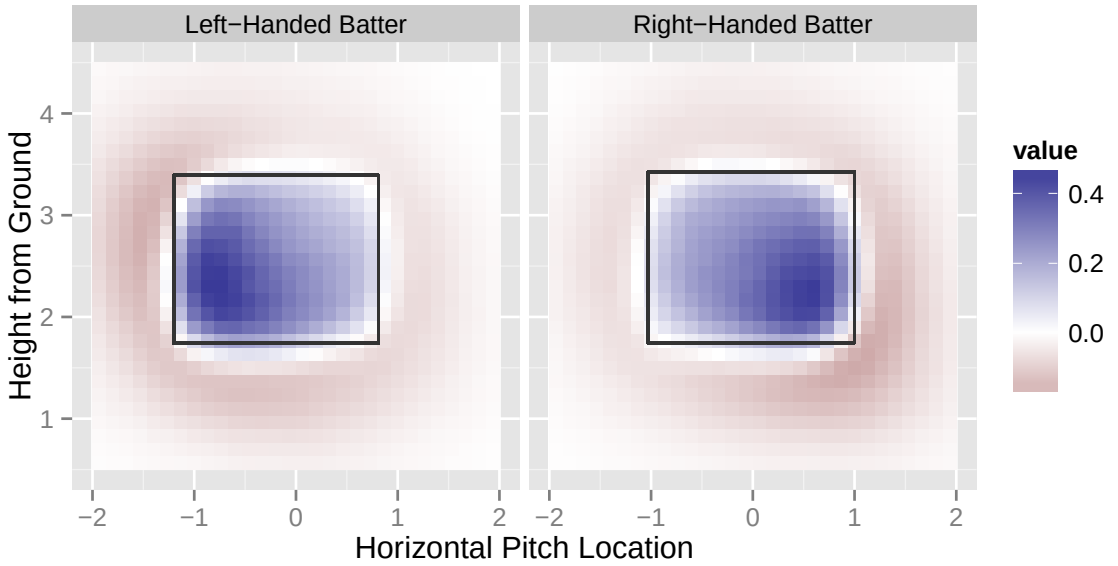


Figure 4.3 Density of called strikes minus density of balls for both right-handed batters and left-handed batters (from 2008 to 2013). The blue region indicates a higher frequency of called strikes and the red region indicates a higher frequency of balls.

These density plots are helpful for visualizing the observed frequency of events; however, they are not very useful for addressing our umpire bias hypothesis. Instead of looking simply at the *density*, we want to model the *probability* of a strike called at each coordinate given the umpire has to make a decision.

4.6.1.2 Probabilistic plots

There are many approaches to probabilistic modeling over a two dimensional spatial region. Since our response is often categorical, generalized additive models (GAMs) is a popular and desirable approach to modeling events over the strike-zone (Mills 2010). There are numerous R package implementations of GAMs, but the `bam` function from the **mgcv** package has several desirable properties (Wood 2006). Most importantly, the smoothing parameter can be estimated using several different methods. In order to have

a reasonable estimate of the smooth 2D surface, GAMs require fairly large amount of observations. As a result, run time can be an issue – especially when modeling 2.5 million observations! Thankfully, the `bam` function has a `cluster` argument which allows one to distribute computations across multiple cores using the built in **parallel** package.

```
library(parallel)
cl <- makeCluster(detectCores() - 1)
library(mgcv)
m <- bam(strike ~ interaction(stand, p_throws, inning_side) +
  s(px, pz, by = interaction(stand, p_throws, inning_side)),
  data = decisions, family = binomial(link = 'logit'), cluster = cl)
```

This formula models the probability of a strike as a function of the baseball's spatial location, the batter's stance, the pitcher's throwing arm, and the side of the inning. Since home pitchers always pitch during the top of the inning, `inning_side` also serves as an indication of whether a pitch is thrown by a home pitcher. In this case, the `interaction` function creates a factor with eight different levels since every input factor has two levels. Consequently, there are 8 different levels of smooth surfaces over the spatial region defined by `px` and `pz`.

The fitted model `m` contains a lot of information which `strikeFX` uses in conjunction with any **ggplot2** facet commands to infer which and how surfaces should be plotted. In particular, the `var.summary` is used to identify model covariates, as well their default conditioning values. In our case, the majority of `decisions` are from right-handed pitchers and the top of the inning. Thus, the default conditioning values are `"top"` for `inning_side` and `"R"` for `p_throws`. If different conditioning values are desired, `var.summary` can be modified accordingly. To demonstrate, Figure 4.4 shows 2 of the 8 possible surfaces that correspond to a right-handed *away* pitcher.

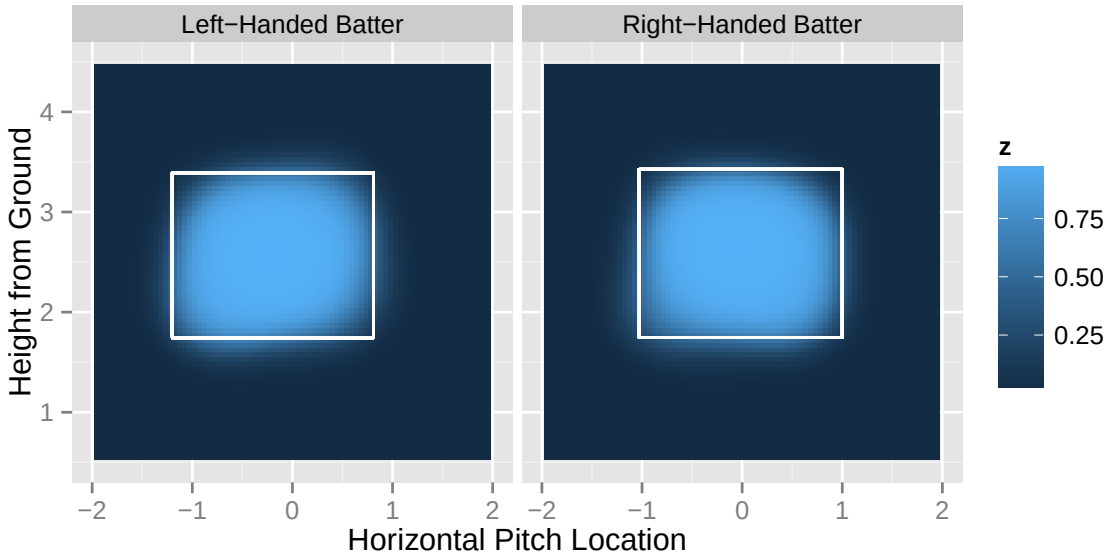


Figure 4.4 Probability that a right-handed away pitcher receives a called strike (provided the umpire has to make a decision). Plots are faceted by the handedness of the batter.

```
away <- list(inning_side = factor("bottom", levels = c("top", "bottom")))
m$var.summary <- modifyList(m$var.summary, away)
strikeFX(decisions, model = m, layer = facet_grid(. ~ stand, labeller = relabel))
```

Using the same intuition exploited earlier to obtain differenced density plots, we can easily obtain differenced probability plots. To obtain Figure 4.5, we simply add `p_throws` as another facet variable and `inning_side` as a differencing variable. In this case, conditioning values do not matter since every one of the 8 surfaces are required in order to produce Figure 4.5.

```
# Function to create better labels for both stand and p_throws
relabel2 <- function(variable, value) {
  if (variable %in% "stand")
    return(sub("^L$", "Left-Handed Batter",
      sub("^R$", "Right-Handed Batter", value)))
```

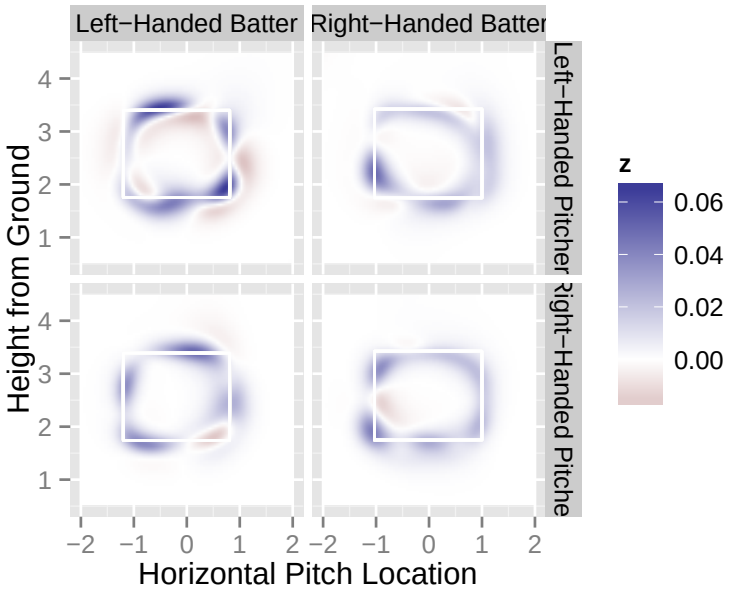


Figure 4.5 Difference between home and away pitchers in the probability of a strike (provided the umpire has to make a decision). The blue regions indicate a higher probability of a strike for home pitchers and red regions indicate a higher probability of a strike for away pitchers. Plots are faceted by the handedness of both the pitcher and the batter.

```
  if (variable %in% "p_throws")
    return(sub("^L$", "Left-Handed Pitcher",
      sub("^R$", "Right-Handed Pitcher", value)))
}
strikeFX(decisions, model = m, layer = facet_grid(p_throws ~ stand, labeller = relabe
  density1 = list(inning_side = "top"), density2 = list(inning_side = "bottom"))
```

The four different plots in Figure 4.5 represent the four different combination of values among `p_throws` and `stand`. In general, provided that a pitcher throws to a batter in the blue region, the pitch is more likely to be called a strike if the pitcher is on their home turf. Interestingly, there is a well-defined blue elliptical band around the boundaries of the typical strike-zone. Thus, home pitchers are more likely to receive a favorable call – especially when the classification of the pitch is in question. In some areas, the

home pitcher has up to a 6 percent higher probability of receiving a called strike than an away pitcher. The subtle differences in spatial patterns across the different values of `p_throws` and `stand` are interesting as well. For instance, pitching at home has a large positive impact for a left-handed pitcher throwing in the lower inside portion of the strike-zone to a right-handed batter, but the impact seems negligible in the mirror opposite case. Differenced probabilistic densities are clearly an interesting visual tool for analyzing PITCHf/x data. With `strikeFX`, one can quickly and easily make all sorts of visual comparisons for various situations. In fact, one can explore and compare the probabilistic structure of any well-defined event over a strike-zone region (for example, the probability a batter reaches base) using a similar approach.

4.6.2 2D animation

`animateFX` provides convenient and flexible functionality for animating the trajectory of any desired set of pitches. For demonstration purposes, this section animates every four-seam and cut fastball thrown by Mariano Rivera and Phil Hughes during the 2011 season. These pitches provide a good example of how facets play an important role in extracting new insights. Similar methods can be used to analyze any MLB player (or combination of players) in greater detail.

`animateFX` tracks three dimensional pitch locations over a sequence of two dimensional plots. The animation takes on the viewpoint of the umpire; that is, each time the plot refreshes, the balls are getting closer to the viewer. This is reflected with the increase in size of the points as the animation progresses. Obviously, some pitches travel faster than others, which explains the different sizes within a particular frame. Animations revert to the initial point of release once *all* of the baseballs have reached home plate. During an interactive session, `animateFX` produces a series of plots that may not viewed easily.

One option available to the user is to wrap `animation::saveHTML` around `animateFX` to view the animation in a browser with proper looping controls (Xie 2013a).

To reduce the time and thinking required to produce these animations, `animateFX` has default settings for the geometry, color, opacity and size associated with each plot. Any of these assumptions can be altered - except for the point geometry. In order for animations to work, a data frame with the appropriately named PITCHf/x parameters (that is, x0, y0, z0, vx0, vy0, vz0, ax0, ay0 and az0) is required. In Figure 4.6, every four-seam and cut fastball thrown by Rivera and Hughes during the 2011 season is visualized using the `pitches` data frame obtained earlier (the animation is available at http://cpsievert.github.io/pitchRx/ani1).

```
animateFX(pitches, layer=list(theme_bw(), coord_equal(),
  facet_grid(pitcher_name~stand, labeller = relabel)))
```

In the animation corresponding to Figure 4.6, the upper right-hand portion (Rivera throwing to right-handed batters) reveals the clearest pattern in flight trajectories. Around the point of release, Rivera's two pitch types are hard to distinguish. However, after a certain point, there is a very different flight path among the two pitch types. Specifically, the drastic left-to-right movement of the cut fastball is noticeably different from the slight right-to-left movement of the four-seam fastball. In recent years, cut fastballs have gained notoriety among the baseball community as a coveted pitch for pitchers have at their disposal. This is largely due to the difficulty that a batter has in distinguishing the cut fastball from another fastball as the ball travels toward home plate. Clearly, this presents an advantage for the pitcher since they can use deception to reduce batter's ability to predict where the ball will cross home plate. This deception factor combined with Rivera's ability to locate his pitches explain his accolades as one of the greatest pitchers of all time (Traub 2010).

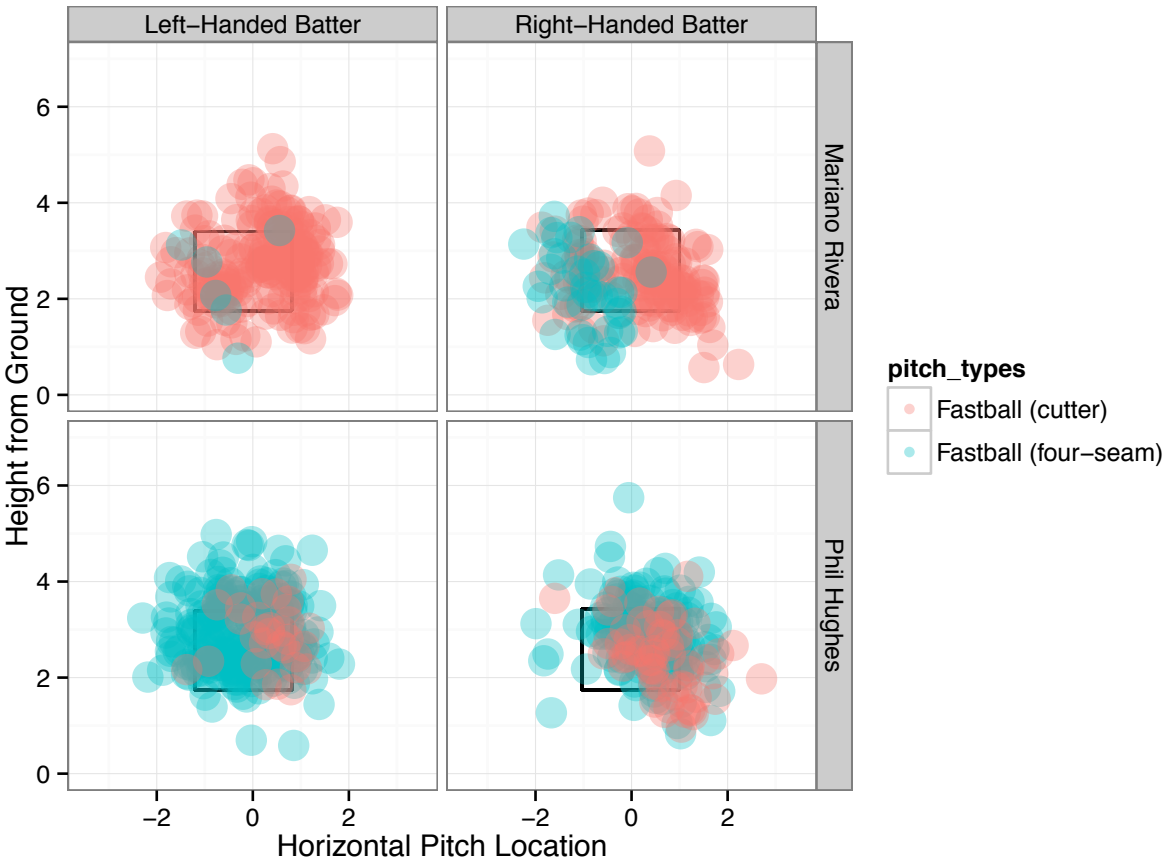


Figure 4.6 The last frame of an animation of every four-seam and cutting fastballs thrown by NY Yankee pitchers Mariano Rivera and Phil Hughes during the 2011 season. The actual animation can be viewed at http://cpsievert.github.io/pitchRx/ani1. Pitches are faceted by pitcher and batting stance. For instance, the top left plot portrays pitches thrown by Rivera to left-handed batters.

Although we see a clear pattern in Rivera's pitches, MLB pitchers are hardly ever that predictable. Animating that many pitches for another pitcher can produce a very cluttered graphic which is hard to interpret (especially when many pitch types are considered). However, we may still want to obtain an indication of pitch trajectory over a set of many pitches. A way to achieve this is to average over the PITCHf/x parameters to produce an overall sense of pitch type behavior (via the `avg.by` option). Note that the facet variables are automatically considered indexing variables. That is, in Figure 4.7, there are eight 'average' pitches since there are two pitch types, two pitchers, and two types of batting stance (the animation is available at http://cpsievert.github.io/pitchRx/ani2).

```
animateFX(pitches, avg.by = "pitch_types", layer = list(coord_equal(), theme_bw(),
  facet_grid(pitcher_name~stand, labeller = relabel)))
```

4.6.3 Interactive 3D graphics

rgl is an R package that utilizes OpenGL for graphics rendering. `interactiveFX` utilizes **rgl** functionality to reproduce flight paths on an interactive 3D platform. Figure 4.8 has two static pictures of Mariano Rivera's 2011 fastballs on this interactive platform. This is great for gaining new perspectives on a certain set of pitches, since the trajectories can be viewed from any angle. Figure 4.8 showcases the difference in trajectory between Rivera's pitch types.

```
Rivera <- subset(pitches, pitcher_name == "Mariano Rivera")
interactiveFX(Rivera, avg.by = "pitch_types")
```

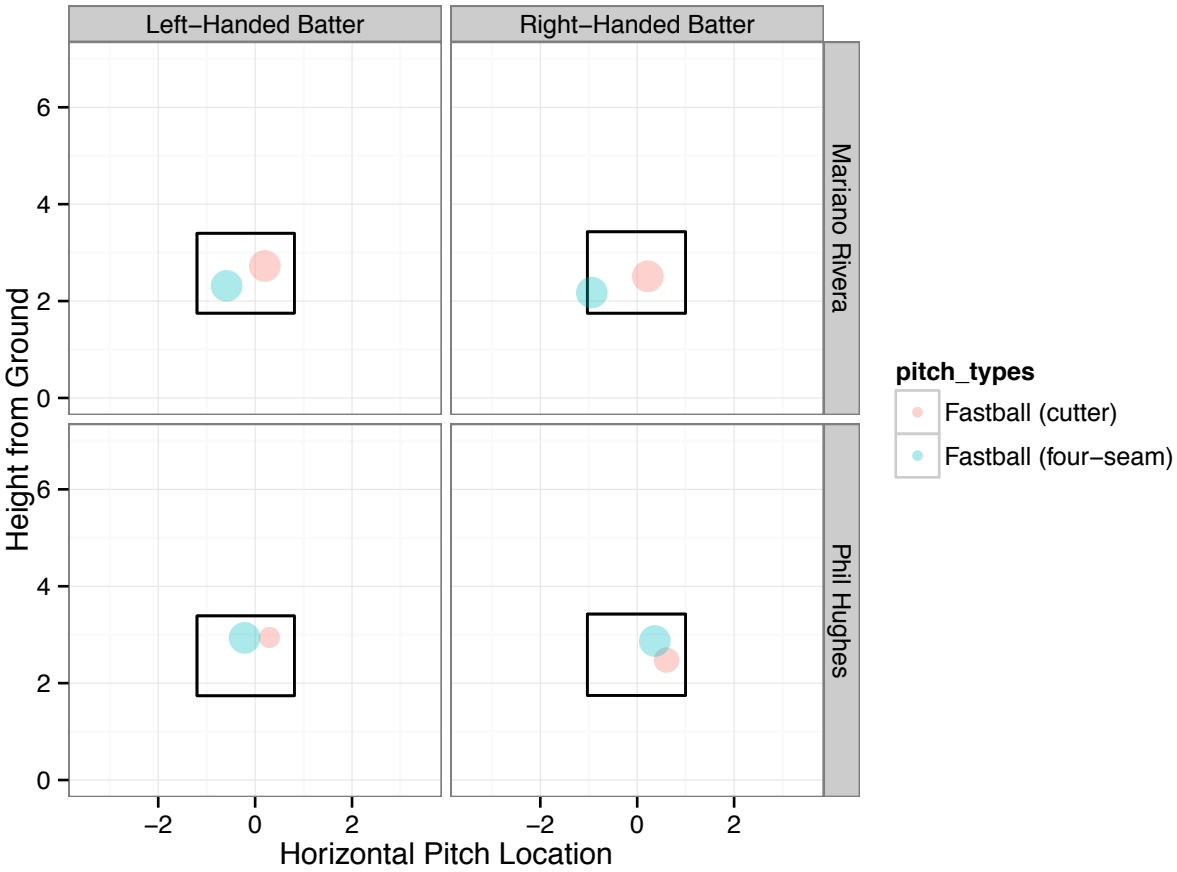


Figure 4.7 The last frame of an animation of averaged four-seam and cutting fastballs thrown by NY Yankee pitchers Mariano Rivera and Phil Hughes during the 2011 season. The actual animation can be viewed at http://cpsievert.github.io/pitchRx/ani2. PITCHf/x parameters are averaged over pitch type, pitcher and batting stance. For instance, the bottom right plot portrays an average four-seam and average cutter thrown by Hughes to right-handed batters.

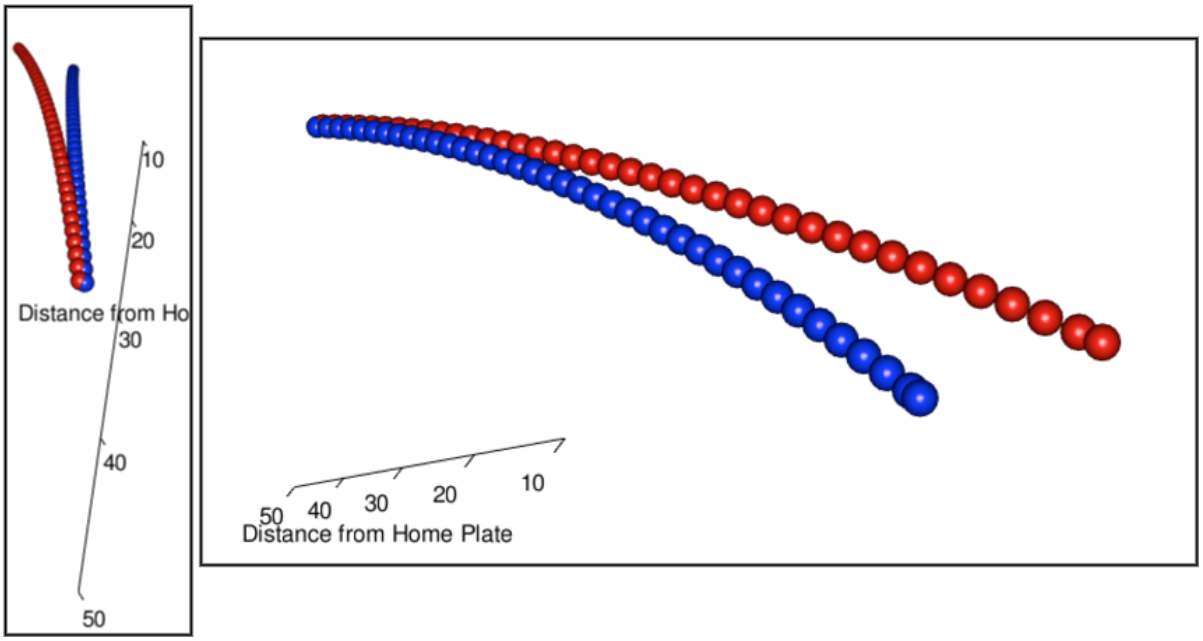


Figure 4.8 3D scatterplot of pitches from Rivera. Pitches are plotted every one-hundredth of a second. Cutting fastballs are shown in red and four-seam fastballs are shown in blue. The left hand plot takes a viewpoint of Rivera and the right hand plot takes a viewpoint near the umpire. Note these are static pictures of an interactive object.

4.7 Conclusion

pitchRx utilizes **XML2R**'s convenient framework for manipulating XML content in order to provide easy access to PITCHf/x and related Gameday data. **pitchRx** removes access barriers which allows the average R user and baseball fan to spend their valuable time analyzing Gameday's enormous source of baseball information. **pitchRx** also provides a suite of functions that greatly reduce the amount of work involved to create popular PITCHf/x graphics. For those interested in obtaining other XML data, **pitchRx** serves as a nice example of leveraging **XML2R** to quickly assemble custom XML data collection mechanisms.

5 LDAvis: A method for visualizing and interpreting topics

This chapter is a paper published in The Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces (ACL 2014) (Sievert and Shirley 2014). I am the primary author of the paper which is avaliable online here http://nlp.stanford.edu/events/illvi2014/papers/sievert-illvi2014.pdf

The formatting of paper has been modified to make for consistent typesetting across the thesis.

ABSTRACT

We present `LDAvis`, an `R` package for creating We present `LDAvis`, a web-based interactive visualization of topics estimated using Latent Dirichlet Allocation that is built using a combination of R and d3. Our visualization provides a global view of the topics (and how they differ from each other), while at the same time allowing for a deep inspection of the tokens most highly associated with each individual topic. First, we propose a novel method for choosing which tokens to present to a user to aid in the task of topic interpretation, in which we define the *relevance* of a token to a topic. Second, we present the results of a user study that illustrates how ranking tokens by their relevance to a given topic relates to that topic's interpretability, and we recommend a default method of computing relevance to maximize topic interpretability. Last, we incorporate relevance into `LDAvis` in a way that allows users to flexibly explore topic-token relationships to better understand a fitted LDA model.

5.1 Introduction

Recently much attention has been paid to visualizing the output of topic models fit using Latent Dirichlet Allocation (LDA) (Matthew J. Gardner and Seppi 2010); (Chaney and Blei 2012); (Jason Chuang and Heer 2012b); (Brynjar Gretarsson and Smyth 2011). Such visualizations are challenging to create because of the high dimensionality of the fitted model – LDA is typically applied to thousands of documents, which are modeled as mixtures of dozens of topics, which themselves are modeled as distributions over thousands of tokens (David M. Blei and Jordan 2012); (Griffiths and Steyvers 2004). The

most promising basic technique for creating LDA visualizations that are both compact and thorough is *interactivity*.

We introduce an interactive visualization system that we call `LDAvis` that attempts to answer a few basic questions about a fitted topic model: (1) What is the meaning of each topic?, (2) How prevalent is each topic?, and (3) How do the topics relate to each other? Different visual components answer each of these questions, some of which are original, and some of which are borrowed from existing tools.

Our visualization (illustrated in Figure 5.1) has two basic pieces. First, the left panel of our visualization presents a "global" view of the topic model, and answers questions 2 and 3. In this view, we plot the topics as circles in the two-dimensional plane whose centers are determined by computing the distance between topics (using a distance measure of the user's choice) and then by using multidimensional scaling to project the inter-topic distances onto two dimensions, as is done in (Jason Chuang and Heer 2012a). We encode each topic's overall prevalence using the areas of the circles, where we sort the topics in decreasing order of prevalence.

Second, the right panel of our visualization depicts a horizontal barchart whose bars represent the individual tokens that are the most useful for interpreting the currently selected topic on the left, and allows users to answer question 1, "What is the meaning of each topic?". A pair of overlaid bars represent both the corpus-wide frequency of a given token as well as the topic-specific frequency of the token, as in (Jason Chuang and Heer 2012b).

The left and right panels of our visualization are linked such that selecting a topic (on the left) reveals the most useful tokens (on the right) for interpreting the selected topic. In addition, selecting a token (on the right) reveals the conditional distribution over topics (on the left) for the selected token. This kind of linked selection allows users to examine a large number of topic-token relationships in a compact manner.

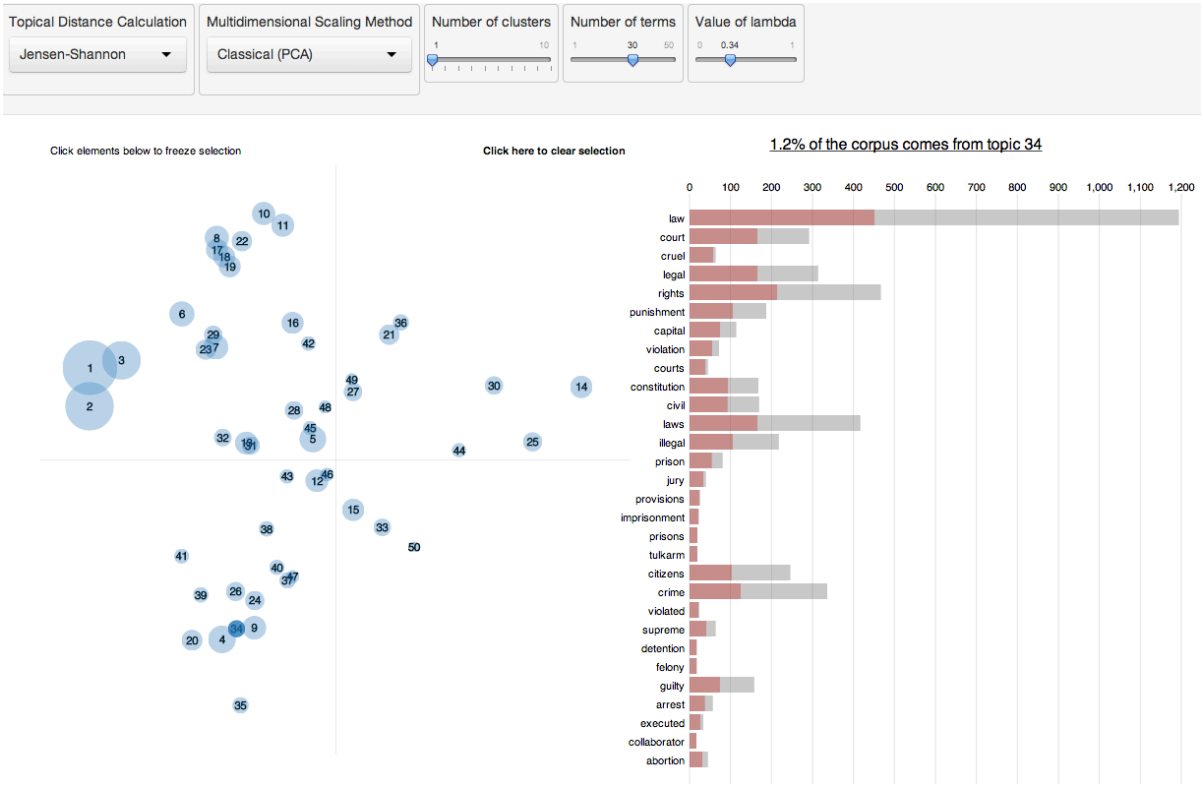


Figure 5.1 The layout of LDAvis, with the global topic view on the left, and the token barcharts on the right. Linked selections allow users to reveal aspects of the topic-token relationships compactly.

A key innovation of our system is how we determine the most useful tokens for interpreting a given topic, and how we allow users to interactively adjust this determination. A topic in LDA is a multinomial distribution over the tokens in the vocabulary, where the vocabulary typically contains thousands of tokens. To interpret a topic, one typically examines a ranked list of the most probable tokens in that topic, using anywhere from three to thirty tokens in the list. The problem with interpreting topics this way is that common tokens in the corpus often appear near the top of such lists for multiple topics, making it hard to differentiate the meanings of these topics.

Bischof and Airoldi (2012) propose ranking tokens for a given topic in terms of both the *frequency* of the token under that topic as well as the token's *exclusivity* to the topic, which accounts for the degree to which it appears in that particular topic to the exclusion of others. We propose a similar measure that we call the *relevance* of a token to a topic to create a flexible method for ranking tokens in order of usefulness for interpreting topics. We discuss our definition of relevance, and its graphical interpretation, in detail in Section 5.3.1. We also present the results of a user study conducted to determine the optimal tuning parameter in the definition of relevance to aid the task of topic interpretation in Section~5.3.2, and we describe how we incorporate relevance into our interactive visualization in Section~5.4.

5.2 Related Work

Much work has been done recently regarding the interpretation of topics (i.e. measuring topic "coherence") as well as visualization of topic models.

5.2.1 Topic Interpretation and Coherence

It is well-known that the topics inferred by LDA are not always easily interpretable by humans. Jonathan Chang and Blei (2009) established via a large user study that standard quantitative measures of fit, such as those summarized by Hanna M. Wallach and Mimno (2009), do not necessarily agree with measures of topic interpretability by humans. Daniel Ramage et al. (2009) assert that "characterizing topics is hard" and describe how using the top-k tokens for a given topic might not always be best, but offer few concrete alternatives.

Loulwah AlSumait and Domeniconi (2009), David Mimno and McCallum (2011), and Jason Chuang and Heer (2013b) develop quantitative methods for measuring the interpretability of topics based on experiments with data sets that come with some notion of topical ground truth, such as document metadata or expert-created topic labels. These methods are useful for understanding, in a global sense, which topics are interpretable (and why), but they don't specifically attempt to aid the user in interpreting *individual* topics.

Blei and Lafferty (2009) developed "Turbo Topics", a method of identifying n-grams within LDA-inferred topics that, when listed in decreasing order of probability, provide users with extra information about the usage of tokens within topics. This two-stage process yields good results on experimental data, although the resulting output is still simply a ranked list containing a mixture of tokens and n-grams, and the usefulness of the method for topic interpretation was not tested in a user study.

David Newman and Baldwin (2010) describe a method for ranking tokens within topics to aid interpretability called Pointwise Mutual Information (PMI) ranking. Under PMI ranking of tokens, each of the ten most probable tokens within a topic are ranked in decreasing order of approximately how often they occur in close proximity to the nine

other most probable tokens from that topic in some large, external "reference" corpus, such as Wikipedia or Google n-grams. Although this method correlated highly with human judgments of token importance within topics, it does not easily generalize to topic models fit to corpora that don't have a readily available external source of word co-occurrences.

In contrast, Taddy (2011) uses an intrinsic measure to rank tokens within topics: a quantity called *lift*, defined as the ratio of a token's probability within a topic to its marginal probability across the corpus. This generally decreases the rankings of globally frequent tokens, which can be helpful. We find that it can be noisy, however, by giving high rankings to very rare tokens that occur in only a single topic, for instance. While such tokens may contain useful topical content, if they are very rare the topic may remain difficult to interpret.

Finally, Bischof and Airoldi (2012) propose and implement a new statistical topic model that infers both a token's frequency as well as its *exclusivity* – the degree to which its occurrences are limited to only a few topics. They introduce a univariate measure called a FREX score ("**FR**equency and **EX**clusivity") which is a weighted harmonic mean of a token's rank within a given topic with respect to frequency and exclusivity, and they recommend it as a way to rank tokens to aid topic interpretation. We propose a similar method that is a weighted average of a token's probability and its lift, and we justify it with a user study and incorporate it into our interactive visualization.

5.2.2 Topic Model Visualization Systems

A number of visualization systems for topic models have arisen in recent years. Several of them focus on allowing users to browse documents, topics, and tokens to learn about the relationships between these three canonical topic model units (Matthew J. Gardner and Seppi 2010); (Chaney and Blei 2012) (Justin Snyder and Wolfe 2013). These

browsers typically use lists of the most probable tokens within topics to summarize the topics, and the visualization elements are limited to barcharts or word clouds of token probabilities for each topic, pie charts of topic probabilities for each document, and/or various barcharts or scatterplots related to document metadata. Although these tools can be useful for browsing a corpus, we seek a more compact visualization, with the more narrow focus of quickly and easily understanding the individual topics themselves (without necessarily visualizing documents).

Jason Chuang and Heer (2012b) develop such a tool, called "Termite", which visualizes the set of topic-token distributions estimated in LDA using a matrix layout. The authors introduce two measures of the usefulness of tokens for understanding a topic model: *distinctiveness* and *saliency*. These quantities measure how much information a token conveys about a topic by computing the Kullback-Liebler divergence between the distribution of topics given the token and the marginal distribution of topics (distinctiveness), optionally weighted by the token's overall frequency (saliency). The authors recommend saliency as a thresholding method for selecting which tokens are included in the visualization, and they further use a seriation method for ordering the most salient tokens to highlight differences between topics.

Termite is a compact, intuitive interactive visualization of the topics in a topic model, but by only including tokens that rank high in saliency or distinctiveness, which are *global* properties of tokens, it is restricted to providing a *global* view of the model, rather than allowing a user to deeply inspect individual topics by visualizing a potentially different set of tokens for every single topic. In fact, Jason Chuang and Heer (2013a) describe the use of a "topic-specific word ordering" as potentially useful future work.

5.3 Relevance of tokens to topics

Here we define *relevance*, our method for ranking tokens within topics, and we describe the results of a user study to learn an optimal tuning parameter in the computation of relevance.

5.3.1 Definition of Relevance

Let ϕ_{kw} denote the probability of token $w \in \{1,...,V\}$ for topic $k \in \{1,...,K\}$, where V denotes the number of unique tokens in the vocabulary, and let p_w denote the marginal probability of token w in the corpus. One typically estimates ϕ in LDA using Variational Bayes methods or Collapsed Gibbs Sampling, and p_w from the empirical distribution of the corpus (optionally smoothed by including prior weights as pseudo-counts).

We define the *relevance* of token w to topic k given a weight parameter λ (where $0 \leq \lambda \leq 1$) as:

$$r(w,k \mid \lambda) = \lambda \log(\phi_{kw}) + (1-\lambda)\log\left(\frac{\phi_{kw}}{p_w}\right),$$

where λ determines the weight given to the probability of token w under topic k relative to its lift. Setting $\lambda = 1$ results in the familiar ranking of tokens in decreasing order of their topic-specific probability, and setting $\lambda = 0$ ranks tokens solely by their lift, which we found anecdotally to result in "noisy" topics full of rare tokens. We wish to learn an "optimal" value of λ for topic interpretation from our user study.

First, though, to see how different values of λ result in different ranked token lists, consider the plot in Figure 5.2. We fit a 50-topic model to the 20 Newsgroups data (details are described in Section~5.3.2) and plotted $\log(\text{lift})$ on the y-axis vs. $\log(\phi_{kw})$ on the x-axis for each token in the vocabulary (which has size $V = 22,524$) for a given topic. Figure 5.2 shows this plot for Topic 29, which occurred mostly in documents posted to the "Motorcycles" newsgroup, but also from documents posted to the "Automobiles"

newsgroup and the "Electronics" newsgroup. Graphically, the line separating the most relevant tokens for this topic, given λ, has slope $-\lambda/(1-\lambda)$ (see Figure 5.2).

For this topic, the top-5 most relevant tokens given $\lambda = 1$ (ranking solely by probability) are {out, #emailaddress, #twodigitnumer, up, #onedigitnumber}, where a '#' symbol denotes a token that is an entity representing a class of things. In contrast to this list, which contains globally common tokens and which provides very little meaning regarding motorcycles, automobiles, or electronics, the top-5 most relevant tokens given $\lambda = 1/3$ are {oil, plastic, pipes, fluid, and lights}. The second set of tokens is much more descriptive of the topic being discussed than the first.

5.3.2 User Study

We conducted a user study to determine whether there was an optimal value of λ in the definition of relevance to aid topic interpretation. First, we fit a 50-topic model to the $D = 13,695$ documents in the 20 Newsgroups data which were posted to a single Newsgroup (rather than two or more Newsgroups). We used the Collapsed Gibbs Sampler algorithm (Griffiths and Steyvers 2004) to sample the latent topics for each of the $N = 1,590,376$ tokens in the data, and we saved their topic assignments from the last iteration (after convergence). We then computed the 20 by 50 table, T, which contains, in cell T_{gk}, the count of the number of times a token from topic $k \in \{1,...,50\}$ was assigned to Newsgroup $g \in \{1,...,20\}$, where we defined the Newsgroup of a token to be the Newsgroup to which the document containing that token was posted. Some of the LDA-inferred topics occurred almost exclusively ($> 90\%$ of occurrences) in documents from a single Newsgroup, such as Topic 38, which was the estimated topic for 15,705 tokens in the corpus, 14,233 of which came from documents posted to the "Medicine" (or "sci.med") Newsgroup. Other topics occurred in a wide variety of Newsgroups. One would expect these "spread-out" topics to be harder to interpret than the "pure" topics like Topic 38.

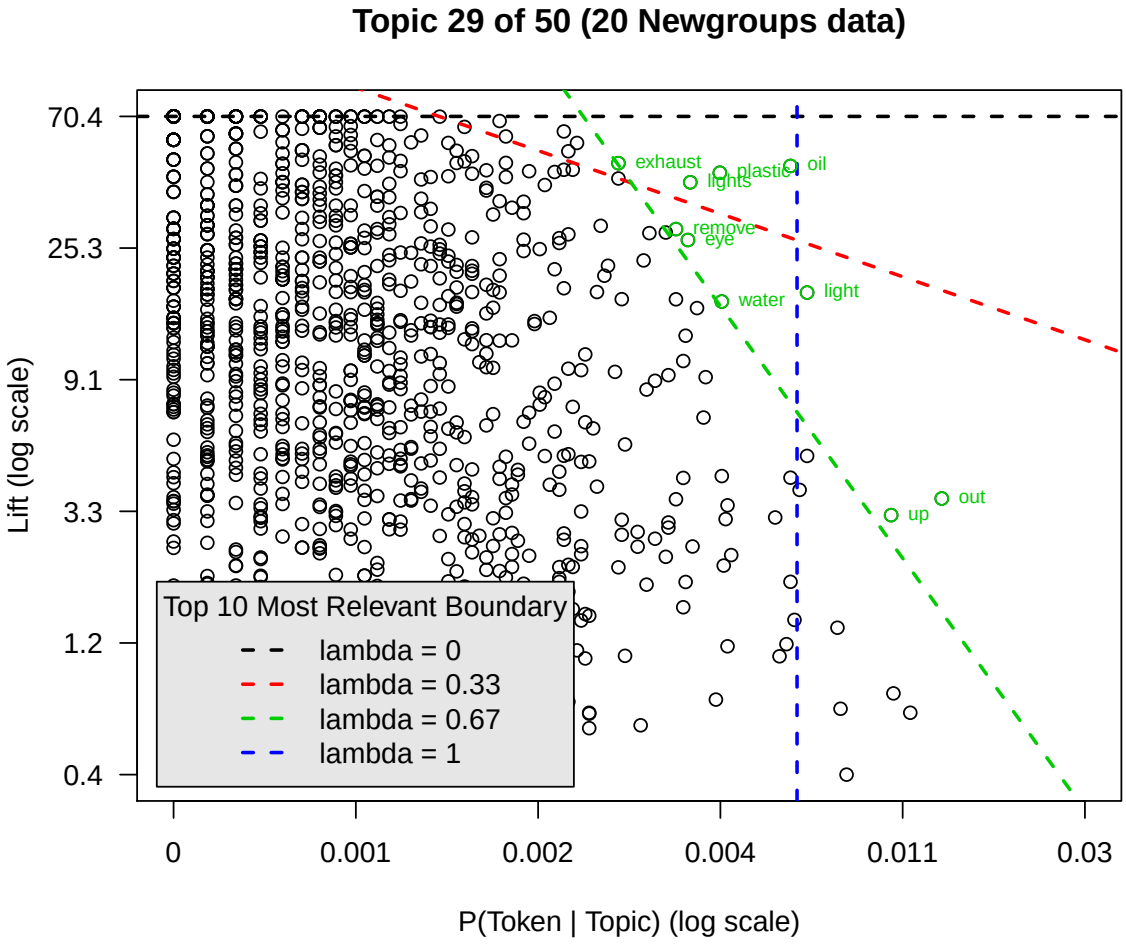


Figure 5.2 Dotted lines separating the top-10 most relevant tokens for different values of λ, with the most relevant tokens for $\lambda = 2/3$ displayed and highlighted in green.

In the study we recruited 29 subjects among our colleagues, and each subject completed an online experiment consisting of 50 tasks, one for each topic in the fitted LDA model. Task k (for $k \in \{1, ..., 50\}$) was to read a list of five tokens, ranked from 1-5 in terms of relevance to topic k, where $\lambda \in (0, 1)$ was randomly sampled to compute relevance. The user was instructed to identify which "topic" the list of tokens discussed from a list of three possible "topics", where their choices were names of the Newsgroups. The correct answer for task k (i.e. our "ground truth") was defined as the Newsgroup that contributed the most tokens to topic k (i.e. the Newsgroup with the largest count in the kth column of the table T), and the two alternative choices were the Newsgroups that contributed the second and third-most tokens to topic k.

We anticipated that the effect of λ on the probability of a user making the correct choice could be different across topics. In particular, for "spread-out" topics that were inherently difficult to interpret, because their tokens were drawn from a wide variety of Newsgroups (similar to a "fused" topic in Jason Chuang and Heer (2013b)), we expected the proportion of correct responses to be roughly 1/3 no matter the value of λ used to compute relevance. Similarly, for very "pure" topics, whose tokens were drawn almost exclusively from one Newsgroup, we expected the task to be easy for any value of λ. To account for this, we analyzed the experimental data by fitting a varying-intercepts logistic regression model to allow each of the fifty topics to have its own baseline difficulty level, where the effect of λ is shared across topics. We used a quadratic function of λ in the model (linear, cubic and quartic functions were explored and rejected).

As expected, the baseline difficulty of each topic varied widely. In fact, seven of the topics were correctly identified by all 29 users, and one topic was incorrectly identified by all 29 users. For the remaining 42 topics we estimated a topic-specific intercept term to control for the inherent difficulty of identifying the topic (not just due to its tokens being spread among multiple Newsgroups, but also to account for the inherent familiarity

of each topic to our subject pool – subjects, on average, were more familiar with "Cars" than "The X Window System", for example).

The estimated effects of λ and λ^2 were 2.74 and -2.34, with standard errors 1.03 and 1.00. Taken together, their joint effect was statistically significant (χ^2 p-value = 0.018). %, but the signs of their coefficients agreed with out intuition, and in a similarly designed large-scale user study (on Mechanical Turk, for instance), we expect that their joint effect would be statistically significant. To see the estimated effect of λ on the probability of correctly identifying a topic, consider Figure 5.3. We plot binned proportions of correct responses (on the y-axis) vs. λ (on the x-axis) for the 14 topics whose estimated topic-specific intercepts fell into the middle tercile among the 42 topics that weren't trivial or impossible to identify. Among these topics there was roughly a 67% baseline probability of correct identification. As Figure 5.3 shows, for these topics, the "optimal" value of λ was about 0.6, and it resulted in a 70% - 75% probability of correct identification, whereas for values of λ near 0 or 1, the proportion of correct responses was closer to 55% or 60%. We view this as evidence that ranking tokens according to relevance, where $\lambda < 1$, can aid topic interpretation, even if this precise task (selecting a known topic label from a list of pre-defined labels associated with each document as metadata) is not always the goal. A similar conclusion might be drawn from an experiment to study the FREX token ranking method of Bischof and Airoldi (2012).

Note that in our experiment, we used the collection of single-posted 20 Newsgroups documents to define our "ground truth" data. An alternative method for collecting "ground truth" data would have been to recruit experts to label topics from an LDA model. We chose against this option because doing so would present a classic "chicken-or-egg" problem: If we use expert-labeled topics in an experiment to learn how to summarize topics so that they can be interpreted (i.e. "labeled"), we would only re-learn the way that our experts were instructed, or allowed, to label the topics in the first place! If, for

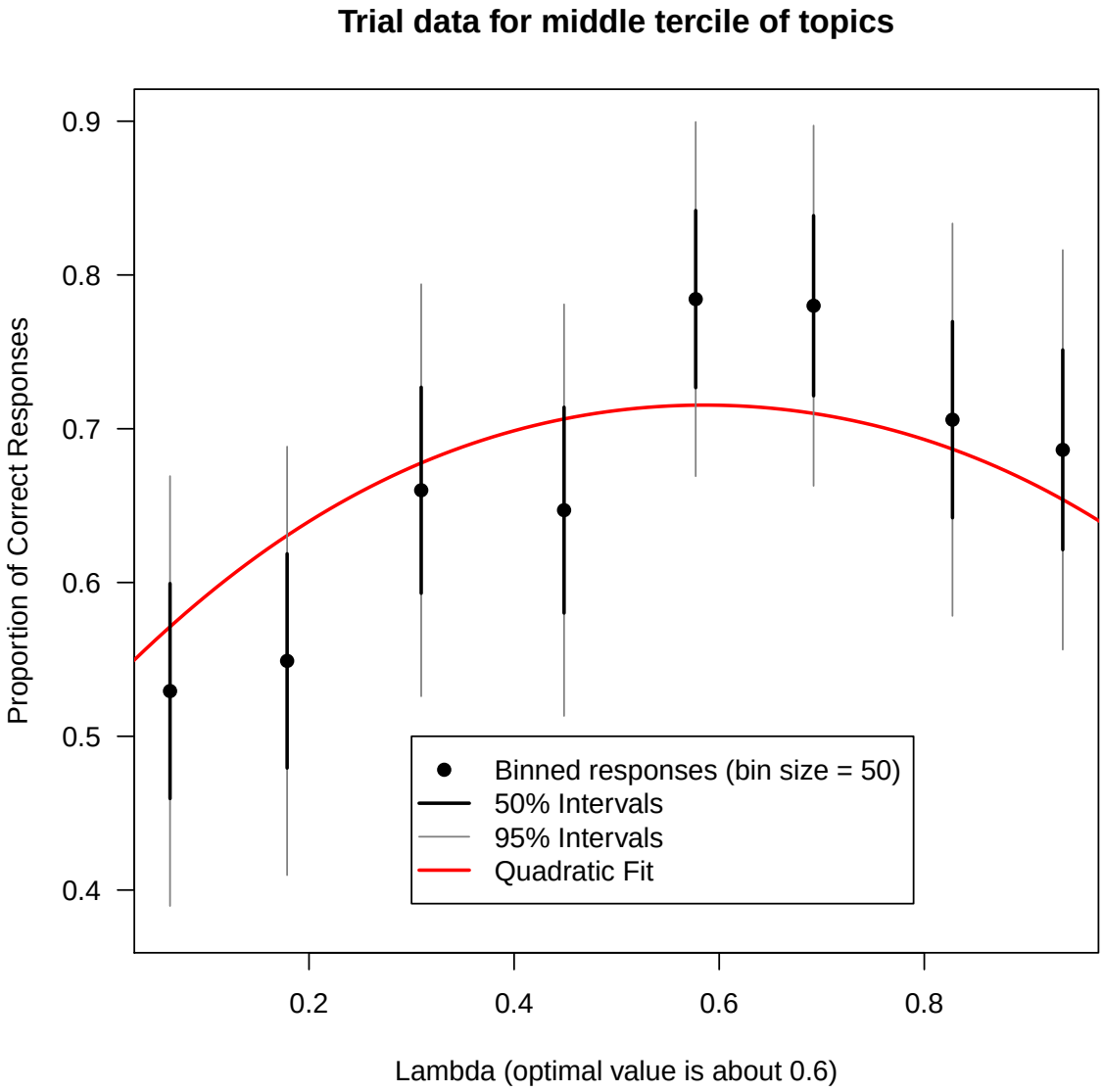


Figure 5.3 A plot of the proportion of correct responses in a user study vs. the value of λ used to compute the most relevant tokens for each topic.

instance, the experts were presented with a ranked list of the most probable tokens for each topic, this would influence the interpretations and labels they give to the topics, and the experimental result would be the circular conclusion that ranking tokens by probability allows users to recover the "expert" labels most easily. To avoid this, we felt strongly that we should use data in which documents have metadata associated with them. The 20 Newsgroups data provides an externally validated source of topic labels, in the sense that the labels were presented to users (in the form of Newsgroup names), and users subsequently filled in the content. It represents, essentially, a crowd-sourced collection of tokens, or content, for a certain set of topic labels.

5.4 Our Visualization System

Our interactive, web-based visualization system, `LDAvis`, has two core functionalities that enable users to understand the topic-token relationships in a fitted LDA model, and a number of extra features that provide additional perspectives on the model. %Usually these questions can not be answered easily with a few simple plots and/or metrics. Instead, an interactive layout such as `LDAvis` allows one to quickly explore model output, form new hypotheses and verify findings.

First and foremost, `LDAvis` allows one to select a topic to reveal the most relevant tokens for that topic. In Figure 5.1, Topic 34 is selected, and its 30 most relevant tokens (given $\lambda = 0.34$, in this case) populate the bar chart to the right (ranked in order of relevance from top to bottom). The widths of the gray bars represent the corpus-wide frequencies of each token, and the widths of the red bars represent the topic-specific frequencies of each token. A slider allows users to change the value of λ, which can alter the rankings of tokens to aid topic interpretation. By default, λ is set to 0.6, as suggested by our user study in Section~5.3.2. If $\lambda = 1$, tokens are ranked solely by ϕ_{kw}, which implies the red bars

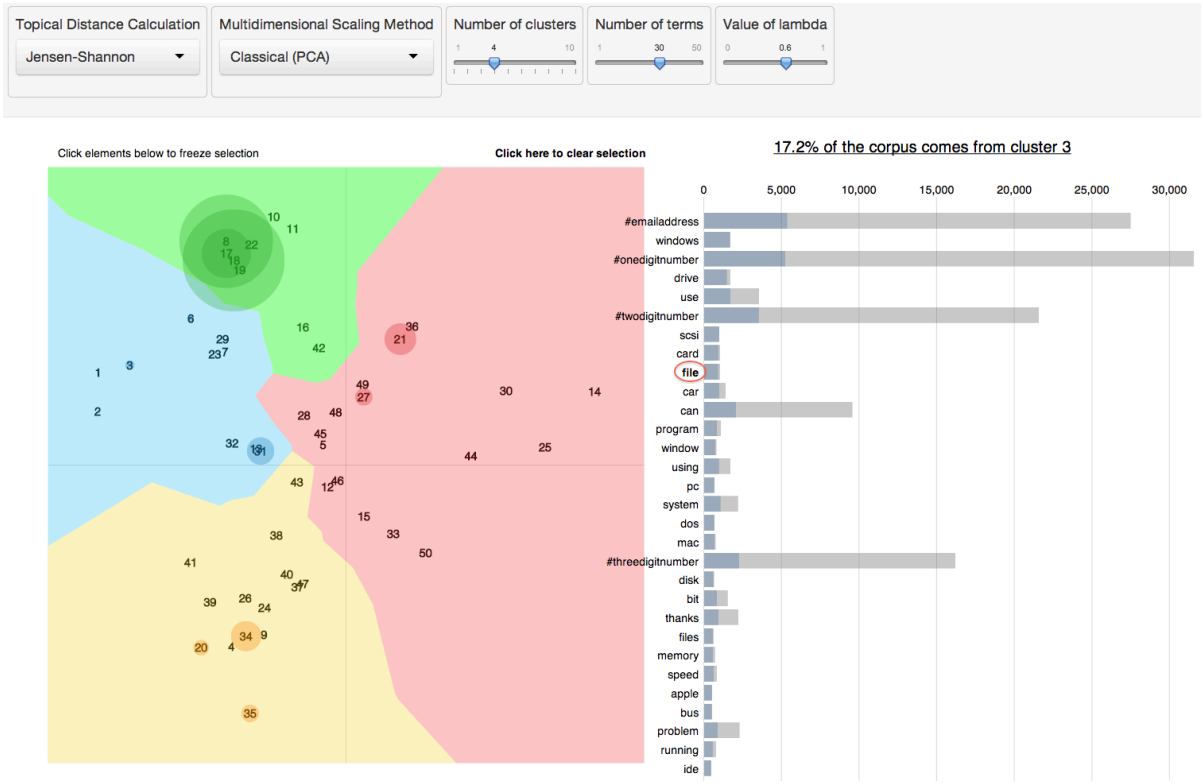


Figure 5.4 The user has chosen to segment the topics into four clusters, and has selected the green cluster to populate the barchart with the most relevant tokens for that cluster. Then, the user hovered over the ninth bar from the top, 'file', to display the conditional distribution over topics for this token.

would be sorted from widest (at the top) to narrowest (at the bottom). By comparing the widths of the red and gray bars for a given token, users can quickly understand whether a token is highly relevant to the selected topic because of its lift (a high ratio of red to gray), or its probability (absolute width of red). The top 3 most relevant tokens in Figure 5.1 are "law", "rights", and "court". Note that "law" is a common word which is generated by Topic 34 in about 40% of its corpus-wide occurrences, whereas "cruel" is a relatively rare word with very high lift – it occurs almost exclusively in Topic 34. Such properties of the topic-token relationship are readily visible in `LDAvis` for every topic.

On the left panel, two visual features provide a global perspective of the topics. First, the areas of the circles are proportional to the relative prevalences of the topics in the corpus, θ_k, which can be computed as $\theta_k = \sum_d N_d \theta_{dk}$ for documents $d = 1, ..., D$, where document d contains N_d tokens. In the 50-topic model fit to the 20 Newsgroups data, the first three topics comprise 12%, 9%, and 6% of the corpus, and all contain common, non-specific tokens (although there are differences: Topic 2 contains formal debate-related language such as "conclusion", "evidence", and "argument", whereas Topic 3 contains slang conversational language such as "kinda", "like", and "yeah"). In addition to visualizing topic prevalence, the left pane shows inter-topic differences. The default for computing inter-topic distances is Jensen-Shannon divergence, although other metrics are enabled. The default for scaling the set of inter-topic distances defaults to Principal Components, but other other algorithms are also enabled.

The second core feature of `LDAvis` is the ability to select a token (by hovering over it) to reveal its conditional distribution over topics. This distribution is visualized by altering the areas of the topic circles such that they are proportional to the token-specific frequencies across the corpus. This allows the user to verify, as discussed in Jason Chuang and Heer (2012a), whether the multidimensional scaling of topics has faithfully clustered similar topics in two-dimensional space. For example, in Figure 5.4, the token "file" is

selected. In the majority of this token's occurrences, it is drawn from one of several topics located in the upper left-hand region of the global topic view. Upon inspection, this group of topics can be interpreted broadly as a discussion of computer hardware and software. This verifies, to some extent, their placement, via multidimensional scaling, into the same two-dimensional region. It also suggests that the word "file" used in this context refers to a computer file. However, there is also conditional probability mass for the token "file" on Topic 34. As shown in Figure 5.1, Topic 34 can be interpreted as discussing the criminal punishment system where "file" refers to court filings. Similar discoveries can be made for any word that exhibits polysemy (such as "drive" appearing in computer- and automobile-related topics, or "ground" occurring in electrical- and baseball-related topics).

Beyond its within-browser interaction capability, `LDAvis` leverages the `R` language to allow users to easily alter the topical distance measurement as well as the multidimensional scaling algorithm to produce the global topic view. In addition, there is an option to apply k-means clustering to the topics (as a function of their two-dimensional locations in the global topic view). This is merely an effort to facilitate semantic zooming in an LDA model with many topics where 'after-the-fact' clustering may be an easier way to learn clusters of topics, rather than fitting a hierarchical topic model (David M. Blei and Tenenbaum 2003), for example. Selecting a cluster (or region) of topics reveals the most relevant tokens for that group of topics, where the token distribution of a cluster of topics is defined as the average of the token distributions of the individual topics in the cluster. In Figure 5.4, the green cluster of topics is selected, and the most relevant tokens are predominantly related to computer hardware and software.

5.5 Discussion

We have described a web-based, interactive visualization system, `LDAvis`, that enables deep inspection of topic-token relationships in an LDA model, while simultaneously providing a "global" view of the topics, via their prevalences and similarities to each other, in a compact space. We also propose a novel way to rank tokens within topics to aid in the task of topic interpretation, and we present a user study that attempts to not only *measure* the interpretability of a topic, but also how to *maximize* the interpretability of the topic.

For future work, we anticipate performing a larger user study to further understand how to facilitate topic interpretation in fitted LDA models, including a comparison of multiple methods, such as ranking by Turbo Topics (Blei and Lafferty 2009) or FREX scores (Bischof and Airoldi 2012), in addition to relevance. We also note the need to visualize correlations between topics, as this can provide insight into what is happening on the document level without actually displaying entire documents. Last, we seek a solution to the problem of visualizing a large number of topics (say, from 100 - 500 topics) in a compact way.

6 Extending ggplot2's grammar of graphics implementation for linked and dynamic graphics on the web

This chapter is a paper currently under revision with intention of submitting to the Journal of Computational and Graphical Statistics. I am the primary author of the paper and there is a working draft available here – https://github.com/tdhock/animint-paper/blob/jcgs/HOCKING-animint.pdf

The formatting of paper has been modified to make for consistent typesetting across the thesis.

ABSTRACT

The web is the most popular medium for sharing interactive data visualizations thanks to the portability of the web browser and the accessibility of the internet. Unfortunately, creating interactive web graphics often requires a working knowledge of numerous web technologies that are foreign to many people working with data. As a result, web graphics are rarely used for exploratory data analysis where quick iteration between different visualizations is of utmost importance. This is the core strength of ggplot2, a popular data visualization package for R, the world's leading open-source statistical programming language. The conceptual framework behind ggplot2 is based on the grammar of graphics, which lays a foundation for describing any static graphic as a small set of independent components. Perhaps the most fundamental component is the mapping from abstract data to the visual space, sometimes referred to as the aesthetic mapping. We propose adding two new aesthetics to the grammar, which together are sufficient for elegantly describing both animations and certain classes of coordinated linked views. We implement this extension in the open-source R package animint, which converts ggplot2 objects to interactive web visualizations via D3.

6.1 Introduction

The world's leading open source statistical programming language, R, has a rich history of interfacing with computational tools for the use of people doing data analysis and statistics research (R Core Team 2015). Understanding R's core audience is important, as they typically want to maximize their time working on data analysis problems, and

minimize time spent learning computational tools. R excels in this regard, as it is designed specifically for interactive use, where users can quickly explore their data using highly expressive interfaces. Another key player in R's success story is its packaging infrastructure, which provides tools for distributing entire research conpendium(s) (code, data, documentation, auxiliary documents, etc) (Gentleman and Lang 2004).

One of the most widely used R packages is ggplot2 (Wickham 2009b), a data visualization package inspired by the grammar of graphics (Wilkinson et al. 2006). In fact, Donoho (2015) writes: "This effort may have more impact on today's practice of data analysis than many highly-regarded theoretical statistics papers". In our experience, ggplot2 has made an impact thanks to its foundation in the grammar of graphics, carefully chosen defaults, and overall usability. This helps data analysts rapidly iterate and discover informative visualizations – an essential task in exploratory data analysis (EDA). When dealing with high-dimensional data, however, it is often useful to produce interactive and/or dynamic graphics, which ggplot2 does not inherently support.

Interactive graphics toolkits in R have been used for decades to enhance the EDA workflow, but these approaches are often not easy to reproduce or distribute to a larger audience. It is true that most graphics generated during EDA are ultimately not useful, but sometimes, understanding gained during this phase is most easily shared via the interactive graphics themselves. Thus, there is value is being able to easily share, and embed interactive graphics inside a larger report. Unfortunately, this is typically hard, if not impossible, using traditional interactive graphics toolkits. As a result, there is a large disconnect between the visualization tools that we use for exploration versus presentation.

We aim to narrow this gap in visualization tools by extending ggplot2's grammar of graphics implementation for interactive and dynamic web graphics. Our extension allows one to create animated transitions and perform database queries via direct manipulation

of linked views like those described in (Ahlberg, Williamson, and Shneiderman 1991) and (Buja et al. 1991). A conceptual model for our extension is provided in Section 6.3.1 and Section 6.3.2. In Section 6.3.3, we demonstrate our extension with an example. In Section 6.3.4, we outline design decisions made in our implementation in the R package animint. In Section 6.4, we provide a sense scope for our system and its performance limitations through a handful of examples. In Section 6.5, we conduct a comparison study by replicating examples with other leading systems. Finally, in Section 6.7, we discuss future work and limitations of our current system.

6.2 Related Work

We aim to provide a system which empowers ggplot2 users to go beyond the confines of static graphics with minimal friction imposed upon their current workflow. We acknowledge that numerous systems which support similar visualization techniques exist outside of the R ecosystem, but we intentionally focus on R interfaces since the surrounding statistical computing environment is crucial for enabling an efficient exploratory data analysis workflow.

It is important to acknowledge that ggplot2 is built on top of the R package grid, a low-level graphics system, which is now bundled with R itself (R Core Team 2015). Neither grid, nor base R graphics, have strong support for handling user interaction creating a need for add-on packages. There are a number of approaches these packages take to rendering, each with their own benefits and drawbacks. Traditionally, they build on low-level R interfaces to graphical systems such as GTK+ (Lawrence and Temple Lang 2010), Qt (Lawrence and Sarkar 2016a); (Lawrence and Sarkar 2016b), or Java GUI frameworks (Urbanek 2015). In general, the resulting system can be very fast and flexible, but sharing ir reproducing output is usually a problem due to the heavy

software requirements. Although there may be sacrifice in performance, using the modern web browser as a canvas is more portable, accessible, and composable (graphics can be embedded within larger frameworks/documents).

Base R does provide a Scalable Vector Graphics (SVG) device, `svg()`, via the Cairo graphics API (Cairo 2016). The R package SVGAnnotation (Nolan and Temple Lang 2012) provides functionality to post-process `svg()` output in order to add interactive and dynamic features. This is a powerful approach, since in theory it can work with any R graphic, but the package is self described as a proof-of-concept which reverse engineers poorly structured `svg()` output. As a result, anyone wishing to extend or alter the core functionality needs a deep understanding of base graphics and SVG.

The lack of well-structured SVG for R graphics motivated the gridSVG package which provides sensible structuring of SVG output for grid graphics (Murrell and Potter 2015). This package also provides some low-level tools for animating or adding interactive features, where grid objects must be referenced by name. As a result, if one wanted to use this interface to add interactivity to a ggplot2 plot, they must know and understand the grid naming scheme ggplot2 uses internally and hope it does not change down the road. An interface where interactivity can be expressed by referencing the data to be visualized, rather than the building blocks of the graphics system, would be preferable since the former interface is decoupled from the implementation and does not require knowledge of grid.

In terms of the user interface, the R package gganimate is very similar to our system (Robinson 2016b). It directly extends ggplot2 by adding a new aesthetic, named `frame`, which splits the data into subsets (one for each unique value of the frame variable), produces a static plot for each subset, and uses the animation package to combine the images into a key frame animation (Xie 2013a). This is quite similar, but not as flexible as our system's support for animation, which we fully describe in Section 6.3.2. Either

system has the ability to control the amount of time that a given frame is displayed, but our system can also animate the transition between frames via the `d3.transition()` API (Bostock, Oglevetsky, and Heer 2011). Smooth transitions help us track positions between frames, which is useful in many scenarios, such as the touring example discussed in Section~6.

Tours are a useful visualization technique for exploring high-dimensional data which requires interactive and dynamic graphics. The open source software ggobi is currently the most fully-featured tool for touring data and has support for interactive techniques such as linking, zooming, panning, and identifying (Cook and Swayne 2007). The R package rggobi (Wickham et al. 2008) provides an R interface to ggobi's graphical interface, but unfortunately, the software requirements for installation and use of this toolchain are heavy and stringent. Furthermore, sharing the interactive versions of these graphics are not possible. The R package cranvas aims to be the successor to ggobi, with support for similar interactive techniques, but with a more flexible interface for describing plots inspired by the grammar of graphics (Yihui Xie 2013). Cranvas also has heavy and stringent software requirements which limits the portability and accessibility of the software.

Another R package for interactive graphics which draws design inspiration from the grammar of graphics is ggvis (Chang and Wickham 2015). It does not directly extend ggplot2, but instead provides a brand new purely functional interface which is designed with interactive graphics in mind. It currently relies on Vega to render the SVG graphics from JSON (A. S. A. K. W. A. J. Heer 2014), and the R package shiny to enable many of its interactive capabilities (Chang et al. 2015). The interface gives tremendous power to R users, as it allows one to write R functions to handle user events. This power does come with a cost, though, as sharing and hosting ggvis graphics typically requires special web server software, even when the interaction logic could be handled entirely client-side.

As we outline in Section 6.3.4, our system does not require a web server, but can also be used inside shiny web applications, when desired.

6.3 Extending the layered grammar of graphics

In this section, we propose an extension to the layered grammar of graphics (Wickham 2010a) which enables declarative expression of animations and database queries via direct manipulation. In the ggplot2 system, there are five essential components that define a layer of graphical markings: data, mappings (i.e., aesthetics), geometry, statistic, and position. These simple components are easily understood in isolation and can be combined in many ways to express a wide array of graphics. For a simple example, here is one way to create a scatterplot in ggplot2 of variables named `<X>` and `<Y>` in `<DATA>`:

```
ggplot() + layer(
  data = <DATA>,
  mapping = aes(x = <X>, y = <Y>),
  geom = "point",
  stat = "identity",
  position = "identity"
)
```

For every geometry, ggplot2 provides a convenient wrapper around `layer()` which provides sensible defaults for the statistic and position (in this case, both are "identity"):

```
ggplot() + geom_point(
  data = <DATA>,
  aes(x = <X>, y = <Y>)
)
```

A single ggplot2 plot can be comprised of multiple layers, and different layers can correspond to different data. Since each graphical mark within a ggplot2 layer corresponds to one (or more) observations in `<DATA>`, aesthetic mappings provide a mechanism for mapping graphical selections to the original data (and vice-versa) which is essential to any interactive graphics system (Andreas Buja and McDonald 1988); (Wickham et al. 2010). Thus, given a way to combine multiple ggplot2 plots into a single view, this design can be extended to support a notion of multiple linked views, as those discussed in (Ahlberg, Williamson, and Shneiderman 1991) and (Buja et al. 1991).

6.3.1 Direct Manipulation of Database Queries

Cook and Swayne (2007) use SQL queries to formalize the direct manipulation methods discussed in Ahlberg, Williamson, and Shneiderman (1991) and Buja et al. (1991). As it turns out, we can embed this framework inside the layered grammar of graphics with two classes of new aesthetics: one class to define a selection source and one to define a target. This is most easily seen using our animint implementation, which has a `clickSelects` aesthetic for defining the selection source (via mouse click) and a `showSelected` aesthetic for defining the target. Here we use animint to create a linked view between a bar chart and a scatter plot, where the user can click on bars to control the points shown in the scatterplot, as shown in the video in Figure 6.1. As a result, we can quickly see how the relationship among tip amount and total bill amount depends on whether the customer is smoker.

```
library(animint)
p1 <- ggplot() + geom_bar(
  data = reshape2::tips,
  aes(x = smoker, clickSelects = smoker)
)
```

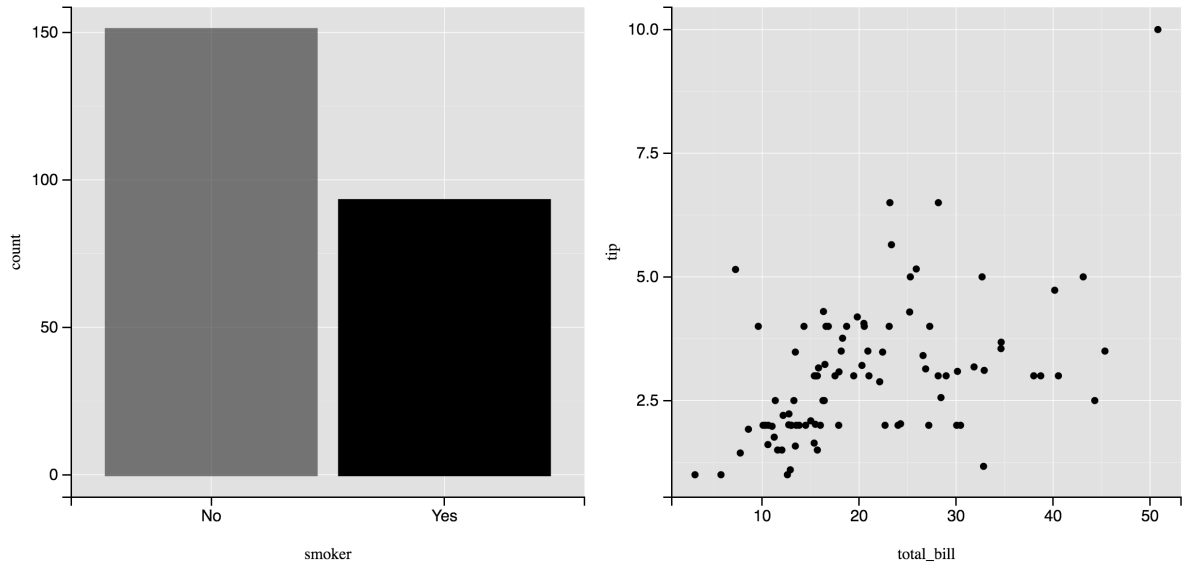


Figure 6.1 Linked database querying via direct manipulation using animint. A video demonstration can be viewed online at https://vimeo.com/160496419

```
p2 <- ggplot() + geom_point(
  data = reshape2::tips,
  aes(x = total_bill, y = tip,
      showSelected = smoker)
)
animint2dir(list(p1 = p1, p2 = p2))
```

In essense, the R code above allows us to use direct manipulation to dynamically perform SQL queries of the form:

```
SELECT total_bill, tip FROM tips
  WHERE smoker IN clickSelects
```

In this example, `clickSelects` is either "Yes" or "No", but as we show in later examples, `clickSelects` can also be an array of values. Although `clickSelects` is tied to a mouseclick event, this same framework supports other selection events, such as hover or click+drag. Statistically speaking, this is useful for visualizing and navigating through

joint distributions conditional upon discrete values. In this sense, our extension is closely related to the same a basis which leads to trellis displays (Richard A. Becker 1996) and linked scatterplot brushing (Becker and Cleveland 1987). The major differences are that conditioning: is layer (i.e., not plot) specific, is not tied to a particular geometry, and can be controlled through direct manipulation or animation controls. %% TODO: make connections to scagnostics? trelliscope?

6.3.2 Adding animation

In some sense, the `showSelected` aesthetic splits the layer into subsets – one for every unique value of the `showSelected` variable. The `clickSelects` aesthetics provides a mechanism to alter the visibility of those subset(s) via direct manipulation, but our system also provides a mechanism for automatically looping through selections to produce animation(s). We acheive this by reserving the name `time` to specify which variable to select as well as the amount of time to wait before changing the selection (in milliseconds). We also reserve the name `duration` to specify the amount of time used to smoothly transition between frames (with linear easing). The code below was used to generate Figure 6.2 which demonstrates a simple animation with smooth transitions between 10 frames of a single point. Note that the resulting web page has controls for interactively altering the `time` and `duration` parameters.

```
d <- data.frame(v = 1:10)
plotList <- list(
  plot = ggplot() + geom_point(
    data = d, aes(x=v, y=v, showSelected=v)
  ),
  time = list(variable = "v", ms = 1000),
  duration = list(v = 1000)
```

```
)

animint2dir(plotList)
```

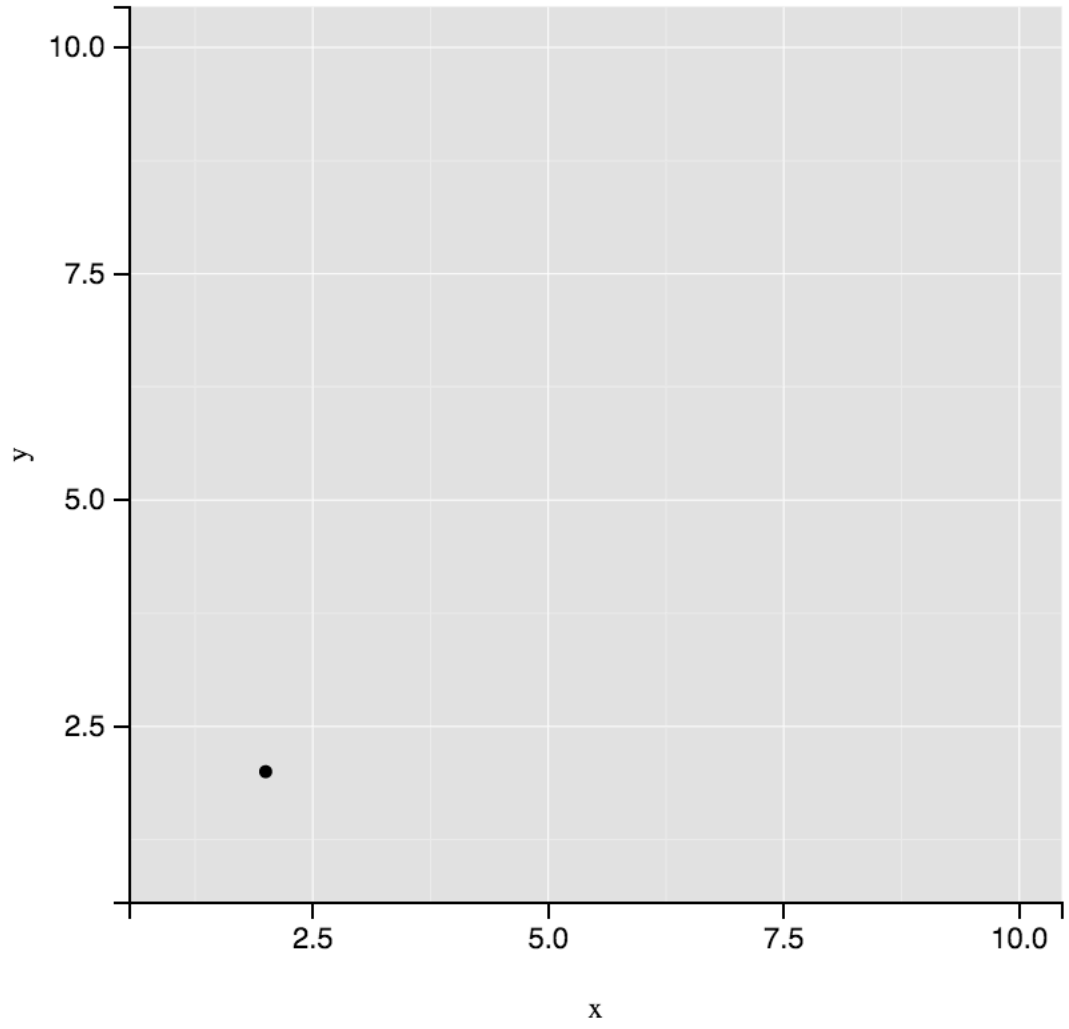
6.3.3 World Bank Example

Figure 6.3 shows an interactive animation of the World Bank data set (World Bank 2012) created with our animint implementation. The visualization helps us explore the change in the relationship between life expectancy and fertility over time for 205 countries. By default, the year 1979 and the countries United States and Vietnam are selected, but readers are encouraged to watch the video of the animation and/or interact the visualization using a web browser.[1] In the interactive version, the selected value of the year variable is automatically incremented every few seconds, using animation to visualize yearly changes in the relationship between life expectancy and fertility rate.

When viewing the interactive version of Figure 6.3, suppose we wish to select Thailand. Direct manipulation is not very useful in this case since it is not easy to identify and select Thailand based on graphical marks on a plot. For this reason, animint also provides dropdown menu(s) for each selection variable to aid the selection process. Figure 6.4 shows what the user sees after typing "th" in the search box. Note that these dropdowns support selection of multiple values and coordinate sensibly with selections made via direct manipulation.

We anticipate that some ggplot2 users will be able to reverse engineer the animint code which creates Figure 6.3, simply by looking at it. In fact, this is a big reason why ggplot2 is so widely used: it helps minimize the amount of time required to translate a figure that exists in your head into computer code. Note that, in the left hand plot of Figure 6.3, we have a time series of the life expectancy where each line is a country (i.e., we `group` by

[1] http://bl.ocks.org/tdhock/raw/8ce47eebb3039263878f/



Show download status table | Hide animation controls

Play **milliseconds**

updates 1000

z 1000

Figure 6.2 A simple animation with smooth transitions and interactively altering transition durations. A video demonstration can be viewed online at https://vimeo.com/160505146

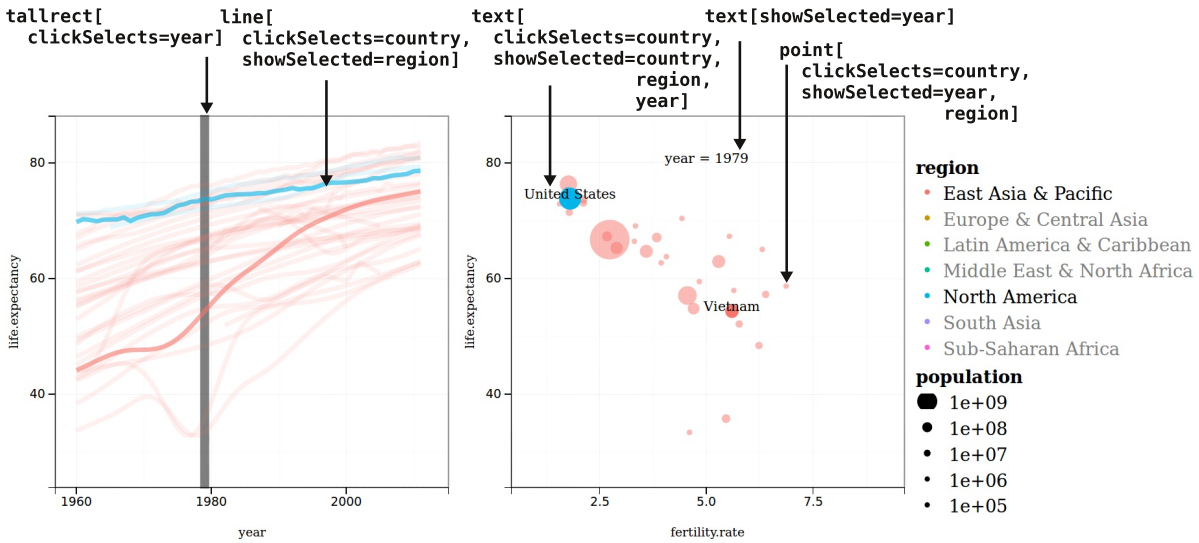


Figure 6.3 An interactive animation of World Bank demographic data of several countries, designed using `clickSelects` and `showSelected` keywords (top). Left: a multiple time series from 1960 to 2010 of life expectancy, with bold lines showing the selected countries and a vertical grey tallrect showing the selected year. Right: a scatterplot of life expectancy versus fertility rate of all countries. The legend and text elements show the current selection: year=1979, country= {United States, Vietnam}, and region={East Asia & Pacific, North America}

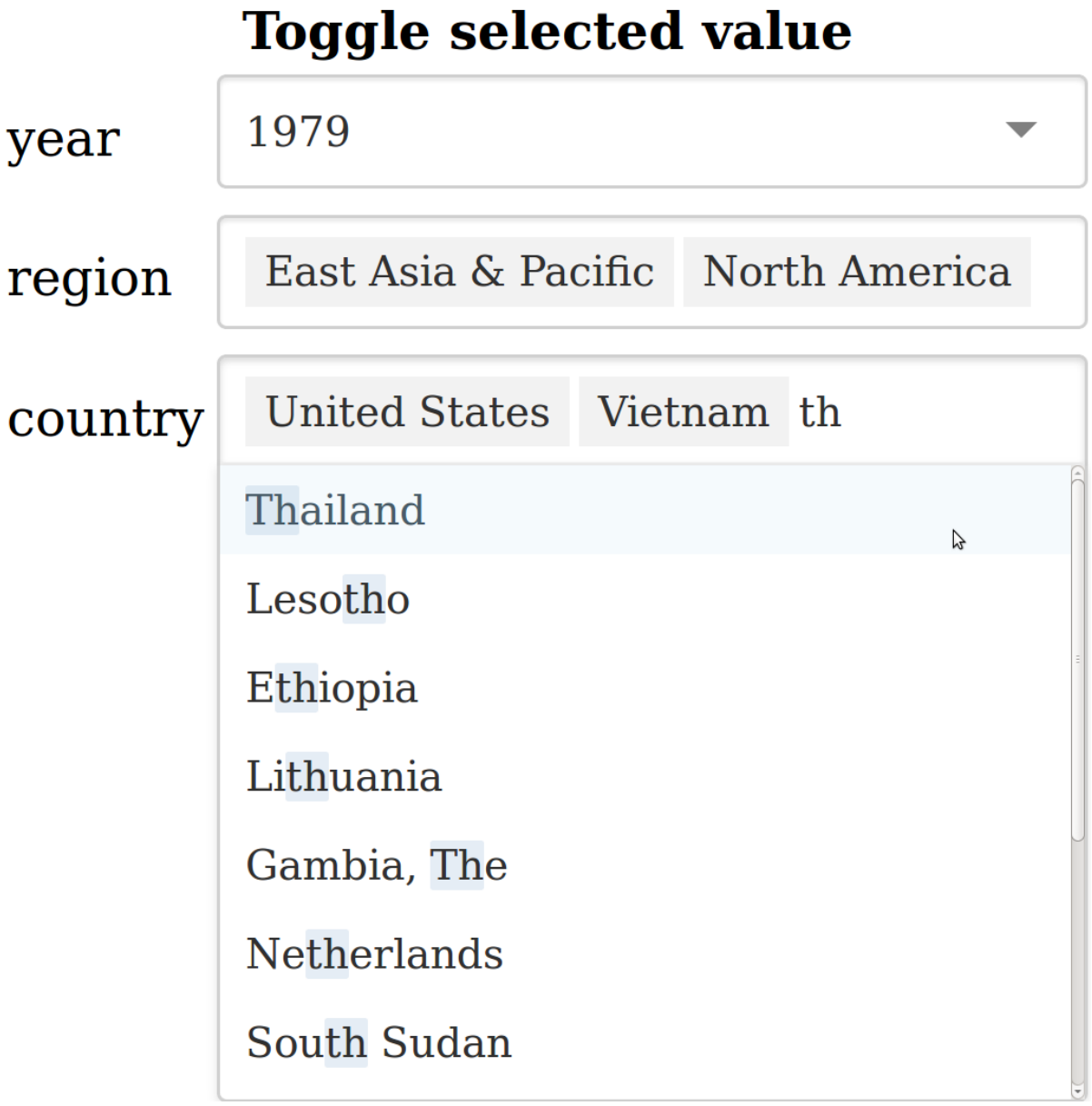


Figure 6.4 Animint provides a menu to update each selection variable. In this example, after typing 'th' the country menu shows the subset of matching countries.

country) and lines are colored by region. By clicking on a line, we also want the country label to appear in the right hand plot, so we also need to set `clickSelects=country`. Lastly, by setting showSelected=region, we can hide/show lines by clicking on the color legend entries.

```
timeSeries <- ggplot() + geom_line(
  data = WorldBank,
  aes(x = year, y = life.expectancy,
      group = country, color = region,
      clickSelects = country,
      showSelected = region)
)
```

We want to provide a visual clue for the selected year in the time series, so we also layer some "tall rectangles" onto the time series. These tall rectangles will also serve as a way to directly modify the selected year. The tallrect geometry is a special case of a rectangle that automatically spans the entire vertical range, so we just have to specify the horizontal range via `xmin` and `xmax`. Also, since the layered grammar of graphics allows for different data in each layer, we supply a data frame with just the unique years in the entire data for this layer.

```
years <- data.frame(year = unique(WorldBank$year))
timeSeries <- timeSeries + geom_tallrect(
  data = years,
  aes(xmin = year - 0.5, xmax = year + 0.5,
      clickSelects = year)
)
```

As for the right hand plot in Figure 6.3, there are three layers: a point layer for countries, a text layer for countries, and a text layer to display the selected year. By clicking on a point, we want to display the country text label and highlight the corresponding time series on the right hand plot, so we set `clickSelects=country` in this layer. Furthermore, we only want to show the points for the selected year and region, so we also need `showSelected=year` and `showSelected2=region`.

```
scatterPlot <- ggplot() + geom_point(
  data = WorldBank,
  aes(x = fertility.rate, y = life.expectancy,
      color = region, size = population,
      clickSelects = country,
      showSelected = year,
      showSelected2 = region)
)
```

The text layer for annotating selected countries is essentially the same as the point layer, except we map the country name to the `label` aesthetic.

```
scatterPlot <- scatterPlot + geom_text(
  data = WorldBank,
  aes(x = fertility.rate, y = life.expectancy,
      label = country,
      showSelected = country,
      showSelected2 = year,
      showSelected3 = region)
)
```

Lastly, to help identify the selected year when viewing the scatterplot, we add another layer of text at a fixed location.

```
scatterPlot <- scatterPlot + geom_text(
  data = years, x = 5, y = 80,
  aes(label = paste("year =", year),
      showSelected = year)
)
```

Now that we have defined the plots in Figure 6.3, we can set the `time` and `duration` options (introduced in Section 6.3.2) to control the animation parameters. Our animint implementation also respects a `selector.types` option which controls whether or not selections for a given variable can accumulate and a `first` option for controlling which values are selected by default. [2] By default, supplying the list of plots and additional options to `animint2dir()` will write all the files necessary to render the visualization to a temporary directory and prompt a web browser to open an HTML file.

```
viz <- list(
  timeSeries = timeSeries,
  scatterPlot = scatterPlot,
  time = list(variable = "year", ms = 3000),
  duration = list(year = 1000),
  selector.types = list(
    year = "single",
    country = "multiple",
    region = "multiple"
  ),
```

[2] We maintain a complete list of (animint specific) options here – https://github.com/tdhock/animint/wiki/Advanced-features-present-animint-but-not-in-ggplot2

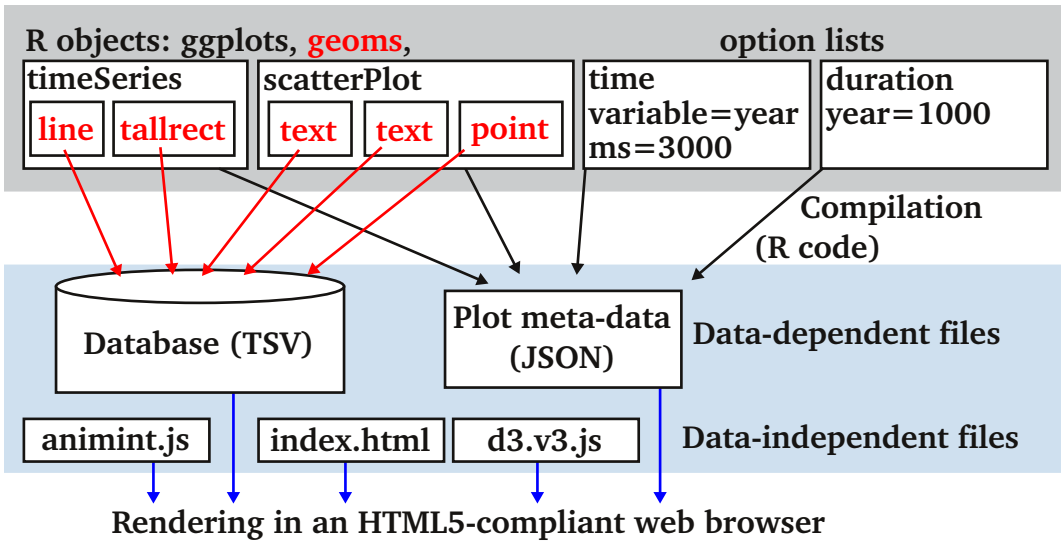


Figure 6.5 A schematic explanation of compilation and rendering in the World Bank visualization. Top: the interactive animation is a list of 4 R objects: 2 ggplots and 2 option lists. Center: animint R code compiles data in ggplot geoms to a database of TSV files (⇨). It also compiles plot meta-data including ggplot aesthetics, animation time options, and transition duration options to a JSON meta-data file (⇨). Bottom: those data-dependent compiled files are combined with data-independent JavaScript and HTML files which render the interactive animation in a web browser (⇨).

```
  first = list(
    country = c("United States", "Thailand")
  )
)
animint2dir(viz)
```

6.3.4 Implementation details

As shown in Figure 6.5, the animint system is implemented in 2 parts: the compiler and the renderer. The compiler is implemented in about 2000 lines of R code that converts a list of ggplots and options to a JSON plot meta-data file and a tab-separated values (TSV) file database.

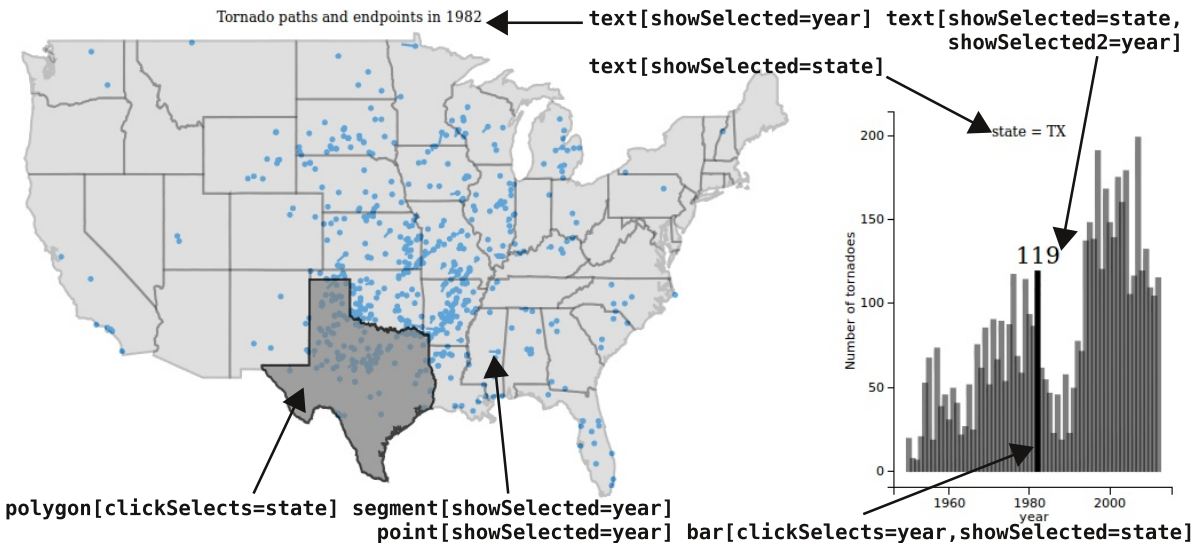


Figure 6.6 Interactive animation of tornadoes recorded from 1950 to 2012 in the United States. Left: map of the lower 48 United States with tornado paths in 1982. The text shows the selected year, and clicking the map changes the selected state, currently Texas. Right: time series of tornado counts in Texas. Clicking a bar changes the selected year, and the text shows selected state and the number of tornadoes recorded there in that year (119 tornadoes in Texas in 1982).

The compiler scans the aesthetics in the ggplots to determine how many selection variables are present, and which geoms to update after a selection variable is updated. It uses ggplot2 to automatically calculate the axes scales, legends, labels, backgrounds, and borders. It outputs this information to the JSON plot meta-data file.

The compiler also uses ggplot2 to convert data variables (e.g. life expectancy and region) to visual properties (e.g. y position and color). The data for each layer/geom are saved in several TSV files, one for each combination showSelected values. Thus for large data sets, the web browser only needs to download the subset of data required to render the current selection (Heer 2013).

When repeated data would be saved in each of the TSV files, an extra common TSV file is created so that the repeated data only need to be stored and downloaded once. In that case, the other TSV files do not store the common data, but are merged with the

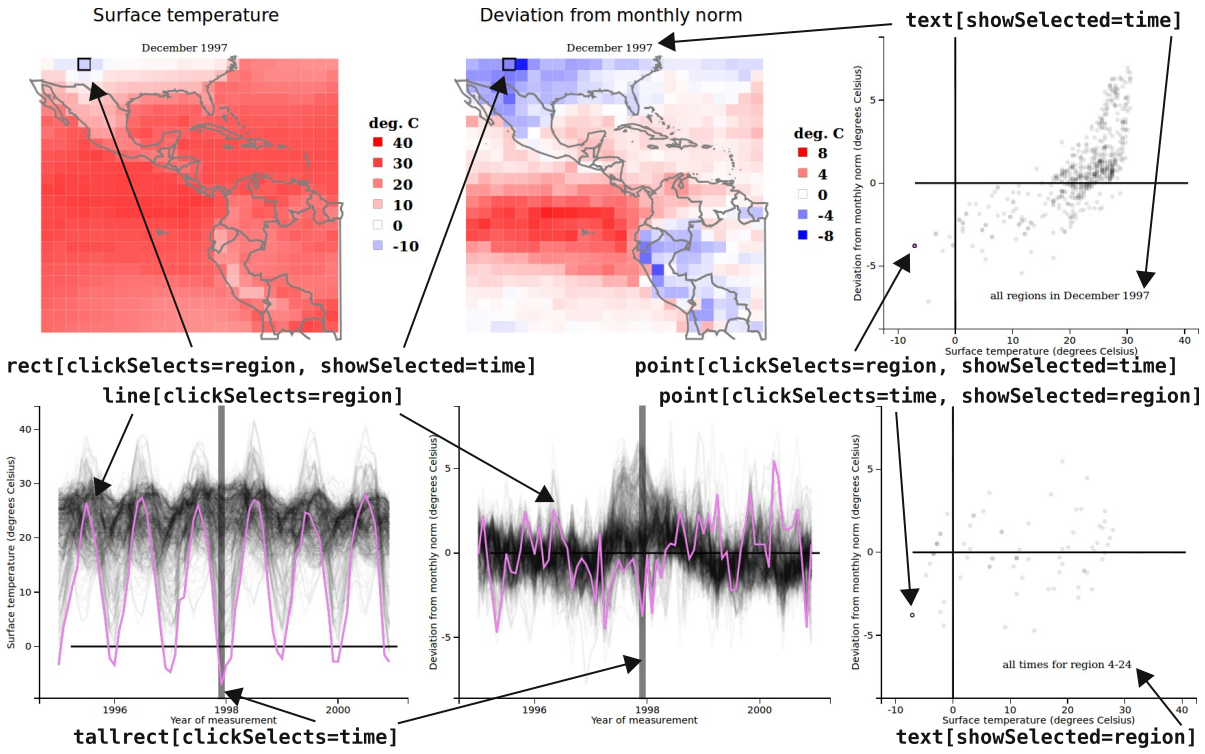


Figure 6.7 Visualization containing 6 linked, interactive, animated plots of Central American climate data. Top: for the selected time (December 1997), maps displaying the spatial distribution of two temperature variables, and a scatterplot of these two variables. The selected region is displayed with a black outline, and can be changed by clicking a rect on the map or a point on the scatterplot. Bottom: time series of the two temperature variables with the selected region shown in violet, and a scatterplot of all times for that region. The selected time can be changed by clicking a background tallrect on a time series or a point on the scatterplot. The selected region can be changed by clicking a line on a time series.

common data after downloading. This method for constructing the TSV file database was developed to minimize the disk usage of animint, particularly for ggplots of spatial maps as in Figure 6.6.

Finally, the rendering engine (`index.html`, `d3.v3.js`, and `animint.js` files) is copied to the plot directory. The `animint.js` renderer is implemented in about 2200 lines of JavaScript/D3 code that renders the TSV and JSON data files as SVG in a web browser. Importantly, animation is achieved by using the JavaScript `setInterval` function, which updates the `time` selection variable every few seconds. Since the compiled plot is just a directory of files, the interactive plots can be hosted on any web server. The interactive plots can be viewed by opening the `index.html` page in any modern web browser.

6.4 Exploring performance & scope with examples

This section attempts to demonstrate a range of visualizations that are supported by animint with more examples. Figure 6.6 shows an interactive animation of tornadoes observed in the United States between 1950 and 2012. At any moment in time, the user can simultaneously view the spatial distribution of tornadoes in the selected year over all states, and see the trend over all years for the selected state. Clicking a state on the map updates the time series bars to show the tornado counts from that state. Clicking a bar on the time series updates the selected year. Figure 6.7 shows an interactive animation of climate time series data observed in Central America. Two maps display the spatial distribution of two temperature variables, which are shown over time in corresponding the time series plots below. Scatterplots also show the relationships between the two temperature variables for the selected time and region. Clicking any of the plots updates all 6 of them. The `clickSelects` and `showSelected` aesthetics make it easy to design this set of 6 linked plots in only 87 lines of code.

Summary statistics describing complexity and performance for examples in this paper, as well as other animint examples, are displayed in Table 6.1. The climate data visualization has noticeably slow animations, since it displays about 88,980 geometric elements at once (http://bit.ly/QcUrhn). We observed this slowdown across all browsers, which suggested that there is an inherent bottleneck when rendering large interactive plots in web browsers using JavaScript and SVG. Another animint with a similar amount of total rows is based on the evolution data (http://bit.ly/O0VTS4), but since it shows less data onscreen (about 2703 elements), it exhibits faster responses to interactivity and animation.

Animint is still useful for creating interactive but non-animated plots when there is not a time variable in the data. In fact, 7 of the 11 examples in Table 6.1 are not animated. For example, linked plots are useful to illustrate complex concepts such as a change point detection model in the breakpoints data (http://bit.ly/1gGYFIV). The user can explore different model parameters and data sets since these are encoded as animint interaction variables.

6.5 Comparison study

In this section we compare our animint implementation with other similar leading systems by creating a given visualization in each system and discussing the pros and cons of the different approaches.

6.5.1 The Grand Tour

The Grand Tour is a well-known method for viewing high dimensional data which requires interactive and dynamic graphics (Asimov 1985). Figure 6.8 shows a grand tour of 300 observations sampled from a correlated tri-variate normal distribution. The left hand view shows the marginal density of each point while the right hand view “tours” through

	LOC	seconds	MB	rows	onscreen	variables	interactive	plots	animated?	Fig
worldPop	17	0.2	0.1	924	624	4	2	2	yes	
WorldBank	20	2.3	2.1	34132	11611	6	2	2	yes	6.3
evolution	25	21.6	12.0	240600	2703	5	2	2	yes	
change	36	2.8	2.5	36238	25607	12	2	3	no	
tornado	39	1.7	6.1	103691	16642	11	2	2	no	6.6
prior	54	0.7	0.2	1960	142	12	3	4	no	
compare	66	10.7	7.9	133958	2140	20	2	5	no	
breakpoints	68	0.5	0.3	4242	667	13	2	3	no	
climate	84	12.8	19.7	253856	88980	15	2	6	yes	6.7
scaffolds	110	56.3	78.5	618740	9051	30	3	3	no	
ChIPseq	229	29.9	78.3	1292464	1156	44	4	5	no	

Table 6.1 Characteristics of 11 interactive visualizations designed with animint. The interactive version of these visualizations can be accessed via http://sugiyama-www.cs.titech.ac.jp/~toby/animint/. From left to right, we show the data set name, the lines of R code (LOC) including data processing but not including comments (80 characters max per line), the amount of time it takes to compile the visualization (seconds), the total size of the uncompressed TSV files in megabytes (MB), the total number of data points (rows), the median number of data points shown at once (onscreen), the number of data columns visualized (variables), the number of `clickSelects`/`showSelected` variables (interactive), the number of linked panels (plots), if the plot is animated, and the corresponding Figure number in this paper (Fig).

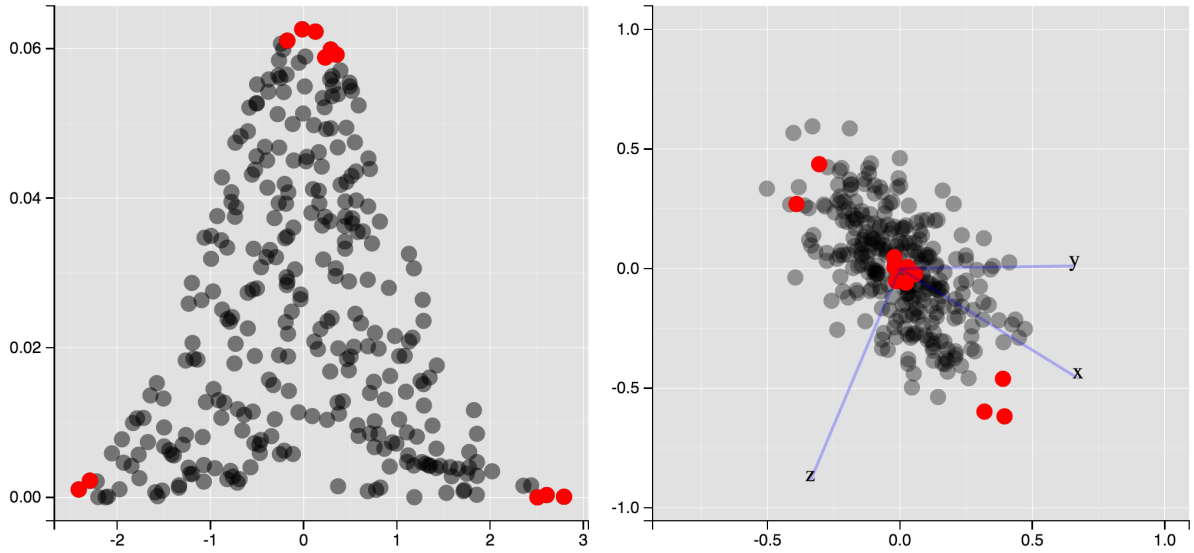


Figure 6.8 Linked selection in a grand tour with animint. A video demonstration can be viewed online at https://vimeo.com/160720834

2D projections of the 3D data. There are many ways to choose projections in a tour, and many ways to interpolate between projections, most of which can be programmed fairly easily using R and relevant add-on packages. In this case, we used the R package tourr, which uses the geodesic random walk (i.e., random 2D projection with geodesic interpolation) in its grand tour algorithm (Wickham et al. 2011).

When touring data, it is generally useful to link low-dimensional displays with the tour itself. The video in Figure 6.8 was generated with our current animint implementation, and points are selected via mouse click which reveal that points with high marginal density are located in the ellipsoid center while points with a low marginal density appear near the ellipsoid border. In this case, it would be convenient to also have brush selection, as we demonstrate in Figure 6.9 which implements the same touring example using the R packages ggvis and shiny. The brush in Figure 6.9 is implemented with shiny's support for brushing static images, which currently does not support multiple brushes, making it difficult to select non-contiguous regions.

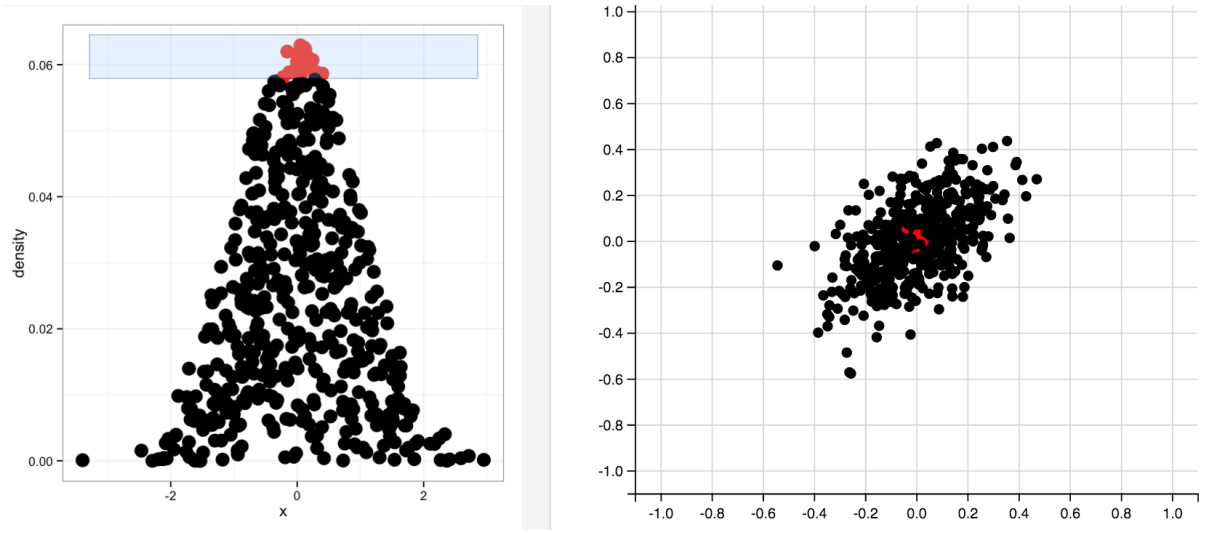


Figure 6.9 Linked selection in a grand tour with ggvis and shiny. A video demonstration can be viewed online at https://vimeo.com/160825528

This example helps point out a few other important differences in using animint versus ggvis+shiny to implement "multiple linked and dynamic views" as described in (Ahlberg, Williamson, and Shneiderman 1991) and (Buja et al. 1991). Maintaining state of the linked brush in Figure 6.9 requires both knowledge and clever use of some sophicated programming techniques such as closures and reactivity. It also requires knowledge of the shiny web application framework and a new approach to the grammar of graphics. On the other hand, maintaining state in Figure 6.8 requires a few different `clickSelects`/`showSelected` mappings. As a result, we believe animint provides a more elegant user interface for this application.

The tourring example also helps point out important consequences of the design and implementation of these two different systems. As mentioned in Section 6.3.4, our current animint implementation requires every subset of data to be precomputed before render time. For visualizations such as tours, where it is more efficient to perform statistical computations on-the-fly, this can be a harsh restriction, but this is a restriction of our current implementation (not a restriction of the framework itself). As a result, when touring a large high-dimensional space, where many projections are needed, ggvis+shiny

may be desirable since the projections are computed on the server and sent to the browser in real-time. This works fine when the application is running and viewed on the same host machine, but viewing such an application hosted on a remote machine can produce staggered animations since client-server requests must be performed, processed, and rendered roughly 30 times a second. Also, generally speaking, the animint system results a more pleasant experience when it comes to hosting and sharing applications since it doesn't require a Web Server with R and special software already installed.

6.5.2 World Bank Example

We also recreated Figure 6.3 using ggvis+shiny (see http://bit.ly/1SsJKlN) and Tableau (see http://bit.ly/worldBank-tableau). Even as experienced ggvis+shiny users, we found it quite difficult to replicate this example, and were not able completely replicate it due to a lack of a mechanism for coordinating indirect and direct manipulations. Overall the visualization is pretty similar, but lacks a few important features. In particular, there is no way to control the selected year using both the slider (indirect) and clicking on the ggvis plot (direct). It also lacks the ability to click on a countries time series and label the corresponding point on the scatterplot. This might be possible, but we could not find a way to update a plot based on a click event on a different plot. Even with this lack of functionality, the ggvis+shiny is significantly more complicated and requires more code (about 100 lines of code compared to 30).

It was also impossible to completely replicate Figure 6.3 using Tableau essentially because the example requires a *layered* approach to the grammar of graphics. In particular, since graphical marks and interaction source/target(s) must derive from the same table in Tableau, it was impossible to control the clickable multiple time series and the clickable tallrects in different ways based on the two different selection variables. In other words,

in Tableau, selections are managed on the plot level, but in animint, selections are specific to each graphical layer.

6.6 User feedback and observations

By working with researchers in several fields of research, we have created a wide variety of interactive visualizations using animint. Typically, the researchers have a complex data set that they wish to visualize, but they do not have the expertise or time to create an interactive data visualization. The animint system made it easy to collaborate with the various domain experts, who were able to provide us with annotated sketches of the desired plots, which we then translated to animint R code. In this section we share comments and constructive criticism that we have obtained from our users.

6.6.1 User perspective

For the `prior` data visualization (http://bit.ly/1peIT7t), the animint user is a machine learning researcher who developed an algorithm and applied it to 4 benchmark data sets. He wanted to explore how his algorithm performed, in comparison to a baseline learning algorithm. He appreciated the intuition about his algorithm's performance that he learned from the interactive plots: "Interactive plotting allows us to explore all relationships of our high-dimensional dataset and gives us an intuitive understanding of the performance of our proposed algorithm. An intuitive understanding of the results is important since it shows under which conditions our proposed method works well and provides avenues for further research."

Another user from a machine learning background found the interactive plots useful for presenting his work: '`the`regularization path' is a difficult concept to demonstrate in my research. The animint (http://bit.ly/1gVb8To) helped greatly by rendering an interac-

tive plot of regularization path, likelihood, and graph at the same time and illustrating their connections. It also reveals an interesting phenomenon that maximizing the testing likelihood actually gives many false positives."

In another application, the animint user was a genomics researcher: "viewing and exploring my complex intestinal microbiome dataset in animint allowed me to grasp the patterns and relationships between samples at an almost intuitive level. The interactive aspect of it was very helpful for browsing through the dataset."

Finally, users also appreciated the simple web interface, and the detail that is possible to show in interactive plots, but impossible to show in publications: "… the web interface is simple and easy to use. It also enables us to publish more detailed interactive results on our website to accompany the results presented in publications."

6.6.2 Developer perspective

R users, and in particular ggplot2 users, have found that animint is easy to learn and use. One statistics Ph.D. student writes, "animint is a fantastic framework for creating interactive graphics for someone familiar with R and ggplot2's grammar of graphics implementation. The API is very intuitive and allows one to quickly bring their static graphics to life in a way that facilitates exploratory data analysis."

6.7 Limitations and future work

A number of limitations derive from the fact that some plot features are computed once during the compilation step and remain static on a rendered plot. For example, users are unable to change variable mappings after compilation. Also, when different data subsets have very different ranges of values, it may be preferable to recompute scales

when `clickSelects` selection(s) change. A future implementation of animint would benefit from changes to the compiler and renderer that allow scales to be updated after each click. Some of these limitations can be resolved by adding interactive widgets to "recompile" components hard-coded in the plot meta information. In fact, animint makes it easy to embed visualizations inside of shiny web applications, and we have an example of interactively redefining variable mappings (http://bit.ly/animint-shiny).

Our compiler also currently takes advantage of ggplot2 internals to compute statistics and positional adjustments before rendering. As a result, statistics/positions will not dynamically recompute based on selections. In other words, using `clickSelects`/`showSelected` with non-identity statistic(s)/position(s) may not generate a sensible result. It would be possible, but a significant amount of work, to transfer these computations from the compiler to the renderer.

Another set of limitations derive our current restriction that all subsets (corresponding to each possible selection) must be precomputed before render time. As eluded to in Section 6.5.1, if there is a large space of possible selections, it is impractical to precompute every subset before viewing. Therefore, it would be useful if the renderer could dynamically compute subsets when new selections are made.

Our implementation is also limited to two specific types of direct manipulation: selecting graphical elements via mouse click (`clickSelects`), and showing/hiding related elements (`showSelected`). However, the framework described in Section 6.3.1 is not restricted to a particular event type, so `hoverSelects` and `brushSelects` aesthetics could be added, for instance. There are other types of interaction that should be added, that wouldn't require additional extensions to the grammar of graphics, such as: zooming, panning, and plot resizing.

6.8 Conclusion

Our R package animint extends ggplot2's layered grammar of graphics implementation for a declarative approach to producing interactive and dynamic web graphics. By adding two aesthetics to specify selection source(s) and target(s), ggplot2 users can quickly and easily create animations with smooth transitions and perform database queries via direct manipulation of linked views. As a result, animint is a useful tool not only for exploratory data analysis, but also for the presentation and distribution of interactive statistical graphics.

Acknowledgements

The authors wish to thank animint users MC Du Plessis, Song Liu, Nikoleta Juretic, and Eric Audemard who have contributed constructive criticism and helped its development.

7 Interactive data visualization on the web using R

7.1 Introduction

Cook, Buja, and Swayne (2007) proposed a taxonomy 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 (Swayne, Cook, and Buja 1998).

This chapter focuses on linking multiple views for posing queries, making comparisons, and exploring relationships in multi-dimensional data. Three different case studies are used to demonstrate these techniques

7.2 Case Studies

7.2.1 Exploring pedestrian counts

The first example uses pedestrian counts from around the city published on the City of Melbourne's open data platform (Melbourne, n.d.). The City currently maintains at least 43 sensors (spread across the central business district), which record the number of pedestrians that walk by every hour. The analysis presented here uses counts starting in 2013 when all 42 of these sensors began recording counts, all the way through July of 2016. This code for obtaining and pre-processing this data, as well as the (cleaned-up) data is made available in the R package **pedestrians** (Sievert 2016). The main dataset of interest is named `pedestrians` and contains nearly 1 million counts, but over 400,000 counts are missing:

```
data(pedestrians, package = "pedestrians")
summary(is.na(pedestrians$Counts))
#>    Mode   FALSE    TRUE    NA's
#> logical  942624  407189       0
```

7.2.1.1 Exploring missingness

Trying to visualize time series of this magnitude in its raw form simply is not useful, but we can certainly extract features and use them to guide our analysis. Figure 7.1 shows the number of missing values broken down by sensor. Southbank has the most missing values by a significant amount and the hand-full of stations with the fewest missing values have nearly the same number of missing values. One thing that Figure 7.1 can not tell us is *where* these missing values actually occur. To investigate this question, it would be helpful to link this information to actual time series.

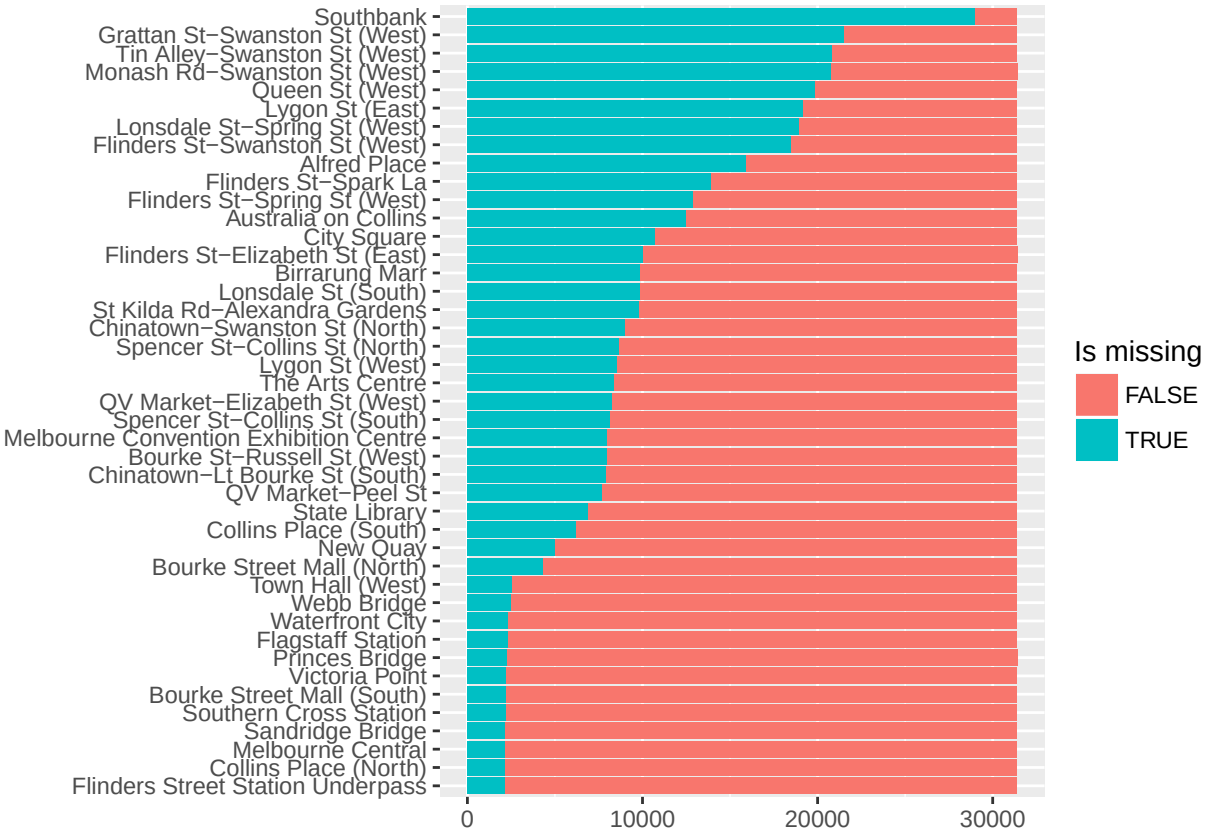


Figure 7.1 Missing values by station.

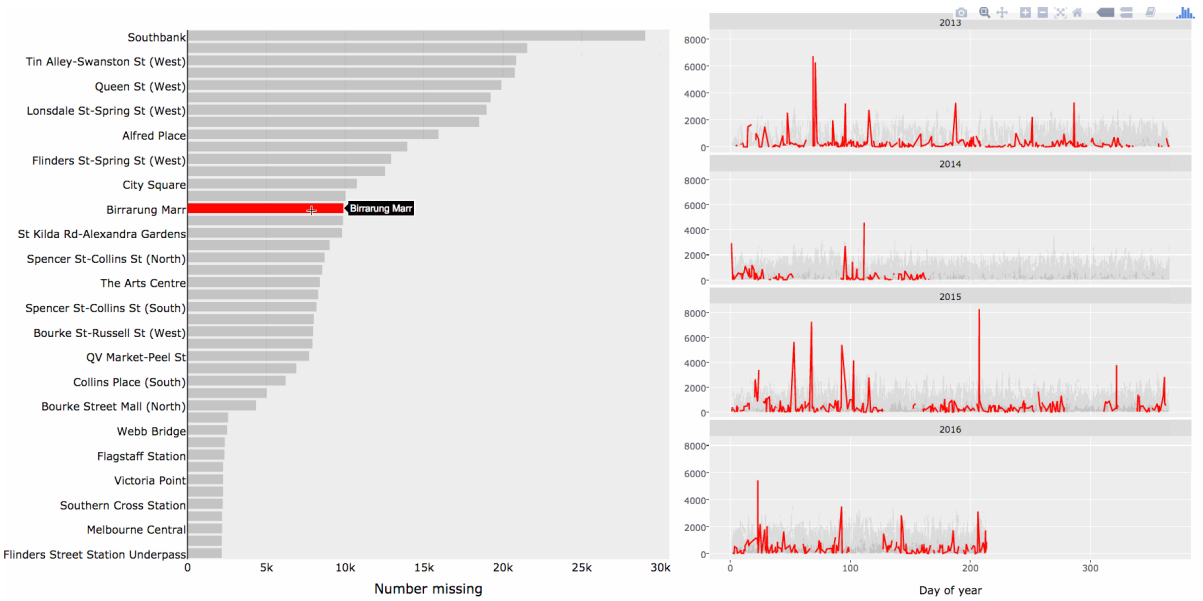


Figure 7.2 An interactive bar chart of the number of missing counts by station linked to a sampled time series of counts. See here for the corresponding video and here for the interactive figure.

Again, visualizing the entire time series all at once is not realistic, but we can still gain an understanding of the relationship between missingness and time via down-sampling techniques. Figure 7.2 displays an interactive version of Figure 7.1 linked to a down-sampled (stratified within sensor location) time series. Clicking on a particular bar reveals the sampled time series for that sensor location. The top one-third of all sensors are relatively new sensors, the middle third generally encounter long periods of down-time, while the bottom third seem to have very little to no pattern in their missingness.

7.2.1.2 Exploring trend and seasonality

A time series Y_t can be thought of as a linear combination of at least three components:

$$Y_t = T_t + S_t + I_t, \quad t \in \{1, \ldots, T\}$$

where T_t is the trend, S_t is the seasonality, and I_t is the "irregular" component (i.e., remainder). For the sensor data, we could imagine having multiple types of seasonality (e.g., hour, day, month, year), but as Figure 7.2 showed, year doesn't seem to have much effect, and as we will see later, hour of day has a significant effect (which is sensible for most traffic data), so we focus on hour of day as a seasonal component. Estimating these components has important applications in time series modeling (e.g., seasonal adjustments), but we could also leverage these estimates to produce further time series "features" to guide our graphical analysis.

There are many to go about modeling and estimating these time series components. Partly due to to its widespread availability in R, the `stl()` function, which is based on LOESS smoothing, is a popular and reasonable approach (R. B. Cleveland and Terpenning 1990); (R Core Team 2015). Both the **anomalous** and **tscognostics** R packages use estimates from `stl()`[1] to measure the strength of trend (as $\hat{Var}(T_t)$) and strength of seasonality (as $\hat{Var}(S_t)$) (Hyndman, Wang, and Laptev 2016); (Wang 2016). From these estimates, they produce other informative summary statistics, such as the seasonal peak ($Max_t(\hat{S}_t)$), trough ($Min_t(\hat{S}_t)$), spike ($Var[(Y_t - \bar{Y})^2]$); as well as trend linearity ($\hat{\beta}_1$) and curvature ($\hat{\beta}_2$) (coefficients from a 2nd degree polynomial fit to the estimated trend $\mu = \beta_0 + \beta_1 \hat{T}_t + \beta_2 \hat{T}_t^2$).

After computing these 7 different summary statistics for each sensor, we may graphically explore the 7-dimensional space that they live in, as shown in Figure 7.3. The top row of Figure 7.3 shows different views of the summary statistics: in the left-hand side is an animated display, which interpolates between random 2D projections (aka, a grand tour) of the 7 dimensions, and on the right-hand side is a parallel coordinates plot (Asimov 1985). For the parallel coordinates plot, each dimension is standardized to have mean 0

[1] If no seasonal component exists, a Generalized Additive Models is used to estimate the trend component (Wood 2006).

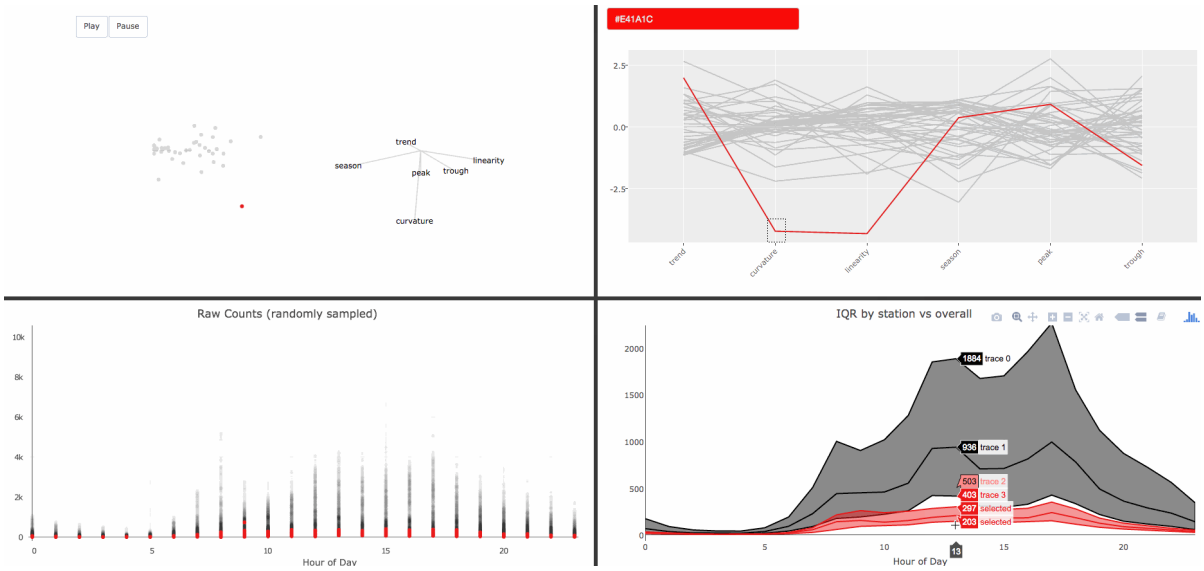


Figure 7.3 Linking views of counts (second row) with features generated from seasonal trend decomposition (first row). The top left panel shows a grand tour of the 7 dimensional space and the top right panel a parallel coordinates plot. In both views, it is apparent that one station (Tin Alley-Swanson St), highlighted in red, has irregular curvature and linearity compared to the other stations. By linking raw counts and the hourly IQR, we can see that this station experiences relatively low traffic (red) compared to overall traffic (black). See here for the corresponding video and here for the interactive figure.

and standard deviation 1. This helps reveal an unusual station (Tin Alley-Swanson St) in terms of both curvature and linearity.

The static image in Figure 7.3 captures the state of the interactive graphic after Tin Alley-Swanson St has been selected and highlighted in red. The same sampled counts behind Figure 7.2 are used to generate the scatterplot (count versus hour of day) in the lower left-hand panel. The lower right-hand panel displays the inter-quartile range (IQR) by hour for the entire dataset. The black ribbon is the overall IQR and the red ribbon is the IQR for Tin Alley-Swanson St. Having both the IQR and the raw counts in the same display is useful for gaining an understanding of the variation in seasonality over time (IQR), while still being able to discover outliers or unusual patterns (raw).

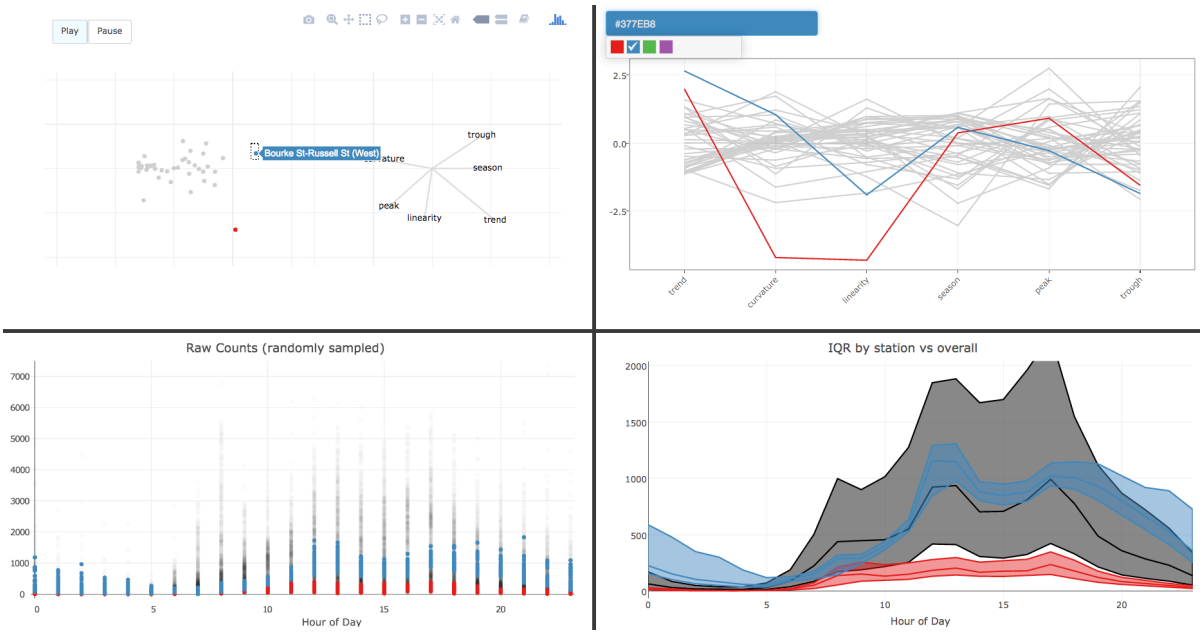


Figure 7.4 Using persistent linked brushing to compare the trend and seasonality of Tin Alley-Swanson St. to Bourke St-Russell St (West). See here for the corresponding video and here for the interactive figure.

A careful inspection of the grand tour in Figure 7.3 hints at another sensor that is relatively close to Tin Alley-Swanson St in the feature space. Figure 7.4 uses a persistent linked brush to highlight this other sensor (Bourke St-Russell St) in blue. Replaying the tour (as shown in the video corresponding to Figure 7.4) with these points highlighted different colors makes it easier to track their distance (relative to each other as well as the other points) throughout the feature space.

A fair amount of insight can be extracted from the interactive graphic in Figure 7.3, but focusing just on the STL-based features is somewhat limiting. There are certainly other features that capture aspects of the time series that these features have missed. In theory, the mathematics and the visualization techniques behind Figure 7.3 can be extended to any number of dimensions. In practice, technology and time typically limits us to tens to hundreds of dimensions. The next section incorporates more features and also links this information to a geographic map so we can investigate the relationship between geographic location and certain features.

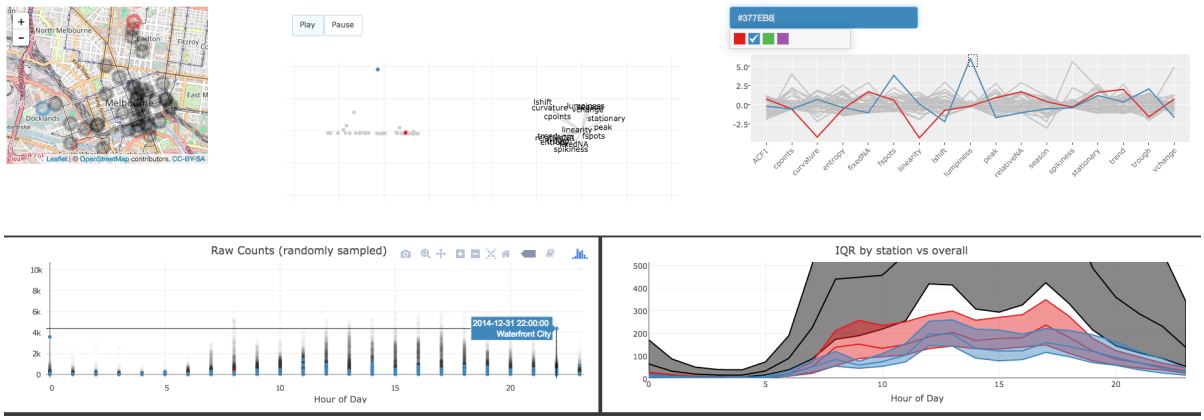


Figure 7.5 Seventeen time series features linked to a geographic map as well as raw counts. This static image was generated using a persistent brush to compare Tin Alley-Swanson St. (in red) to Waterfront City (in blue). In addition to being unusual in the feature space, these sensors are also on the outskirts of the city. See here for the corresponding video and here for the interactive figure.

7.2.1.3 Exploring many features

Figure 7.5 is an extension of Figure 7.3 to incorporate 10 other time series features, as well as a map of Melbourne. In this much larger feature space, Tin Alley-Swanson St. (highlighted in red) is still an unusual sensor, but not quite as unusual as Waterfront City (highlighted in blue). Also, rather interestingly, both of these sensors are located on the outskirts of the city, relative to the other sensors. It appears Waterfront City is so noticeably unusual due to its very large value of lumpiness (defined as the variance of block variances of size 24). Inspecting the unusually high raw counts for this station reveals some insight as to why that is case – the counts are relatively low year-round, but then spike dramatically on new years eve and on January 26th. A Google search reveals that Waterfront City is a popular place to watch fireworks. This is a nice example of how interactive graphics can help us discover and explain *why* unusual patterns occur.

In addition to discovering interesting details, we can also use Figure 7.5 to make meaningful comparisons in overall behavior across numerous sensors. Figure 7.6 uses a persistent

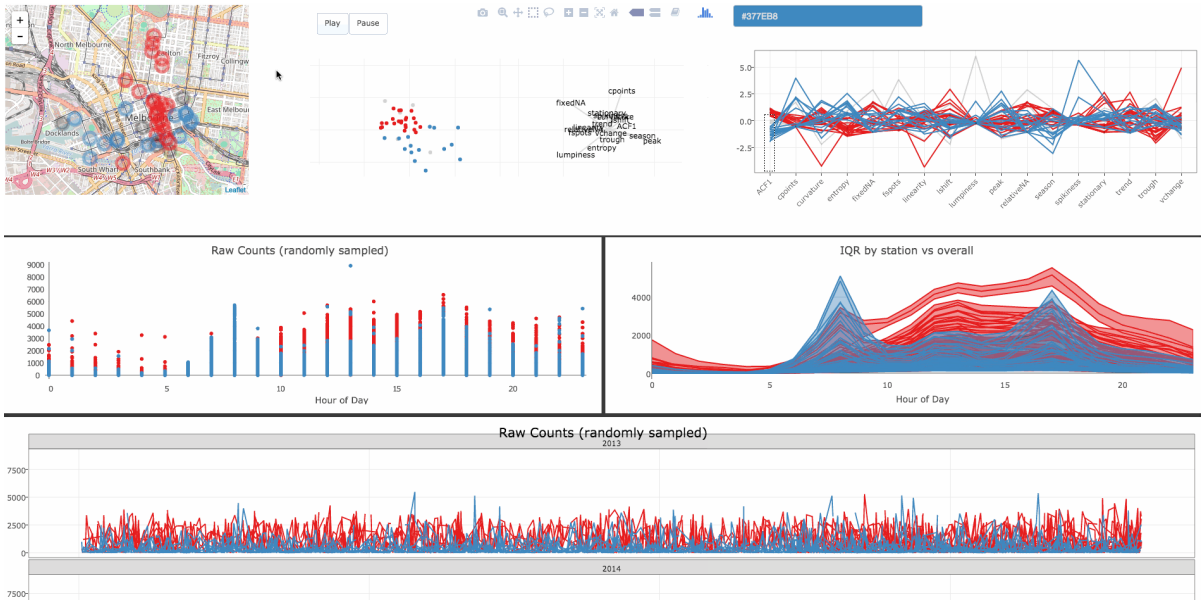


Figure 7.6 Sensors with high first order autocorrelation (in red) versus sensors with low autocorrelation (in blue). See here for the corresponding video and here for the interactive figure.

linked brush to compare sensors with a high first order autocorrelation ($Corr(Y_t, Y_{t-1})$), in red, against sensors with low autocorrelation, in blue. A few interesting observations can be made from this selection set.

The most striking relationship with respect to autocorrelation in Figure 7.6 is in the geographic locations. Sensors with high autocorrelation (red) appear along Swanson St. – the heart of the central business district in Melbourne. These stations experience a fairly steady flow of traffic throughout the day since both tourists and people going to/from work use nearby trains/trams to get from place to place. On the other hand, sensors with a low autocorrelation[2] see the bulk of their traffic at the start and end of the work day. It seems that this feature alone would provide a fairly good criteria for splitting these sensors into 2 groups, which we could verify and inspect further via hierarchical clustering.

[2]It should be noted that the (raw) autocorrelation is positive for each station with a minimum of 0.66, median of 0.83, and max of 0.94.

Figure 7.7 Linking a dendrogram of hierarchical clustering results to See here for the corresponding video and here for the interactive figure.

Figure 7.7 links a dendrogram of a hierarchical cluster analysis using the complete linkage method (via the `hclust()` function in R) to other views of the data. A persistent brush selects all the sensors under a given node – effectively providing a tool to choose a number of clusters and visualize model results in the data space (in real-time). Splitting the dendrogram at the root node splits the sensors into 2 groups (red and green) confirms our prior suspicion from earlier – sensors on Swanson St (high autocorrelation) are most different from sensors on the outskirts of the city (low autocorrelation). Increasing the number of clusters to 3-4 splits off the unusual sensors that we identified in out previous analysis (Waterfront City, Birrarung Marr, and Tin Alley-Swanson St).

7.2.2 Exploring Australian election data

7.2.3 Exploring disease outbreaks

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

- Explain how

7.3 Discussion

TODO: discussion the tech, costs of server-client infrastructure

7.4 Acknowledgements

Thank you to the organizers (Nicholas Tierney, Miles McBain, Jessie Roberts) of the rOpenSci hackathon as well as my group members (Heike Hofmann, Di Cook, Rob Hyndman, Ben Marwick, Earo Wang) where the **eechidna** package first took flight. Thank you to Di Cook and Earo Wang for sparking my interest in the pedestrians data, helping implement the **pedestrians** R package, and many fruitful discussions (some with Heike Hofmann and Rob Hyndman).

8 plotly for R

This chapter contains many web-based interactive graphics and moving images. Since moving images are not allowed on the University's publishing platform, I highly suggest viewing the web-based version of this chapter – https://cpsievert.github.io/plotly_book/

8.1 Two approaches, one object

There are two main ways to initiate a plotly object in R. The `plot_ly()` function transforms *data* into a plotly object, while the `ggplotly()` function transforms a *ggplot object* into a plotly object (Wickham 2009a); (Sievert et al. 2016). Regardless of how a plotly object is created, printing it results in an interactive web-based visualization with tooltips, zooming, and panning enabled by default. The R package also has special semantics for arranging, linking, and animating plotly objects. This chapter discusses some of the philosophy behind each approach, explores some of their similarities, and explains why understanding both approaches is extremely powerful.

The initial inspiration for the `plot_ly()` function was to support plotly.js chart types that **ggplot2** doesn't support, such as 3D surface and mesh plots. Over time, this effort snowballed into an interface to the entire plotly.js graphing library with additional abstractions inspired by the grammar of graphics (Wilkinson 2005). This newer "non-ggplot2" interface to plotly.js is currently not, and may never be, as fully featured as **ggplot2**. Since we can already translate a fairly large amount of ggplot objects to plotly

objects, I'd rather not reinvent those same abstractions, and advance our ability to link multiple views.

The next section uses a case study to introduce some of the similarities between `ggplotly()`/`plot_ly()`, introduces the concept of a data-plot-pipeline, and also demonstrates how to extend `ggplotly()` with functions that can modify plotly objects.

8.1.1 A case study of housing sales in Texas

The **plotly** package depends on **ggplot2** which bundles a data set on monthly housing sales in Texan cities acquired from the TAMU real estate center. After the loading the package, the data is "lazily loaded" into your session, so you may reference it by name:

```
library(plotly)
txhousing
#> # A tibble: 8,602 × 9
#>      city  year month sales   volume median listings inventory  date
#>     <chr> <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
#> 1 Abilene  2000     1    72  5380000  71400      701       6.3  2000
#> 2 Abilene  2000     2    98  6505000  58700      746       6.6  2000
#> 3 Abilene  2000     3   130  9285000  58100      784       6.8  2000
#> 4 Abilene  2000     4    98  9730000  68600      785       6.9  2000
#> 5 Abilene  2000     5   141 10590000  67300      794       6.8  2000
#> 6 Abilene  2000     6   156 13910000  66900      780       6.6  2000
#> # ... with 8,596 more rows
```

In attempt to understand house price behavior over time, we could plot `date` on x, `median` on y, and group the lines connecting these x/y pairs by `city`. Using **ggplot2**, we can *initiate* a ggplot object with the `ggplot()` function which accepts a data frame and a

mapping from data variables to visual aesthetics. By just initiating the object, **ggplot2** won't know how to geometrically represent the mapping until we add a layer to the plot via one of `geom_*()` (or `stat_*()`) functions (in this case, we want `geom_line()`). In this case, it is also a good idea to specify alpha transparency so that 5 lines plotted on top of each other appear as solid black, to help avoid overplotting.

If you're new to **ggplot2**, the ggplot2 cheatsheet provides a nice quick overview. The online docs or R graphics cookbook are helpful for learning by example, and the ggplot2 book provides a nice overview of the conceptual underpinnings.

```
p <- ggplot(txhousing, aes(date, median)) +
  geom_line(aes(group = city), alpha = 0.2)
```

8.1.1.1 The `ggplotly()` function

Now that we have a valid **ggplot2** object, `p`, the **plotly** package provides the `ggplotly()` function which converts a ggplot object to a plotly object. By default, it supplies the entire aesthetic mapping to the tooltip, but the `tooltip` argument provides a way to restrict tooltip info to a subset of that mapping. Furthermore, in cases where the statistic of a layer is something other than the identity function (e.g., `geom_bin2d()` and `geom_hex()`), relevant "intermediate" variables generated in the process are also supplied to the tooltip. This provides a nice mechanism for decoding visual aesthetics (e.g., color) used to represent a measure of interest (e.g, count/value). Figure 8.1 demonstrates tooltip functionality for a number of scenarios, and uses `subplot()` function from the **plotly** package (discussed in more detail in Arranging multiple views) to concisely display numerous interactive versions of ggplot objects.

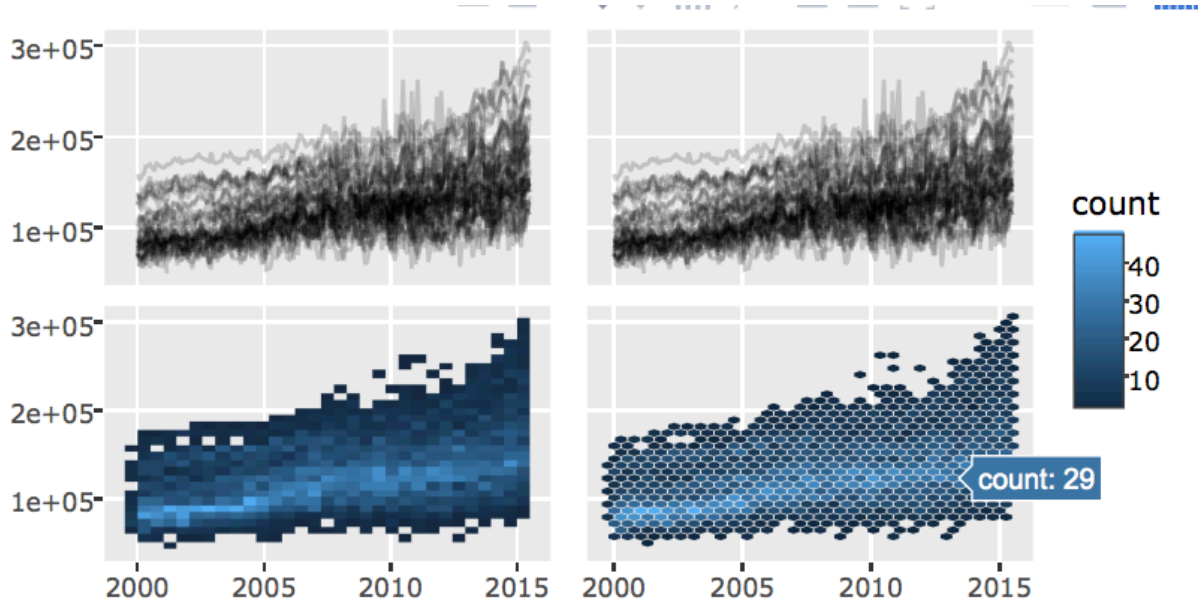


Figure 8.1 Monthly median house price in the state of Texas. The top row displays the raw data (by city) and the bottom row shows 2D binning on the raw data. The binning is helpful for showing the overall trend, but hovering on the lines in the top row helps reveal more detailed information about each city.

```
subplot(
  p, ggplotly(p, tooltip = "city"),
  ggplot(txhousing, aes(date, median)) + geom_bin2d(),
  ggplot(txhousing, aes(date, median)) + geom_hex(),
  nrows = 2, shareX = TRUE, shareY = TRUE,
  titleY = FALSE, titleX = FALSE
)
```

Although **ggplot2** does not have a `text` aesthetic, the `ggplotly()` function recognizes this aesthetic and displays it in the tooltip by default. In addition to providing a way to supply "meta" information, it also provides a way to customize your tooltips (do this by restricting the tooltip to the text aesthetic – `ggplotly(p, tooltip = "text")`)

The `ggplotly()` function translates most things that you can do in **ggplot2**, but not quite everything. To help demonstrate the coverage, I've built a plotly version of the ggplot2 docs. This version of the docs displays the `ggplotly()` version of each plot in a static form (to reduce page loading time), but you can click any plot to view its interactive version. The next section deomnstrates how to create plotly.js visualizations via the R package, without **ggplot2**, via the `plot_ly()` function. We'll then leverage those concepts to extend `ggplotly()`.

8.1.1.2 The `plot_ly()` interface

8.1.1.2.1 The Layered Grammar of Graphics

The cognitive framework underlying the `plot_ly()` interface draws inspiration from the layered grammar of graphics (Wickham 2010b), but in contrast to `ggplotly()`, it provides a more flexible and direct interface to plotly.js. It is more direct in the sense that it doesn't call **ggplot2**'s sometimes expensive plot building routines, and it is more flexible in the sense that data frames are not required, which is useful for visualizing matrices, as shown in Get Started. Although data frames are not required, using them is highly recommended, especially when constructing a plot with multiple layers or groups.

When a data frame is associated with a **plotly** object, it allows us to manipulate the data underlying that object in the same way we would directly manipulate the data. Currently, `plot_ly()` borrows semantics from and provides special plotly methods for generic functions in the **dplyr** and **tidyr** packages (Wickham and Francois 2014); (Wickham 2016a). Most importantly, `plot_ly()` recognizes and preserves groupings created with **dplyr**'s `group_by()` function.

```
library(dplyr)

tx <- group_by(txhousing, city)
```

```
# initiate a plotly object with date on x and median on y
p <- plot_ly(tx, x = ~date, y = ~median)
# plotly_data() returns data associated with a plotly object, note the group attrib
plotly_data(p)
#> Source: local data frame [8,602 x 9]
#> Groups: city [46]
#>
#>       city  year month sales   volume median listings inventory  date
#>      <chr> <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
#> 1 Abilene  2000     1    72  5380000  71400      701       6.3  2000
#> 2 Abilene  2000     2    98  6505000  58700      746       6.6  2000
#> 3 Abilene  2000     3   130  9285000  58100      784       6.8  2000
#> 4 Abilene  2000     4    98  9730000  68600      785       6.9  2000
#> 5 Abilene  2000     5   141 10590000  67300      794       6.8  2000
#> 6 Abilene  2000     6   156 13910000  66900      780       6.6  2000
#> # ... with 8,596 more rows
```

Defining groups in this fashion ensures `plot_ly()` will produce at least one graphical mark per group.[1] So far we've specified `x`/`y` attributes in the plotly object `p`, but we have not yet specified the geometric relation between these x/y pairs. Similar to `geom_line()` in **ggplot2**, the `add_lines()` function connects (a group of) x/y pairs with lines in the order of their `x` values, which is useful when plotting time series as shown in Figure 8.2.

```
# add a line highlighting houston
add_lines(
  # plots one line per city since p knows city is a grouping variable
```

[1] In practice, it's easy to forget about "lingering" groups (e.g., `mtcars %>% group_by(vs, am) %>% summarise(s = sum(mpg))`), so in some cases, you may need to `ungroup()` your data before plotting it.

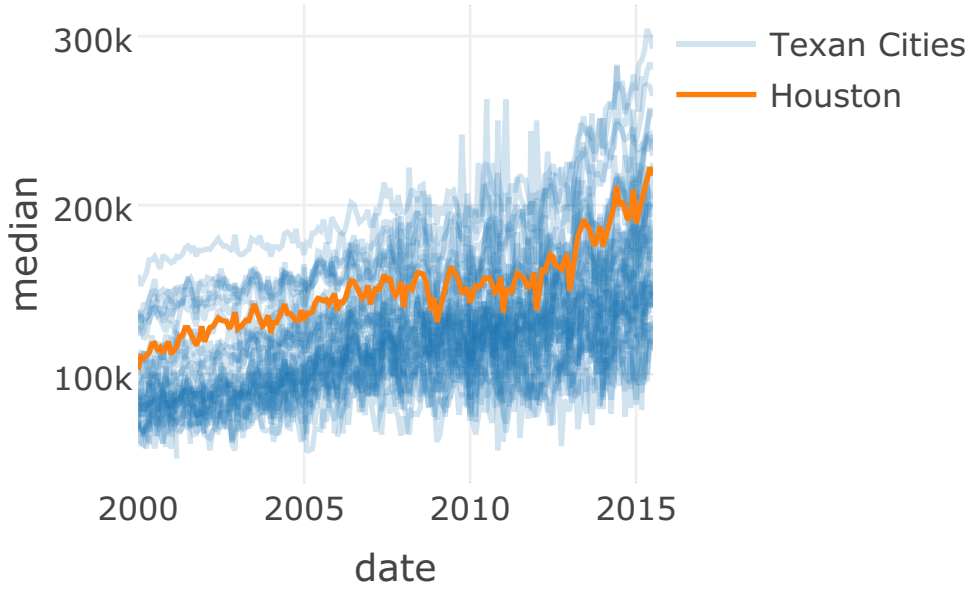


Figure 8.2 Monthly median house price in Houston in comparison to other Texan cities.

```
  add_lines(p, alpha = 0.2, name = "Texan Cities", hoverinfo = "none"),
  name = "Houston", data = filter(txhousing, city == "Houston")
)
```

The **plotly** package has a collection of `add_*()` functions, all of which inherit attributes defined in `plot_ly()`. These functions also inherit the data associated with the plotly object provided as input, unless otherwise specified with the `data` argument. I prefer to think about `add_*()` functions like a layer in **ggplot2**, which is slightly different, but related to a plotly.js trace. In Figure 8.2, there is a 1-to-1 correspondence between layers and traces, but `add_*()` functions do generate numerous traces whenever mapping a discrete variable to a visual aesthetic (e.g., color). In this case, since each call to `add_lines()` generates a single trace, it makes sense to `name` the trace, so a sensible legend entry is created.

In the first layer of Figure 8.2, there is one line per city, but all these lines belong a single trace. We *could have* produced one trace for each line, but this is way more computationally expensive because, among other things, each trace produces a legend

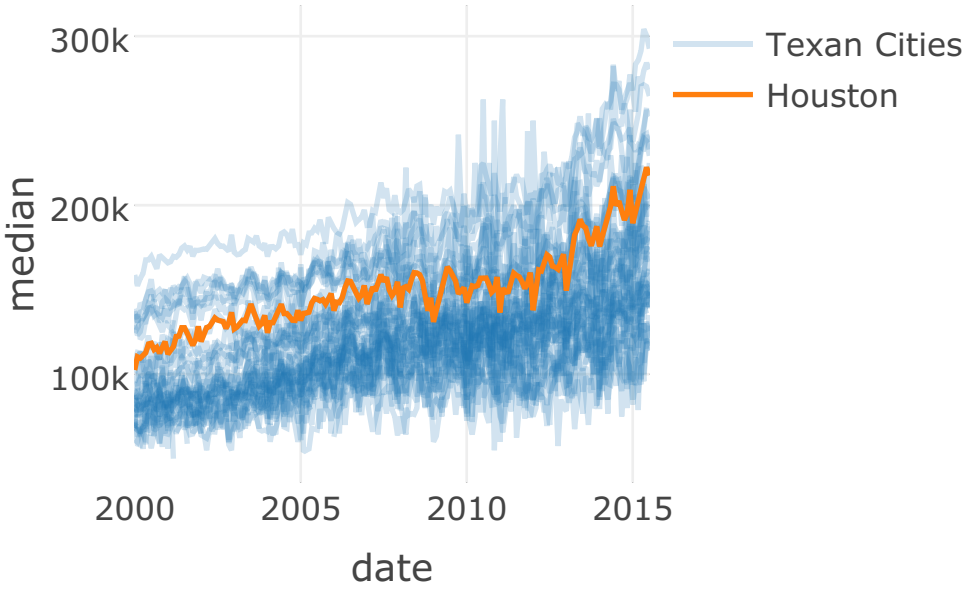
entry and tries to display meaningful hover information. It is much more efficient to render this layer as a single trace with missing values to differentiate groups. In fact, this is exactly how the group aesthetic is translated in `ggplotly()`; otherwise, layers with many groups (e.g., `geom_map()`) would be slow to render.

8.1.1.2.2 The data-plot-pipeline

Since every **plotly** function modifies a plotly object (or the data underlying that object), we can express complex multi-layer plots as a sequence (or, more specifically, a directed acyclic graph) of data manipulations and mappings to the visual space. Moreover, **plotly** functions are designed to take a plotly object as input, and return a modified plotly object, making it easy to chain together operations via the pipe operator (`%>%`) from the **magrittr** package (Bache and Wickham 2014). Consequently, we can re-express Figure 8.2 in a much more readable and understandable fashion.

```
allCities <- txhousing %>%
  group_by(city) %>%
  plot_ly(x = ~date, y = ~median) %>%
  add_lines(alpha = 0.2, name = "Texan Cities", hoverinfo = "none")


allCities %>%
  filter(city == "Houston") %>%
  add_lines(name = "Houston")
```



Sometimes the directed acyclic graph property of a pipeline can be too restrictive for certain types of plots. In this example, after filtering the data down to Houston, there is no way to recover the original data inside the pipeline. The `add_fun()` function helps to work-around this restriction[2] – it works by applying a function to the plotly object, but does not affect the data associated with the plotly object. This effectively provides a way to isolate data transformations within the pipeline[3]. Figure 8.3 uses this idea to highlight both Houston and San Antonio.

```
allCities %>%
  add_fun(function(plot) {
    plot %>% filter(city == "Houston") %>% add_lines(name = "Houston")
  }) %>%
  add_fun(function(plot) {
    plot %>% filter(city == "San Antonio") %>% add_lines(name = "San Antonio")
  })
```

[2]Credit to Winston Chang and Hadley Wickham for this idea. The `add_fun()` is very much like `layer_f()` function in **ggvis**.

[3]Also, effectively putting a pipeline inside a pipeline

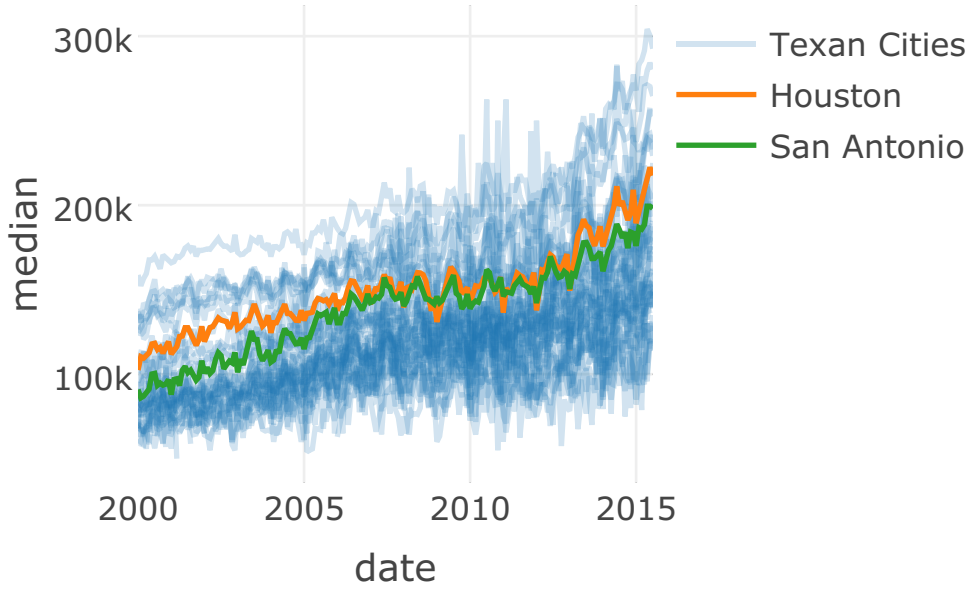


Figure 8.3 Monthly median house price in Houston and San Antonio in comparison to other Texan cities.

It is useful to think of the function supplied to `add_fun()` as a "layer" function – a function that accepts a plot object as input, possibly applies a transformation to the data, and maps that data to visual objects. To make layering functions more modular, flexible, and expressive, the `add_fun()` allows you to pass additional arguments to a layer function. Figure **??** makes use of this pattern, by creating a reusable function for layering both a particular city as well as the first, second, and third quartile of median monthly house sales (by city).

```
# reusable function for highlighting a particular city
layer_city <- function(plot, name) {
  plot %>% filter(city == name) %>% add_lines(name = name)
}


# reusable function for plotting overall median & IQR
layer_iqr <- function(plot) {
  plot %>%
```

```
    group_by(date) %>%
    summarise(
      q1 = quantile(median, 0.25, na.rm = TRUE),
      m = median(median, na.rm = TRUE),
      q3 = quantile(median, 0.75, na.rm = TRUE)
    ) %>%
    add_lines(y = ~m, name = "median", color = I("black")) %>%
    add_ribbons(ymin = ~q1, ymax = ~q3, name = "IQR", color = I("black"))
}


allCities %>%
  add_fun(layer_iqr) %>%
  add_fun(layer_city, "Houston") %>%
  add_fun(layer_city, "San Antonio")
#> Error in x[!is.na(x)]: object of type 'closure' is not subsettable
```

A layering function does not have to be a data-plot-pipeline itself. Its only requirement on a layering function is that the first argument is a plot object and it returns a plot object. This provides an opportunity to say, fit a model to the plot data, extract the model components you desire, and map those components to visuals. Furthermore, since **plotly**'s `add_*()` functions don't require a data.frame, you can supply those components directly to attributes (as long as they are well-defined), as done in Figure **??** via the **forecast** package (Hyndman 2016).

```
library(forecast)
layer_forecast <- function(plot) {
  d <- plotly_data(plot)
  series <- with(d,
```

```
    ts(median, frequency = 12, start = c(2000, 1), end = c(2015, 7))
  )
  fore <- forecast(ets(series), h = 48, level = c(80, 95))
  plot %>%
    add_ribbons(x = time(fore$mean), ymin = fore$lower[, 2],
                ymax = fore$upper[, 2], color = I("gray95"),
                name = "95% confidence", inherit = FALSE) %>%
    add_ribbons(x = time(fore$mean), ymin = fore$lower[, 1],
                ymax = fore$upper[, 1], color = I("gray80"),
                name = "80% confidence", inherit = FALSE) %>%
    add_lines(x = time(fore$mean), y = fore$mean, color = I("blue"),
              name = "prediction")
}

txhousing %>%
  group_by(city) %>%
  plot_ly(x = ~date, y = ~median) %>%
  add_lines(alpha = 0.2, name = "Texan Cities", hoverinfo = "none") %>%
  add_fun(layer_iqr) %>%
  add_fun(layer_forecast)
#> Error in x[!is.na(x)]: object of type 'closure' is not subsettable
```

In summary, the "data-plot-pipeline" is desirable for a number of reasons: (1) makes your code easier to read and understand, (2) encourages you to think of both your data and plots using a single, uniform data structure, which (3) makes it easy to combine and reuse transformations. As it turns out, we can even use these ideas when creating a plotly

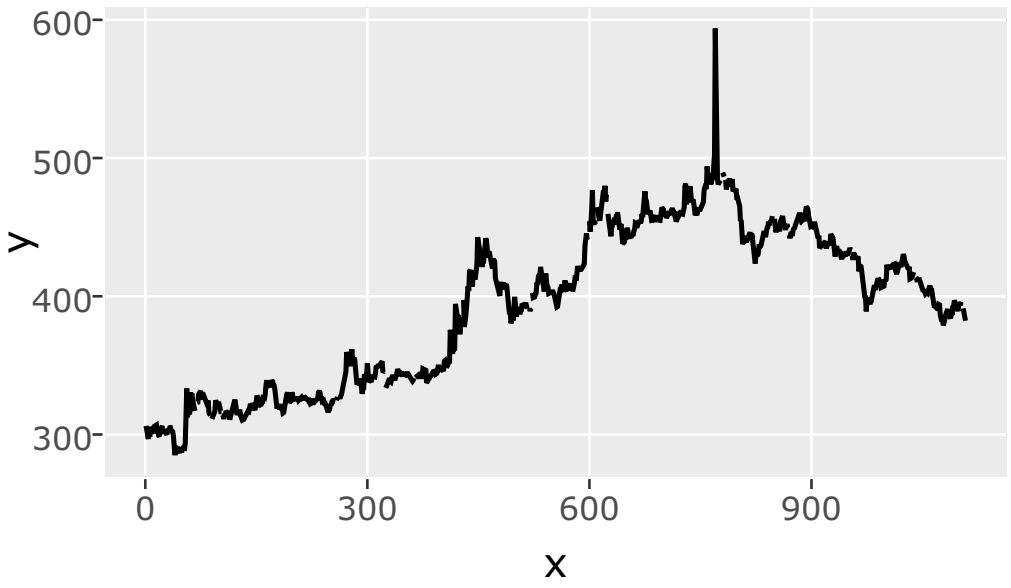


Figure 8.4 Customizing the dragmode of an interactive ggplot2 graph.

object via `ggplotly()`, as discused in the next section Extending `ggplotly()`(extending-ggplotly.

8.1.2 Extending `ggplotly()`

8.1.2.1 Customizing the layout

Since the `ggplotly()` function returns a plotly object, we can manipulate that object in the same way that we would manipulate any other plotly object. A simple and useful application of this is to specify interaction modes, like plotly.js' layout.dragmode for specifying the mode of click+drag events. Figure 8.4 demonstrates how the default for this attribute can be modified via the `layout()` function.

```
p <- ggplot(fortify(gold), aes(x, y)) + geom_line()
gg <- ggplotly(p)
layout(gg, dragmode = "pan")
```

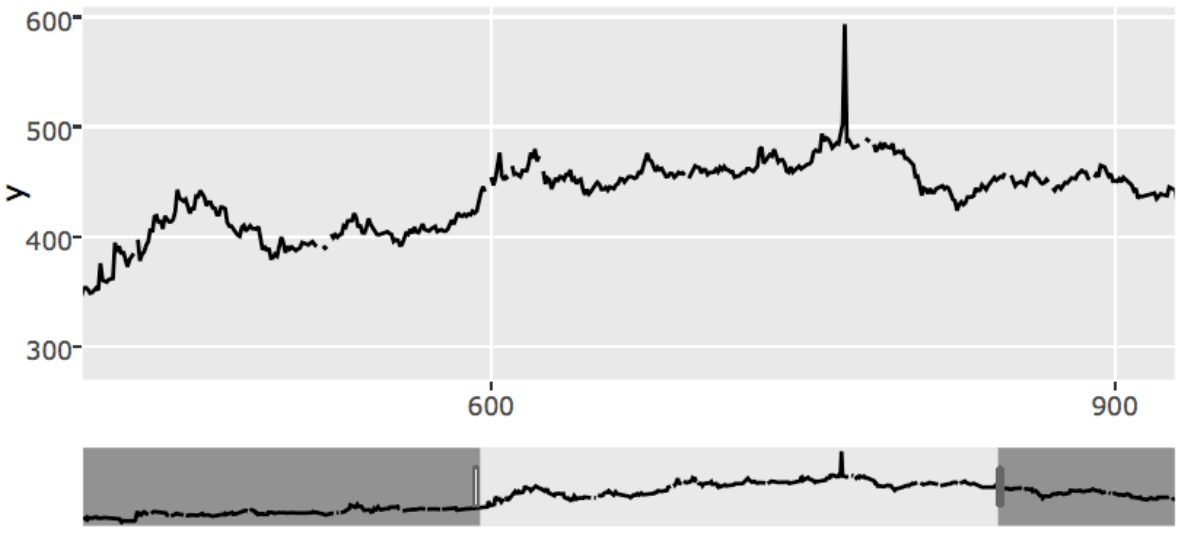


Figure 8.5 Adding a rangeslider to an interactive ggplot2 graph.

Perhaps a more useful application is to add a range slider to the x-axis, which allows you to zoom on the x-axis, without losing the global context. This is quite useful for quickly altering the limits of your plot to achieve an optimal aspect ratio for your data (William S. Cleveland 1988), without losing the global perspective. Figure 8.5 uses the `rangeslider()` function to add a rangeslider to the plot.

```
rangeslider(gg)
```

Since a single plotly object can only have one layout, modifying the layout of `ggplotly()` is fairly easy, but it's trickier to add and modify layers.

8.1.2.2 Adding layers

Since `ggplotly()` returns a plotly object, and plotly objects have data associated with them, we can effectively associate data from a ggplot object with a plotly object, before or after summary statistics have been applied. Since each ggplot layer owns a data frame, it is useful to have some way to specify the particular layer of data of interest, which is the point of the `layerData` argument in `ggplotly()`. Also, when a particular layer

applies a summary statistic (e.g., `geom_bin()`), or applies a model (e.g., `geom_smooth()`) to the data, it might be useful to access the output of that transformation, which is the point of the `originalData` argument in `ggplotly()`.

```
p <- ggplot(mtcars, aes(x = wt, y = mpg)) +
   geom_point() + geom_smooth()
p %>%
  ggplotly(layerData = 2, originalData = FALSE) %>%
  plotly_data()
#> # A tibble: 80 × 13
#>       x     y  ymin  ymax    se PANEL group  colour   fill  size
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <int> <int>   <chr>  <chr> <dbl>
#> 1  1.51  32.1  28.1  36.0  1.92     1    -1 #3366FF grey60     1
#> 2  1.56  31.7  28.2  35.2  1.72     1    -1 #3366FF grey60     1
#> 3  1.61  31.3  28.1  34.5  1.54     1    -1 #3366FF grey60     1
#> 4  1.66  30.9  28.0  33.7  1.39     1    -1 #3366FF grey60     1
#> 5  1.71  30.5  27.9  33.0  1.26     1    -1 #3366FF grey60     1
#> 6  1.76  30.0  27.7  32.4  1.16     1    -1 #3366FF grey60     1
#> # ... with 74 more rows, and 3 more variables: linetype <dbl>,
#> #   weight <dbl>, alpha <dbl>
```

This is the dataset ggplot2 uses to actually draw the fitted values (as a line) and standard error bounds (as a ribbon). Figure 8.6 uses this data to add additional information about the model fit; in particular, it adds a vertical lines and annotations at the x-values that are associated with the highest and lowest amount uncertainty in y.

```
p %>%
  ggplotly(layerData = 2, originalData = F) %>%
  add_fun(function(p) {
```

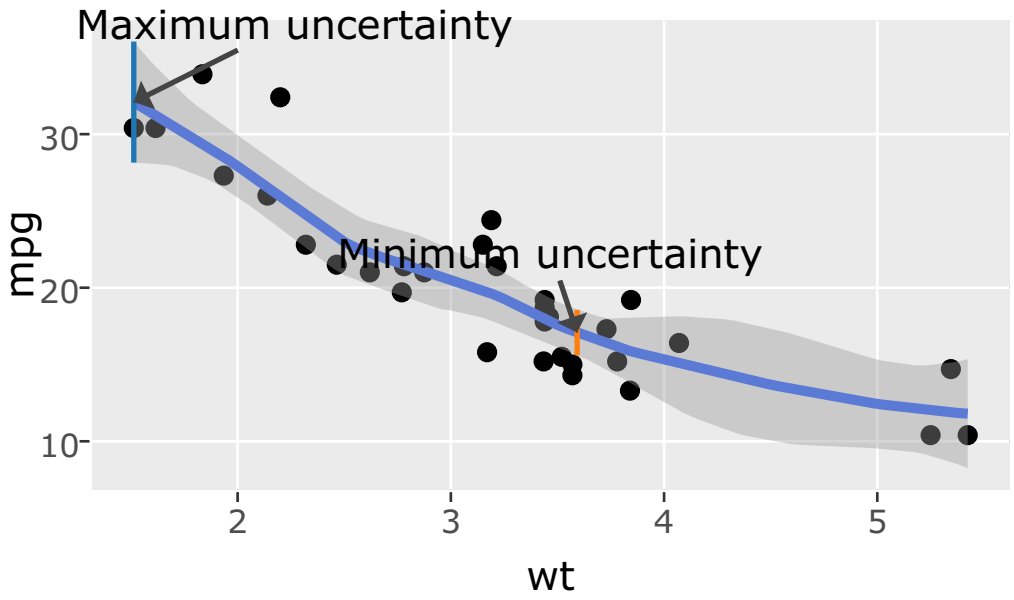


Figure 8.6 Leveraging data associated with a `geom_smooth()` layer to display additional information about the model fit.

```
    p %>% slice(which.max(se)) %>%
      add_segments(x = ~x, xend = ~x, y = ~ymin, yend = ~ymax) %>%
      add_annotations("Maximum uncertainty", ax = 60)
  }) %>%
  add_fun(function(p) {
    p %>% slice(which.min(se)) %>%
      add_segments(x = ~x, xend = ~x, y = ~ymin, yend = ~ymax) %>%
      add_annotations("Minimum uncertainty")
  })
```

Although it is not used in this example, it worth noting that when adding plotly layers to the output of `ggplotly()`, it will inherit mappings from the ggplot aesthetic mapping, which may or may not be desired, but the `inherit` argument in any of the `add_*()` functions may be set to `FALSE` to avoid this behavoir.

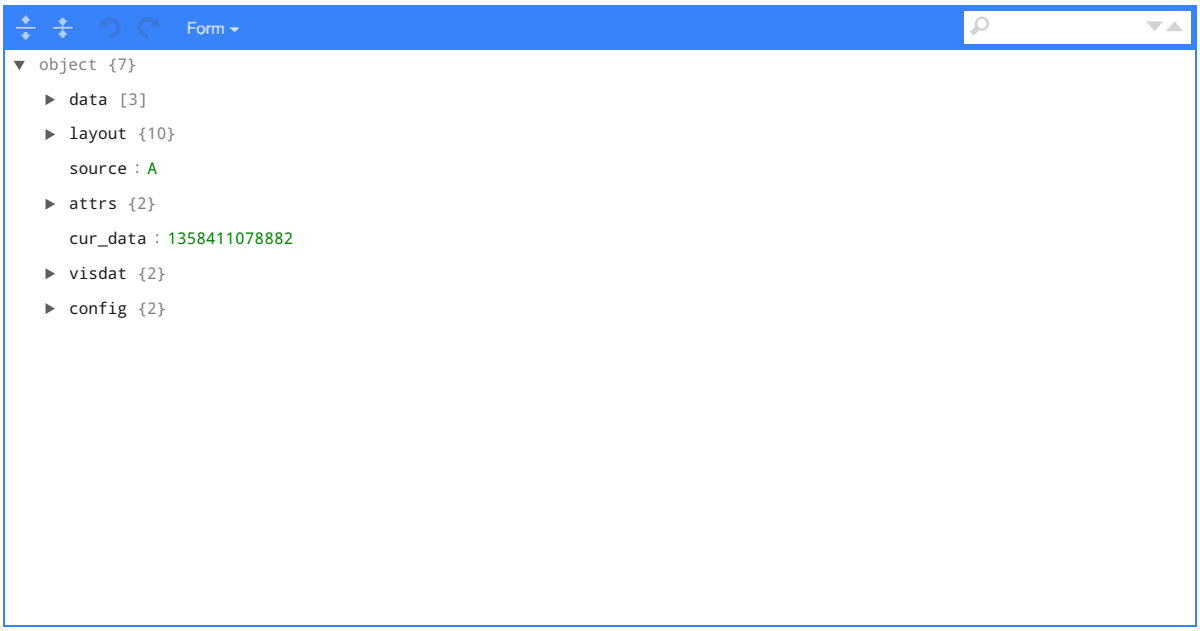


Figure 8.7 Using listviewer to inspect a plotly object.

8.1.2.3 Modifying layers

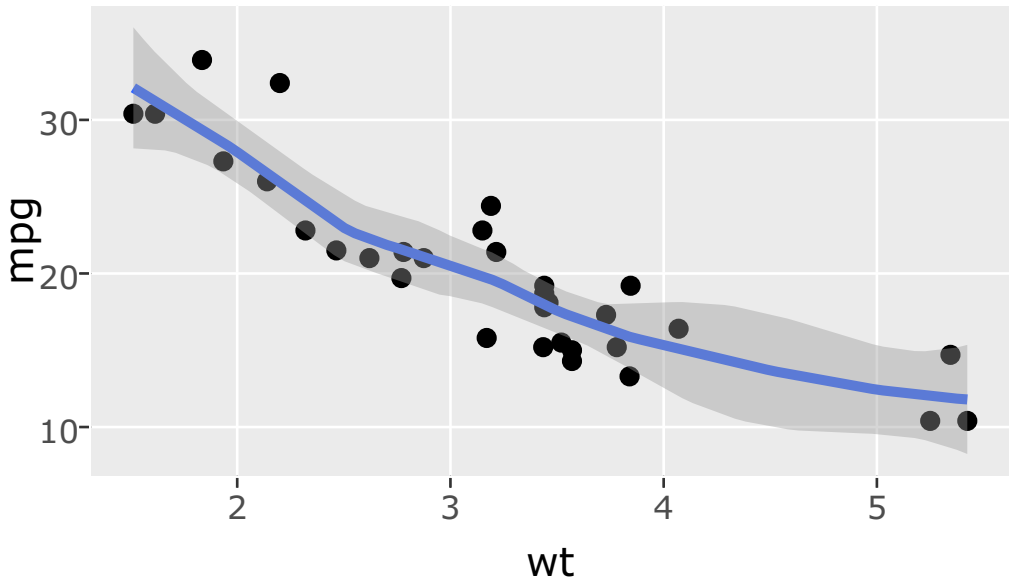
As mentioned previously, `ggplotly()` translates each ggplot2 layer into one or more plotly.js traces. In this translation, it is forced to make a number of assumptions about trace attribute values that may or may not be appropriate for the use case. The `style()` function is useful in this scenario, as it provides a way to modify trace attribute values in a plotly object. Before using it, you may want to inspect the actual traces in a given plotly object using the `plotly_json()` function. This function uses the **listviewer** package to display a convenient interactive view of the JSON object sent to plotly.js (de Jong and Russell 2016). By clicking on the arrow next to the data element, you can see the traces (data) behind the plot. In this case, we have three traces: one for the `geom_point()` layer and two for the `geom_smooth()` layer.

```
plotly_json(p)
```

Say, for example, we'd like to display information when hovering over points, but not when hovering over the fitted values or error bounds. The ggplot2 API has no semantics

for making this distinction, but this is easily done in plotly.js by setting the hoverinfo attribute to `"none"`. Since the fitted values or error bounds are contained in the second and third traces, we can hide the information on just these traces using the `traces` attribute in the `style()` function:

```
style(p, hoverinfo = "none", traces = 2:3)
```



8.2 The plotly cookbook

This chapter demonstrates the rendering capabilities of `plot_ly()` through a series of examples. The `plot_ly()` function provides a direct interface to plotly.js, so anything in the figure reference can be specified via `plot_ly()`, but this chapter will focus more on the special semantics unique to the R package that can't be found on the figure reference. Along the way, we will touch on some best practices in visualization.

8.2.1 Scatter traces

A plotly visualization is composed of one (or more) trace(s), and every trace has a `type`. The default trace type, "scatter", can be used to draw a large amount of geometries, and actually powers many of the `add_*()` functions such as `add_markers()`, `add_lines()`, `add_paths()`, `add_segments()`, `add_ribbons()`, and `add_polygons()`. Among other things, these functions make assumptions about the mode of the scatter trace, but any valid attribute(s) listed under the scatter section of the figure reference may be used to override defaults.

The `plot_ly()` function has a number of arguments that make it easier to scale data values to visual aesthetics (e.g., `color`/`colors`, `symbol`/`symbols`, `linetype`/`linetypes`, `size`/`sizes`). These arguments are unique to the R package and dynamically determine what objects in the figure reference to populate (e.g., `marker.color` vs `line.color`). Generally speaking, the singular form of the argument defines the domain of the scale (data) and the plural form defines the range of the scale (visuals). To make it easier to alter default visual aesthetics (e.g., change all points from blue to black), "AsIs" values (values wrapped with the `I()` function) are interpreted as values that already live in visual space, and thus do not need to be scaled. The next section on scatterplots explores detailed use of the `color`/`colors`, `symbol`/`symbols`, & `size`/`sizes` arguments. The section on lineplots explores detailed use of the `linetype`/`linetypes`.

8.2.1.1 Scatterplots

The scatterplot is useful for visualizing the correlation between two quantitative variables. If you supply a numeric vector for x and y in `plot_ly()`, it defaults to a scatterplot, but you can also be explicit about adding a layer of markers/points via the `add_markers()` function. A common problem with scatterplots is overplotting, meaning that there are

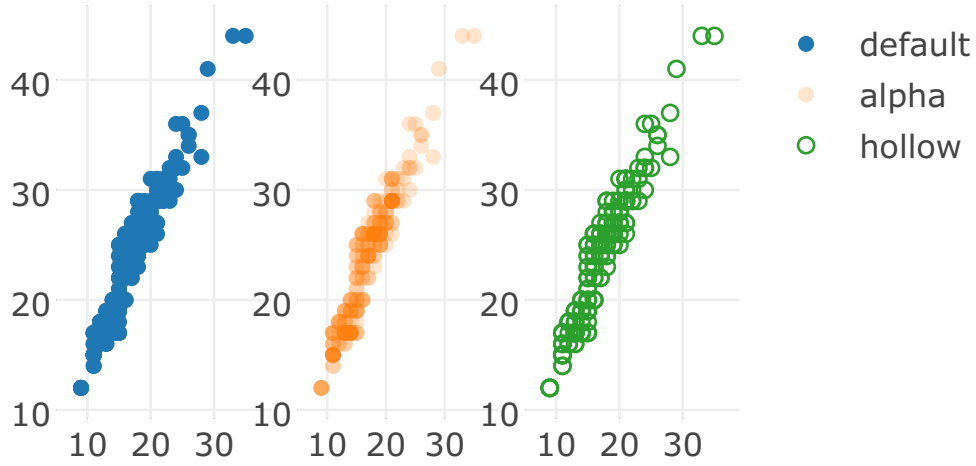


Figure 8.8 Three versions of a basic scatterplot

multiple observations occupying the same (or similar) x/y locations. There are a few ways to combat overplotting including: alpha transparency, hollow symbols, and 2D density estimation. Figure 8.8 shows how alpha transparency and hollow symbols can provide an improvement over the default.

```
subplot(
  plot_ly(mpg, x = ~cty, y = ~hwy, name = "default"),
  plot_ly(mpg, x = ~cty, y = ~hwy) %>% add_markers(alpha = 0.2, name = "alpha"),
  plot_ly(mpg, x = ~cty, y = ~hwy) %>% add_markers(symbol = I(1), name = "hollow")
)
```

In Figure 8.8, hollow circles are specified via `symbol = I(1)`. By default, the `symbol` argument (as well as the `color`/`size`/`linetype` arguments) assumes value(s) are "data", which need to be mapped to a visual palette (provided by `symbols`). Wrapping values with the `I()` function notifies `plot_ly()` that these values should be taken "AsIs". If you compare the result of `plot(1:25, 1:25, pch = 1:25)` to Figure 8.9, you'll see that `plot_ly()` can translate R's plotting characters (pch), but you can also use plotly.js' symbol syntax, if you desire.

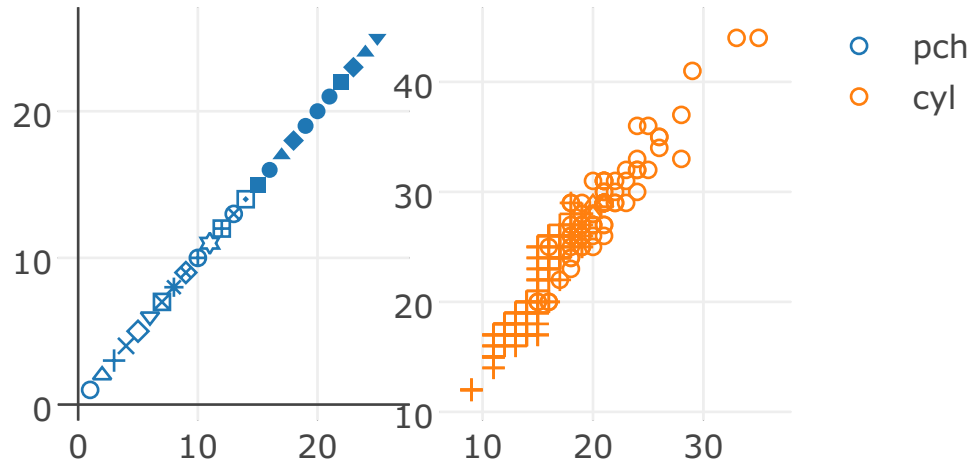


Figure 8.9 Specifying symbol in a scatterplot

```
subplot(
  plot_ly(x = 1:25, y = 1:25, symbol = I(1:25), name = "pch"),
  plot_ly(mpg, x = ~cty, y = ~hwy, symbol = ~cyl, symbols = 1:3, name = "cyl")
)
```

When mapping a numeric variable to `symbol`, it creates only one trace, so no legend is generated. If you do want one trace per symbol, make sure the variable you're mapping is a factor, as Figure 8.10 demonstrates. When plotting multiple traces, the default plotly.js color scale will apply, but you can set the color of every trace generated from this layer with `color = I("black")`, or similar.

```
p <- plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.3)
subplot(
  add_markers(p, symbol = ~cyl, name = "A single trace"),
  add_markers(p, symbol = ~factor(cyl), color = I("black"))
)
```

The `color` argument adheres to similar rules as `symbol`:

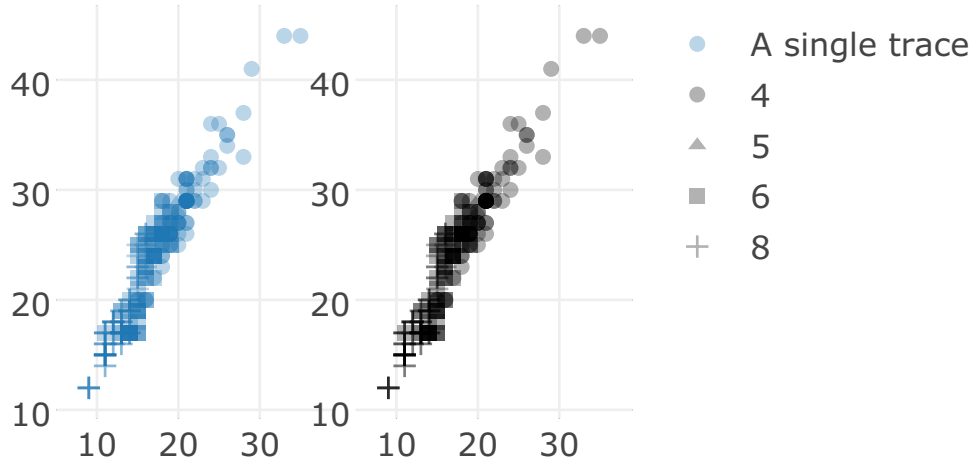


Figure 8.10 Mapping symbol to a factor

- If numeric, `color` produces one trace, but colorbar is also generated to aide the decoding of colors back to data values. The `colorbar()` function can be used to customize the appearance of this automatically generated guide. The default colorscale is viridis, a perceptually-uniform colorscale (even when converted to black-and-white), and perceivable even to those with common forms of color blindness (Data Science 2016).

- If discrete, `color` produces one trace per value, meaning a legend is generated. If an ordered factor, the default colorscale is viridis (Garnier 2016); otherwise, it is the "Set2" palette from the **RColorBrewer** package (Neuwirth 2014)

```
p <- plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.5)
subplot(
  add_markers(p, color = ~cyl, showlegend = FALSE) %>%
    colorbar(title = "Viridis", len = 1/2, y = 1),
  add_markers(p, color = ~factor(cyl))
) %>% layout(showlegend = TRUE)
```

There are a number of ways to alter the default colorscale via the `colors` argument. This argument excepts: (1) a color brewer palette name (see the row names of

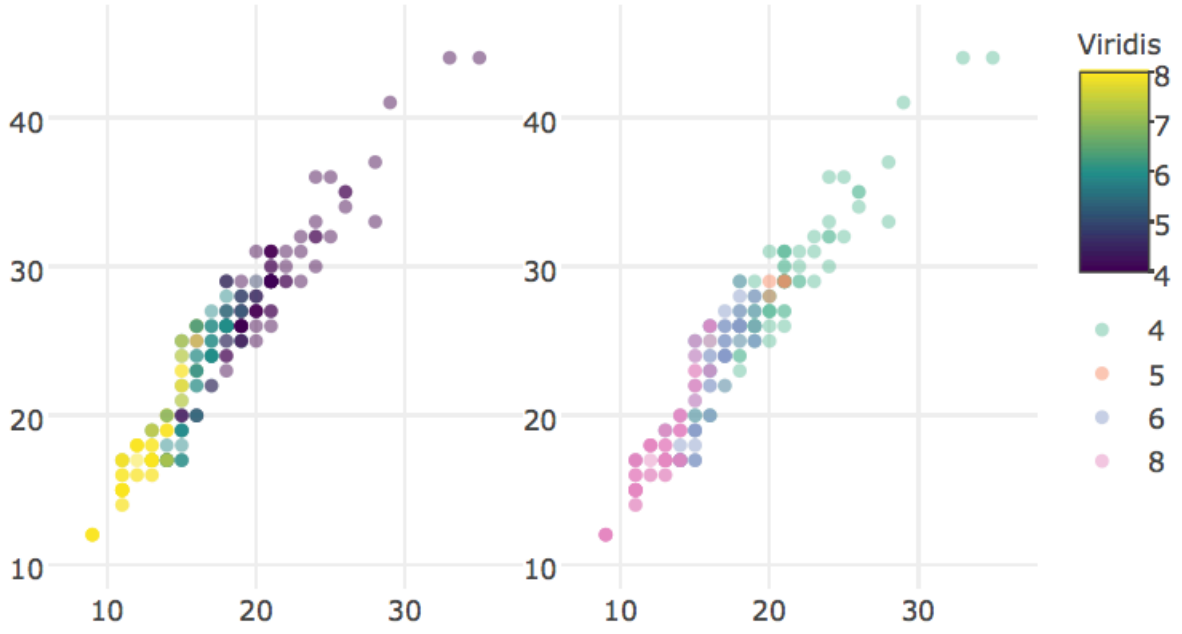


Figure 8.11 Variations on a numeric color mapping.

`RColorBrewer::brewer.pal.info` for valid names), (2) a vector of colors to interpolate, or (3) a color interpolation function like `colorRamp()` or `scales::colour_ramp()`. Although this grants a lot of flexibility, one should be concious of using a sequential colorscale for numeric variables (& ordered factors) as shown in 8.12, and a qualitative colorscale for discrete variables as shown in 8.13. (TODO: touch on lurking variables?)

```
subplot(
  add_markers(p, color = ~cyl, colors = c("#132B43", "#56B1F7")) %>%
    colorbar(title = "ggplot2 default", len = 1/3, y = 1),
  add_markers(p, color = ~cyl, colors = viridisLite::inferno(10)) %>%
    colorbar(title = "Inferno", len = 1/3, y = 2/3),
  add_markers(p, color = ~cyl, colors = colorRamp(c("red", "white", "blue"))) %>%
    colorbar(title = "colorRamp", len = 1/3, y = 1/3)
)
```

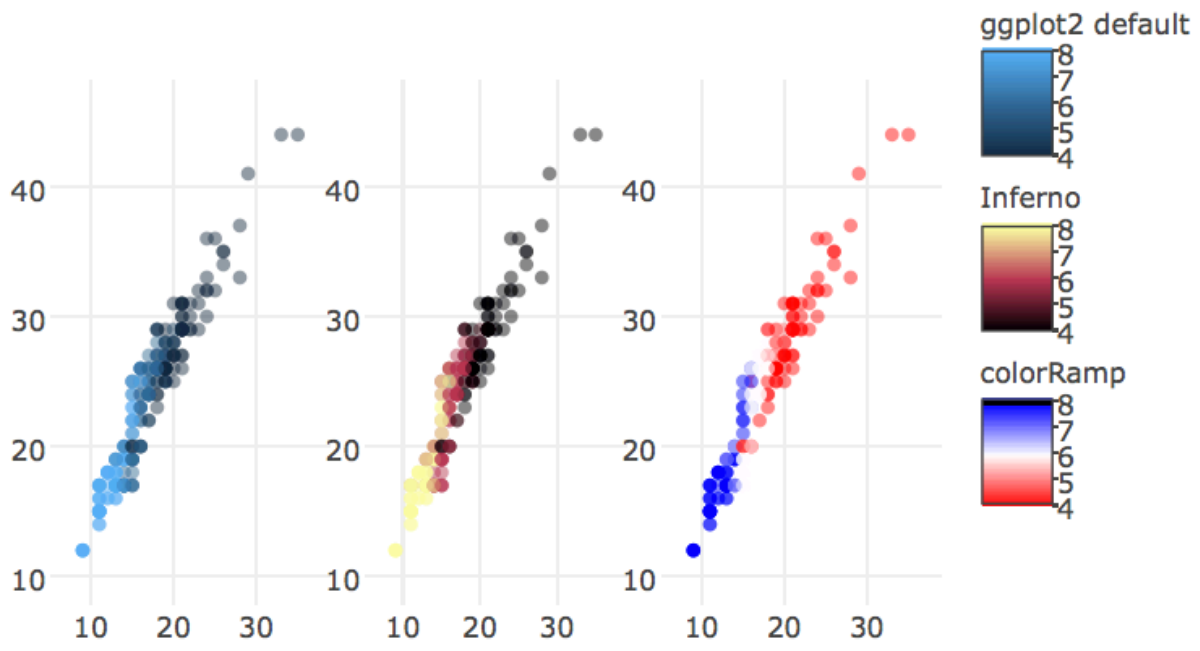


Figure 8.12 Three variations on a numeric color mapping

```
subplot(
  add_markers(p, color = ~factor(cyl), colors = "Pastel1"),
  add_markers(p, color = ~factor(cyl), colors = colorRamp(c("red", "blue"))),
  add_markers(p, color = ~factor(cyl),
              colors = c(`4` = "red", `5` = "black", `6` = "blue", `8` = "green"))
) %>% layout(showlegend = FALSE)
```

For scatterplots, the `size` argument controls the area of markers (unless otherwise specified via sizemode), and *must* be a numeric variable. The `sizes` argument controls the minimum and maximum size of circles, in pixels:

```
subplot(
  add_markers(p, size = ~cyl, name = "default"),
  add_markers(p, size = ~cyl, sizes = c(1, 500), name = "custom")
)
```

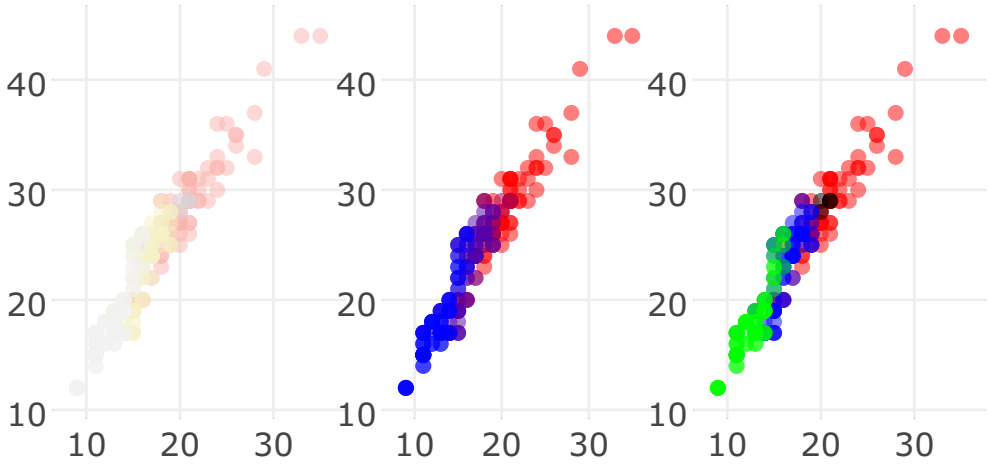
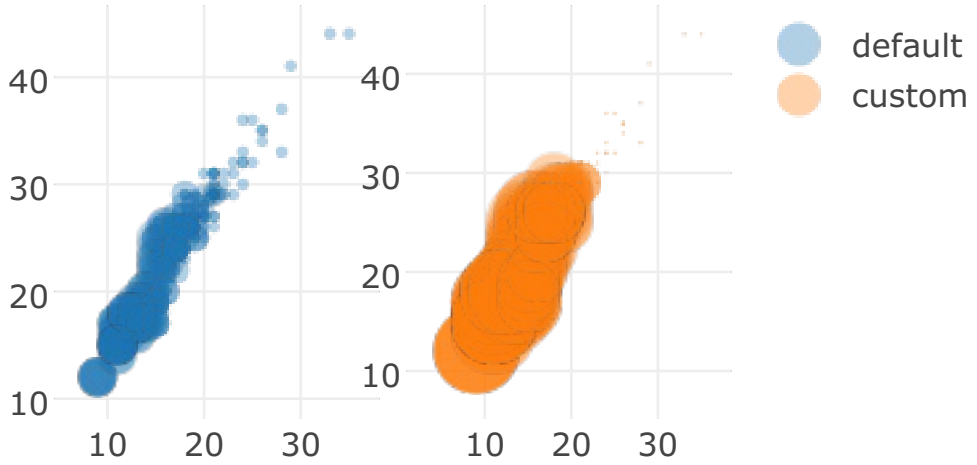


Figure 8.13 Three variations on a discrete color mapping



8.2.1.1.1 3D scatterplots

To make a 3D scatterplot, just add a `z` attribute:

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_markers(color = ~cyl)
```

8.2.1.1.2 Scatterplot matrices

Scatterplot matrices *can* be made via `plot_ly()` and `subplot()`, but `ggplotly()` has a special method for translating ggmatrix objects from the **GGally** package to plotly

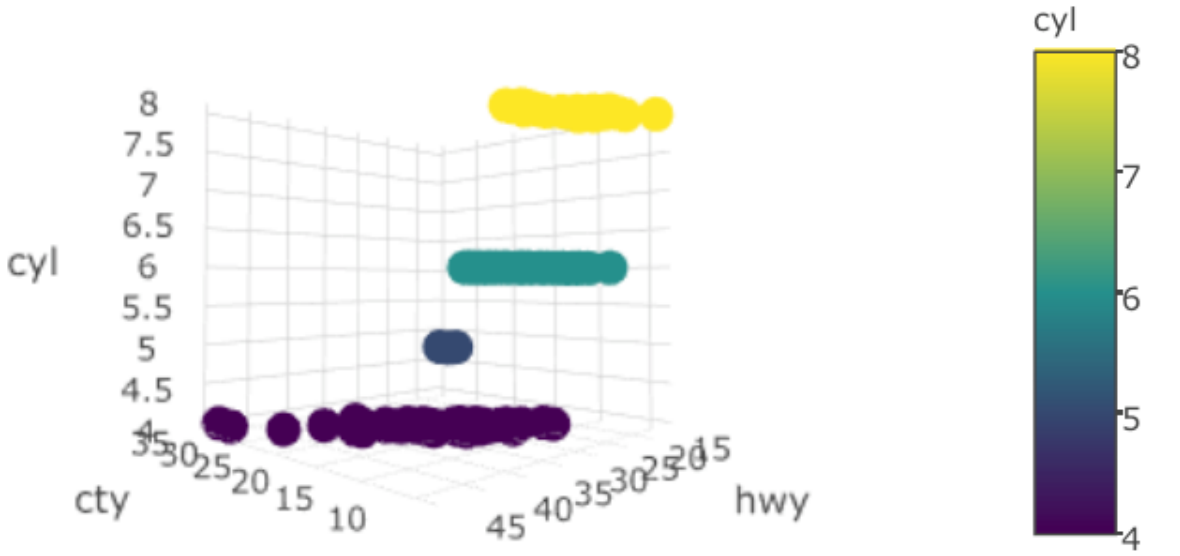


Figure 8.14 A 3D scatterplot

objects (Schloerke et al. 2016). These objects are essentially a matrix of ggplot objects and are the underlying data structure which powers higher level functions in **GGally**, such as `ggpairs()` – a function for creating a generalized pairs plot (Emerson et al. 2013). The generalized pairs plot can be motivated as a generalization of the scatterplot matrix with support for categorical variables and different visual representations of the data powered by the grammar of graphics. Figure 8.15 shows an interactive version of the generalized pairs plot made via `ggpairs()` and `ggplotly()`. In Linking views without shiny, we explore how this framework can be extended to enable linked brushing in the generalized pairs plot.

```
pm <- GGally::ggpairs(iris)

ggplotly(pm)
```

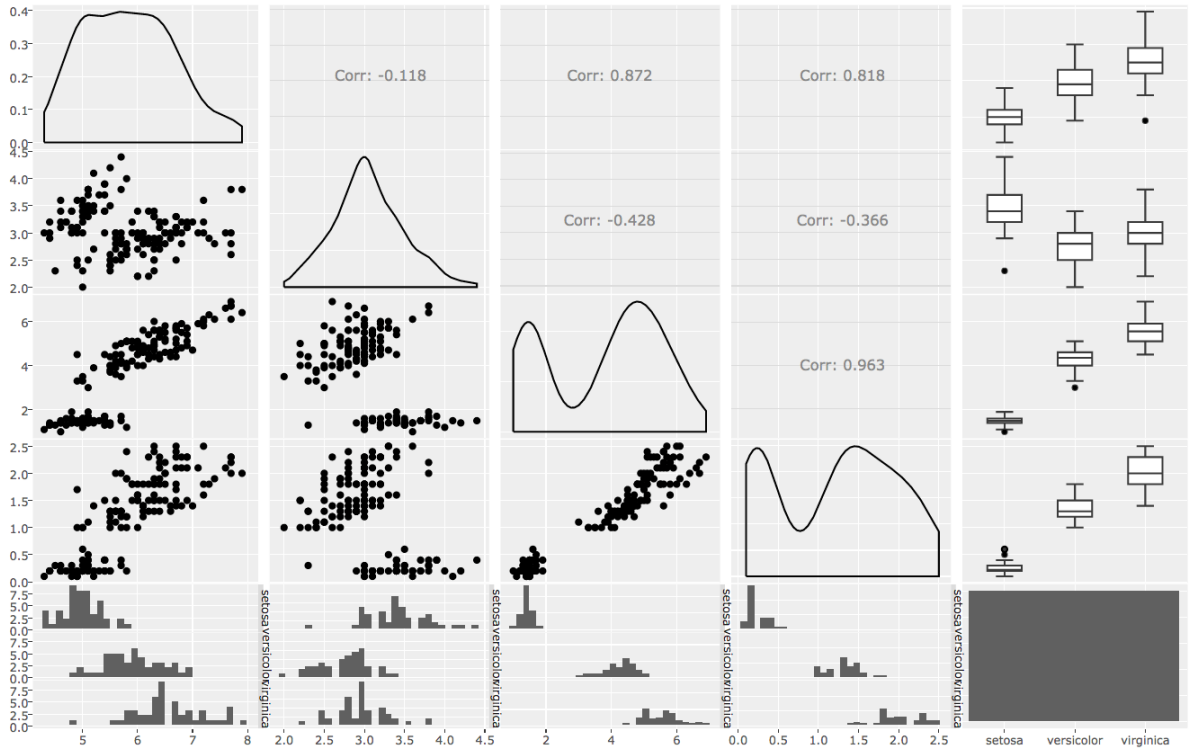


Figure 8.15 An interactive version of the generalized pairs plot made via the `ggpairs()` function from the **GGally** package

8.2.1.2 Dotplots & error bars

A dotplot is similar to a scatterplot, except instead of two numeric axes, one is categorical. The usual goal of a dotplot is to compare value(s) on a numerical scale over numerous categories. In this context, dotplots are preferrable to pie charts since comparing position along a common scale is much easier than comparing angle or area (Cleveland and McGill 1984); (Bostock 2010). Furthermore, dotplots can be preferrable to bar charts, especially when comparing values within a narrow range far away from 0 (Few 2006). Also, when presenting point estimates, and uncertainty associated with those estimates, bar charts tend to exaggerate the difference in point estimates, and lose focus on uncertainty (Messing 2012).

A popular application for dotplots (with error bars) is the so-called "coefficient plot" for visualizing the point estimates of coefficients and their standard error. The `coefplot()` function in the **coefplot** package (Lander 2016) and the `ggcoef()` function in the **GGally** both produce coefficient plots for many types of model objects in R using **ggplot2**, which we can translate to plotly via `ggplotly()`. Since these packages use points and segments to draw the coefficient plots, the hover information is not the best, and it'd be better to use error objects. Figure 8.16 uses the `tidy()` function from the **broom** package (Robinson 2016a) to obtain a data frame with one row per model coefficient, and produce a coefficient plot with error bars along the x-axis.

```
m <- lm(Sepal.Length ~ Sepal.Width * Petal.Length * Petal.Width, data = iris)
# arrange by estimate, then make term a factor to order categories in the plot
d <- broom::tidy(m) %>%
  arrange(desc(estimate)) %>%
  mutate(term = factor(term, levels = term))
plot_ly(d, x = ~estimate, y = ~term) %>%
  add_markers(error_x = ~list(value = std.error)) %>%
```

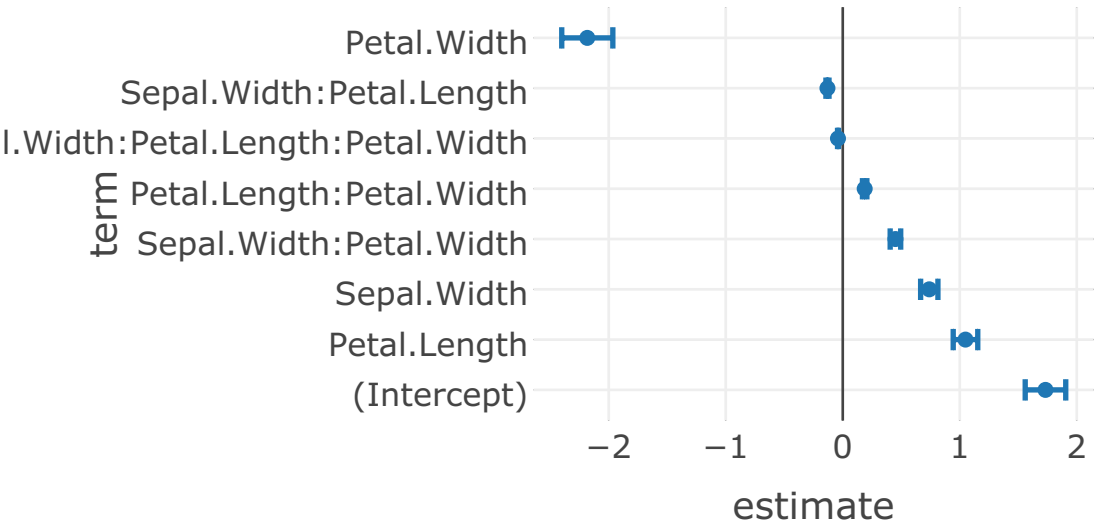


Figure 8.16 A coefficient plot

```
layout(margin = list(l = 200))
```

8.2.1.3 Line plots

This section surveys useful applications of `add_lines()` and `add_paths()`. The only difference between these functions is that `add_lines()` connects x/y pairs from left to right, instead of the order in which the data appears. Both functions understand the `color`, `linetype`, and `alpha` attributes[4], as well as groupings defined by `group_by()`.

Figure 8.2 uses `group_by()` to plot one line per city in the `txhousing` dataset using a *single* trace. Since there can only be one tooltip per trace, hovering over that plot does not reveal useful information. Although plotting many traces can be computationally expensive, it is necessary in order to display better information on hover. Since the `color` argument produces one trace per value (if the variable (`city`) is discrete), hovering on

[4]plotly.js currently does not support data arrays for `scatter.line.width` or `scatter.line.color`, meaning a single line trace can only have one width/color in 2D line plot, and consequently numeric `color`/`size` mappings won't work

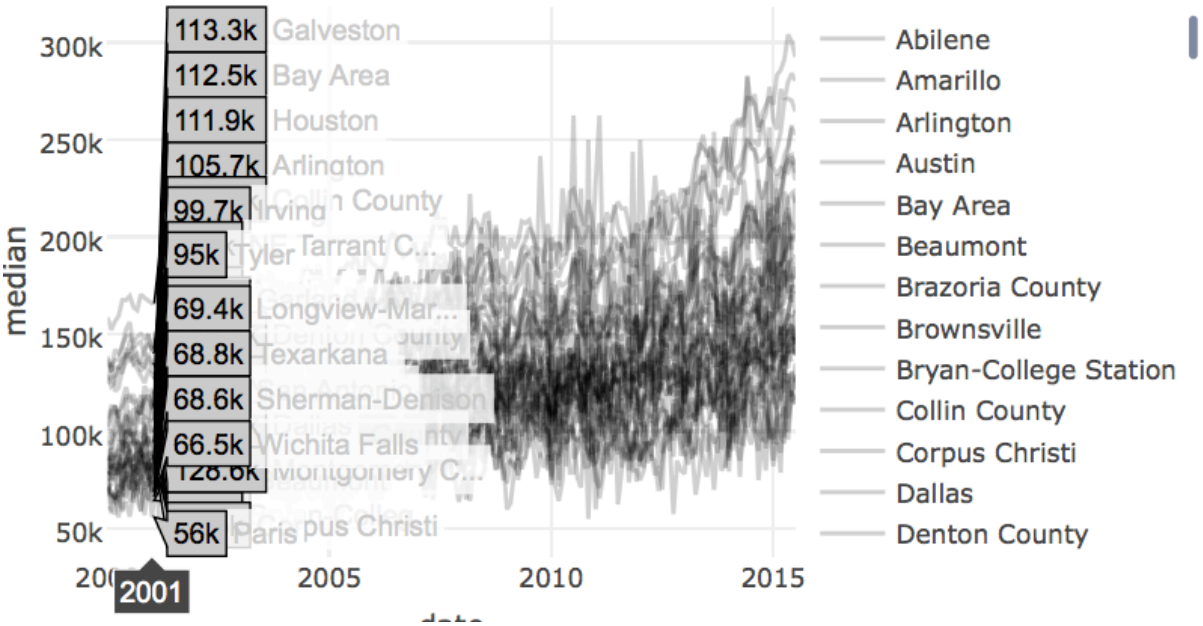


Figure 8.17 Median house sales with one trace per city.

Figure 8.17 reveals the top ~10 cities at a given x value. Since 46 colors is too many to perceive in a single plot, Figure 8.17 also restricts the set of possible `colors` to black.

```
plot_ly(txhousing, x = ~date, y = ~median) %>%
  add_lines(color = ~city, colors = "black", alpha = 0.2)
```

Generally speaking, it's hard to perceive more than 8 different colors/linetypes/symbols in a given plot, so sometimes we have to filter data to use these effectively. Here we use the **dplyr** package to find the top 5 cities in terms of average monthly sales (`top5`), then effectively filter the original data to contain just these cities via `semi_join()`. Once we have the data is filtered, mapping city to `color` or `linetype` is trivial. The color palette can be altered via the `colors` argument, and follows the same rules as scatterplots. The linetype palette can be altered via the `linetypes` argument, and accepts R's `lty` values or plotly.js dash values.

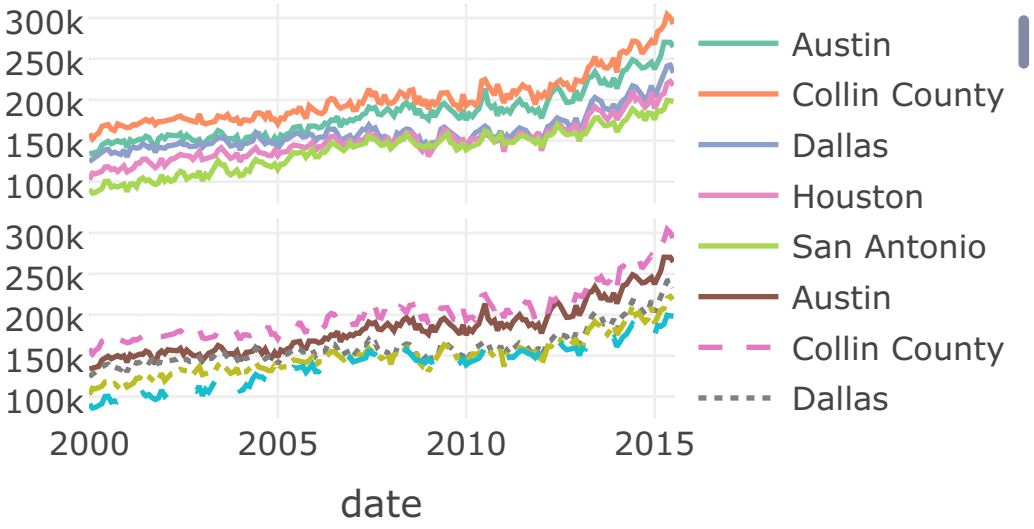
```
library(dplyr)

top5 <- txhousing %>%
```

```
  group_by(city) %>%
  summarise(m = mean(sales, na.rm = TRUE)) %>%
  arrange(desc(m)) %>%
  top_n(5)

p <- semi_join(txhousing, top5, by = "city") %>%
  plot_ly(x = ~date, y = ~median)

subplot(
  add_lines(p, color = ~city),
  add_lines(p, linetype = ~city),
  shareX = TRUE, nrows = 2
)
```



8.2.1.3.1 Density plots

In Bars & histograms, we leveraged a number of algorithms in R for computing the "optimal" number of bins for a histogram, via `hist()`, and routing those results to

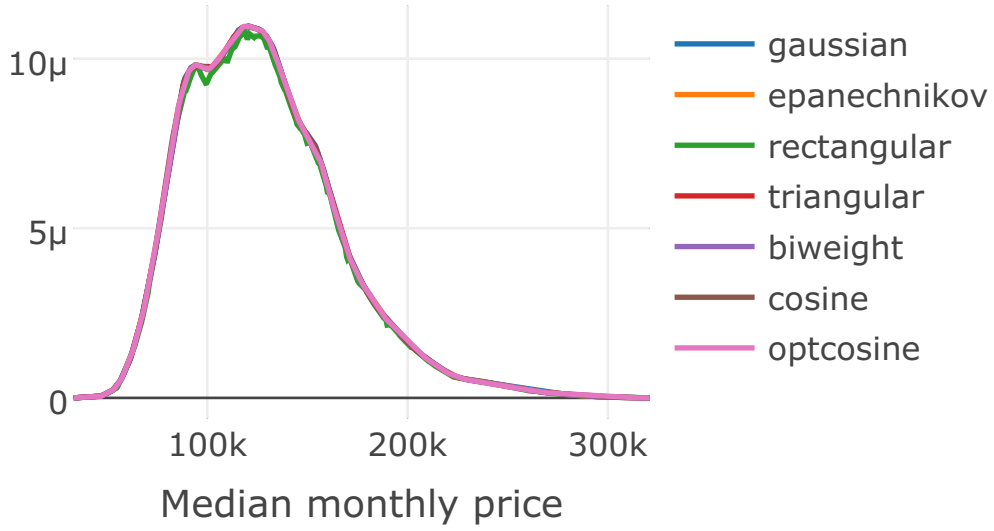


Figure 8.18 Various kernel density estimates.

`add_bars()`. We can leverage the `density()` function for computing kernel density estimates in a similar way, and routing the results to `add_lines()`, as is done in 8.18.

```
kerns <- c("gaussian", "epanechnikov", "rectangular",
           "triangular", "biweight", "cosine", "optcosine")
p <- plot_ly()
for (k in kerns) {
  d <- density(txhousing$median, kernel = k, na.rm = TRUE)
  p <- add_lines(p, x = d$x, y = d$y, name = k)
}
layout(p, xaxis = list(title = "Median monthly price"))
```

8.2.1.3.2 Parallel Coordinates

One very useful, but often overlooked, visualization technique is the parallel coordinates plot. Parallel coordinates provide a way to compare values along a common (or non-aligned) positional scale(s) – the most basic of all perceptual tasks – in more than 3 dimensions (Cleveland and McGill 1984). Usually each line represents every measurement

for a given row (or observation) in a data set. When measurements are on very different scales, some care must be taken, and variables must transformed to be put on a common scale. As Figure 8.19 shows, even when variables are measured on a similar scale, it can still be a informative to transform variables in different ways.

```
iris$obs <- seq_len(nrow(iris))
iris_pcp <- function(transform = identity) {
  iris[] <- purrr::map_if(iris, is.numeric, transform)
  tidyr::gather(iris, variable, value, -Species, -obs) %>%
    group_by(obs) %>%
    plot_ly(x = ~variable, y = ~value, color = ~Species) %>%
    add_lines(alpha = 0.3)
}
subplot(
  iris_pcp(),
  iris_pcp(scale),
  iris_pcp(scales::rescale)
) %>% hide_legend()
```

It is also worth noting that the **GGally** offers a `ggparcoord()` function which creates parallel coordinate plots via **ggplot2**, which we can convert to plotly via `ggplotly()`. In linked highlighting, parallel coordinates are linked to lower dimensional (but sometimes higher resolution) graphics of related data to guide multi-variate data exploration.

8.2.1.3.3 3D paths

To make a path in 3D, use `add_paths()` in the same way you would for a 2D path, but add a third variable `z`, as Figure 8.20 does.

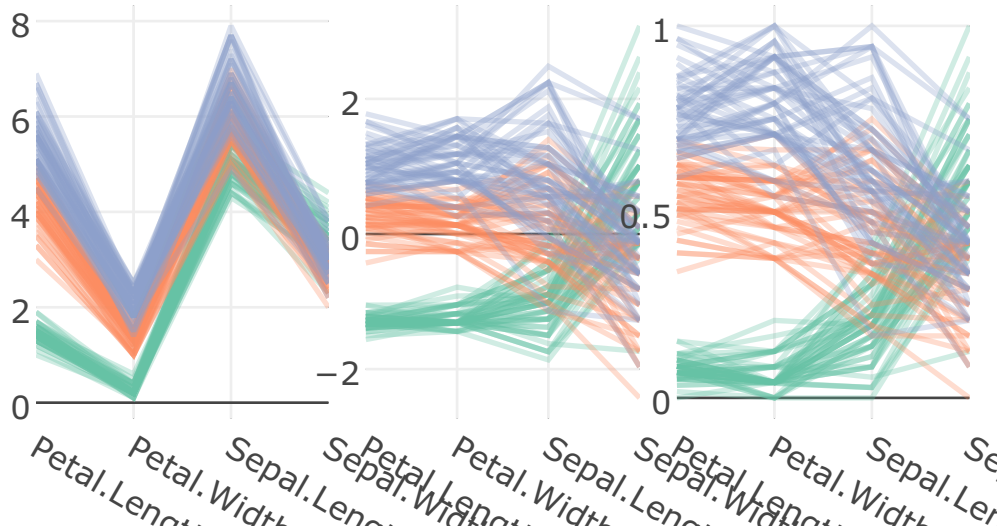


Figure 8.19 Parallel coordinates plots of the Iris dataset. On the left is the raw measurements. In the middle, each variable is scaled to have mean of 0 and standard deviation of 1. On the right, each variable is scaled to have a minimum of 0 and a maximum of 1.

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_paths(color = ~displ)
```

Figure 8.21 uses `add_lines()` instead of `add_paths()` to ensure the points are connected by the x axis instead of the row ordering.

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_lines(color = ~displ)
```

8.2.1.4 Segments

The `add_segments()` function essentially provides a way to connect two points ((`x`, `y`) to (`xend`, `yend`)) with a line. Segments form the building blocks for many useful chart types, including candlestick charts, a popular way to visualize stock prices. Figure 8.22 uses the **quantmod** package (Ryan 2016) to obtain stock price data for Microsoft and

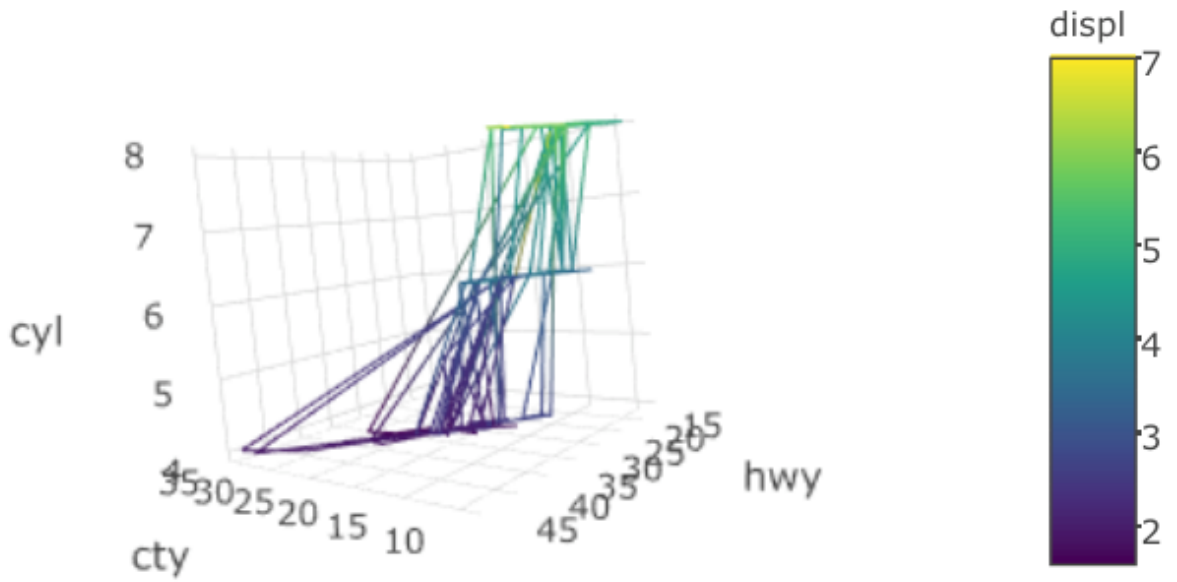


Figure 8.20 A path in 3D

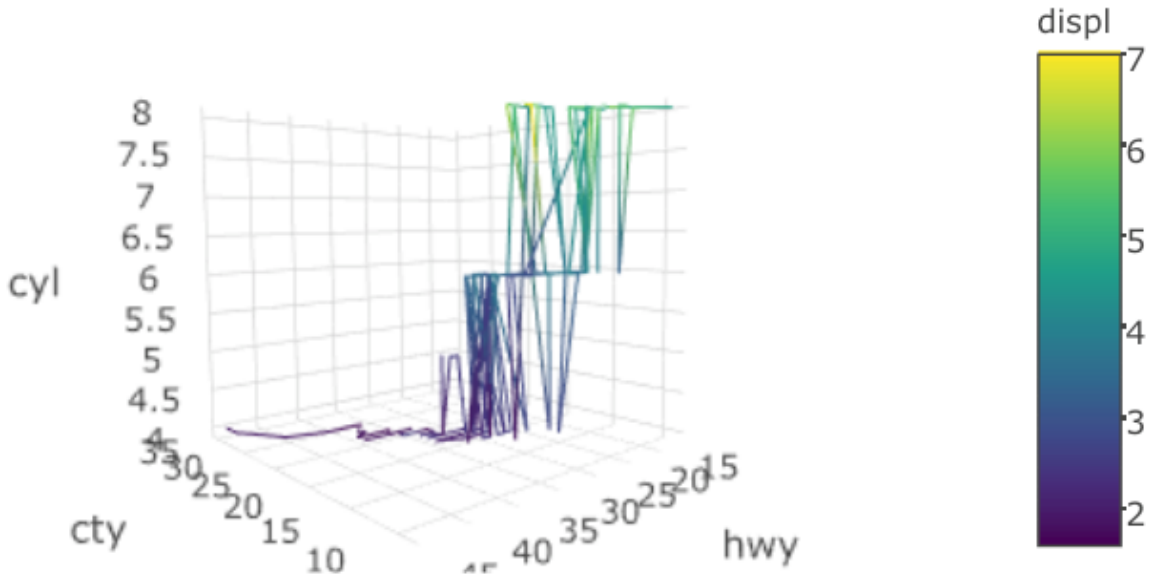


Figure 8.21 A 3D line plot

Figure 8.22 A candelstick chart

plots two segments for each day: one to encode the opening/closing values, and one to encode the daily high/low.

```
library(quantmod)
msft <- getSymbols("MSFT", auto.assign = F)
dat <- as.data.frame(msft)
dat$date <- index(msft)
dat <- subset(dat, date >= "2016-01-01")

names(dat) <- sub("^MSFT\\.", "", names(dat))

plot_ly(dat, x = ~date, xend = ~date, color = ~Close > Open,
        colors = c("red", "forestgreen"), hoverinfo = "none") %>%
  add_segments(y = ~Low, yend = ~High, size = I(1)) %>%
  add_segments(y = ~Open, yend = ~Close, size = I(3)) %>%
  layout(showlegend = FALSE, yaxis = list(title = "Price")) %>%
  rangeslider()
```

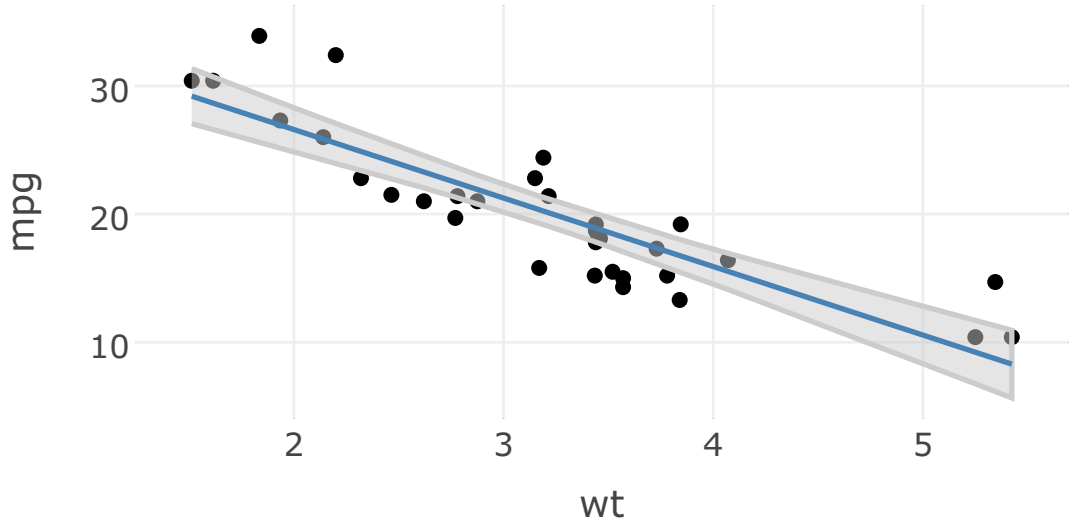


Figure 8.23 Plotting fitted values and uncertainty bounds of a linear model via the **broom** package.

8.2.1.5 Ribbons

Ribbons are useful for showing uncertainy bounds as a function of x. The `add_ribbons()` function creates ribbons and requires the arguments: `x`, `ymin`, and `ymax`. The `augment()` function from the **broom** package appends observational-level model components (e.g., fitted values stored as a new column `.fitted`) which is useful for extracting those components in a convenient form for visualization. Figure 8.23 shows the fitted values and uncertainty bounds from a linear model object.

```
m <- lm(mpg ~ wt, data = mtcars)

broom::augment(m) %>%
  plot_ly(x = ~wt, showlegend = FALSE) %>%
  add_markers(y = ~mpg, color = I("black")) %>%
  add_ribbons(ymin = ~.fitted - 1.96 * .se.fit,
              ymax = ~.fitted + 1.96 * .se.fit, color = I("gray80")) %>%
  add_lines(y = ~.fitted, color = I("steelblue"))
```

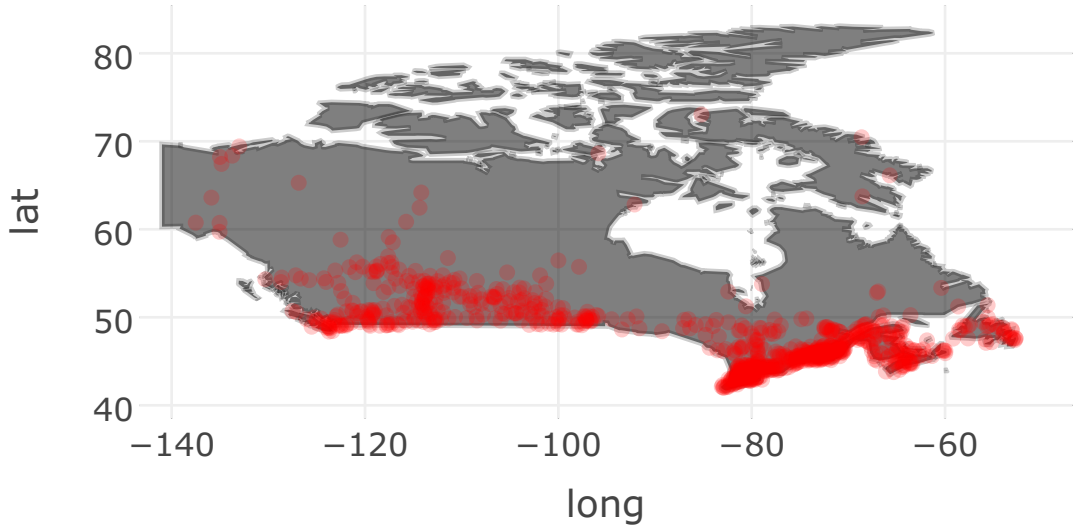


Figure 8.24 A map of Canada using the default cartesian coordinate system.

8.2.1.6 Polygons

The `add_polygons()` function is essentially equivalent to `add_paths()` with the fill attribute set to "toself". Polygons from the basis for other, higher-level, geometries such as `add_ribbons()`, but can be useful in their own right.

```
map_data("world", "canada") %>%
  group_by(group) %>%
  plot_ly(x = ~long, y = ~lat, alpha = 0.2) %>%
  add_polygons(hoverinfo = "none", color = I("black")) %>%
  add_markers(text = ~paste(name, "<br />", pop), hoverinfo = "text",
              color = I("red"), data = maps::canada.cities) %>%
  layout(showlegend = FALSE)
```

8.2.2 Maps

8.2.2.1 Using scatter traces

As shown in polygons, it is possible to create maps using plotly's default (cartesian) coordinate system, but plotly.js also has support for plotting scatter traces on top of either a custom geo layout or a mapbox layout. Figure 8.25 compares the three different layout options in a single subplot.

```
dat <- map_data("world", "canada") %>% group_by(group)

map1 <- plot_ly(dat, x = ~long, y = ~lat) %>%
  add_paths(size = I(1)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75)

map2 <- plot_mapbox(dat, x = ~long, y = ~lat) %>%
  add_paths(size = I(2)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75) %>%
  layout(mapbox = list(zoom = 0,
      center = list(lat = ~median(lat), lon = ~median(long))
   ))

# geo() is the only object type which supports different map projections
map3 <- plot_geo(dat, x = ~long, y = ~lat) %>%
  add_markers(size = I(1)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75) %>%
  layout(geo = list(projection = list(type = "mercator")))
```

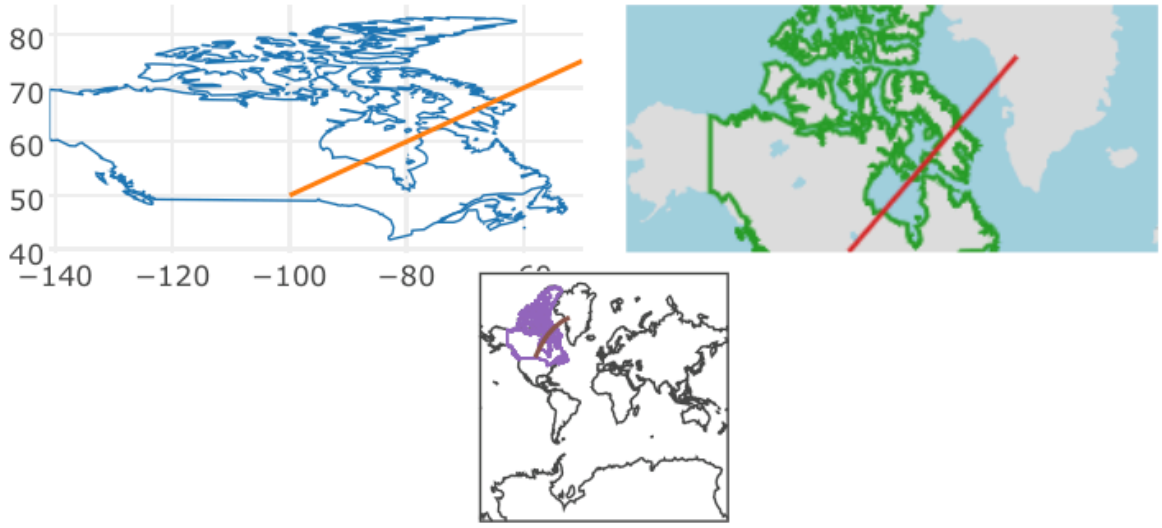


Figure 8.25 Three different ways to render a map. On the top left is plotly's default cartesian coordinate system, on the top right is plotly's custom geographic layout, and on the bottom is mapbox.

```
subplot(map1, map2) %>%
  subplot(map3, nrows = 2) %>%
  hide_legend()
```

Any of the `add_*()` functions found under scatter traces should work as expected on plotly-geo (initialized via `plot_geo()`) or plotly-mapbox (initialized via `plot_mapbox()`) objects. You can think of `plot_geo()` and `plot_mapbox()` as special cases (or more opiniated versions) of `plot_ly()`. For one, they won't allow you to mix scatter and non-scatter traces in a single plot object, which you probably don't want to do anyway. In order to enable Figure 8.25, plotly.js *can't* make this restriction, but since we have `subplot()` in R, we *can* make this restriction without sacrificing flexibility.

8.2.2.2 Choropleths

In addition to scatter traces, plotly-geo objects can also create a choropleth trace/layer. Figure 8.26 shows the population density of the U.S. via a choropleth, and also layers on

markers for the state center locations, using the U.S. state data from the **datasets** package (R Core Team 2016). By simply providing a `z` attribute, plotly-geo objects will try to create a choropleth, but you'll also need to provide `locations` and a `locationmode`.

```
density <- state.x77[, "Population"] / state.x77[, "Area"]


g <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  lakecolor = toRGB('white')
)


plot_geo() %>%
  add_trace(
    z = ~density, text = state.name,
    locations = state.abb, locationmode = 'USA-states'
  ) %>%
  add_markers(
    x = state.center[["x"]], y = state.center[["y"]],
    size = I(2), symbol = I(8), color = I("white"), hoverinfo = "none"
  ) %>%
  layout(geo = g)
```

8.2.3 Bars & histograms

The `add_bars()` and `add_histogram()` functions wrap the bar and histogram plotly.js trace types. The main difference between them is that bar traces require bar heights (both `x` and `y`), whereas histogram traces require just a single variable, and plotly.js handles

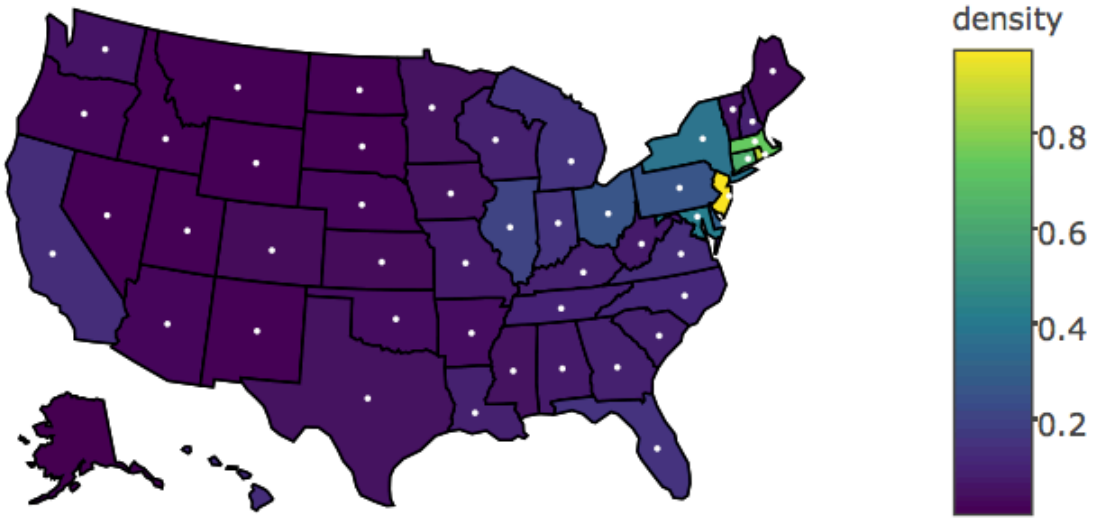


Figure 8.26 A map of U.S. population density using the `state.x77` data from the **datasets** package.

binning in the browser.[5] And perhaps confusingly, both of these functions can be used to visualize the distribution of either a numeric or a discrete variable. So, essentially, the only difference between them is where the binning occurs.

Figure 8.27 compares the default binning algorithm in plotly.js to a few different algorithms available in R via the `hist()` function. Although plotly.js has the ability to customize histogram bins via xbins/ybins, R has diverse facilities for estimating the optimal number of bins in a histogram that we can easily leverage.[6] The `hist()` function alone allows us to reference 3 famous algorithms by name (Sturges 1926); (Freedman and Diaconis 1981); (Scott 1979), but there are also packages (e.g. the **histogram** package) which extend this interface to incorporate more methodology (Mildenberger, Rozenholc, and Zasada. 2009). The `price_hist()` function below wraps the `hist()` function to

[5] This has some interesting applications for linked highlighting as it allows for summary statistics to be computed on-the-fly based on a selection

[6] Optimal in this context is the number of bins which minimizes the distance between the empirical histogram and the underlying density.

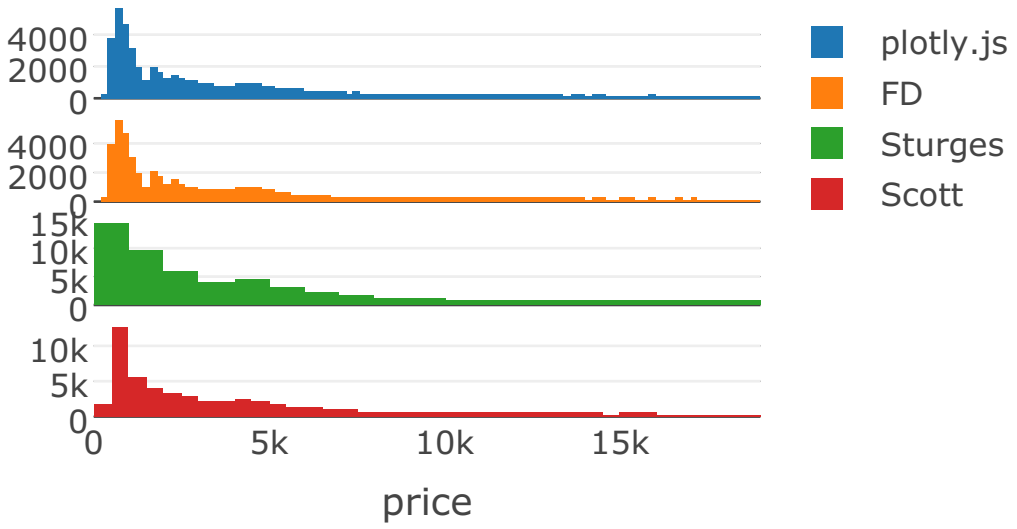


Figure 8.27 plotly.js's default binning algorithm versus R's `hist()` default

obtain the binning results, and map those bins to a plotly version of the histogram using `add_bars()`.

```
p1 <- plot_ly(diamonds, x = ~price) %>% add_histogram(name = "plotly.js")

price_hist <- function(method = "FD") {
  h <- hist(diamonds$price, breaks = method, plot = FALSE)
  plot_ly(x = h$mids, y = h$counts) %>% add_bars(name = method)
}

subplot(
  p1, price_hist(), price_hist("Sturges"),  price_hist("Scott"),
  nrows = 4, shareX = TRUE
)
```

Figure 8.28 demonstrates two ways of creating a basic bar chart. Although the visual results are the same, its worth noting the difference in implementation. The `add_histogram()` function sends all of the observed values to the browser and lets

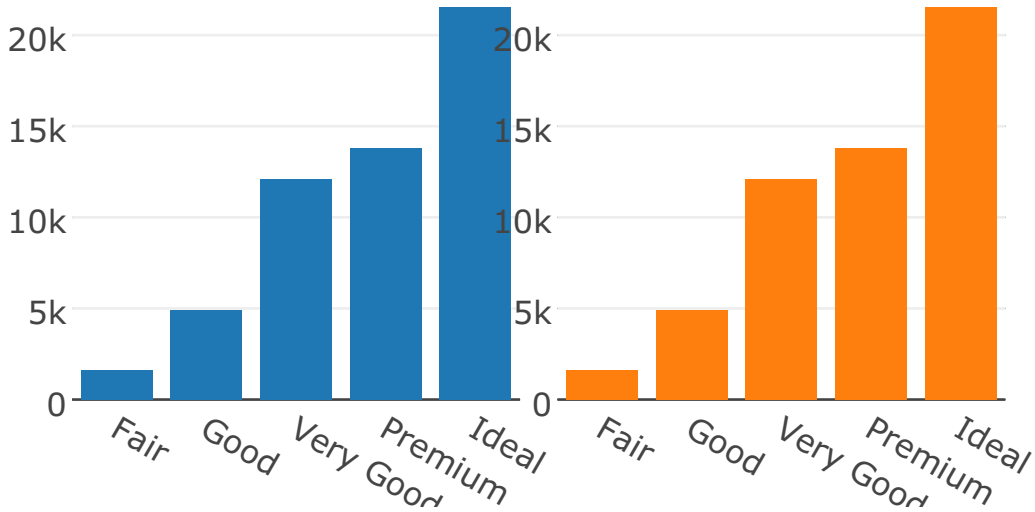


Figure 8.28 Number of diamonds by cut.

plotly.js perform the binning. It takes more human effort to perform the binning in R, but doing so has the benefit of sending less data, and requiring less computation work of the web browser. In this case, we have only about 50,000 records, so there is much of a difference in page load times or page size. However, with 1 Million records, page load time more than doubles and page size nearly doubles.[7]

```
p1 <- plot_ly(diamonds, x = ~cut) %>% add_histogram()


p2 <- diamonds %>%
  dplyr::count(cut) %>%
  plot_ly(x = ~cut, y = ~n) %>%
  add_bars()


subplot(p1, p2) %>% hide_legend()
```

[7]These tests were run on Google Chrome and loaded a page with a single bar chart. Here are the results for `add_histogram()` and here are the results for `add_bars()`

8.2.3.1 Multiple numeric distributions

It is often useful to see how the numeric distribution changes with respect to a discrete variable. When using bars to visualize multiple numeric distributions, I recommend plotting each distribution on its own axis, rather than trying to overlay them on a single axis.[8]. This is where the `subplot()` infrastructure, and its support for trellis displays, comes in handy. Figure 8.29 shows a trellis display of diamond price by diamond color. Note how the `one_plot()` function defines what to display on each panel, then a split-apply-recombine strategy is employed to generate the trellis display.

```
one_plot <- function(d) {
  plot_ly(d, x = ~price) %>%
    add_annotations(
      ~unique(clarity), x = 0.5, y = 1,
      xref = "paper", yref = "paper", showarrow = FALSE
    )
}

diamonds %>%
  split(.$clarity) %>%
  lapply(one_plot) %>%
  subplot(nrows = 2, shareX = TRUE, titleX = FALSE) %>%
  hide_legend()
```

[8]It's much easier to visualize multiple numeric distributions on a single axis using lines

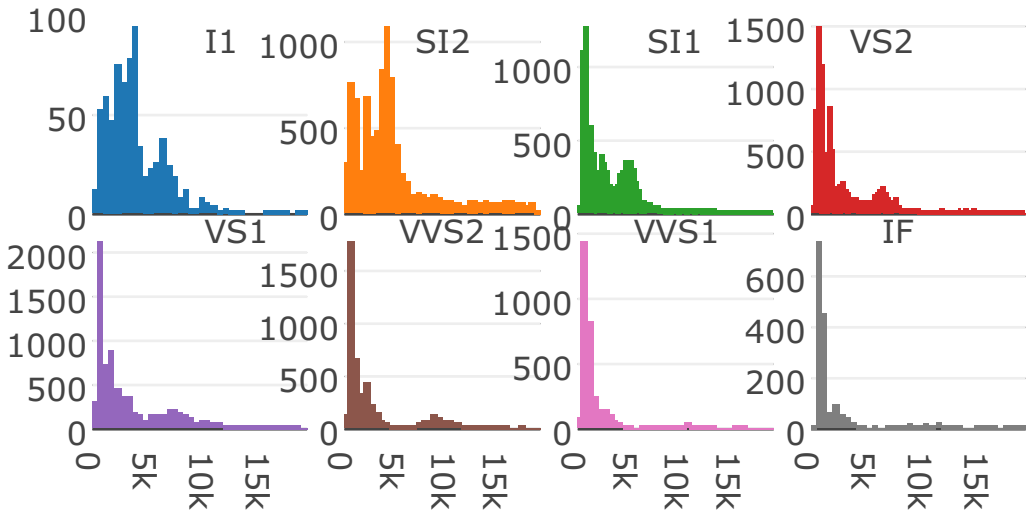


Figure 8.29 A trellis display of diamond price by diamond clarity.

8.2.3.2 Multiple discrete distributions

Visualizing multiple discrete distributions is difficult. The subtle complexity is due to the fact that both counts and proportions are important for understanding multi-variate discrete distributions. Figure 8.30 presents diamond counts, divided by both their cut and clarity, using a grouped bar chart.

```
plot_ly(diamonds, x = ~cut, color = ~clarity) %>%
  add_histogram()
```

Figure 8.30 is useful for comparing the number of diamonds by clarity, given a type of cut. For instance, within "Ideal" diamonds, a cut of "VS1" is most popular, "VS2" is second most popular, and "I1" the least popular. The distribution of clarity within "Ideal" diamonds seems to be fairly similar to other diamonds, but it's hard to make this comparison using raw counts. Figure 8.31 makes this comparison easier by showing the relative frequency of diamonds by clarity, given a cut.

```
# number of diamonds by cut and clarity (n)
cc <- count(diamonds, cut, clarity)
```

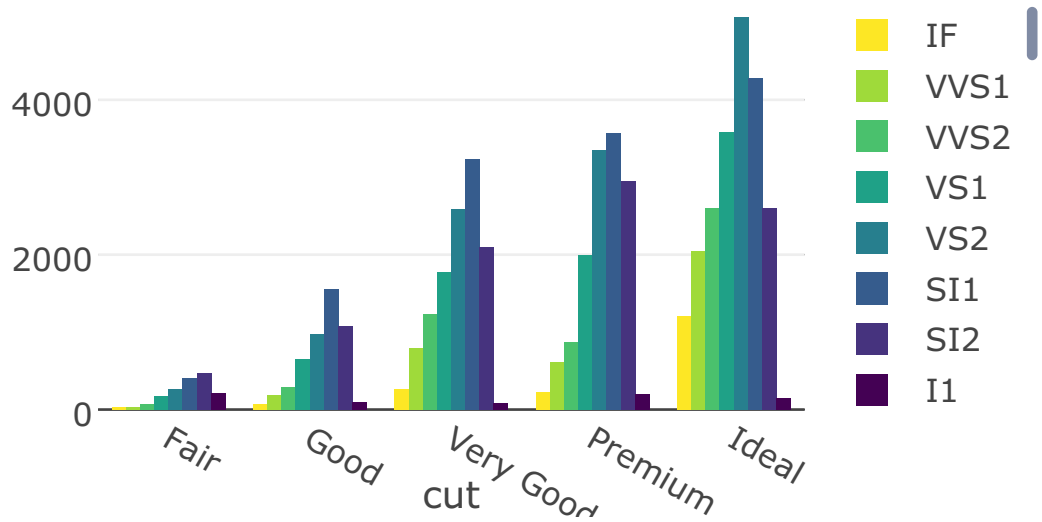


Figure 8.30 A grouped bar chart

```
# number of diamonds by cut (nn)
cc2 <- left_join(cc, count(cc, cut, wt = n))
cc2 %>%
  mutate(prop = n / nn) %>%
  plot_ly(x = ~cut, y = ~prop, color = ~clarity) %>%
  add_bars() %>%
  layout(barmode = "stack")
```

This type of plot, also known as a spine plot, is a special case of a mosaic plot. In a mosaic plot, you can scale both bar widths and heights according to discrete distributions. For mosaic plots, I recommend using the **ggmosaic** package (Jeppson, Hofmann, and Cook, n.d.), which implements a custom **ggplot2** geom designed for mosaic plots, which we can convert to plotly via `ggplotly()`. Figure 8.32 show a mosaic plot of cut by clarity. Notice how the bar widths are scaled proportional to the cut frequency.

```
library(ggmosaic)
p <- ggplot(data = cc) +
  geom_mosaic(aes(weight = n, x = product(cut), fill = clarity))
```

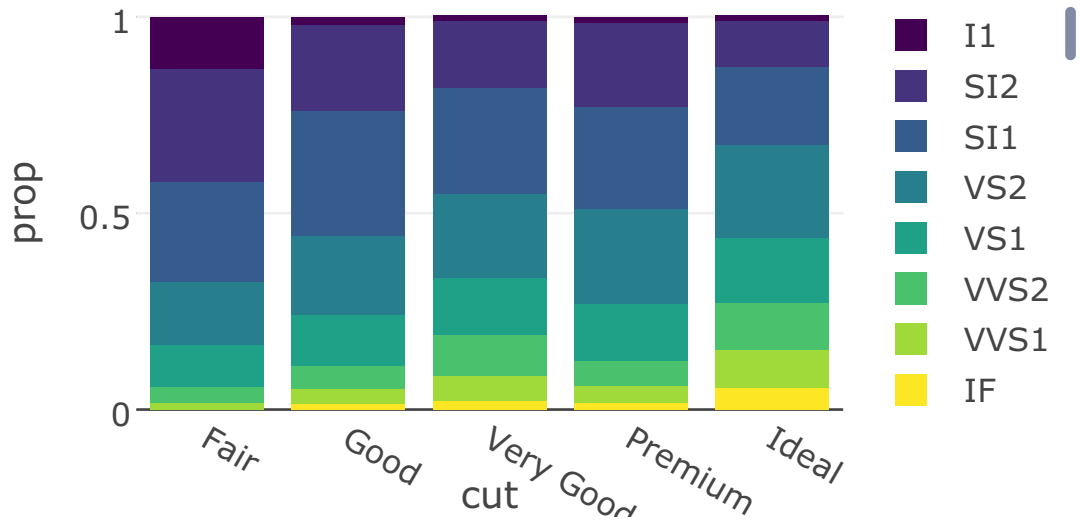


Figure 8.31 A stacked bar chart showing the proportion of clarity within

```
#> Error: GeomMosaic was built with an incompatible version of ggproto.
#> Please reinstall the package that provides this extension.
ggplotly(p)
```

8.2.4 Boxplots

Boxplots encode the five number summary of a numeric variable, and are more efficient than trellis displays of histograms for comparing many numeric distributions. The `add_boxplot()` function requires one numeric variable, and guarantees boxplots are oriented correctly, regardless of whether the numeric variable is placed on the x or y scale. As Figure 8.33 shows, on the axis orthogonal to the numeric axis, you can provide a discrete variable (for conditioning) or supply a single value (to name the axis category).

```
p <- plot_ly(diamonds, y = ~price, color = I("black"),
             alpha = 0.1, boxpoints = "suspectedoutliers")
p1 <- p %>% add_boxplot(x = "Overall")
p2 <- p %>% add_boxplot(x = ~cut)
```

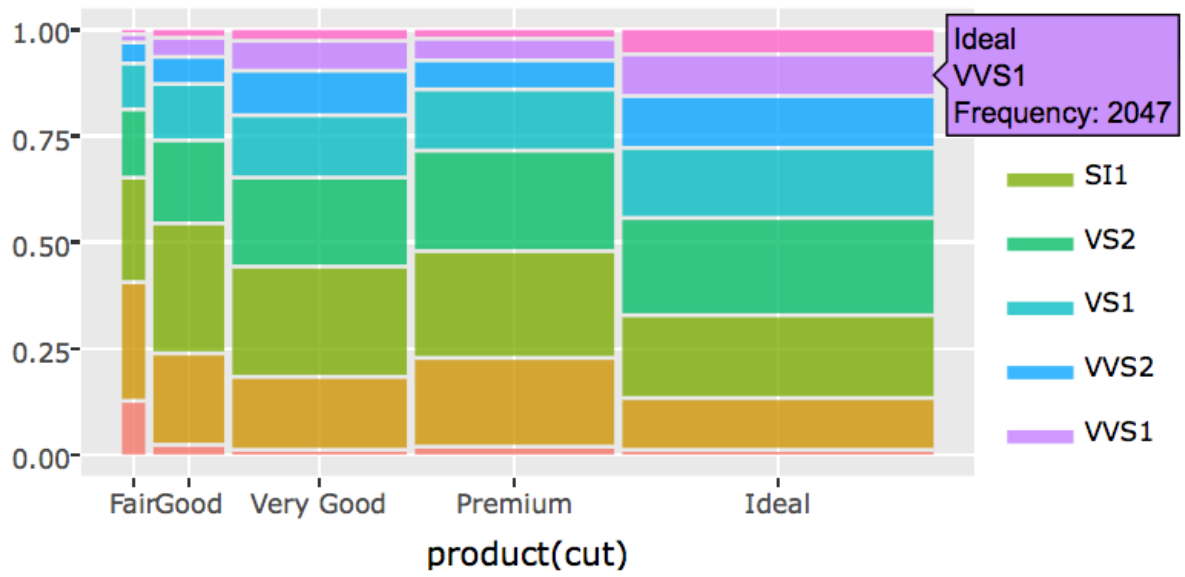


Figure 8.32 Using ggmosaic and ggplotly() to create advanced interactive visualizations of categorical data

```
subplot(
  p1, p2, shareY = TRUE,
  widths = c(0.2, 0.8), margin = 0
) %>% hide_legend()
```

If you want to partition by more than one discrete variable, I recommend mapping the interaction of those variables to the discrete axis, and coloring by the nested variable, as Figure 8.34 does with diamond clarity and cut.

```
plot_ly(diamonds, x = ~price, y = ~interaction(clarity, cut)) %>%
  add_boxplot(color = ~clarity) %>%
  layout(yaxis = list(title = ""), margin = list(l = 100))
```

It is also helpful to sort the boxplots according to something meaningful, such as the median price. Figure 8.35 presents the same information as Figure 8.34, but sorts the boxplots by their median, and makes it immediately clear that diamonds with a cut of "SI2" have the highest diamond price, on average.

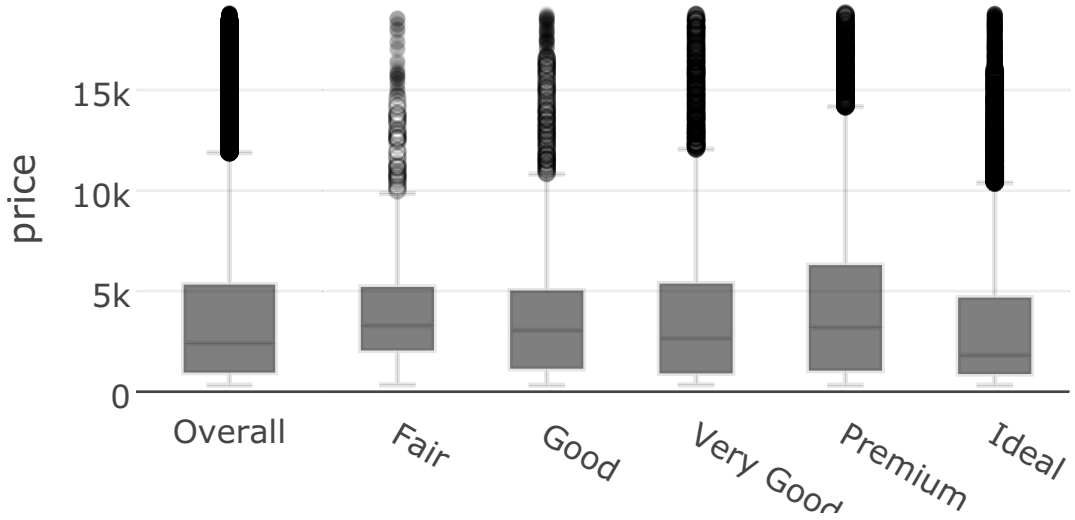


Figure 8.33 Overall diamond price and price by cut.

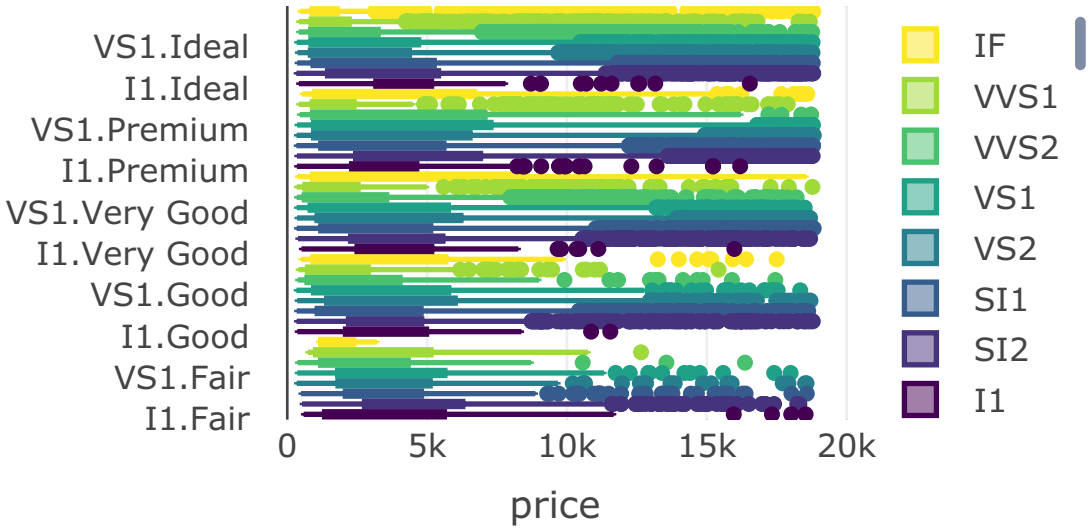


Figure 8.34 Diamond prices by cut and clarity.

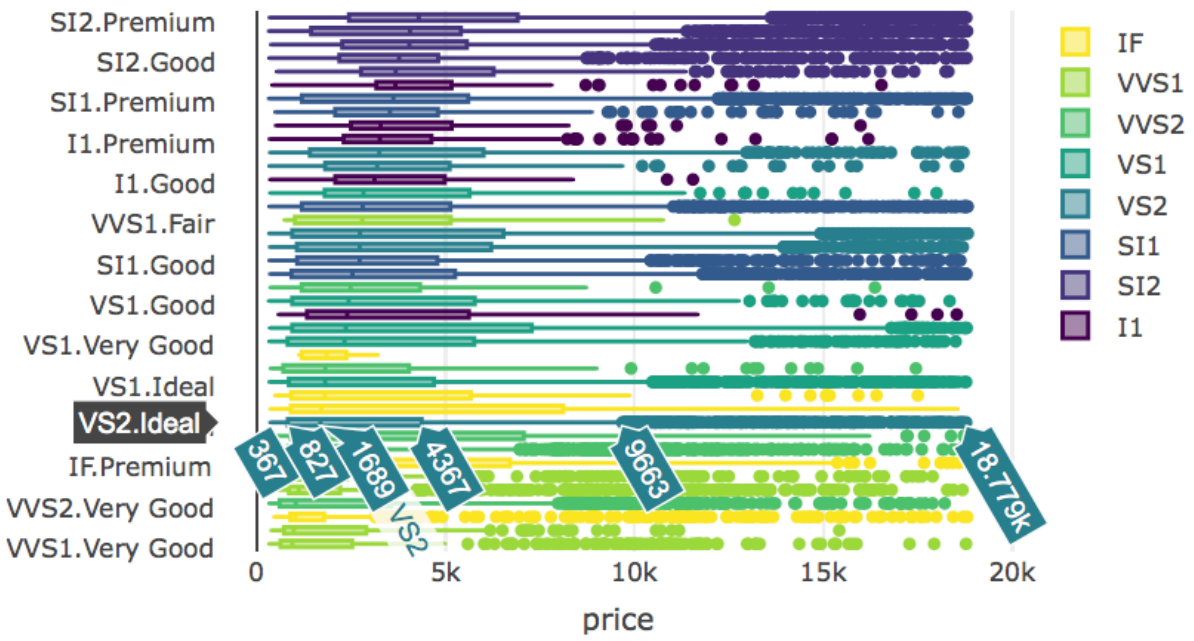


Figure 8.35 Diamond prices by cut and clarity, sorted by price median.

```
d <- diamonds %>%
  mutate(cc = interaction(clarity, cut))


# interaction levels sorted by median price
lvls <- d %>%
  group_by(cc) %>%
  summarise(m = median(price)) %>%
  arrange(m) %>%
  .[["cc"]]


plot_ly(d, x = ~price, y = ~factor(cc, lvls)) %>%
  add_boxplot(color = ~clarity) %>%
  layout(yaxis = list(title = ""), margin = list(l = 100))
```

Similar to `add_histogram()`, `add_boxplot()` sends the raw data to the browser, and lets plotly.js compute summary statistics. Unfortunately, plotly.js does not yet allow precomputed statistics for boxplots.[9]

8.2.5 2D frequencies

8.2.5.1 Rectangular binning in plotly.js

The **plotly** package provides two functions for displaying rectangular bins: `add_heatmap()` and `add_histogram2d()`. For numeric data, the `add_heatmap()` function is a 2D analog of `add_bars()` (bins must be pre-computed), and the `add_histogram2d()` function is a 2D analog of `add_histogram()` (bins can be computed in the browser). Thus, I recommend `add_histogram2d()` for exploratory purposes, since you don't have to think about how to perform binning. It also provides a useful `zsmooth` attribute for effectively increasing the number of bins (currently, "best" performs a bi-linear interpolation, a type of nearest neighbors algorithm), and nbinsx/nbinsy attributes to set the number of bins in the x and/or y directions. Figure 8.36 compares three different uses of `add_histogram()`: (1) plotly.js' default binning algorithm, (2) the default plus smoothing, (3) setting the number of bins in the x and y directions. Its also worth noting that filled contours, instead of bins, can be used in any of these cases by using `histogram2dcontour()` instead of `histogram2d()`.

```
p <- plot_ly(diamonds, x = ~log(carat), y = ~log(price))
subplot(
  add_histogram2d(p) %>%
    colorbar(title = "default", len = 1/3, y = 1) %>%
    layout(xaxis = list(title = "default")),
```

[9] Follow the issue here https://github.com/plotly/plotly.js/issues/242

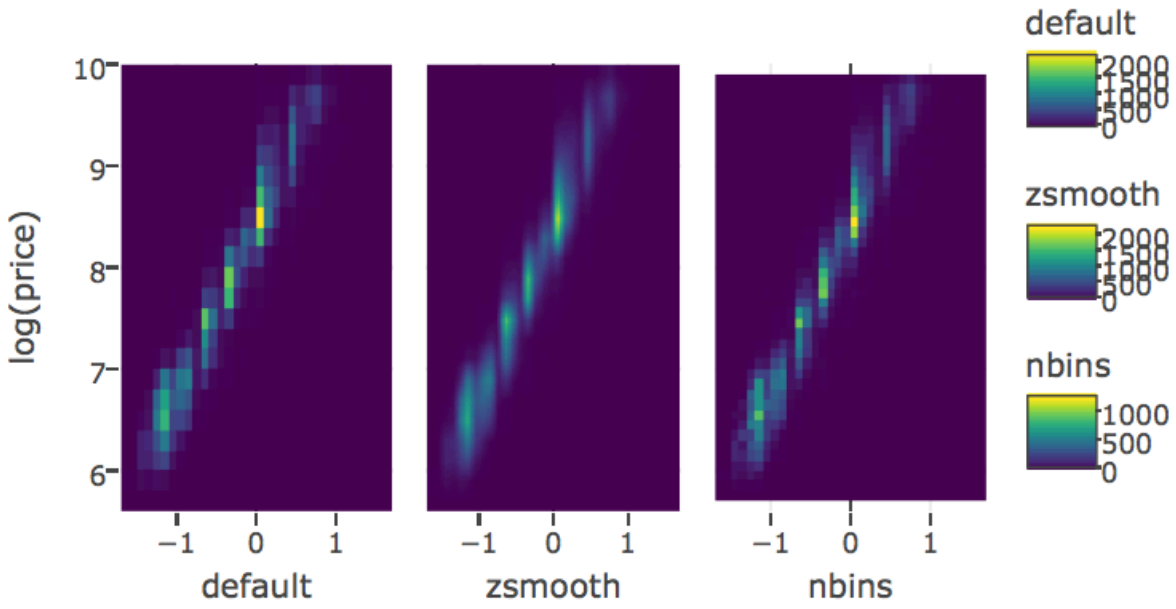


Figure 8.36 Three different uses of `histogram2d()`

```
  add_histogram2d(p, zsmooth = "best") %>%
    colorbar(title = "zsmooth", len = 1/3, y = 2/3 - 0.05) %>%
    layout(xaxis = list(title = "zsmooth")),
  add_histogram2d(p, nbinsx = 60, nbinsy = 60) %>%
    colorbar(title = "nbins", len = 1/3, y = 1/3 - 0.1) %>%
    layout(xaxis = list(title = "nbins")),
  shareY = TRUE, titleX = TRUE
)
```

8.2.5.2 Rectangular binning in R

In Bars & histograms, we leveraged a number of algorithms in R for computing the "optimal" number of bins for a histogram, via `hist()`, and routing those results to `add_bars()`. There is a surprising lack of research and computational tools for the 2D analog, and among the research that does exist, solutions usually depend on charac-

teristics of the unknown underlying distribution, so the typical approach is to assume a Gaussian form (Scott 1992). Practically speaking, that assumption is not very useful, but 2D kernel density estimation provides a useful alternative that tends to be more robust to changes in distributional form. Although kernel density estimation requires choice of kernel and a bandwidth parameter, the `kde2d()` function from the **MASS** package provides a well-supported rule-of-thumb for estimating the bandwidth of a Gaussian kernel density (Venables and Ripley 2002). Figure 8.37 uses `kde2d()` to estimate a 2D density, scales the relative frequency to an absolute frequency, then uses the `add_heatmap()` function to display the results as a heatmap.

```
kde_count <- function(x, y, ...) {
  kde <- MASS::kde2d(x, y, ...)
  df <- with(kde, setNames(expand.grid(x, y), c("x", "y")))
  # The 'z' returned by kde2d() is a proportion, but we can scale it to a count
  df$count <- with(kde, c(z) * length(x) * diff(x)[1] * diff(y)[1])
  data.frame(df)
}

kd <- with(diamonds, kde_count(log(carat), log(price), n = 30))
plot_ly(kd, x = ~x, y = ~y, z = ~count) %>%
  add_heatmap() %>%
  colorbar(title = "Number of diamonds")
```

8.2.5.3 Categorical axes

The functions `add_histogram()`, `add_histogram2contour()`, and `add_heatmap()` all support categorical axes. Thus, `add_histogram()` *can* be used to easily display 2-way contingency tables, but since its easier to compare values along a common scale rather

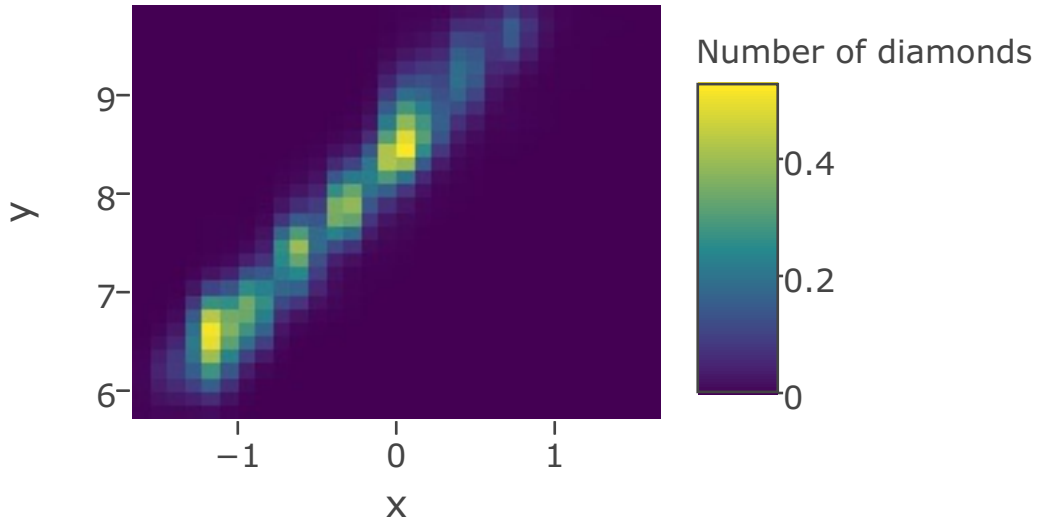


Figure 8.37 2D Density estimation via the `kde2d()` function

than compare colors (Cleveland and McGill 1984), I recommend creating grouped bar charts instead. The `add_heatmap()` function can still be useful for categorical axes, however, as it allows us to display whatever quantity we want along the z axis (color).

Figure 8.38 uses `add_heatmap()` to display a correlation matrix. Notice how the `limits` arguments in the `colorbar()` function can be used to expand the limits of the color scale to reflect the range of possible correlations (something that is not easily done in plotly.js).

```
corr <- cor(diamonds[vapply(diamonds, is.numeric, logical(1))])
plot_ly(x = rownames(corr), y = colnames(corr), z = corr) %>%
  add_heatmap() %>%
  colorbar(limits = c(-1, 1))
```

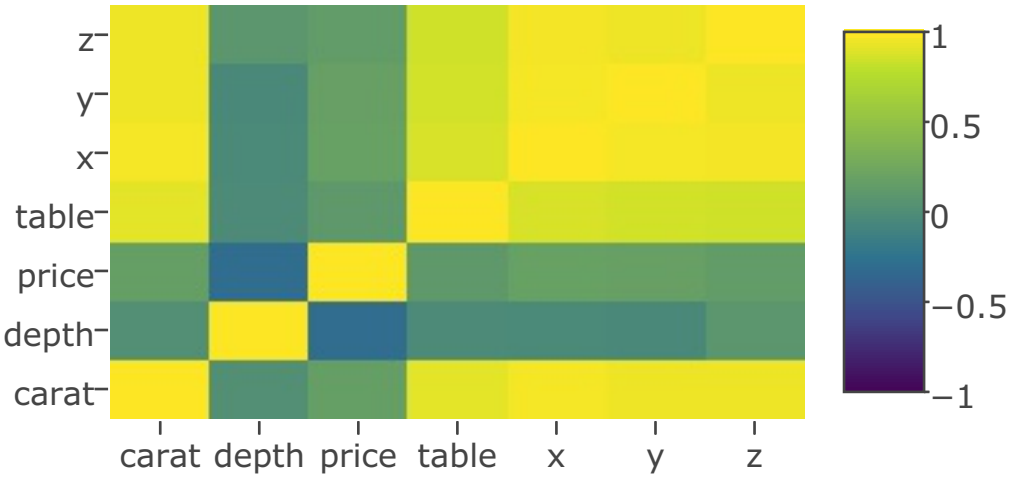


Figure 8.38 Displaying a correlation matrix with `add_heatmap()` and controling the scale limits with `colorbar()`.

8.2.6 Other 3D plots

In scatter traces, we saw how to make 3D scatter plots and 3D paths/lines, but plotly.js also supports 3D surface and triangular mesh surfaces (aka trisurf plots). For a nice tutorial on creating trisurf plots in R via `plot_ly()`, I recommend visiting this tutorial.

Creating 3D surfaces with `add_surface()` is a lot like creating heatmaps with `add_heatmap()`. In fact, you can even create 3D surfaces over categorical x/y (try changing `add_heatmap()` to `add_surface()` in Figure 8.38)! That being said, there should be a sensible ordering to the x/y axes in a surface plot since plotly.js interpolates z values. Usually the 3D surface is over a continous region, as is done in Figure 8.39 to display the height of a volcano. If a numeric matrix is provided to z as in Figure 8.39, the x and y attributes do not have to be provided, but if they are, the length of x should match the number of rows in the matrix and y should match the number of columns.

```
x <- seq_len(nrow(volcano)) + 100
y <- seq_len(ncol(volcano)) + 500
plot_ly() %>% add_surface(x = ~x, y = ~y, z = ~volcano)
```

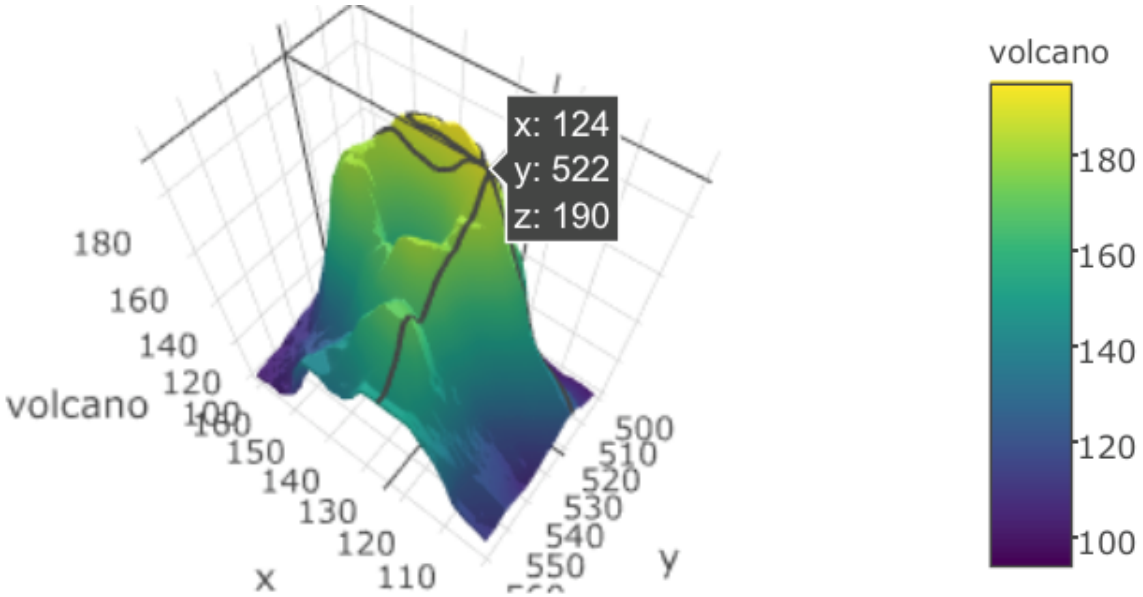


Figure 8.39 A 3D surface of volcano height.

8.3 Arranging multiple views

One technique essential to high-dimensional data analysis is the ability to arrange multiple views. Ideally, these views are linked in some way to foster comparisons (the next chapter discusses linking techniques). The next section, Arranging htmlwidgets describes techniques for arranging htmlwidget objects, which many R packages for creating web-based data visualizations build upon, including **plotly**. Typically interactivity is isolated *within* an htmlwidget object, but Linking views without shiny explores some more recent work on enabling interactivity *across* htmlwidget objects. The following section, Subplots describes the `subplot()` function, which is useful for *merging* multiple plotly objects into a single htmlwidget object. The main benefit of merging (rather than arranging) plotly objects is that it gives us the ability to synchronize zoom and pan events across multiple axes. The last section, Navigating many views discusses some useful tools for restricting focus on interesting views when there are more views than you can possibly digest visually.

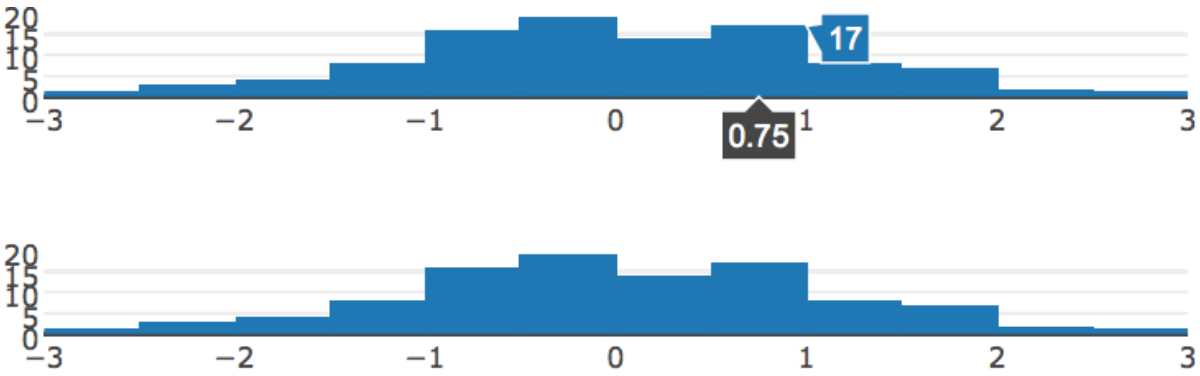


Figure 8.40 Printing multiple htmlwidget objects with `tagList()`. To render tag lists at the command line, wrap them in `browsable()`

8.3.1 Arranging htmlwidgets

Since plotly objects inherit properties from an htmlwidget object, any method that works for arranging htmlwidgets also works for plotly objects. In some sense, an htmlwidget object is just a collection of HTML tags, and the **htmltools** package provides some useful functions for working with HTML tags (RStudio and Inc. 2016). The `tagList()` function gathers multiple HTML tags into a tag list, and when printing a tag list inside of a **knitr**/**rmarkdown** document (Xie 2013b); (Allaire et al. 2016), it knows to render as HTML. When printing outside of this context (e.g., at the command line), a tag list prints as a character string by default. In order to view the rendered HTML, provide the tag list to the `browsable()` function.

```
library(htmltools)
library(plotly)
p <- plot_ly(x = rnorm(100))
tagList(p, p)
```

Figure 8.40 renders two plots, each in its own row spanning the width of the page, because each htmlwidget object is an HTML `<div>` tag. More often than not, it is desirable to arrange multiple plots in a given row, and there are a few ways to do that.

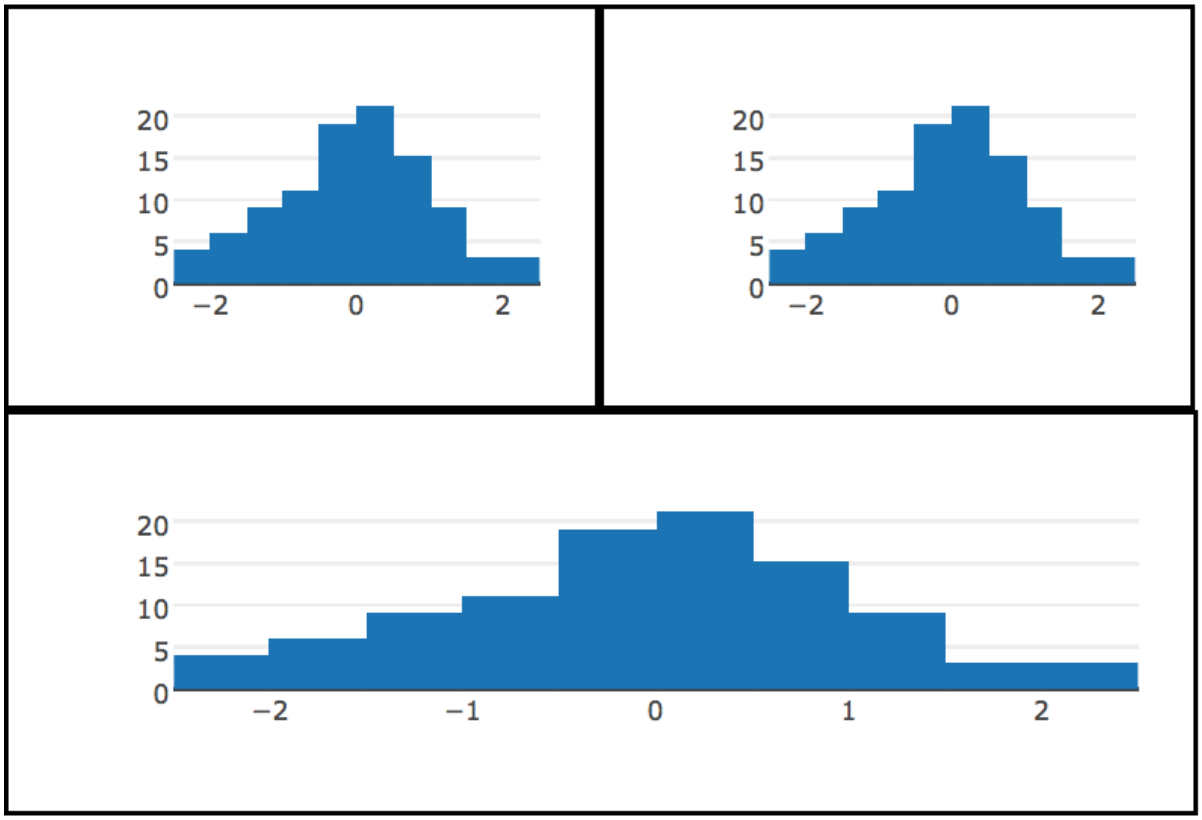


Figure 8.41 Arranging multiple htmlwidgets with flexbox

A very flexible approach is to wrap all of your plots in a flexbox (i.e., an HTML `<div>` with `display: flex` Cascading Style Sheets (CSS) property). The `tags$div()` function from **htmltools** provides a way to wrap a `<div>` around both tag lists and htmlwidget objects, and set attributes, such as `style`. As Figure 8.41 demonstrates, this approach also provides a nice way to add custom styling to the page, such as borders around each panel.

```
tags$div(
  style = "display: flex; flex-wrap: wrap",
  tags$div(p, style = "width: 50%; padding: 1em; border: solid;"),
  tags$div(p, style = "width: 50%; padding: 1em; border: solid;"),
  tags$div(p, style = "width: 100%; padding: 1em; border: solid;")
)
```

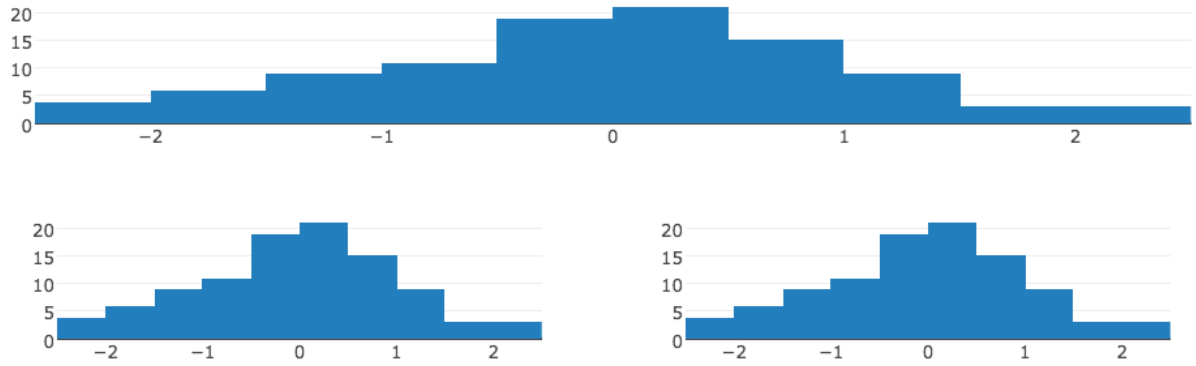


Figure 8.42 Arranging multiple htmlwidgets with `fluidPage()` from the **shiny** package.

Another way to arrange multiple htmlwidget objects on a single page is to leverage the `fluidPage()`, `fluidRow()`, and `column()` functions from the **shiny** package.

```
library(shiny)
fluidPage(
  fluidRow(p),
  fluidRow(
    column(6, p), column(6, p)
  )
)
```

All the arrangement approaches discussed thus far are agnostic to output format, meaning that they can be used to arrange htmlwidgets within *any* **knitr**/**rmarkdown** document.[10] If the htmlwidgets do not need to be embedded within a larger document that requires an opininated output format, the **flexdashboard** package provides a **rmarkdown** template for generating dashboards, with a convenient syntax for arranging views (Allaire 2016).

[10] Although HTML can not possibly render in a pdf or word document, **knitr** can automatically detect a non-HTML output format and embed a static image of the htmlwidget via the **webshot** package (Chang 2016).

8.3.2 Merging plotly objects

The `subplot()` function provides a flexible interface for merging multiple plotly objects into a single object (i.e., view). It is more flexible than most trellis display frameworks (e.g., ggplot2's `facet_wrap()`) as you don't have to condition on a value of common variable in each display (Richard A. Becker 1996). Its capabilities and interface is similar to the `grid.arrange()` function from the **gridExtra** package, which allows you to arrange multiple **grid** grobs in a single view, effectively providing a way to arrange (possibly unrelated) **ggplot2** and/or **lattice** plots in a single view (R Core Team 2016); (Auguie 2016); (Sarkar 2008). Figure 8.43 shows the most simple way to use `subplot()` which is to directly supply plotly objects.

```
library(plotly)
p1 <- plot_ly(economics, x = ~date, y = ~unemploy) %>%
  add_lines(name = "unemploy")
p2 <- plot_ly(economics, x = ~date, y = ~uempmed) %>%
  add_lines(name = "uempmed")
subplot(p1, p2)
```

Although `subplot()` accepts an arbitrary number of plot objects, passing a *list* of plots can save typing and redundant code when dealing with a large number of plots. Figure 8.44 shows one time series for each variable in the `economics` dataset and share the x-axis so that zoom/pan events are synchronized across each series:

```
vars <- setdiff(names(economics), "date")
plots <- lapply(vars, function(var) {
  plot_ly(economics, x = ~date, y = as.formula(paste0("~", var)), name = var) %>%
    add_lines()
})
```

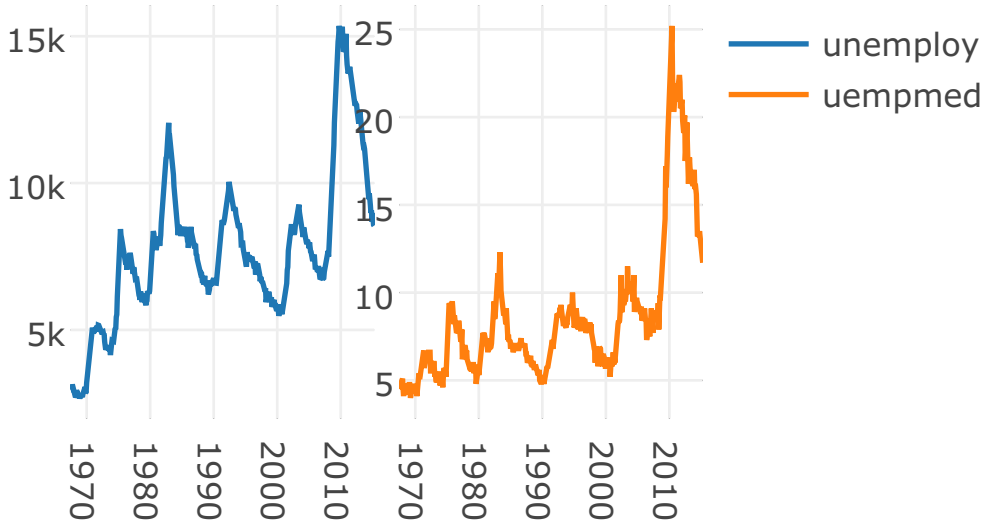


Figure 8.43 The most basic use of `subplot()` to merge multiple plotly objects into a single plotly object.

```
subplot(plots, nrows = length(plots), shareX = TRUE, titleX = FALSE)
```

A plotly subplot is a single plotly graph with multiple traces anchored on different axes. If you pre-specify an axis ID for each trace, `subplot()` will respect that ID. Figure 8.45 uses this fact in correspondence with the fact that mapping a discrete variable to `color` creates one trace per value. In addition to providing more control over trace placement, this provides a convenient way to control coloring (we could have `symbol`/`linetype` to acheive the same effect).

```
economics %>%
  tidyr::gather(variable, value, -date) %>%
  transform(id = as.integer(factor(variable))) %>%
  plot_ly(x = ~date, y = ~value, color = ~variable, colors = "Dark2",
          yaxis = ~paste0("y", id)) %>%
  add_lines() %>%
  subplot(nrows = 5, shareX = TRUE)
```

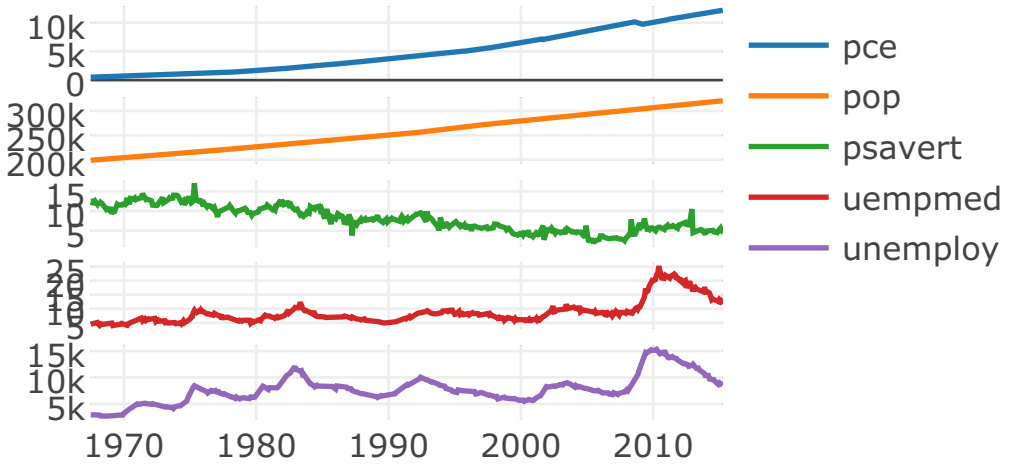


Figure 8.44 Five different economic variables on different y scales and a common x scale. Zoom and pan events in the x-direction are synchronized across plots.

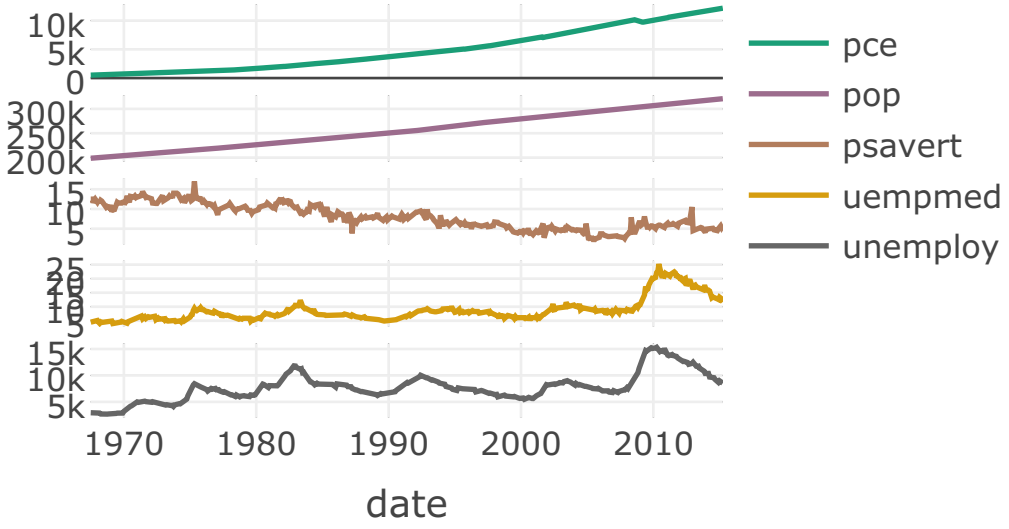


Figure 8.45 Pre-populating y axis IDs.

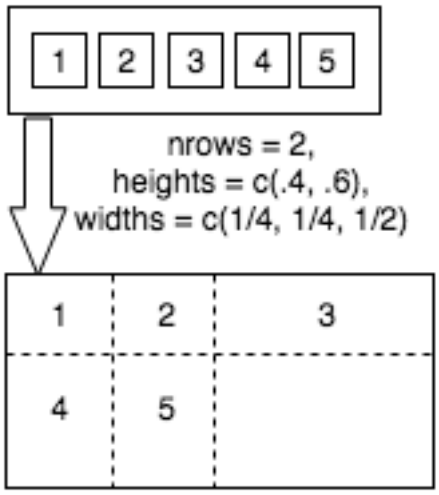


Figure 8.46 A visual diagram of controling the `heights` of rows and `widths` of columns.

Conceptually, `subplot()` provides a way to place a collection of plots into a table with a given number of rows and columns. The number of rows (and, by consequence, the number of columns) is specified via the `nrows` argument. By default each row/column shares an equal proportion of the overall height/width, but as shown in Figure 8.46 the default can be changed via the `heights` and `widths` arguments.

This flexibility is quite useful for a number of visualizations, for example, as shown in Figure 8.47, a joint density plot is really of subplot of joint and marginal densities. The **heatmaply** package is great example of leveraging `subplot()` in a similar way to create interactive dendograms (Galili 2016).

```
x <- rnorm(100)
y <- rnorm(100)
s <- subplot(
  plot_ly(x = x, color = I("black")),
  plotly_empty(),
  plot_ly(x = x, y = y, color = I("black")),
  plot_ly(y = y, color = I("black")),
  nrows = 2, heights = c(0.2, 0.8), widths = c(0.8, 0.2),
```

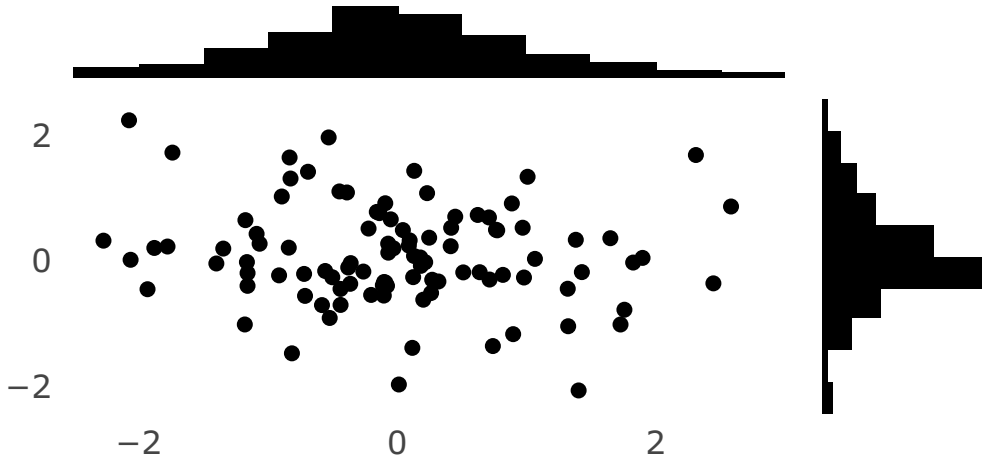


Figure 8.47 A joint density plot with synchronized axes.

```
  shareX = TRUE, shareY = TRUE, titleX = FALSE, titleY = FALSE
)
layout(s, showlegend = FALSE)
```

8.3.2.1 Recursive subplots

The `subplot()` function returns a plotly object so it can be modified like any other plotly object. This effectively means that subplots work recursively (i.e., you can have subplots within subplots). This idea is useful when your desired layout doesn't conform to the table structure described in the previous section. In fact, you can think of a subplot of subplots like a spreadsheet with merged cells. Figure 8.48 gives a basic example where each row of the outer-most subplot contains a different number of columns.

```
plotList <- function(nplots) {
  lapply(seq_len(nplots), function(x) plot_ly())
}
s1 <- subplot(plotList(6), nrows = 2, shareX = TRUE, shareY = TRUE)
s2 <- subplot(plotList(2), shareY = TRUE)
```

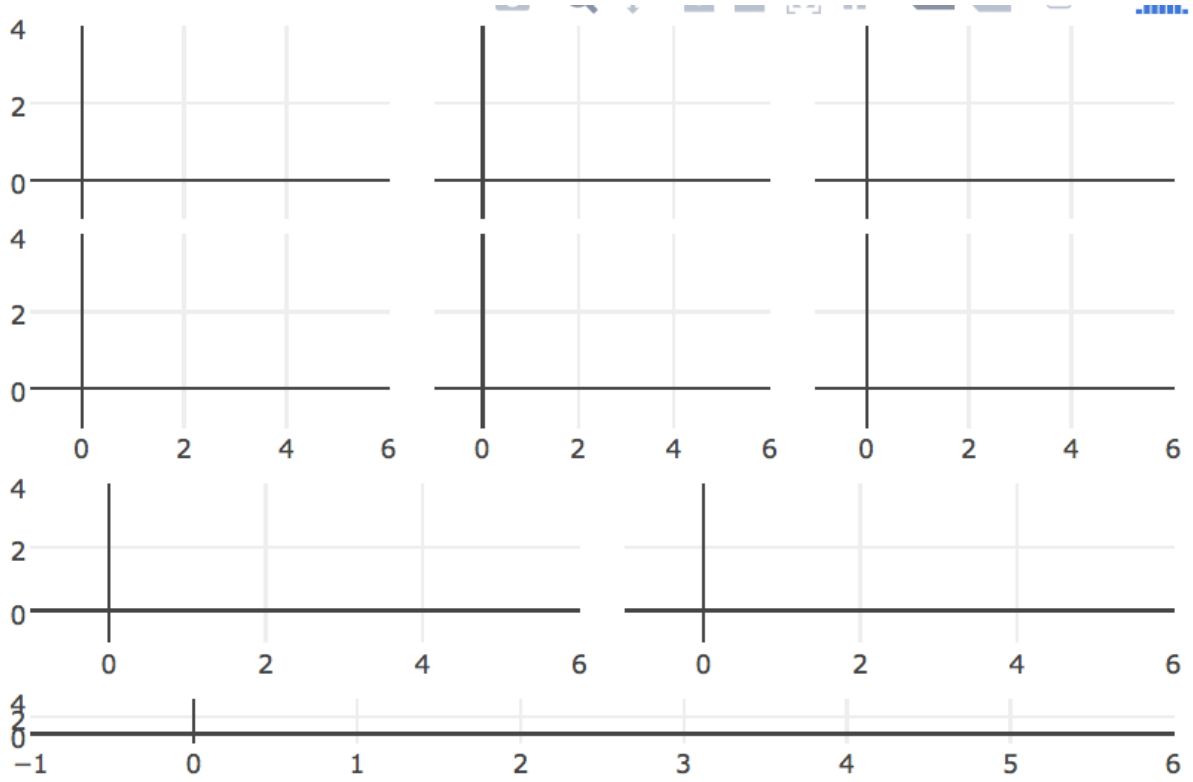


Figure 8.48 Recursive subplots.

```
subplot(s1, s2, plot_ly(), nrows = 3, margin = 0.04, heights = c(0.6, 0.3, 0.1))
```

The concept is particularly useful when you want plot(s) in a given row to have different widths from plot(s) in another row. Figure 8.49 uses this recursive behavior to place many bar charts in the first row, and a single choropleth in the second row.

```
# specify some map projection/options
g <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  lakecolor = toRGB('white')
)
# create a map of population density
```

```
density <- state.x77[, "Population"] / state.x77[, "Area"]
map <- plot_geo(z = ~density, text = state.name,
                locations = state.abb, locationmode = 'USA-states') %>%
  layout(geo = g)
# create a bunch of horizontal bar charts
vars <- colnames(state.x77)
barcharts <- lapply(vars, function(var) {
  plot_ly(x = state.x77[, var], y = state.name) %>%
    add_bars(orientation = "h", name = var) %>%
    layout(showlegend = FALSE, hovermode = "y",
           yaxis = list(showticklabels = FALSE))
})
subplot(
  subplot(barcharts, margin = 0.01), map,
  nrows = 2, heights = c(0.3, 0.7), margin = 0.1
)
```

8.3.2.2 ggplot2 subplots

Underneath the hood, ggplot2 facets are implemented as subplots, which enables the synchronized zoom events on shared axes. Since subplots work recursively, it is also possible to have a subplot of ggplot2 faceted plots, as Figure 8.50 shows. Moreover, `subplot()` can understand ggplot objects, so there is no need to translate them to plotly object via `ggplotly()` (unless you want to leverage some of the `ggplotly()` arguments, such as `tooltip` for customizing information displayed on hover).

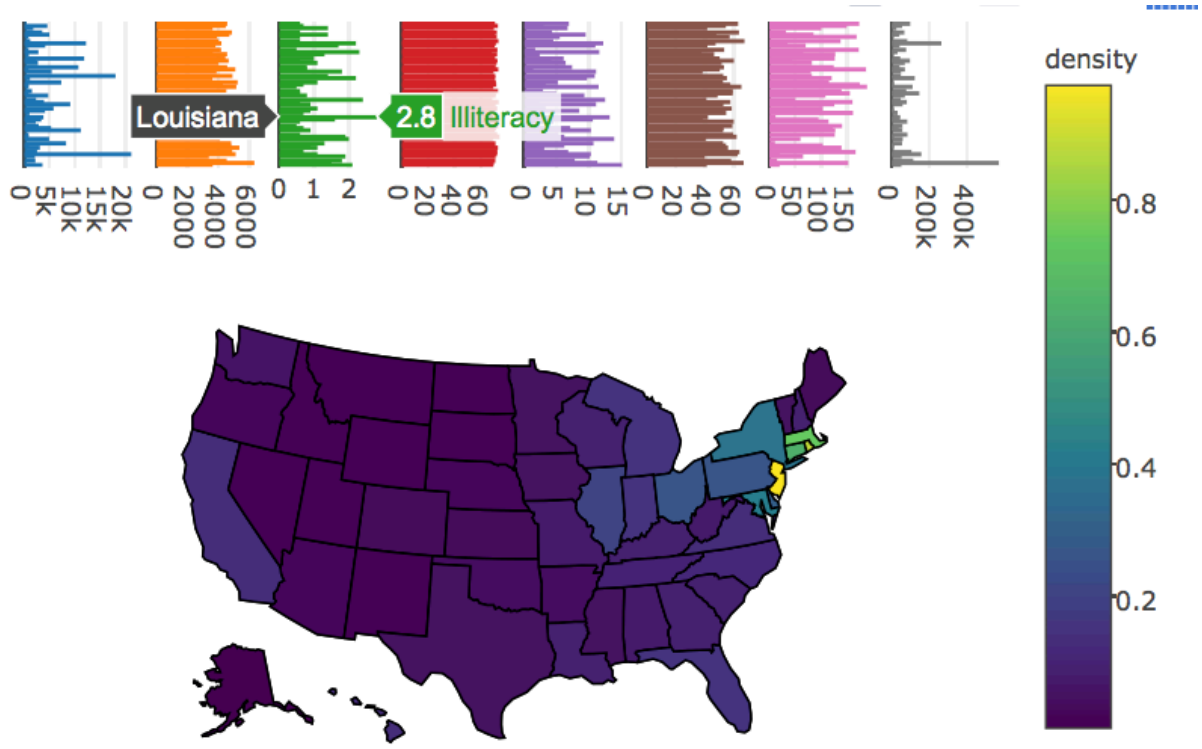


Figure 8.49 Multiple bar charts of US statistics by state in a subplot with a choropleth of population density

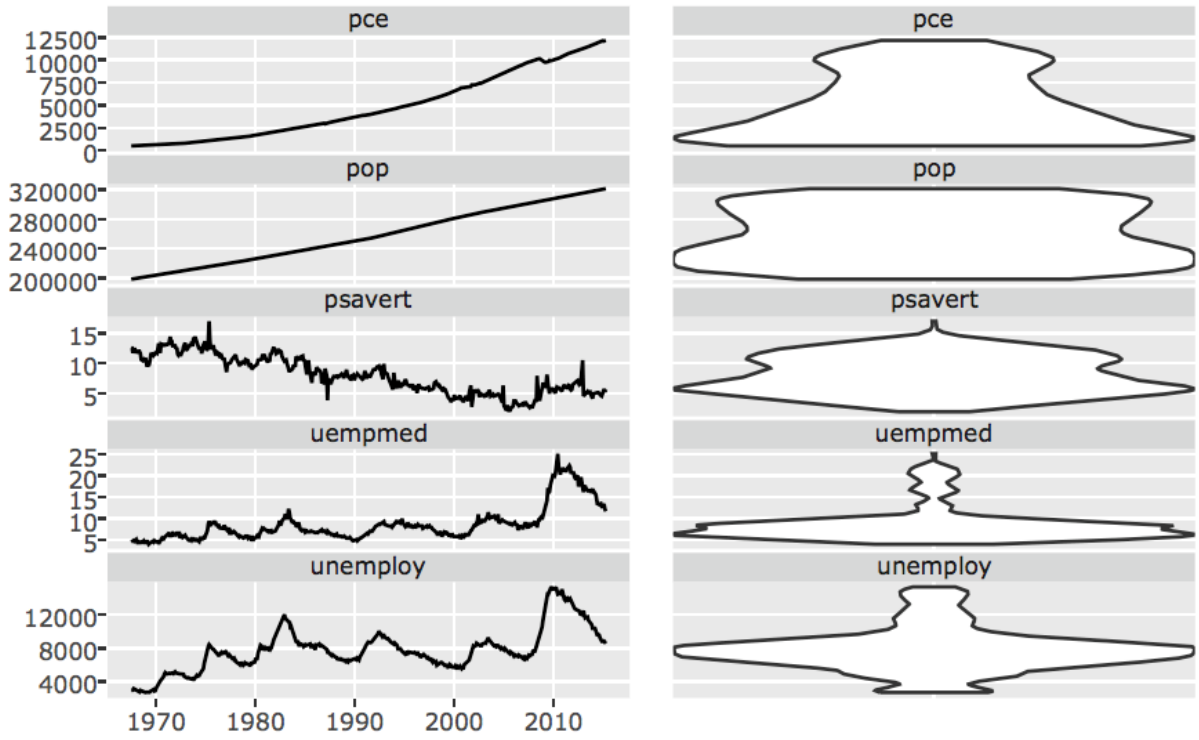


Figure 8.50 Arranging multiple faceted ggplot2 plots into a plotly subplot.

```
e <- tidyr::gather(economics, variable, value, -date)
gg1 <- ggplot(e, aes(date, value)) + geom_line() +
  facet_wrap(~variable, scales = "free_y", ncol = 1)
gg2 <- ggplot(e, aes(factor(1), value)) + geom_violin() +
  facet_wrap(~variable, scales = "free_y", ncol = 1) +
  theme(axis.text = element_blank(), axis.ticks = element_blank())
subplot(gg1, gg2) %>% layout(margin = list(l = 50))
```

8.3.3 Navigating many views

Sometimes you have to consider way more views than you can possibly digest visually. In Multiple linked views, we explore some useful techniques for implementing the popular visualization mantra from Shneiderman (1996):

"Overview first, zoom and filter, then details-on-demand."

In fact, Figure 8.54 from that section provides an example of this mantra put into practice. The correlation matrix provides an overview of the correlation structure between all the variables, and by clicking a cell, it populates a scatterplot between those two specific variables. This works fine with tens or hundreds or variables, but once you have thousands or tens-of-thousands of variables, this technique begins to fall apart. At that point, you may be better off defining a range of correlations that you're interested in exploring, or better yet, incorporating another measure (e.g., a test statistic), then focusing on views that match a certain criteria.

Tukey and Tukey (n.d.) first described the idea of using quantitative measurements of scatterplot characteristics (e.g. correlation) to help guide exploratory analysis of many variables. This idea, coined scagnostics (short for scatterplot diagnostics), has since been made explicit, and many measures have been explored, even measures specifically useful for time-series have been proposed (Wilkinson 2005); (Wilkinson and Wills 2008); (Dang and Wilkinson 2012). Probably the most universally useful scagnostic is the outlying measure which helps identify projections of the data space that contain outlying observations. Of course, the idea of associating quantitative measures with a graphical display of data can be generalized to include more that just scatterplots, and in this more general case, these measures are sometimes referred to as cognostics.

The same problems and principles that inspired scagnostics has inspired work on more general divide & recombine technique(s) for working with navigating through many statistical artifacts (Cleveland and Hafen 2014); (Saptarshi Guha and Cleveland 2012), including visualizations (Hafen et al. 2013). The **trelliscope** package provides a system for computing arbitrary cognostics on each panel of a trellis display as well as an interactive graphical user interface for defining (and navigating through) interesting panels based on those cognostics (Hafen 2016). This system also allows users to define the graph-

ical method for displaying each panel, so **plotly** graphs can easily be embedded. The **trelliscope** package is currently built upon **shiny**, but as Figure 8.51 demonstrates, the **trelliscopecore** package provides lower-level tools that allow one to create trelliscope displays without **shiny**.

```
library(trelliscopecore)
library(dplyr)
library(plotly)

# cognostics data frame
iris_cog_df <- iris %>%
  group_by(Species) %>%
  summarise(
    mean_sl = cog(mean(Sepal.Length), desc = "mean sepal length"),
    mean_sw = cog(mean(Sepal.Width), desc = "mean sepal length"),
    mean_pl = cog(mean(Petal.Length), desc = "mean sepal length"),
    mean_pw = cog(mean(Petal.Width), desc = "mean sepal length")
  )

iris_cog_df <- as_cognostics(iris_cog_df, cond_cols = "Species", key_col = "Species")

# list of panels
panels <- iris %>%
  split(iris$Species) %>%
  lapply(function(x) {
    plot_ly(x, x = ~Sepal.Length, y = ~Sepal.Width) %>% config(collaborate = FALSE)
  })
```

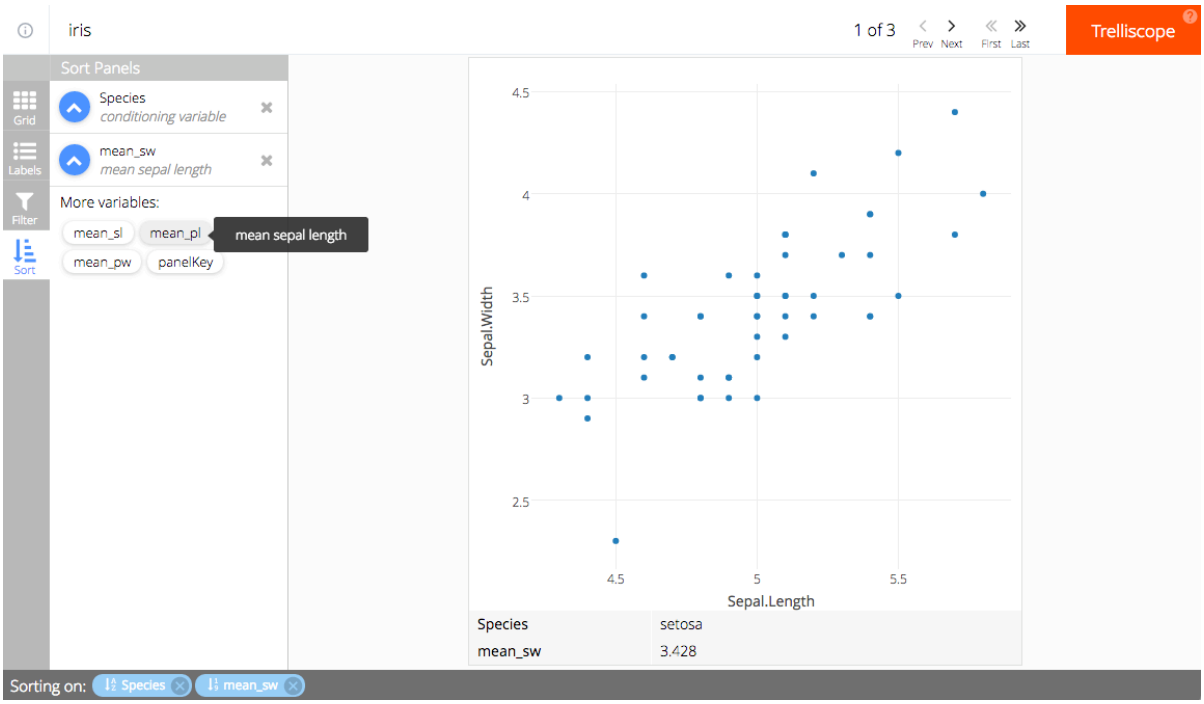


Figure 8.51 Using plotly within a trelliscope

```
base_path <- tempdir()


write_panels(panels, base_path = base_path, name = "iris")


write_display_obj(
  iris_cog_df,
  panel_example = panels[[1]],
  base_path = base_path,
  name = "iris"
)


prepare_display(base_path)


view_display(base_path)
```

8.4 Multiple linked views

8.4.1 Linking views with shiny

8.4.1.1 Accessing events in shiny

The plotly.js library emits custom events when a user interacts directly with a graph. The `event_data()` function provides a mechanism for accessing the data corresponding to those events within a shiny app. The shiny app in Figure 8.52 is designed to demonstrate the most useful plotly events one may access via `event_data()`: mouse hover (`"plotly_hover"`), click (`"plotly_click"`), and click+drag (`"plotly_selected"`). All of these events return selections on the data scale, not on a pixel scale, which is useful for updating views.

There are currently four different modes for click+drag interactions in plotly.js, but only two will trigger a `"plotly_selected"` event: rectangular and lasso selection. The other two drag modes, zoom and pan, both emit a `"plotly_relayout"` event which could be useful for say, providing global context in relation to a zoom event and/or recomputing a model based on new x/y limits. In Figure 8.52, the default click+drag mode was set to rectangular selection set via the dragmode attribute, but the mode can also be changed interactively via the mode bar at the top of the graph.

The video in Figure 8.52 helps demonstrate how different user events cause different blocks of code to be evaluated on the R server.[11] Conceptually, you can think of events as different inputs that becomes invalidated when the event is triggered by plotly.js. Moreover, similar to restrictions placed on references to input value(s) in shiny, `event_data()` has to be called *within* a reactive expressions. As RStudio's lesson on reactive expres-

[11] You can also run the example yourself using the following code – `shiny::runApp(system.file("examples", "plotlyEvents", package = "plotly"))`

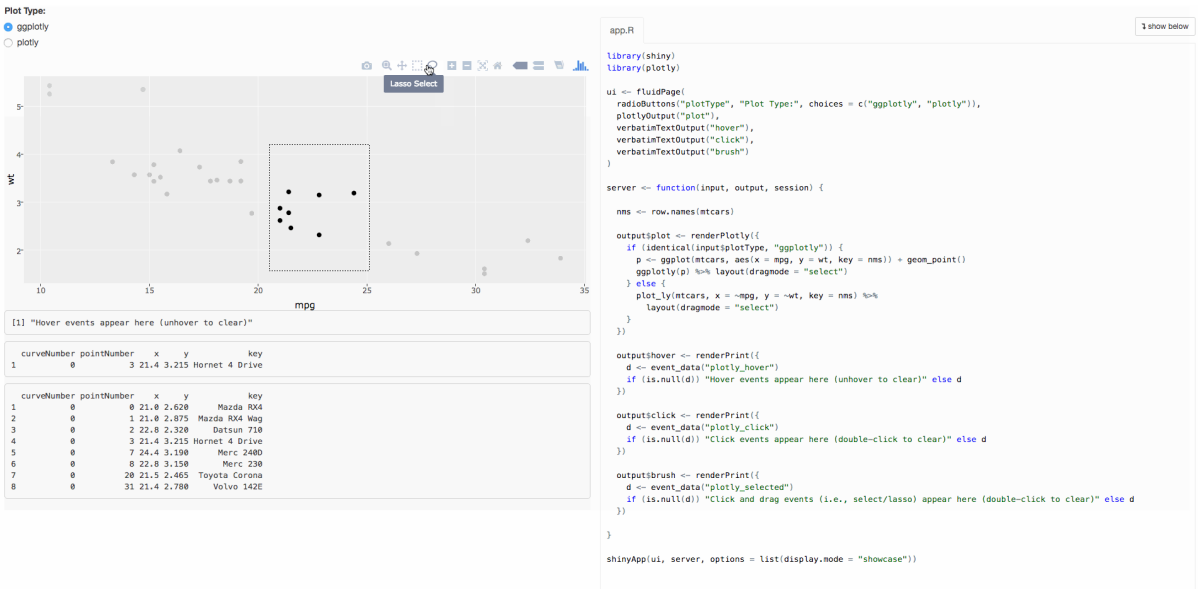


Figure 8.52 A video demonstration of plotly events in shiny. The video can be accessed here

sions points out: "A reactive expression is an R expression that uses widget input [(e.g., `event_data()`)] and returns a value."

Any of the `render*()` functions in **shiny** turn a regular R expression into a reactive expression. In Figure 8.52, every use of `event_data()` appears within `renderPrint()` since we only need to display the result of the event on the user interface with `verbatimTextOutput()`. In the next section, we use the return result of `event_data()` to display more interesting and informative views of user events.

8.4.1.2 Updating views

Obtaining data from a plotly event is easy, but updating view(s) based on the result of an event can be difficult. To start with something fairly easy, consider two scatterplots showing the same observations, but on different axes (i.e.., a subset of a scatterplot matrix). Figure 8.53 shows a linked lasso brush between two scatterplots. The main idea is that we first plot all the observations in black, then highlight the selection by adding

an additional layer of selected points in red using the data returned by `event_data()`. In order to guarantee that we can uniquely identify observations in the event data, it is also crucial that we attach a `key` attribute to each observation (here the rownames of the data), which we can then use to filter the original data down to the selected observations.

Figure 8.53 consciously updates the source of the selection (the top plot) to match the visual characteristics of the target (the bottom plot). In general, whenever linking views to display graphical selection(s), matching the visual characteristics of the selection both the source and target(s) can aide interpretation, especially when using interactive graphics to present results to others. Although the update rule in Figure 8.53 is to simply layer on additional points, a full redraw is performed during the update, which can impact performance when dealing with a large amount of graphical elements.

Figure 8.53 could be made slightly more efficient by just changing the color of selected points, or dimming the non-selected points, rather than plotting an extra layer of points. However, this technique does not work for chart types that display aggregate values (e.g., how do you dim non-selected values in a box plot?). For this reason, in Linking views without shiny, selections are implemented as an additional layer, but avoid the full redraw required when updating plot via **shiny** reactive framework.[12]

Since the update rule is the same for each view in Figure 8.53, we end up with a lot of redundant code that can be made more modular, as shown here. Making code more modular not only makes for less reading, but it leaves you less prone to making mistakes. Since the only difference between the two plots is the x/y variables, we can write a function that accepts x/y variables as input, and output a plotly object. Since this function outputs a plotly object, and is dependent upon `event_data()`, which can only be called within a reactive expression, this function can only be called within the `renderPlotly()` function in the **plotly** package.

[12] To my knowledge, the **leaflet** package is the only R package which provides a way to update a plot in a shiny app without a full redraw.

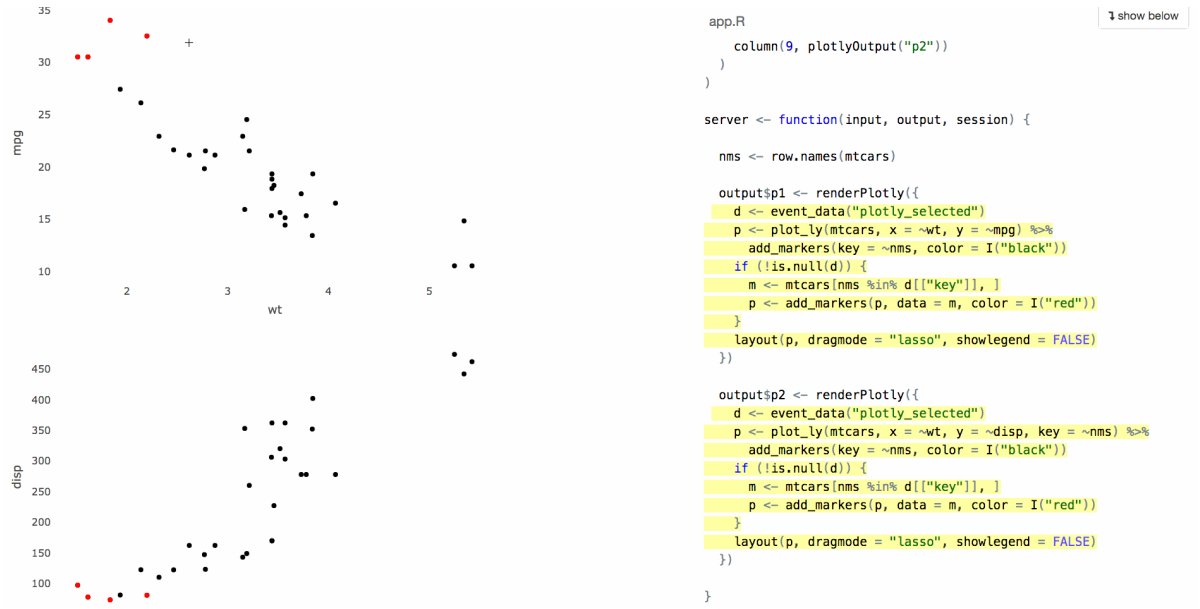


Figure 8.53 A video demonstration of linked brushing in a shiny app. The video can be accessed here and the code to run the example is here

8.4.1.3 Targeting views

The linked brushing example in Figure 8.53 has bi-directional communication – a `"plotly_selected"` event deriving from either view impacts the other view. In other words, each view can be either the source or target of the selection. Often times, we want *one* view to be the source of a selection, and related view(s) to be the target. Figure 8.54 shows a heatmap of a correlation matrix (the source of a selection) linked to a scatterplot (the target of a selection). By clicking on a cell in the correlation matrix, a scatterplot of the two variables is displayed below the matrix.

To update the scatterplot view, Figure 8.54 accesses `"plotly_click"` events via the `event_data()` function, but it also careful to not access click events triggered from the scatterplot. By strategically matching the value of the `source` argument in the `plot_ly()` and `event_data()` functions, Figure 8.54 effectively restricts the scope of events to a specific plot (the heatmap).

Figure 8.54 A video demonstration of clicking on a cell in a correlation matrix to view the corresponding scatterplot. The video can be accessed here and the code to run the example is here

Another aspect of Figure 8.54 that makes it an interesting example is that the `key` attribute is a matrix, matching the same dimensions of `z` (i.e, the values displayed in each cell). For good reason, most linked views paradigms (including the paradigm discussed in Linking views without shiny) restrict linkage definitions to relational database schema. In this case, it is more efficient to implement the relation with a key matrix, rather than a column.

8.4.2 Linking views without shiny

The code is this section is still under development and is likely to change. To run any of the code you see in this section, you'll need this developmental version of the package: `devtools::install_github("ropensci/plotly#554")`

8.4.2.1 Motivating examples

As shown in Linking views with shiny, the `key` attribute provides a way to attach a key (i.e., ID) to graphical elements – an essential feature when making graphical queries. When linking views in **plotly** outside of **shiny**, the suggested way to attach a key to graphical elements is via the `SharedData` class from the **crosstalk** package (Cheng 2015a). At the very least, the `new()` method for this class requires a data frame, and a key variable. Lets suppose we're interested in making comparisons of housing sales across cities for a given year using the `txhousing` dataset. Given that interest, we may want to make graphical queries that condition on a year, so we start by creating a `SharedData` object with `year` as the shared key.

```
# devtools::install_github("ropensci/crosstalk")
library(crosstalk)
sd <- SharedData$new(txhousing, ~year)
```

As far as `ggplotly()` and `plot_ly()` are concerned, `SharedData` object(s) act just like a data frame, but with a special `key` attribute attached to graphical elements. Since both interfaces are based on the layered grammar of graphics, `key` attributes can be attached at the layer level, and those attributes can also be shared across multiple views. Figure 8.55 leverages both of these features to link multiple views of median house sales in various Texan cities. As the video shows, hovering over a line in any panel selects that particular year, and all corresponding panels update to highlight that year. The result is an incredibly powerful tool for quickly comparing house sale prices, not only across cities for a given year, but also across years for a given city.

```
p <- ggplot(sd, aes(month, median)) +
  geom_line(aes(group = year)) +
  geom_smooth(data = txhousing, method = "gam") +
```

Figure 8.55 Monthly median house sales by year and city. Each panel represents a city and panels are linked by year. A video demonstrating the graphical queries can be viewed here

```
  facet_wrap(~ city)


ggplotly(p, tooltip = "year") %>%
  highlight(on = "plotly_hover", defaultValues = 2015, color = "red")
```

Figure 8.55 uses the `highlight()` function from the **plotly** package to specify the type of plotly event for triggering a selection (via the `on` argument), the color of the selection (via the `color` argument), and set a default selection (via the `defaultValues` argument). The `off` argument controls the type of event that clears selections, and by default, is set to a `plotly_relayout` event, which can be triggered by clicking the home icon in the mode bar (or via zoom/pan). The `highlight()` function can also be used to control Transient versus persistent selection modes, and dynamically control selection colors, which is very useful for making comparisons.

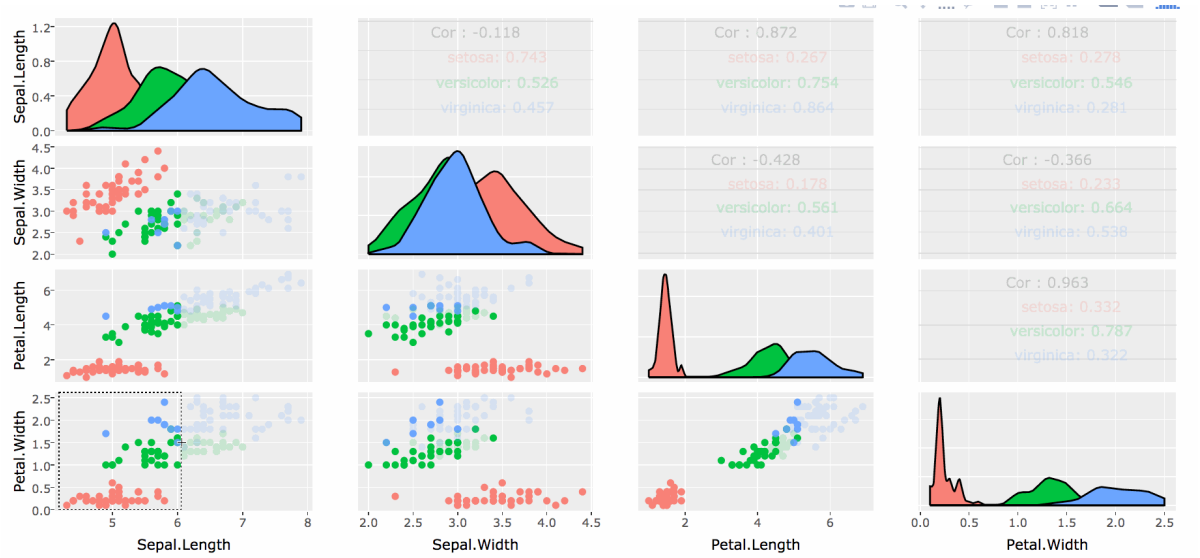


Figure 8.56 Brushing a scatterplot matrix via the `ggpairs()` function in the **GGally** package. A video demonstrating the graphical queries can be viewed here

Figure 8.56 shows another example of using `SharedData` objects to link multiple views, this time to enable linked brushing in a scatterplot matrix via the `ggpairs()` function from the **GGally** package. As discussed in Scatterplot matrices, the `ggpairs()` function implements the generalized pairs plot – a generalization of the scatterplot matrix – an incredibly useful tool for exploratory data analysis. Since the `Species` variable (as discrete variable) is mapped to color in Figure 8.56, we can inspect both correlations, and marginal densities, dependent upon Species type. By adding the brushing capabilities via `ggplotly()`, we add the ability to examine the dependence between a continuous conditional distribution and other variables. For this type of interaction, a unique key should be attached to each observation in the original data, which is the default behavior of the `SharedData` object's `new()` method when no key is provided.

```
d <- SharedData$new(iris)
p <- GGally::ggpairs(d, aes(color = Species), columns = 1:4)
layout(ggplotly(p), dragmode = "select")
```

When the graphical query is made is 8.56, the marginal densities do not update. This points out one of the weaknesses of implementing multiple linked views without shiny (or some other R backend). The browser knows nothing about the algorithm **GGally** (or **ggplot2**) uses to compute a density, so updating the densities in a consistent way is not realistic without being able to call R from the browser. It is true that we could try to precompute densities for every possible selection state, but this does not generally scale well when the number of selection states is large, even as large as Figure 8.56. As discussed briefly in bars & histograms, Boxplots, and 2D distributions, plotly.js does have some statistical functionality that we can leverage to display Dynamic aggregates, but this currently covers only a few types of statistical displays.

8.4.2.2 Transient versus persistent selection

The examples in the previous section use transient selection, meaning that when a value is selected, previous selection(s) are forgotten. Sometimes it is more useful to allow selections to accumulate – a type of selection known as persistent selection. To demonstrate the difference, Figure 8.57 presents two different takes a single view, one with transient selection (on the left) and one with persistent selection (on the right). Both selection modes can be used when linking multiple views, but as Figure 8.57 shows, highlighting graphical elements, even in a single view, can be useful tool to avoid overplotting.

```
sd <- SharedData$new(txhousing, ~city)
p <- ggplot(sd, aes(date, median)) + geom_line()
gg <- ggplotly(p, tooltip = "city")

highlight(gg, on = "plotly_hover", dynamic = TRUE)
highlight(gg, on = "plotly_hover", dynamic = TRUE, persistent = TRUE)
```

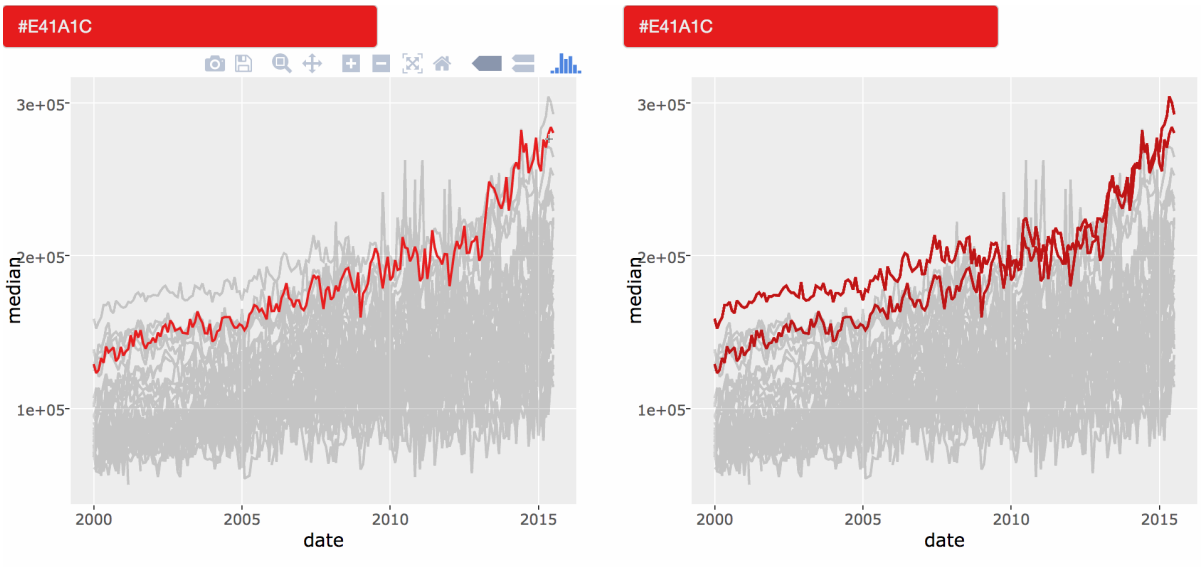


Figure 8.57 Highlighting lines with transient versus persistent selection. In the left hand panel, transient selection (the default); and in the right hand panel, persistent selection. The video may be accessed here

Figure 8.57 also sets the `dynamic` argument to `TRUE` to populate a widget, powered by the **colourpicker** package (Attali 2016), for dynamically altering the selection color. When paired with persistent selection, this makes for a powerful tool for making comparisons between two selection sets. However, for Figure 8.57, transient selection is probably the better mode for an initial look at the data (to help reveal any structure in missingness or anomalies for a given city), whereas persistent selection is better for making comparisons once have a better idea of what cities might be interesting to compare.

8.4.2.3 Linking with other htmlwidgets

Perhaps the most exciting thing about building a linked views framework on top of the **crosstalk** package is that it provides a standardized protocol for working with selections that other **htmlwidget** packages may build upon. If implemented carefully, this effectively provides a way to link views between two independent graphical systems – a fairly foreign technique within the realm of interactive statistical graphics. This grants

a tremendous amount of power to the analyst since she/he may leverage the strengths of multiple systems in a single linked views analysis. Figure 8.58 shows an example of linked views between plotly and leaflet for exploring the relationship between the magnitude and geographic location of earthquakes.

```
library(plotly)
library(leaflet)

sd <- SharedData$new(quakes)

p <- plot_ly(sd, x = ~depth, y = ~mag) %>%
  add_markers(alpha = 0.5) %>%
  layout(dragmode = "select") %>%
  highlight(dynamic = TRUE, persistent = TRUE)

map <- leaflet(sd) %>%
  addTiles() %>%
  addCircles()

htmltools::tagList(p, map)
```

In Figure 8.58, the user first highlights earthquakes with a magnitude of 5 or higher in red (via plotly), then earthquakes with a magnitude of 4.5 or lower, and the corresponding earthquakes are highlighted in the leaflet map. This immediately reveals an interesting relationship in magnitude and geographic location, and leaflet provides the ability to zoom and pan on the map to investigate regions that have a high density of quakes. It's worth noting that the **crosstalk** package itself does not provide semantics for describing

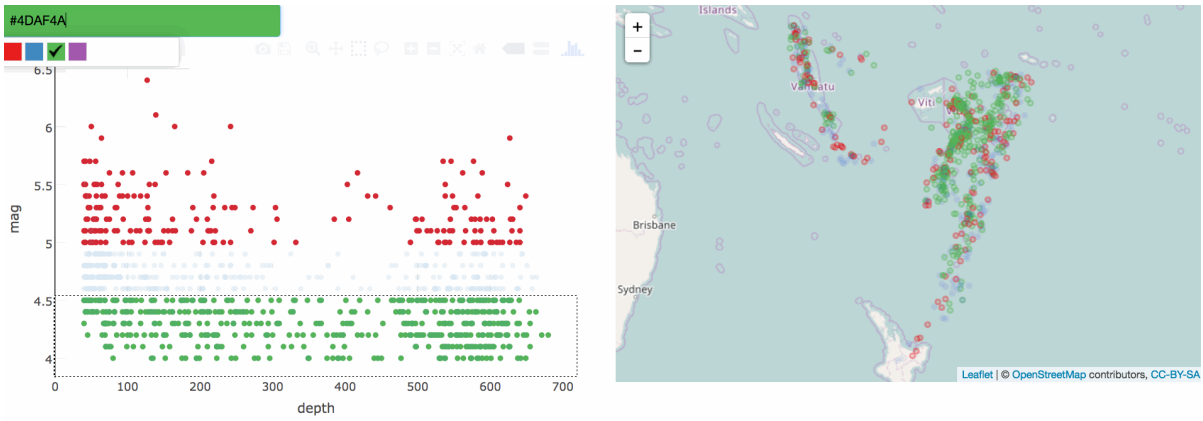


Figure 8.58 Linking views between plotly and leaflet to explore the relation between magnitude and geographic location of earthquakes around Fiji. The video may be accessed here

persistent/dynamic selections, but **plotly** does inform crosstalk about these semantics which other **htmlwidget** authors can access in their JavaScript rendering logic.

8.4.2.4 Selection via indirect manipulation

The interactions described thus far in Linking views without shiny is what Cook and Swayne (2007) calls direct manipulation, where the user makes graphical queries by directly interacting with graphical elements. In Figure 8.59, cities are queried indirectly via a dropdown powered by the selectize.js library (B. R. & Contributors 2016). Indirect manipulation is especially useful when you have unit(s) of interest (e.g. your favorite city), but can not easily find that unit in the graphical space. The combination of direct and indirect manipulation is powerful, especially when the interactive widgets for indirect manipulation are synced with direct manipulation events. As shown in Figure 8.59, when cities are queried indirectly, the graph updates accordingly, and when cities are queried directly, the select box updates accordingly. If the time series was linked to other view(s), as it is in the next section, selecting a city via the dropdown menu would highlight all of the relevant view(s).

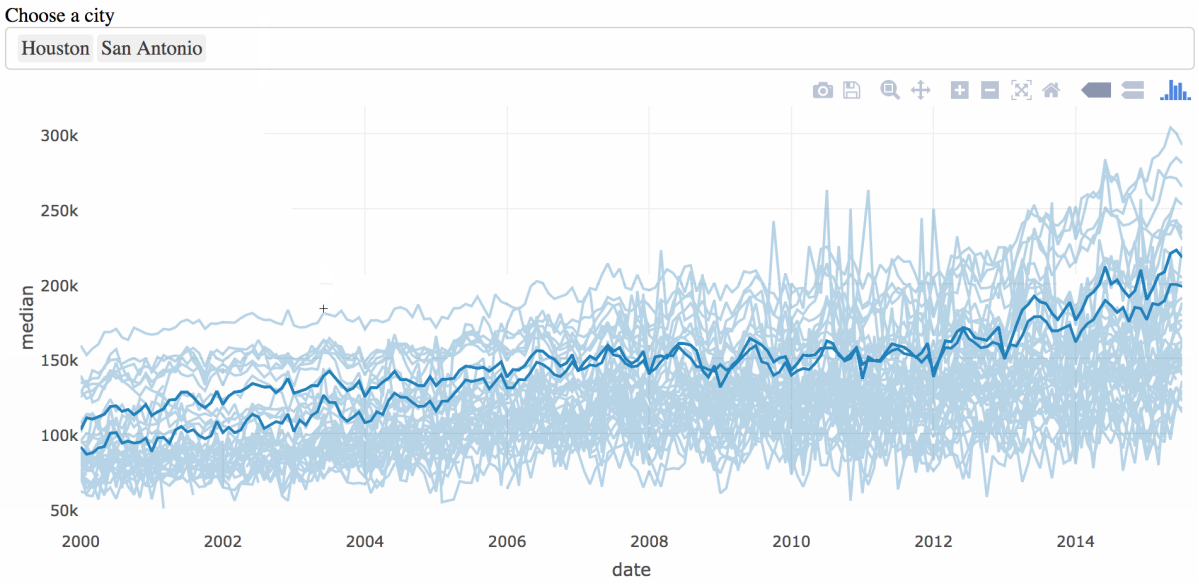


Figure 8.59 Selecting cities by indirect manipulation. The video may be accessed here

```
# The group name is currently used to populate a title for the selectize widget
sd <- SharedData$new(txhousing, ~city, "Choose a city")
plot_ly(sd, x = ~date, y = ~median) %>%
  add_lines(text = ~city, hoverinfo = "text") %>%
  highlight(on = "plotly_hover", persistent = TRUE, selectize = TRUE)
```

8.4.2.5 The SharedData plot pipeline

Sometimes it is useful to display a summary (i.e., overview) in one view and link that summary to more detailed views. Figure 8.60 is one such example that displays a bar chart of all Texan cities with one or more missing values (the summary) linked with their values over time (the details). By default, the bar chart allows us to quickly see which cities have the most missing values, and by clicking a specific bar, it reveals the relationship between missing values and time for a given city. In cities with the most missing values, data did not start appearing until somewhere around 2006-2010, but

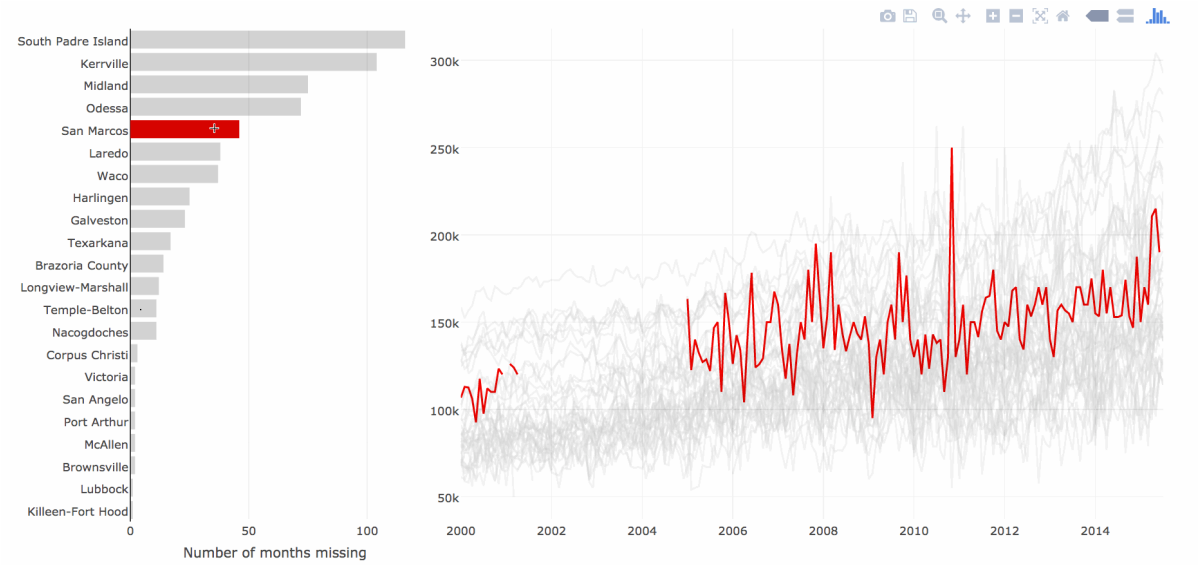


Figure 8.60 A bar chart of cities with one or more missing median house sales linked to a time series of those sales over time. The video may be accessed here

for most other cities (e.g., Harlingen, Galveston, Temple-Belton, etc), values started appearing in 2000, but for some reason go missing around 2002-2003.

When implementing linked views like Figure 8.60, it can be helpful to conceptualize a pipeline between a central data frame and the corresponding views. Figure 8.61 is a visual depiction of this conceptual model between the central data frame and the eventual linked views in Figure 8.60. In order to generate the bar chart on the left, the pipeline contains a function for computing summary statistics (the number of missing values per city). On the other hand, the time series does not require any summarization – implying the pipeline for this view is the identity function.

Since the pipeline from data to graphic is either an identity function or a summarization of some kind, it is good idea to use the most granular form of the data for the `SharedData` object, and use the data-plot-pipeline to define a pipeline from the data to the plot. As Wickham et al. (2010) writes, a true interactive graphics system is aware of the both the function from the central data object to the graphic, as well as the inverse function (i.e., the function from the graphic back to the central data object). As it currently

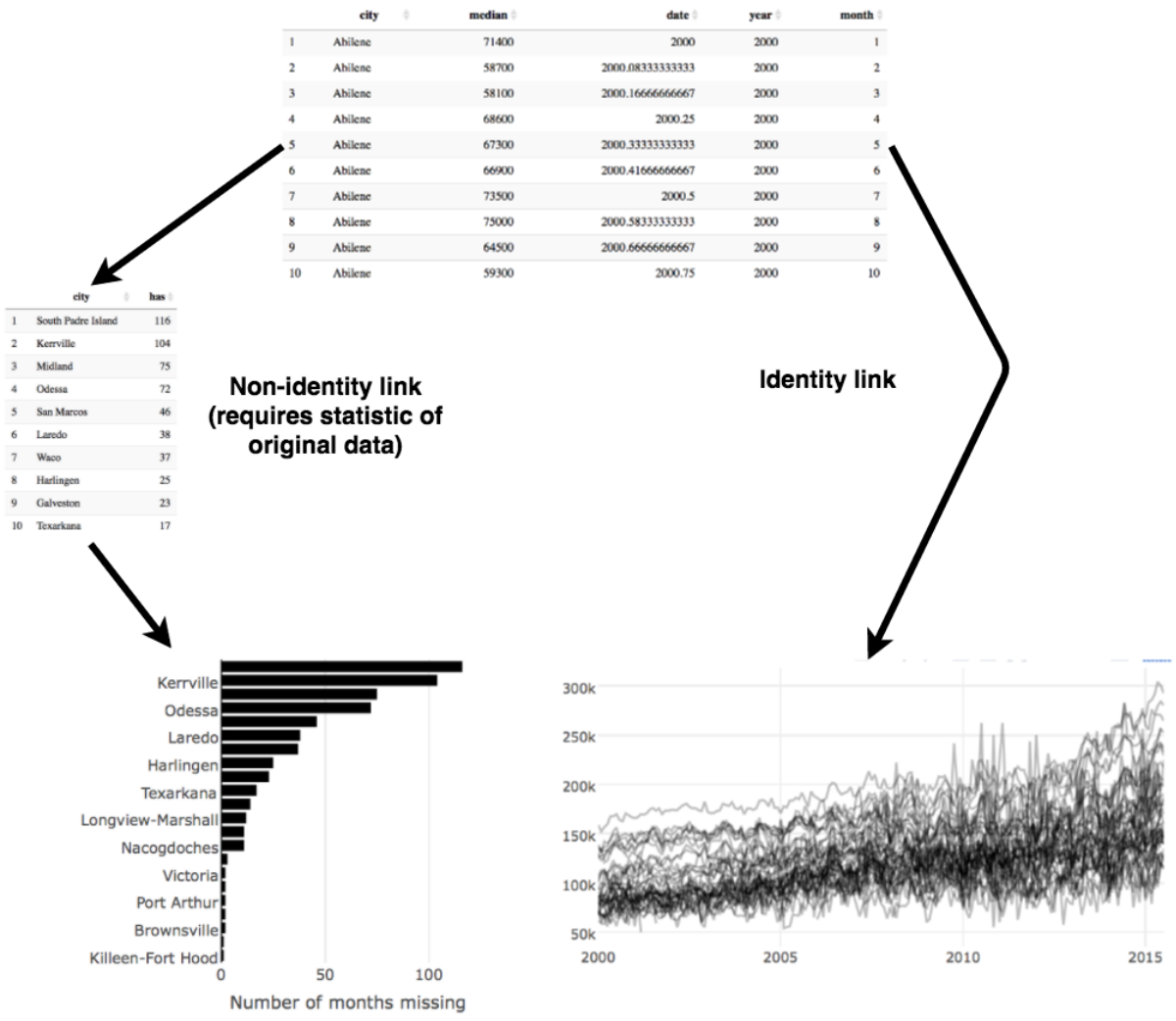


Figure 8.61 A diagram of the pipeline between the data and graphics.

stands, **plotly** loses this information when the result is pushed to the web browser, but that does not matter for Figure 8.60 since pipelines do not have to re-execute upon user selections.[13]

```
sd <- SharedData$new(txhousing, ~city)

base <- plot_ly(sd, color = I("black")) %>%
  group_by(city)

p1 <- base %>%
  summarise(has = sum(is.na(median))) %>%
  filter(has > 0) %>%
  arrange(has) %>%
  add_bars(x = ~has, y = ~factor(city, levels = city), hoverinfo = "none") %>%
  layout(
    barmode = "overlay",
    xaxis = list(title = "Number of months missing"),
    yaxis = list(title = "")
  )

p2 <- base %>%
  add_lines(x = ~date, y = ~median, alpha = 0.3) %>%
  layout(xaxis = list(title = ""))

subplot(p1, p2, titleX = TRUE, widths = c(0.3, 0.7)) %>%
  layout(margin = list(l = 120)) %>%
```

[13]Since **dplyr** semantics translate to `SQL` primitives, you could imagine a system that translates a data-plot-pipeline to `SQL` queries, and dynamically re-executes within the browser via something like `SQL.js` (O. L. & Contributors 2016).

```
  highlight(on = "plotly_click", off = "plotly_unhover", color = "red")
```

8.4.2.6 Dynamic aggregates

As discussed in the plotly cookbook, there are a number of way to compute statistical summaries in the browser via plotly.js (e.g., `add_histogram()`, `add_boxplot()`, `add_histogram2d()`, and `add_histogram2dcontour()`). When linking views with the **plotly** package, we can take advantage of this functionality to display aggregated views of selections. Figure 8.62 shows a basic example of brushing a scatterplot to select cars with 10-20 miles per gallon, then a 5 number summary of the corresponding engine displacement is dynamically computed and displayed as a boxplot.

```
d <- SharedData$new(mtcars)
scatterplot <- plot_ly(d, x = ~mpg, y = ~disp) %>%
  add_markers(color = I("black")) %>%
  layout(dragmode = "select")

subplot(
  plot_ly(d, y = ~disp, color = I("black")) %>% add_boxplot(name = "overall"),
  scatterplot, shareY = TRUE
) %>% layout(dragmode = "select")
```

Figure 8.62 is very similar to Figure 8.63, but uses `add_histogram()` to link to a bar chart of the number of cylinders rather than a boxplot of engine displacement. By brushing to select cars with a low engine displacement, we can see that (obviously) displacement is related with to the number of cylinders.

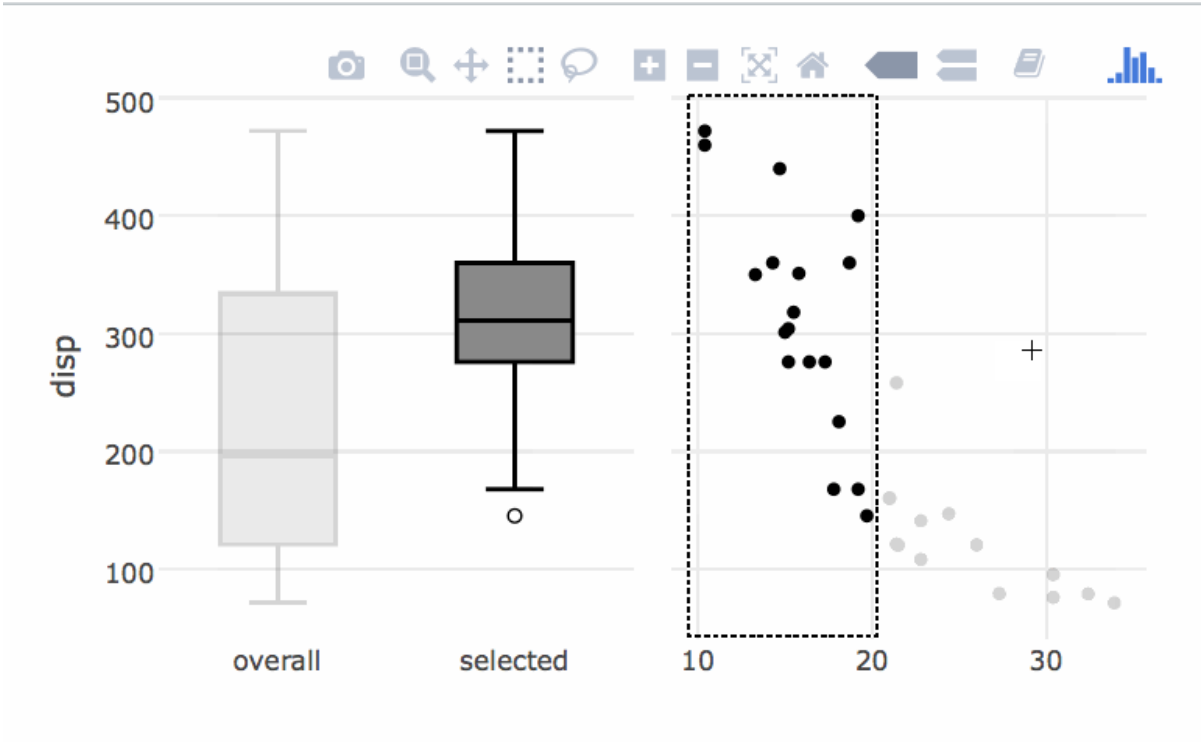


Figure 8.62 Dynamically populating a boxplot reflecting brushed observations

```
p <- subplot(
  plot_ly(d, x = ~factor(vs)) %>% add_histogram(color = I("black")),
  scatterplot
)

# Selections are actually additional traces, and, by default,
# plotly.js will try to dodge bars placed under the same category
layout(p, barmode = "overlay")
```

8.4.2.7 Hierarchial categorical selection

All of the sections under linking views without shiny thus far have attached a *single value* to a graphical mark or group of marks. However, there are also situations where it

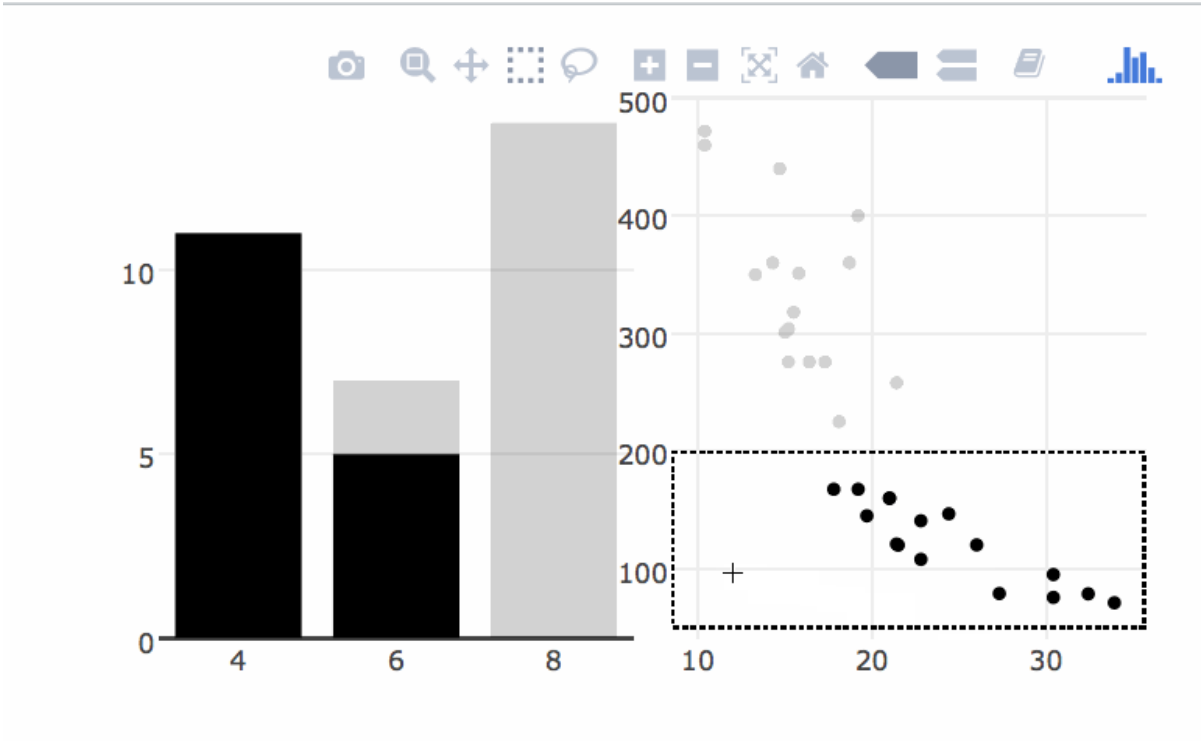


Figure 8.63 Dynamically populating a bar chart reflecting brushed observations

makes sense to attach multiple values to mark(s). Instead of using multiple keys to assign multiple values, it is actually more powerful to use a single *nested* key, since multiple keys can not support a different number of values for each mark. For example, suppose I have 4 (x, y) pairs, and each pair is associated with a different set of categorical values:

```
# data frames do support list columns,
# but tibble::tibble() provides a nicer API for this...
d <- data.frame(x = 1:4, y = 1:4)
d$key <- lapply(1:4, function(x) letters[seq_len(x)])
d
#>   x y        key
#> 1 1 1          a
#> 2 2 2       a, b
#> 3 3 3    a, b, c
```

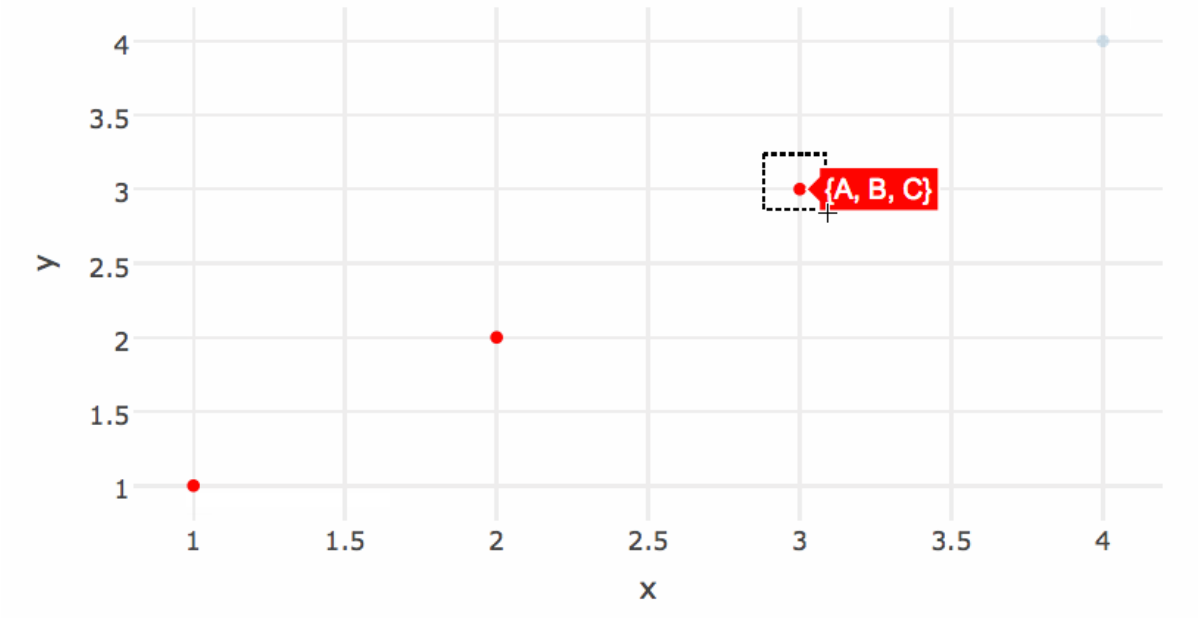


Figure 8.64 A simple example of hierarchial selection

```
#> 4 4 4 a, b, c, d
```

Suppose point (3, 3) is selected – implying the set $\{a,b,c\}$ is of interest – what key sets should be considered a match? The most sensible approach is to match any subsets of the selected set, so for example, this key set would match the sets $\{a\}$, $\{b\}$, $\{c\}$, $\{a,b\}$, $\{a,c\}$, and $\{b,c\}$. This leads to a type of selection I'll refer to as *hierarchical* categorical selection. Figure 8.64 provides a visual demo of this example in action:

```
plot_ly(SharedData$new(d, ~z), x = ~x, y = ~y)
```

More informally, hierarchical categorical selection provides a nice way to select all the "children" of a given "parent" selection. This type of selection is useful when their is a natural hierarchy to a dataset, such as for a dendrogram, and 8.65 demonstrates:

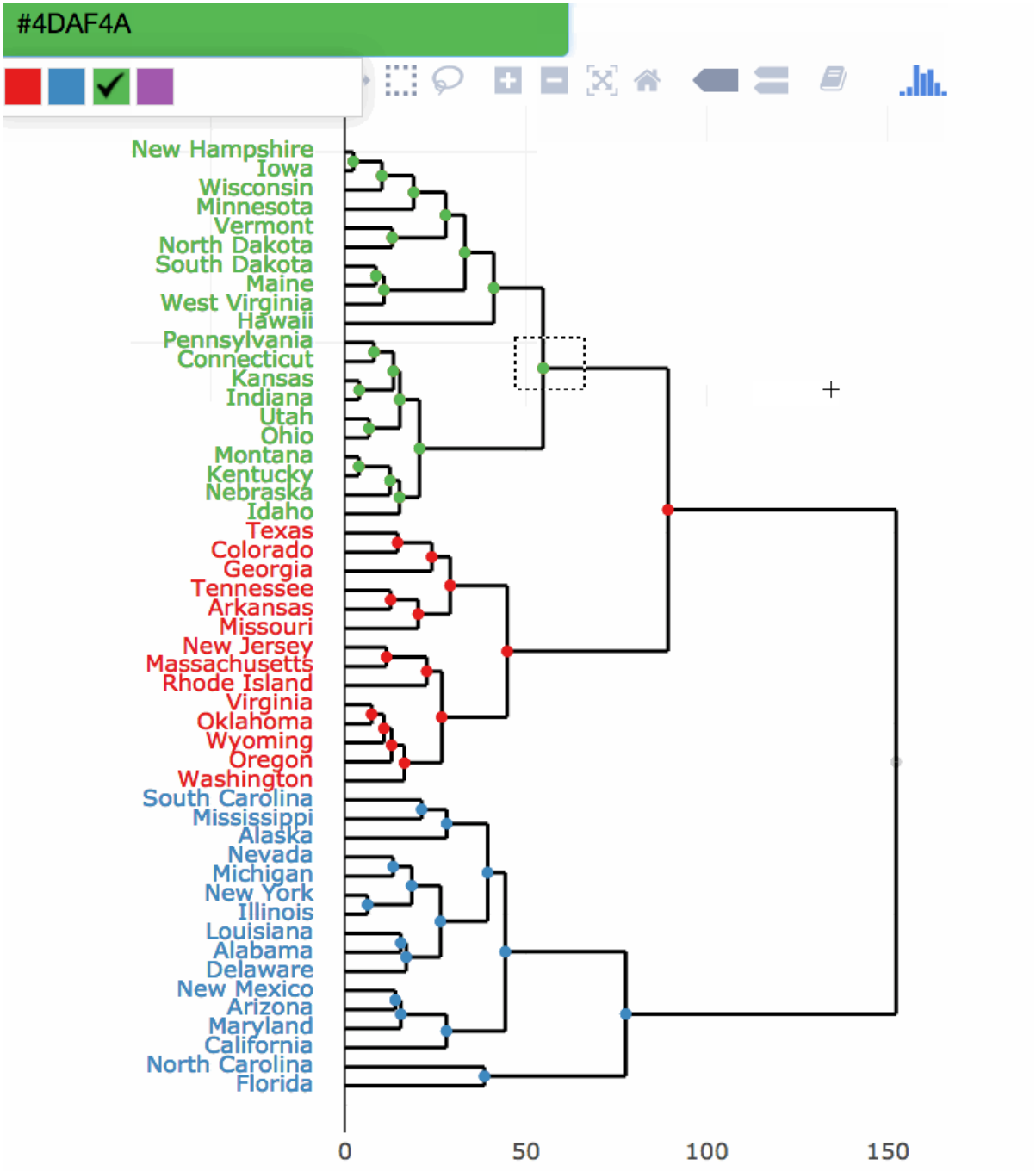


Figure 8.65 Leveraging hierarchical selection and persistent brushing to paint branches of a dendrogram.

8.4.2.8 More examples

The **plotly** package bundles a bunch of demos that illustrate all the options available when linking views without **shiny** via **crosstalk**'s `SharedData` class and **plotly**'s `highlight()` function. To list all the examples, enter `demo(package = "plotly")` into your R prompt, and pick a topic (e.g., `demo("highlight-intro", package = "plotly")`).

8.5 Advanced topics

This section describes some advanced topics regarding the **plotly** package. Some of the content found here may be useful for the following people:

- R users that know some JavaScript and want to enable custom features that **plotly** and plotly.js does not natively support.
- R developers that have authored a custom **ggplot2** geom and want to inform `ggplotly()` about the rendering rules of their geom.
- R developers that want to build a similar interface to another Javascript graphing library.

8.5.1 Custom behavior via JavaScript

The section on linking views with shiny shows how to acquire data tied to plotly.js events from a **shiny** app. Since **shiny** adds a lot of additional infrastructure, **plotly** also provides a way to link views without shiny, but this definitely does not encompass every type of interactivity. Thankfully the **htmlwidgets** package provides a way to invoke a JavaScript function on the widget element (after it is done rendering) from R via the `onRender()` function (Vaidyanathan et al. 2015). The JavaScript function

should have at least two arguments: (1) the DOM element containing the htmlwidget (`el`) and (2) the data passed from R (`x`). This enables, for instance, the ability to author custom behavior tied to a particular plotly.js event. Figure 8.66 uses `onRender()` to open a relevant google search upon clicking a point.

```
library(plotly)
library(htmlwidgets)
search <- paste0("http://google.com/#q=", curl::curl_escape(rownames(mtcars)))
plot_ly(mtcars, x = ~wt, y = ~mpg, color = ~factor(vs)) %>%
  add_markers(text = ~search, hoverinfo = "x+y") %>%
  onRender("
    function(el, x) {
      el.on('plotly_click', function(d) {
        // d.points is an array of objects which, in this case,
        // is length 1 since the click is tied to 1 point.
        var pt = d.points[0];
        var url = pt.data.text[pt.pointNumber];
        // DISCLAIMER: this won't work from RStudio
        window.open(url);
      });
    }
  ")
```

8.5.2 Translating custom ggplot2 geoms

Version 2.0.0 of **ggplot2** introduced a way for other R packages to implement custom geoms. Some great examples include: **ggrepel**, **ggalt**, **ggraph**, **geomnet**, **ggmosaic** and **ggtern** (Rudis 2016); (Pedersen 2016); (Tyner and Hofmann 2016); (Jeppson, Hofmann,

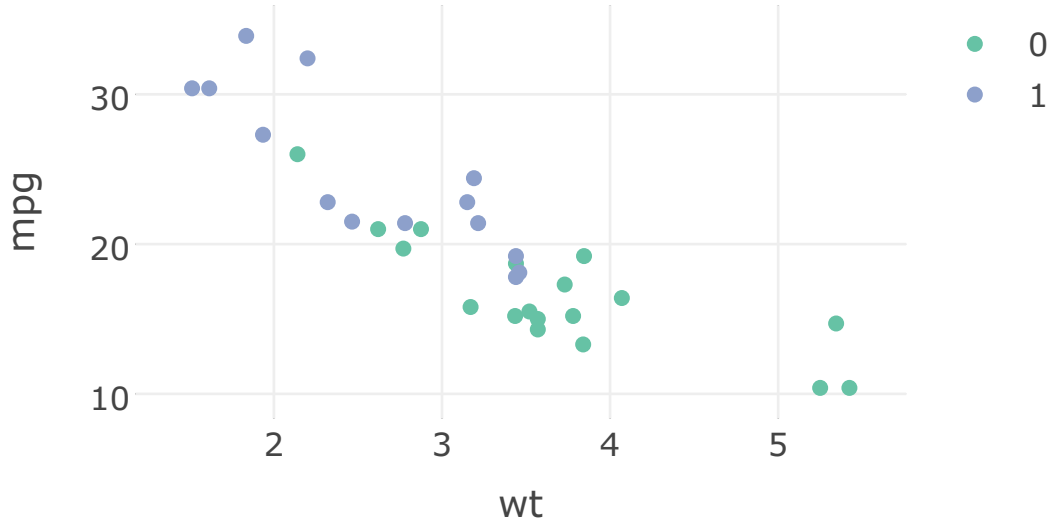


Figure 8.66 Using `onRender()` to register a JavaScript callback that opens a google search upon a 'plotly_click' event.

and Cook, n.d.); (Hamilton 2016).[14] Although the `ggplotly()` function translates most of the geoms bundled with the **ggplot2** package, it has no way of knowing about the rendering rules for custom geoms. The **plotly** package does, however, provide 2 generic functions based on the S3 scheme that can leveraged to inform `ggplotly()` about these rules (Chambers 1992).[15] To date, the **ggmosaic** and **ggalt** packages have taken advantage of this infrastructure to provide translations of their custom geoms to plotly.

In **ggplot2**, many geoms are special cases of other geoms. For example, `geom_line()` is equivalent to `geom_path()` once the data is sorted by the x variable. For cases like this, when a geom can be reduced to another lower-level (i.e., basic) geom, authors just have to write a method for the `to_basic()` generic function in **plotly**. In fact, within the package itself, the `to_basic()` function has a `GeomLine` method which simply sorts the data by the x variable then returns it with a class of `GeomPath` prefixed.

[14] There are many other useful extension packages that are listed on this website – https://www.ggplot2-exts.org

[15] For those new to S3, http://adv-r.had.co.nz/S3.html provides an approachable introduction and overview (Wickham 2014a).

```
getS3method("to_basic", "GeomLine")
#> function (data, prestats_data, layout, params, p, ...)
#> {
#>     data <- data[order(data[["x"]]), ]
#>     prefix_class(data, "GeomPath")
#> }
#> <environment: namespace:plotly>
```

If you have implemented a custom geom, say `GeomCustom`, rest assured that the data passed to `to_basic()` will be of class `GeomCustom` when `ggplotly()` is called on a plot with your geom. And assuming `GeomCustom` may be reduced to another lower-level geom support by plotly, a `to_basic.GeomCustom()` method that transforms the data into a form suitable for that lower-level geom is sufficient for adding support. Moreover, note that the data passed to `to_basic()` is essentially the last form of the data *before* the render stage and *after* statistics have been performed. This makes it trivial to add support for geoms like `GeomXspline` from the **ggalt** package.

```
# devtools::install_github("hrbrmstr/ggalt")
library(ggalt)
getS3method("to_basic", "GeomXspline")
#> function (data, prestats_data, layout, params, p, ...)
#> {
#>     data <- data[order(data[["x"]]), ]
#>     prefix_class(data, "GeomPath")
#> }
#> <environment: namespace:plotly>
```

As shown in Figure 8.67, once the conversion has been provided. Users can call `ggplotly()` on the ggplot object containing the custom geom just like any other ggplot object.

```
# example from `help(geom_xspline)`
set.seed(1492)
dat <- data.frame(
  x = c(1:10, 1:10, 1:10),
  y = c(sample(15:30, 10), 2 * sample(15:30, 10), 3 * sample(15:30, 10)),
  group = factor(c(rep(1, 10), rep(2, 10), rep(3, 10)))
)
p <- ggplot(dat, aes(x, y, group = group, color = factor(group))) +
  geom_point(color = "black") +
  geom_smooth(se = FALSE, linetype = "dashed", size = 0.5) +
  geom_xspline(spline_shape = 1, size = 0.5)
#> Error: GeomXspline was built with an incompatible version of ggproto.
#> Please reinstall the package that provides this extension.
ggplotly(p) %>% hide_legend()
```

In more complicated cases, where your custom geom can not be converted to a lower level geom, a custom method for the `geom2trace()` generic is required (`methods(geom2trace)` lists all the basic geoms that we natively support). This method should involve a conversion from a data frame to a list-like object conforming to the plotly.js figure reference.

8.5.3 Designing an htmlwidget interface

The plotly.js library, as with many other JavaScript graphing libraries, strives to describe any plot through a plot specification defined via JavaScript Object Notation (JSON).

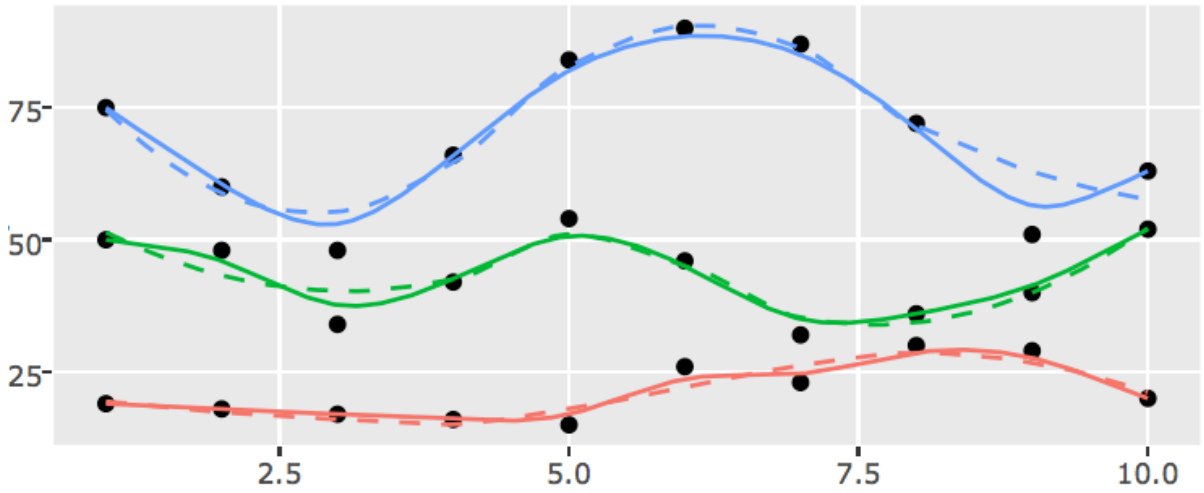


Figure 8.67 Converting GeomXspline from the **ggalt** package to plotly.js via `ggplotly()`.

JSON is a language independent data-interchange format that was originally designed for JavaScript, but parsers for many different languages now exist, including R (Temple Lang 2014b); (Ooms 2014a). JSON is a recursive key-value data structure (similar to a list in R), and essentially any valid JavaScript value has a natural R equivalent (e.g., `NULL`/`null`). As a result, any JSON object can be created from an appropriate R list, meaning that theoretically any plotly.js plot can be described via an R list. However, simply providing a bridge between R lists and JSON does not guarantee a powerful or usable interface, especially for a general purpose graphing library.

Although it can be complicated to implement, R interfaces to JavaScript graphing libraries should leverage R's strong resources for computing on the language to design a more expressive interface (Wickham 2014a). It should also look and feel like (and work well with!) other commonly used interfaces in R. A good way to do this is to embrace (pure and predictable) functional programming. Most importantly, this implies that every function *modifies* a central type of object – meaning that every function input and output the same type of object (predictable). Furthermore, if the output of a function can be determined completely by the input (i.e., pure), it removes any need to search

for other code that may be affecting the output. In the case of providing an interface to a JavaScript graphing library, there are a number of reasons why the central object should inherit from the central object provided by the **htmlwidgets** package.

The idea of interfacing R with JavaScript libraries via JSON data transfer has been popular approach for quite some time (Vaidyanathan 2013); (Hocking, VanderPlas, and Sievert 2015); (Sievert and Shirley 2014). The R package **htmlwidgets** standardized this bridge, and provides some additional infrastructure for making sure the HTML output works as expected in multiple contexts (in the R console or RStudio, within **rmarkdown** documents, and even embedded inside **shiny** apps). The **htmlwidgets** package itself is opininated about the data structure used to represent the widget in R since it needs to retain meta-information about the widget, such as the sizing policy. To avoid surprise, widget authors should strive to have all functions in their interface modify this data structure.[16]

JavaScript graping libraries usually have strong requirements about the JSON structure used to create a plot. In some cases, the R interface needs to know about these requirements in order to faithfully translate R objects to JSON. For example, in plotly.js some attributes must *always* be an array (e.g. x/y), even if they are length 1, while other attributes cannot be an array must be a literal constant (e.g. name). This leads to a situation where the translation rules from R to JSON cannot be simply "box all vectors of length 1 into an array (or not)":

```
list(x = 1, y = 1, name = "A point") => {x: [1], y: [1], name: "A point"}
```

Thankfully plotly.js provides a plot schema which declares types for each attribute that **plotly** leverages internally. At print time, it ensures each attribute is of the correct type,

[16]The **plotly** package initially fought this advice and represented plotly objects using a special data frame with a special print method to enable the data-plot-pipeline. I have since changed my mind and decided special methods for popular generic functions should be provided instead.

and also searches for any unsupported attributes that may have been specified by the user (and throws a warning that the attribute will be ignored).

9 Impact and Future Work

9.1 Impact

9.1.1 plotly

At the time of writing, **plotly** is the most widely used R package for interactive web graphics according to RStudio's anonymized CRAN mirror download logs. Figure 9.1 shows CRAN downloads for the past 6 months among the leading R packages for interactive web graphics. The recent spike in CRAN downloads was due to a major update in **plotly** which introduced a lot of new features.

As shown in Figure 9.2, **plotly** also enjoys the most GitHub stars among all R packages built on the **htmlwidgets** framework. Github stars provide a mechanism for users of open-source software to indicate their interest in a project on the worlds largest repository of software. Recently, it has started being used to study popularity in open-source projects (Hudson Borges 2016).

9.1.2 LDAvis

Github stars and python port?

9.2 Future work

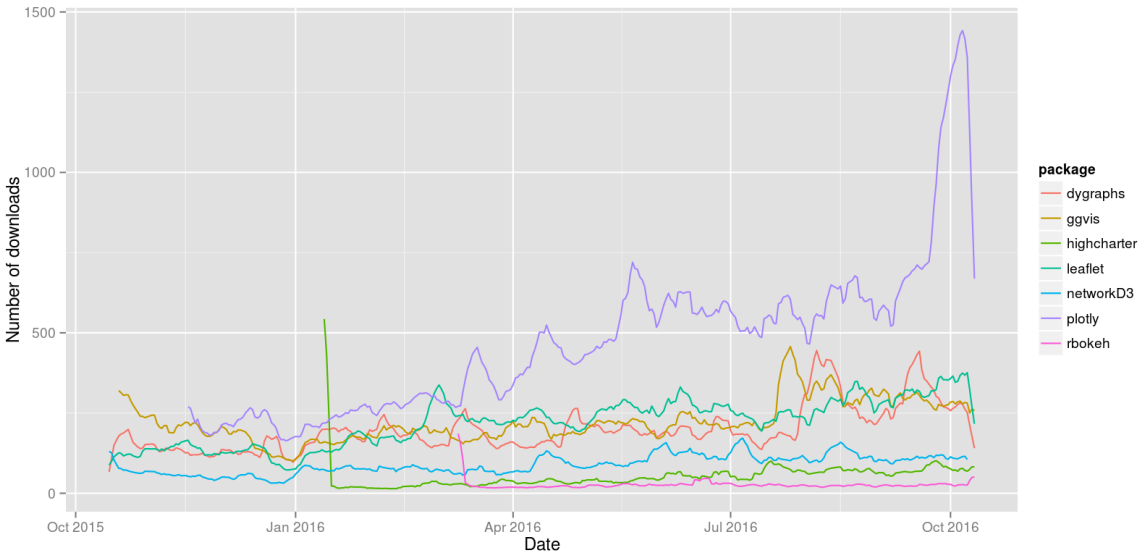


Figure 9.1 CRAN downloads over the past 6 months from RStudio's anonymized CRAN mirror download logs. Shown are common packages for interactive web graphics.

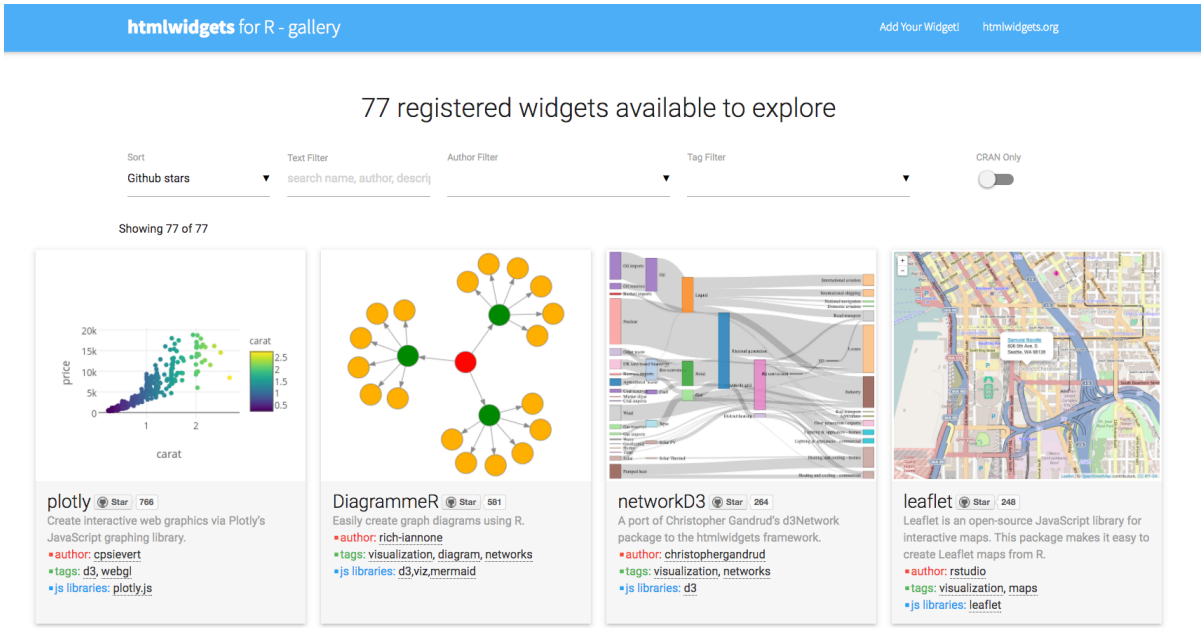


Figure 9.2 A screenshot of the htmlwidgets gallery website. This website allows you to browse R packages built on the htmlwidgets framework and sort widgets by the number of GitHub stars. On October 17th, the date this screenshot was taken, plotly had 769 stars which is the most among all htmlwidgets.

BIBLIOGRAPHY

Adler, Daniel, Duncan Murdoch, and others. 2016. *Rgl: 3D Visualization Using Opengl*. https://CRAN.R-project.org/package=rgl.

Ahlberg, Christopher, Christopher Williamson, and Ben Shneiderman. 1991. "Dynamic Queries for Information Exploration: An Implementation and Evaluation." In *ACM Chi '92 Conference Proceedings*, 21:619–26.

Allaire, JJ. 2016. *Flexdashboard: R Markdown Format for Flexible Dashboards*. https://CRAN.R-project.org/package=flexdashboard.

Allaire, JJ, Joe Cheng, Yihui Xie, Jonathan McPherson, Winston Chang, Jeff Allen, Hadley Wickham, Aron Atkins, and Rob Hyndman. 2016. *Rmarkdown: Dynamic Documents for R*. https://CRAN.R-project.org/package=rmarkdown.

Alt, Alina, and Marvin S. White. 2008. Tracking an object with multiple asynchronous cameras. US 70219509. Patent, issued September 11, 2008. http://www.patentlens.net/patentlens/patent/US_7062320/.

Andreas Buja, Catherine Hurley, Daniel Asimov, and John A. McDonald. 1988. "Elements of a Viewing Pipeline for Data Analysis." In *Dynamic Graphics for Statistics*,

edited by William S. Cleveland and Marylyn E. McGill. Belmont, California: Wadsworth, Inc.

Asimov, Daniel. 1985. "The Grand Tour: A Tool for Viewing Multidimensional Data." *SIAM J. Sci. Stat. Comput.* 6 (1). Philadelphia, PA, USA: Society for Industrial; Applied Mathematics: 128–43. doi:10.1137/0906011.

Attali, Dean. 2016. *Colourpicker: A Colour Picker Widget for Shiny Apps, Rstudio, R-Markdown, and 'Htmlwidgets'*. https://CRAN.R-project.org/package=colourpicker.

Auguie, Baptiste. 2016. *GridExtra: Miscellaneous Functions for "Grid" Graphics*. https://CRAN.R-project.org/package=gridExtra.

Bache, Stefan Milton, and Hadley Wickham. 2014. *Magrittr: A Forward-Pipe Operator for R*. https://CRAN.R-project.org/package=magrittr.

Baumer, Ben, and Carson Sievert. 2016. *Etl: Extract-Transfer-Load Framework for Medium Data*. http://github.com/beanumber/etl.

Bååth, Rasmus. 2016. "An Implementation of a Small Mcmc Framework and Some Likelihood Functions for Doing Bayes Stats in the Browser." https://github.com/rasmusab/bayes.js.

Becker, R. A., and J. M. Chambers. 1978. "Design and Implementation of the 'S' System for Interactive Data Analysis." *Proceedings of COMPSAC*, 626–29.

Becker, RA, and WS Cleveland. 1987. "Brushing Scatterplots." *Technometrics* 29 (2): 127–42.

Bischof, Jonathan M., and Edoardo M. Airoldi. 2012. "Summarizing Topical Content with Word Frequency and Exclusivity." In *Icml*.

Blei, David M., and John Lafferty. 2009. "Visualizing Topics with Multi-Word Expressions." *Arxiv*. https://arxiv.org/pdf/0907.1013.pdf.

Bostock, Jeffrey Heer AND Michael. 2010. "Crowdsourcing Graphical Perception: Using Mechanical Turk to Assess Visualization Design." In *ACM Human*

Factors in Computing Systems (Chi), 203–12. http://vis.stanford.edu/papers/crowdsourcing-graphical-perception.

Bostock, Michael, Vadim Oglevetsky, and Jeffrey Heer. 2011. "D3 Data-Driven Documents." *IEEE Transactions on Visualization and Computer Graphics* 17 (12): 2301–9.

Brynjar Gretarsson, Svetlin Bostandjieb, John O'Donovan, and Padhraic Smyth. 2011. "TopicNets: Visual Analysis of Large Text Corpora with Topic Modeling." In *ACM Transactions on Intelligent Systems and Technology*.

Buja, Andreas, Dianne Cook, Heike Hofmann, Michael Lawrence, Eun-Kyung Lee, Deborah F Swayne, and Hadley Wickham. 2009. "Statistical inference for exploratory data

analysis and model diagnostics." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 367 (1906): 4361–83.

Buja, Andreas, John Alan McDonald, John Michalak, and Werner Stuetzle. 1991. "Interactive data visualization using focusing and linking." *IEEE Proceedings of Visualization*, February, 1–8.

Cairo. 2016. "Cairo: A Vector Graphics Library." http://cairographics.org/.

Chambers, John. 1992. "Classes and Methods: Object-Oriented Programming in S." In *Statistical Models in S*, edited by J. M. Chambers and T. J. Hastie. Wadsworth & Brooks/Cole.

———. 1999. *Programming with Data*. Springer Verlag.

Chaney, Allison J.B., and David M. Blei. 2012. "Visualizing Topic Models." In *Icwsm*.

Chang, Winston. 2016. *Webshot: Take Screenshots of Web Pages*. https://CRAN.R-project.org/package=webshot.

Chang, Winston, and Hadley Wickham. 2015. *Ggvis: Interactive Grammar of Graphics*. http://CRAN.R-project.org/package=ggvis.

Chang, Winston, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. 2015. *Shiny: Web Application Framework for R*. http://CRAN.R-project.org/package=shiny.

Cheng, Joe. 2015a. *Crosstalk*. https://github.com/rstudio/crosstalk.

———. 2015b. *D3scatter: Demo of D3 Scatter Plot; Testbed for Crosstalk Library*. https://github.com/jcheng5/d3scatter.

Cheng, Joe, and Yihui Xie. 2015. *Leaflet: Create Interactive Web Maps with the Javascript 'Leaflet' Library*. http://rstudio.github.io/leaflet/.

Cleveland, William S, and Robert McGill. 1984. "Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods." *Journal of the American Statistical Association* 79 (September): 531–54.

Cleveland, William S., and Ryan Hafen. 2014. "Divide and Recombine (d&R): Data

Pairs Plot." *Journal of Computational and Graphical Statistics* 22 (1): 79–91. doi:10.1080/10618600.2012.694762.

Fast, Mike. 2007. "How to Build a Pitch Database." http://fastballs.wordpress.com/2007/08/23/how-to-build-a-pitch-database/.

———. 2011. "Spinning Yarn: A Zone of Their Own." http://www.baseballprospectus.com/article.php?articleid=14572.

Few, Stephen. 2006. "Data Visualization: Rules for Encoding Values in Graph." https://web.archive.org/web/20160404214629/http://www.perceptualedge.com/articles/b-eye/encoding_values_in_graph.pdf.

Freedman, D., and P. Diaconis. 1981. "On the Histogram as a Density Estimator: L_2 Theory." *Zeitschrift Für Wahrscheinlichkeitstheorie Und Verwandte Gebiete* 57: 453–76.

Friedman, Daniel P., and Mitchell Wand. 2008. *Essentials of Programming Languages, Third Edition*. MIT Press.

Fujino, Tomokazu. 2015. *VdmR: Visual Data Mining Tools for R*. http://CRAN.R-project.org/package=vdmR.

Galili, Tal. 2016. *Heatmaply: Interactive Heat Maps Using 'Plotly'*. https://CRAN.R-project.org/package=heatmaply.

Gandrud, Christopher, J.J. Allaire, and Kenton Russell. 2015. *NetworkD3: D3 Javascript Network Graphs from R*. http://CRAN.R-project.org/package=networkD3.

Garnier, Simon. 2016. *ViridisLite: Default Color Maps from 'Matplotlib' (Lite Version)*. https://CRAN.R-project.org/package=viridisLite.

Gentleman, Robert, and Duncan Temple Lang. 2004. "Statistical Analyses and Reproducible Research." *Bioconductor Project Working Papers*, November, 1–38.

Gohel, David. 2016a. *Ggiraph: Make 'Ggplot2' Graphics Interactive Using 'Htmlwidgets'*. https://CRAN.R-project.org/package=ggiraph.

———. 2016b. *Rvg: R Graphics Devices for Vector Graphics Output*. https://CRAN.

Large Complex Data." In *Large-Scale Data Analysis and Visualization (Ldav), 2013 Ieee Symposium on*, 105–12. doi:10.1109/LDAV.2013.6675164.

Hafen, Ryan. 2016. *Trelliscope: Create and Navigate Large Multi-Panel Visual Displays*. https://CRAN.R-project.org/package=trelliscope.

Hafen, Ryan, and Bokeh team. 2015. *Rbokeh: R Interface for Bokeh*.

Hamilton, Nicholas. 2016. *Ggtern: An Extension to 'Ggplot2', for the Creation of Ternary Diagrams*. https://CRAN.R-project.org/package=ggtern.

Hanna M. Wallach, Ruslan Salakhutdinov, Iain Murray, and David Mimno. 2009. "Evaluation Methods for Topic Models." In *ICML*.

Harrison, John. 2014. *RSelenium: R Bindings for Selenium Webdriver*. http://CRAN.R-project.org/package=RSelenium.

Heer, Arvind Satyanarayan AND Kanit Wongsuphasawat AND Jeffrey. 2014. "Declarative Interaction Design for Data Visualization." In *ACM User Interface Software & Technology (Uist)*. http://idl.cs.washington.edu/papers/reactive-vega.

Heer, J, and GG Robertson. 2007. "Animated Transitions in Statistical Data Graphics." *IEEE Transaction on Visualization and Computer Graphics* 13 (6): 1240–7.

Heer, Michael Bostock AND Vadim Ogievetsky AND Jeffrey. 2011. "D3: Data-Driven Documents." *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*. http://vis.stanford.edu/papers/d3.

Heer, Zhicheng Liu AND Biye Jiang AND Jeffrey. 2013. "ImMens: Real-Time Visual Querying of Big Data." *Computer Graphics Forum (Proc. EuroVis)* 32 (3). http://vis.stanford.edu/papers/immens.

Heer, Zhicheng Liu AND Jeffrey. 2014. "The Effects of Interactive Latency on Exploratory Visual Analysis." *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*. http://idl.cs.washington.edu/papers/latency.

Hocking, Toby Dylan, Susan VanderPlas, and Carson Sievert. 2015. *Animint: Interac-*

Analysis Workflows." In *NIPS 2013 Topic Models: Computation, Application, and Evaluation*.

Jason Chuang, Christopher D. Manning, and Jeffrey Heer. 2012b. "Termite: Visualization Techniques for Assessing Textual Topic Models." In *Advanced Visual Interfaces*.

Jason Chuang, Christopher D. Manning, Daniel Ramage, and Jeffrey Heer. 2012a. "Interpretation and Trust: Designing Model-Driven Visualizations for Text Analysis." In *ACM Human Factors in Computing Systems (Chi)*.

Jason Chuang, Christopher D. Manning, Sonal Gupta, and Jeffrey Heer. 2013b. "Topic Model Diagnostics: Assessing Domain Relevance via Topical Alignment." In *Icml*.

Jeppson, Haley, Heike Hofmann, and Di Cook. n.d. *Ggmosaic: Mosaic Plots in the 'Ggplot2' Framework*. http://github.com/haleyjeppson/ggmosaic.

Jonathan Chang, Sean Gerrish, Jordan Boyd-Graber, and David M. Blei. 2009. "Reading Tea Leaves: How Humans Interpret Topic Models." In *Nips*.

Justin Snyder, Mark Dredze, Rebecca Knowles, and Travis Wolfe. 2013. "Topic Models and Metadata for Visualizing Text Corpora." In *Proceedings of the 2013 Naacl Hlt Demonstration Session*.

Lab, UW Interactive Data. 2016a. "JavaScript Data Utility Library." https://github.com/vega/datalib.

———. 2016b. "Reactive Dataflow Processing." https://github.com/uwdata/vega-dataflow.

Lander, Jared P. 2016. *Coefplot: Plots Coefficients from Fitted Models*. https://CRAN.R-project.org/package=coefplot.

Lang, Duncan Temple. 2006. "R as a Web Client the RCurl package." *Journal of Statistical Software*, July, 1–42.

———. 2013. *XML: Tools for Parsing and Generating Xml Within R and S-Plus*. http://CRAN.R-project.org/package=XML.

Lawrence, Michael. 2002. "The Ggobi Data Pipeline." http://web.archive.org/web/

(2). [American Statistical Association, Taylor & Francis, Ltd., Institute of Mathemati-

cal Statistics, Interface Foundation of America]: 123–55. http://www.jstor.org/stable/1390777.

Ripley, Brian D. 2001. "Connections." *R News* 1 (1): 1–32.

———. 2004. "Selecting Amongst Large Classes of Models." *Symposium in Honour of David Coxs 80th Birthday*.

Riutta et. al., Anders, and Kent Russell. 2015. *SvgPanZoom: R 'Htmlwidget' to Add Pan and Zoom to Almost Any R Graphic*. http://CRAN.R-project.org/package=svgPanZoom.

Robinson, David. 2016a. *Broom: Convert Statistical Analysis Objects into Tidy Data Frames*. https://CRAN.R-project.org/package=broom.

———. 2016b. *Gganimate: Create Easy Animations with Ggplot2*. http://github.com/dgrtwo/gganimate.

RStudio, and Inc. 2016. *Htmltools: Tools for Html*. https://CRAN.R-project.org/package=htmltools.

Rudis, Bob. 2016. *Ggalt: Extra Coordinate Systems, Geoms and Statistical Transformations for 'Ggplot2'*. https://CRAN.R-project.org/package=ggalt.

Ryan, Jeffrey A. 2016. *Quantmod: Quantitative Financial Modelling Framework*. https://CRAN.R-project.org/package=quantmod.

Saptarshi Guha, Jeremiah Rounds, Ryan Hafen, and William S. Cleveland. 2012. "Large Complex Data: Divide and Recombine (d&R) with Rhipe." *The ISI's Journal for the Rapid Dissemination of Statistics Research*, August, 53–67.

Sarkar, Deepayan. 2008. *Lattice: Multivariate Data Visualization with R*. New York: Springer. http://lmdvr.r-forge.r-project.org.

Schloerke, Barret, Jason Crowley, Di Cook, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Joseph Larmarange. 2016. *GGally: Extension to 'Ggplot2'*.

Scott, David W. 1979. "On Optimal and Data-Based Histograms." *Biometrika* 66: 605–

sualization, and Interfaces, June, 1–8. http://nlp.stanford.edu/events/illvi2014/papers/sievert-illvi2014.pdf.

Sievert, Carson, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Corvellec, and Pedro Despouy. 2016. *Plotly: Create Interactive Web-Based Graphs via Plotly's Api*. https://github.com/ropensci/plotly.

Sturges, Herbert A. 1926. "The Choice of a Class Interval." *Journal of the American Statistical Association* 21 (153): 65–66. doi:10.1080/01621459.1926.10502161.

Swayne, Deborah F, Dianne Cook, and Andreas Buja. 1998. "XGobi: Interactive Dynamic Data Visualization in the X Window System." *Journal of Computational and Graphical Statistics* 7 (1): 113–30.

Swayne, Deborah F., and Sigbert Klinke. 1999. "Introduction to the Special Issue on Interactive Graphical Data Analysis: What Is Interaction?" *Computational Statistics* 14 (1).

Taddy, Matthew A. 2011. "On Estimation and Selection for Topic Models." In *AISTATS*.

Temple Lang, Duncan. 2014a. *RCurl: General Network (Http/Ftp/.) Client Interface for R*. http://CRAN.R-project.org/package=RCurl.

———. 2014b. *RJSONIO: Serialize R Objects to Json, Javascript Object Notation*. http://CRAN.R-project.org/package=RJSONIO.

Theus, Martin, and Simon Urbanek. 2008. *Interactive Graphics for Data Analysis: Principles and Examples*. Chapman & Hall / CRC.

Thomas Leeper, Patrick Mair, Scott Chamberlain. n.d. "CRAN Task View: Web Technologies and Services." https://CRAN.R-project.org/view=WebTechnologies.

Traub, James. 2010. "Mariano Rivera, King of the Closers." http://www.nytimes.com/2010/07/04/magazine/04Rivera-t.html.

Trestle Technology, LLC. 2016. *Plumber: An Api Generator for R*. https://CRAN.R-project.org/package=plumber.

fredo Rizzi and Maurizio Vichi, 221–30. Physica-Verlag HD. http://dx.doi.org/10.1007/978-3-7908-1709-6_17.

Unwin, Antony, and Heike Hofmann. 2009. "GUI and Command-line - Conflict or Synergy?" *Proceedings of the St Symposium on the Interface*, September, 1–11.

Unwin, Antony, Chris Volinsky, and Sylvia Winkler. 2003. "Parallel Coordinates for Exploratory Modelling Analysis." *Computational Statistics & Data Analysis* 43 (4): 553–64.

Urbanek, Simon. 2004. "Model Selection and Comparison Using Interactive Graphics." PhD thesis.

———. 2015. *RJava: Low-Level R to Java Interface*. http://CRAN.R-project.org/package=rJava.

Urbanek, Simon, and Jeffrey Horner. 2015. *FastRWeb: Fast Interactive Framework for Web Scripting Using R*. http://CRAN.R-project.org/package=FastRWeb.

Vaidyanathan, Ramnath. 2013. *RCharts: Interactive Charts Using Javascript Visualization Libraries*. https://github.com/ramnathv/rCharts/.

Vaidyanathan, Ramnath, Yihui Xie, JJ Allaire, Joe Cheng, and Kenton Russell. 2015. *Htmlwidgets: HTML Widgets for R*. http://CRAN.R-project.org/package=htmlwidgets.

Vanderkam, Dan, and JJ Allaire. 2015. *Dygraphs: Interface to Dygraphs Interactive Time Series Charting Library*. http://CRAN.R-project.org/package=dygraphs.

Venables, W. N., and B. D. Ripley. 2002. *Modern Applied Statistics with S*. Fourth. New York: Springer. http://www.stats.ox.ac.uk/pub/MASS4.

Wang, Earo. 2016. *Tscognostics: Time Series Cognostics*. https://github.com/earowang/tscognostics.

Wickham, Hadley. 2007. "Meifly: Models explored interactively." *Website ASA Sections on Statistical Computing and Graphics (Student Paper Award Winner)*.

———. 2009a. *Ggplot2: Elegant Graphics for Data Analysis*. New York: Springer.

[American Statistical Association, Taylor & Francis, Ltd.]: 289–300. http://www.jstor.org/stable/2288843.

Wood, S.N. 2006. *Generalized Additive Models: An Introduction with R*. Chapman; Hall/CRC.

World Bank. 2012. "World Development Indicators." http://data.worldbank.org/data-catalog/world-development-indicators.

Xie, Yihui. 2013a. "Animation: An R Package for Creating Animations and Demonstrating Statistical Methods." *Journal of Statistical Software* 53 (1): 1–27. http://www.jstatsoft.org/v53/i01/.

———. 2013b. *Dynamic Documents with R and Knitr*. Chapman; Hall/CRC.

———. 2015. *DT: A Wrapper of the Javascript Library 'Datatables'*. http://rstudio.github.io/DT.

Xie, Yihui, Heike Hofmann, and Xiaoyue Cheng. 2014. "Reactive Programming for Interactive Graphics." *Statistical Science* 29 (2): 201–13.

Yihui Xie, Di Cook, Heike Hofmann. 2013. *Interactive Statistical Graphics Based on Qt*.