GENERALIZED PLOT MATRICES, AUTOMATIC COGNOSTICS, AND EFFICIENT DATA EXPLORATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Barret Schloerke

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. William Cleveland, Co-Chair

Shanti S. Gupta Distinguished Professor of Statistics

Dr. Ryan Hafen, Co-Chair

Adjunct Assistant Professor of Statistics

Dr. Bowei Xi

Associate Professor of Statistics

Dr. Vinayak Rao

Assistant Professor of Statistics

Approved by:

Dr. Hao Zhang

Department Head of Statistics

ACKNOWLEDGMENTS

On the academic front, I would like to thank my co-chair advisors, Ryan Hafen and Bill Cleveland. I am grateful for their consistent guidance and willingness to take risks on new ideas. I would like to thank my long-time academic advisors: Di Cook and Hadley Wickham. Without their encouragement and support over the past decade, I would not be where I am today. *If I have seen further, it is only by standing on the shoulders of giants.*

I am grateful to Purdue Statistics for providing an excellent environment for both learning and research. Thank you Rebecca Doerge and Mark Ward for helping steer my early stages at Purdue and always lending me an ear. Thank you Doug Crabill for his willingness to share his deep technical knowledge at any moment. I am honored to have been able to attend numerous Wednesday Night Probability Seminars.

To my parents, I can not thank you enough for your unconditional love and support. Whether it is your well timed advice or your nodding and smiles as I ramble about a topic, I can not tell you how much I appreciate it.

And to Tracy, from traveling all across the globe to spending late nights in the library, I am truly thankful for having you in my life. Your never ending patience, encouragement, and love always makes me smile.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

API	Application Protocol Interface
CCA	Canonical Correlation Analysis
HTTP	Hypertext Transfer Protocol
PDF	Portable Document Format
REST	Representational State Transfer
RESTful	API that has REST qualities
URL	Uniform Resource Locator

GLOSSARY

Axes	All axis areas of a plot. Typically only X and Y.
Browser	Computer application that visits websites using the internet
Directed Graph	A graph whose edges have a defined start and end
Host	a web server with a unique URL
Interface	A point where two systems, subjects, organizations, etc., meet and interact
Jittered	Points slightly altered from their original position to help overcome overplotting
Overplotted	What occurs too many points are displayed to see the underlying structure
Panel	Plotting area. This is not limited to the strips, axes, and plot content. When used in the context of a plot matrix, it refers to a single sub plot within the plot matrix.
Plot	A statistical graphic displaying the relationship between variables.
Plot content	Plot area within the axes.
Server	A machine that calculates, stores, retrieves, and communicates information
Strip	A panel label
Web Server	A server that returns websites or data related to a website
URL, web address	Uniform Resource Locator. Also the address of a World Wide Web page

ABSTRACT

Schloerke, Barret Ph.D., Purdue University, December 2017. Generalized Plot Matrices, Automatic Cognostics, and Efficient Data Exploration. Major Professors: Dr. William Cleveland and Dr. Ryan Hafen.

Statistical visualization of large-scale data has become an increasingly essential task in the era of *big data*. In particular, exploratory data analysis and visualization is the first step towards any in-depth statistical modeling and analysis. Being able to rapidly specify and generate visualizations regardless of data-scale is crucial. Trelliscope handles data visualization at scale by attaching cognostics (univariate metrics) to each panel aiding in the organization of panels of interest. While Trelliscope provides a general framework for visualizing data at scale, there are several aspects that can be improved to help users generate displays more rapidly (such as cognostics, axis scales, etc.). When visually modeling complex data with Trelliscope, traditional two-grouped plot matrices do not allow for a mixed-scale axis to display both continuous and discrete data natively. Web-based visualization systems like Trelliscope, that retrieve information from a back-end service such as R, must maximize performance for an engaging user experience. Addressing the mixed-scale plot matrix axis, a generalized plot matrix is developed for two-grouped data which displays both continuous and discrete data using appropriate visualization methods for each panel. To compliment Trelliscope's panel organization, automatic cognostic summaries are established by mapping the context of what is visualized to classes of metrics that are meaningful for each type of visualization layer at no additional user effort. Finally, communication from web-based visualization systems to back-end R services is greatly improved by leveraging the GraphQL query language which minimizes the number of required data queries needed to perform data extraction. Together, these three contributions curtail the increasing complexity and scale of data visualization.

1. INTRODUCTION

Statistical visualizations of large-scale data has become an increasingly essential task in the era of *big data*. In particular, exploratory data analysis and visualization is the first step towards any in-depth statistical modeling and analysis. Since its release in 2000, the R programming language and environment [1] has becoming the top ranked open source data analysis tool [2] [3] and has arose as a powerful and convenient platform for performing data analysis and visualization with over 11,000 active, user submitted packages (as of Sept. 2017 [4]) and a number of more packages are in development on GitHub [5]. Various successful implementations have achieved in building a scientific visualization library in R, including `lattice` [6], `ggplot2` [7], `rbokeh` [8], and `plotly` [9], to name a few. `lattice` implements *trellis* graphics for R with powerful yet elegant high-level data visualization functions emphasizing on multivariate displays. `ggplot2` deconstructs higher level plots into a lower level of data visualization grammar using layered graphics which facilitate publication-ready data visualization. In this thesis, I develop

i) `ggduo`, an R function in `GGally` that produces generalized plot matrices for two groups of variables,

ii) `autocogs`, an R package that automatically generates cognostics for a set of plots, and

iii) `gqlr`, an R package which implements the GraphQL data query application protocol interface.

1.1 `ggduo`: Generalized pairs plot for two-grouped data

`ggplot2` offers a powerful graphics language for creating elegant and complex plots, however it has certain limitations. For instance, it does not allow the displaying of data sets with mixed scales (e.g., simultaneously display discrete and continuous scales) on the same axis. To incorporate this functionality, the R package `GGally` provides several composite plots (i.e., multi-layered plots) that build on the basic `ggplot2` [7] plotting framework. `GGally` functions produce multivariate plots such as generalized scatterplot matrices, and parallel coordinate plots are provided, as well as network plots, survival models, and glyph maps for spatiotemporal data. One important functionality in `GGally` is `ggpairs`. `ggpairs` an implementation of the generalized pairs plot [10]. The generalized pairs plot displays a plot matrix (all bivariate combinations of a single set of variables) that allows for a mixture of both continuous and discrete variable types using the `ggplot2` plotting framework. Furthermore, `ggpairs`'s plot matrix was generalized one step further to a generalized plot matrix which handles arbitrary `ggplot2` plot objects in a variable number of rows and columns.

In the first part of this thesis, I introduce a new function, `ggduo` that builds on the structure of the `ggmatrix` function used to produce the generalized pairs plot, `ggpairs`. Specifically, `ggduo` produces generalized plots for two groups of variables (e.g. a matrix of X variables and a matrix of Y variables), as might be modeled by multivariate regression diagnostics, canonical correlation analysis, or even multivariate time series. For the case of multivariate regression diagnostics, I develop `ggnostic` which displays common linear model diagnostic data against each of the model's explanatory variables. Figure 1.1 contains an example of `ggnostic` using a linear model of `flea` data where each plot displays model diagnostic information against each model explanatory variable. This plot would not have been possible in either `lattice` or `ggplot2` due to the mixture of axis scales and would have been

uninformative if rendered using `ggpairs`. I believe that the new function `ggduo` will help analysts to look at their data to support better modeling.

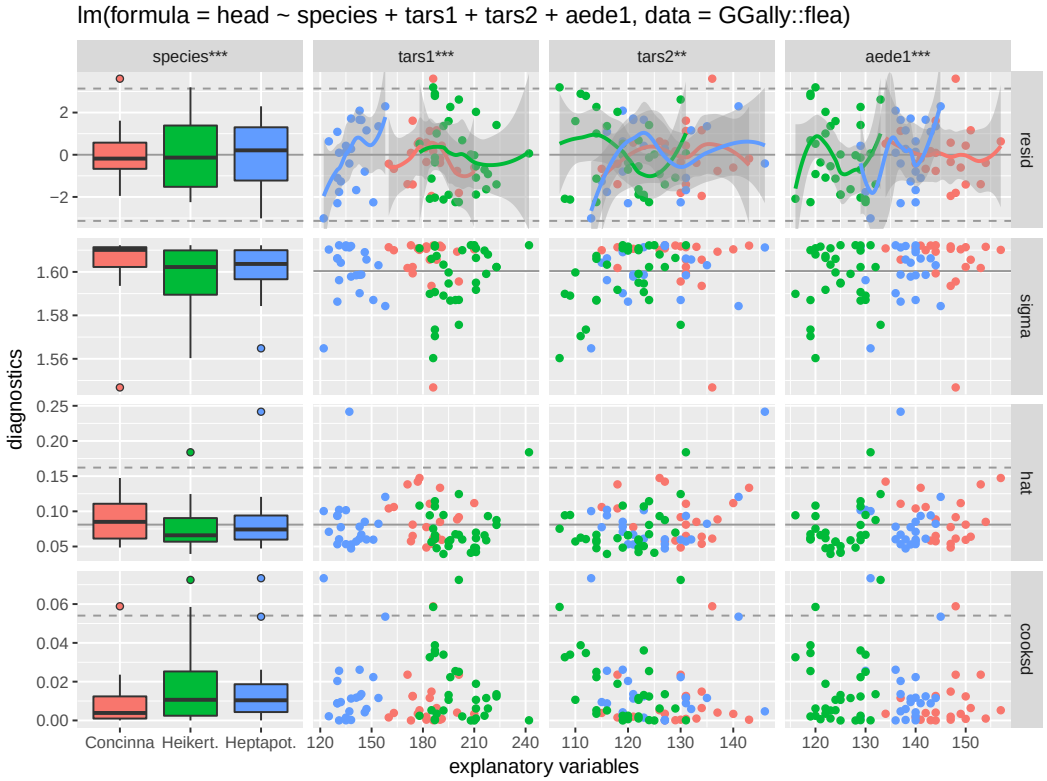


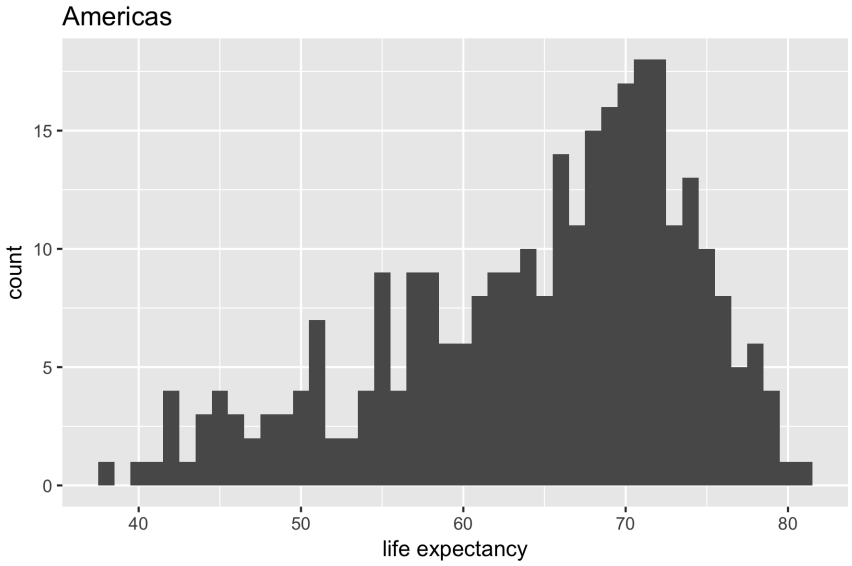
Figure 1.1. Model diagnostics are displayed in the plot matrix above using the `GGally` function `ggnostic`, which calls `ggduo`. Each panel of the plot matrix displays the same original data with different combinations of response and explanatory variables in each panel. Significance lines are displayed as dashed lines while solid lines represent expected values.

1.2 `autocogs`: Metrics enabling detailed interactive data visualization

To examine the difference between independent subsets of the same data set, Edward Tufte introduced the notion of *small multiples* using *trellis displays* [11]. In trellis displays, data are separated into independent subsets and a consistent visualization method is applied to each subset. The result is a set of panels that are displayed in a grid, resembling a garden trellis. These multi-panel display systems have proven to be very effective tools for visualizing complex data sets in detail. How-

ever, when the count of data subsets becomes very large, it is often the case that there are too many panels for the analyst to consume at one time. A simple idea put forth by John Tukey [12] is to compute *cognostics*, metrics that help bring different, interesting sets of panels in a display to the analyst's attention and allow the analyst to interactively sort and filter the panels. Cognostics can include statistical summaries, descriptive variables, goodness-of-fit metrics, etc.

Groups of cognostics and `ggplot2` layers are intimately connected. For instance, a simple histogram in `ggplot2` is created via a single histogram layer and is associated with four cognostic groups: univariate continuous cognostics, density cognostics, histogram cognostics, and count cognostics. In the second part of this thesis, I develop `autocogs` [13], an R package which automatically produces sets of standard cognostic groups that would be commonly useful to the data analyst given their supplied visualization objects. Figure 1.2 displays four groups of cognostics for the single layer (histogram) plot in two side-by-side tables. Eighteen cognostics in total are calculated. As an application, we demonstrate how `autocogs`can greatly enhance the functionalities of `trelliscopejs` [14]. `trelliscopejs`is an HTML widget that plot panels in an interactive trellis display which allows for sorting and filtering plot panels according to supplied cognostics. While it is possible to manually specify all the cognostics in `trelliscopejs`, `autocogs` greatly simplifies the user experience by automatically providing default groups of cognostics complimentary to the panel visualization.



group	cog	value
_x	min	37.58
	max	80.65
	mean	64.66
	median	67.05
	var	87.33
_density_x	max_density	0.05
	max_density_location	70.14
	unimodal_p_value	0.99
	skew	-0.74
	kurt	2.81

group	cog	value
_hist_x	count_min	0.00
	count_max	18.00
	count_mean	6.82
	count_median	6.00
	count_var	25.78
	chisq	0.00
_n	n	300.00
	n_na	0.00

Figure 1.2. `autocogs` automatically produces four cognostic groups for a single layer (histogram) plot shown above. These cognostic groups are shown in the two tables describing the cognostic group, name, and value. All eighteen cognostics may be used within a `trelliscopejs` HTML widget to aid in sorting and filtering plot panels.

1.3 `gqlr`: An R server GraphQL implementation

In 2012, Facebook began development of GraphQL, a backend agnostic data query language and runtime. Data query API that allows data to be queried without requiring knowledge how the data is stored. In doing so, GraphQL drastically reduces the number of server requests created by the browser by using a dynamic and nested query structure. For instance, when inspecting the names of a person's friends of

their friends, it would normally require $O(n^2)$ query commands to finally realize the full answer where n is the number of friends for each person. With GraphQL, the dynamic query structure allows for the full request to be sent to the server and a single, albeit, larger answer is returned. The submitted query command is separated from the actual backend service, moving the implementation complexity to the data backend service rather than the query submission process. Figure 1.3 displays the reduction in communication between a web browser client and the data server for both a naive REST implementation and a GraphQL implementation. The naive REST implementation requires $O(n^2)$ communications with the data server, while GraphQL executes a single communication with a single return value. By decoupling the data servers with web pages, the development cycles of both the web pages and data servers are improved. However, there is no prior interface of GraphQL to R. Therefore, in the third part of this thesis, I develop the `gqlr` R package, which implements a full GraphQL server within R. `gqlr` allows R users to supply their own functions to satisfy the data requirements of a submitted GraphQL query, thus enjoying the rapid iteration time of R and production iteration time of GraphQL.

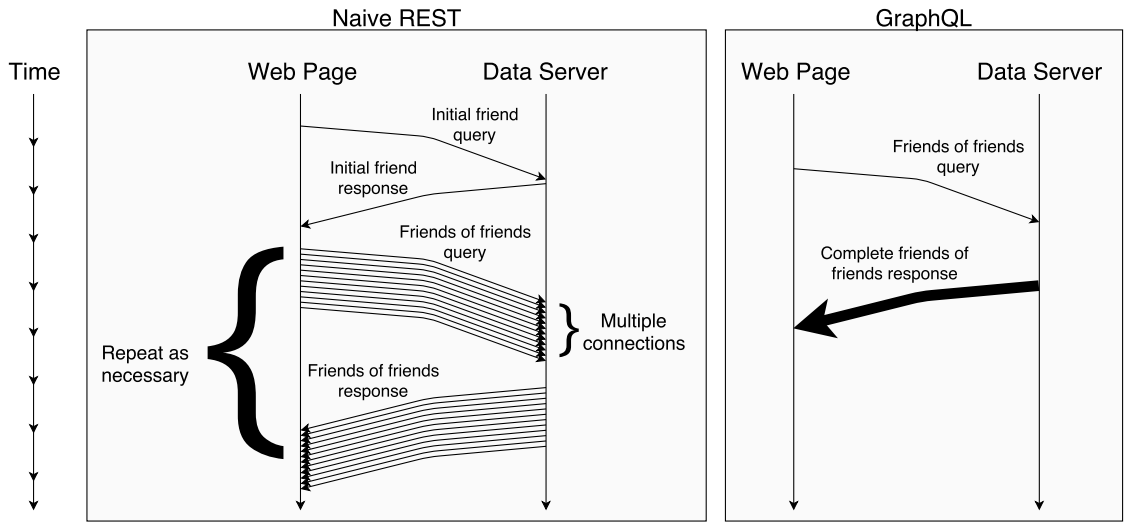


Figure 1.3. Time passes from top to bottom in this communication graph. In "friends of my friends" example, the Naive REST implementation requires $O(n^2)$ queries (where n is the number of friends for each person), while the GraphQL implementation solves the example query in a single communication to the data server.

1.4 Thesis Organization

The structure of this thesis is presented in Figure 1.4. The main contributions of this thesis include:

i) `ggduo`, an R function in `GGally` that produces generalized plot matrices for two groups of variables (Chapter 2),

ii) `autocogs`, an R package that automatically generates cognostics for a set of plots (Chapter 3), and

iii) `gqlr`, an R package which implements the GraphQL data query application protocol interface (Chapter 4).

All components of this thesis are developed within the R environment and relate to the visualization and exploration of data.

Building on a grammar of graphics, `ggplot2` is a layered plotting framework that `GGally` utilizes for visualizing data. Each plot within the cell of a `ggmatrix` plot matrix is a fully defined `ggplot2` plot object. Using `ggmatrix` as a plot matrix foundation, two main functions apply different variable combinations to produce different plot matrices: `ggpairs`, a generalized pairs plot, and `ggduo`, a generalized plot matrix for two-grouped data. `ggduo` is further extended by `ggnostic`, a generalized plot matrix for model diagnostics, and `ggts`, a generalized plot matrix for time series data. In Chapter 2, I develop the generalized plot matrix for two-grouped data and its extensions.

In Chapter 3, I develop `autocogs`, an R package that inspects a plot's data visualization layers to produce standard groups of cognostics. Currently, `ggplot2` plot objects are understood by `autocogs`; in future work, `autocogs` will be able to understand how to automatically produce cognostics for plot objects produced by the popular R plotting libraries `plotly`, `lattice`, and `rbokeh`. Once the cognostic groups are produced, they may be utilized within the interactive HTML widget `trelliscopejs` to sort and filter visualization panels. The R package `htmlwidgets` [15] generates HTML widgets which facilitate interactive web visualizations in R. `trelliscopejs` can display data visualization panels from many R packages, of which include `ggplot2`, `plotly`, `lattice`, and `rbokeh`.

In Chapter 4, I explore the GraphQL data query API and develop the R package `gqlr`. GraphQL was originally built to provide a consistent communication link between websites in the browser and data servers. The package `gqlr` implements a GraphQL data server within the R environment. `gqlr` is built to handle GraphQL data queries and expose statistical routines provided by R and its packages. While `gqlr` was originally pursued to aid data extraction with `trelliscopejs`, it will be integrated within `trelliscopejs` as development is continued.

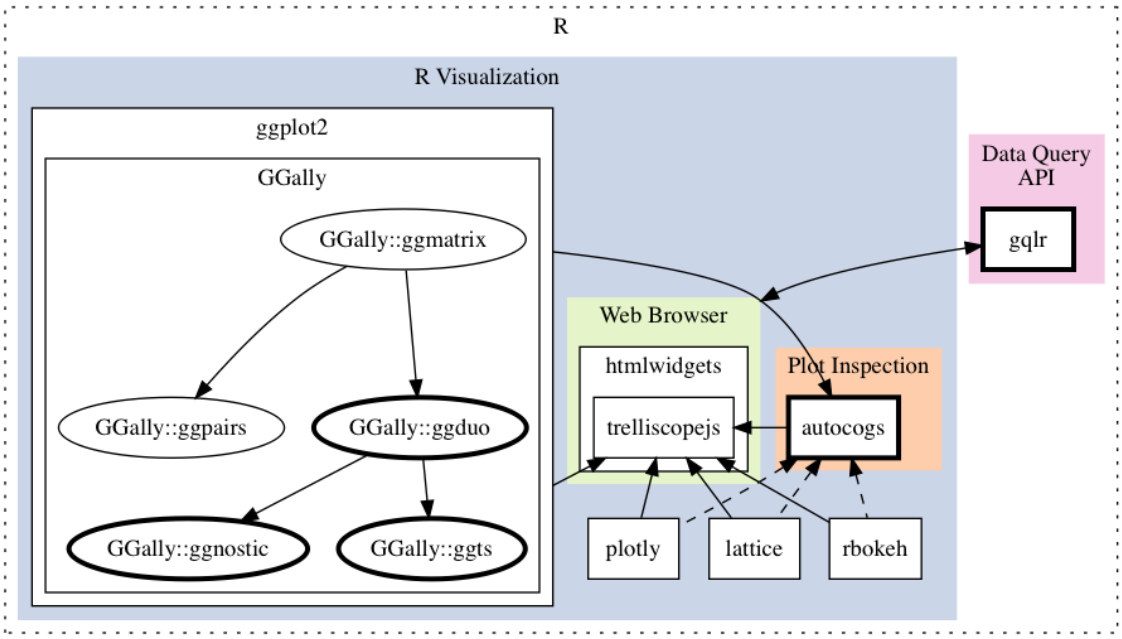


Figure 1.4. Thesis Organization. Shaded regions represent different contexts within the R environment. Black bordered square boxes represent R packages. `GGally` is built upon `ggplot2`, and ellipses within `GGally` represent package functions. Solid arrows represent functional dependencies or R package interactions. Dashed arrows represent possible future interactions. The bolded R packages and `GGally` functions correspond to different chapters within this thesis. Summary descriptions of these packages are provided in Appendix A.

2. GGDUO: GENERALIZED PAIRS PLOT FOR TWO-GROUPED DATA

2.1 `ggplot2`

2.1.1 *Layered Grammar of Graphics*

The R data visualization package `ggplot2` is based on a *Layered Grammar of Graphics* [16]. During consulting meetings with students and faculty in helping them produce statistical visualizations of their data, the authors of the package noticed that many clients had trouble producing plots quickly and have difficulties understanding how the plots were generated. This motivated them to develop the `ggplot2` package, which is based on the foundations of *The Grammar of Graphics* [].

2.1.2 `ggplot2` layers

Most of the standard statistical graphical displays take the form of single-layered plots. For example, a scatterplot consists of a point layer while a boxplot contains a boxplot layer. There is no formal name for the data graphic where horizontally jittered points are displayed on top of a vertical boxplot. Each layer is understood as a component to the plot, therefore the plot as a whole can be understood. Wickham did state that while a layered grammar guides a well formed graphic [16], he analogizes "good grammar is just the first step in creating a good sentence" [16].

Each plot consists of three components: the data, geom, and scales. Respectively, each of the component defines what is being displayed, how it is displayed and where it is being displayed. By inferring from the supplied data source and geom defaults, many plots can be displayed without much coding.

```
p <- ggplot(tips, aes(total_bill, tip)) +
  geom_point()
p
```

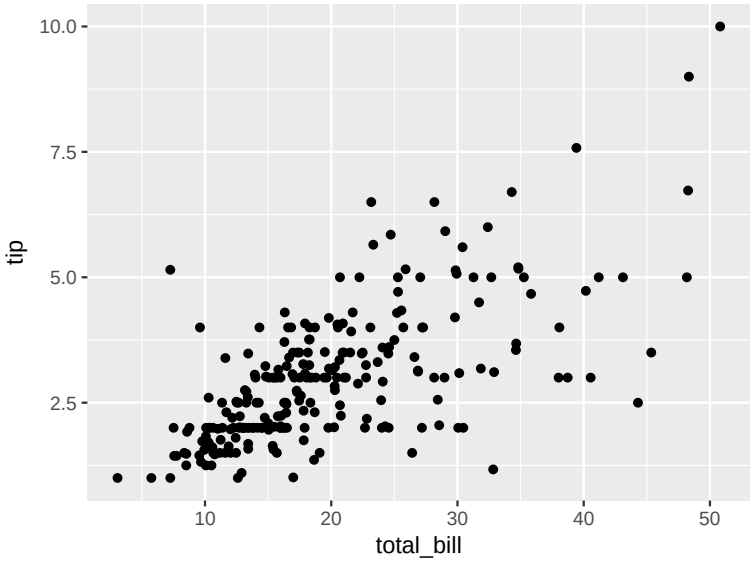


Figure 2.1. Minimal coding for a scatterplot of `tips` data.

Typically, each layer of a plot uses the same data source to explore different aspects of the data set. Multi-layered plots may use multiple data sources where it would not be appropriate to combine into a single data source. Added data sources usually display contextual information (such as a map) or summary statistics (such as a mean or linear model). Figure 2.2 manually adds a linear model line on top of a scatterplot.

```
tip_lm <- broom::tidy(lm(tip ~ total_bill, data = tips))$estimate
(tip_lm_dt <- data.frame(
  intercept = tip_lm[1],
  slope = tip_lm[2]
))
##   intercept     slope
## 1 0.9202696 0.1050245
p +
  geom_abline(
    data = tip_lm_dt,
    aes(slope = slope, intercept = intercept)
  )
```

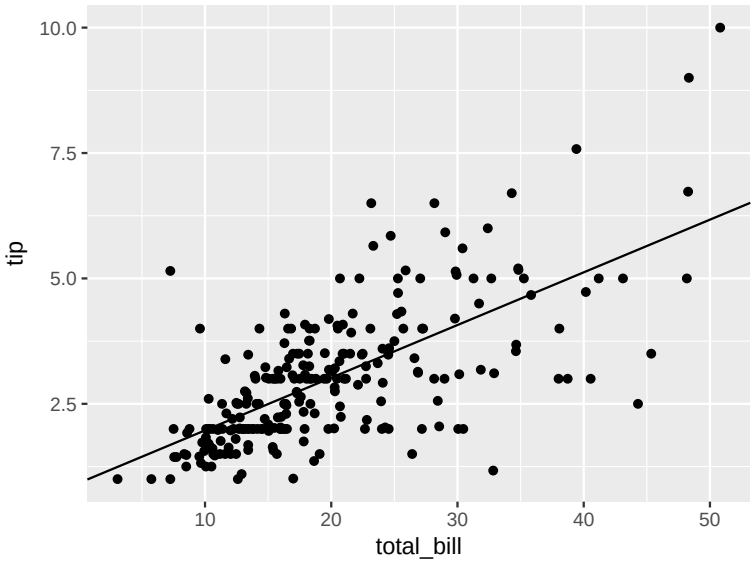


Figure 2.2. Scatterplot with a linear model line added using a different data source.

2.1.3 Plot creation

All plot layers do not need to exist at the time of plot object inception. Each layer of a `ggplot2` plot may be added one by one to the original plot object at different times of the code execution. The simplist of plot objects consists of a default data set and a default set of aesthetics.

```
minimal_plot <- ggplot(data = tips, aes(total_bill, tip))
# perform extra calculations
1 + 1
## [1] 2
# plot display delayed until print time
minimal_plot
```

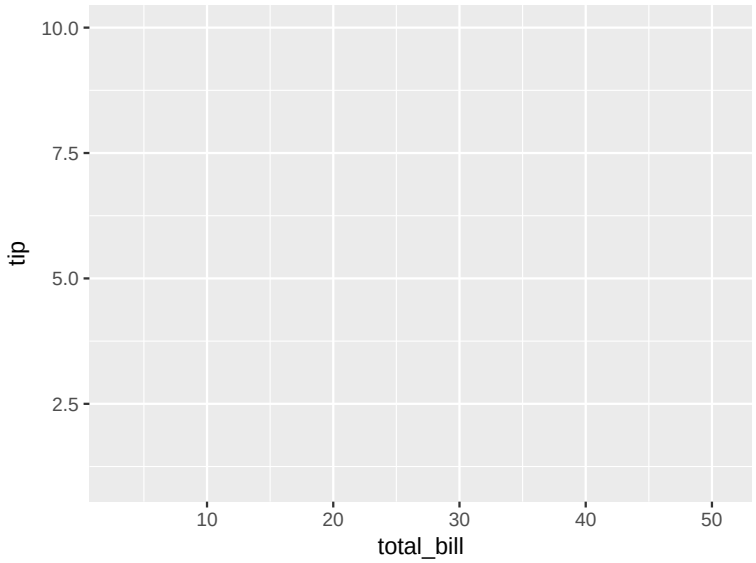


Figure 2.3. Empty `ggplot2` plot with no layers displayed after extra calculations.

No layers are displayed as none have been provided. However, the scales have been infered from the data aesthetics: X: `total_bill` and Y: `tip`.

2.1.4 Comparison

Building the plot up layer by layer and storing it as an R object until printed does not follow the existing patterns in standard R graphics. In Figure 2.4, the R core package `graphics` [1] displays information in a plot immediately upon function call. There is no ability to delay the display of the plot after the initial plot function has been called when using the `graphics` package.

```
g <- graphics::plot(tip ~ total_bill, data = tips)
```

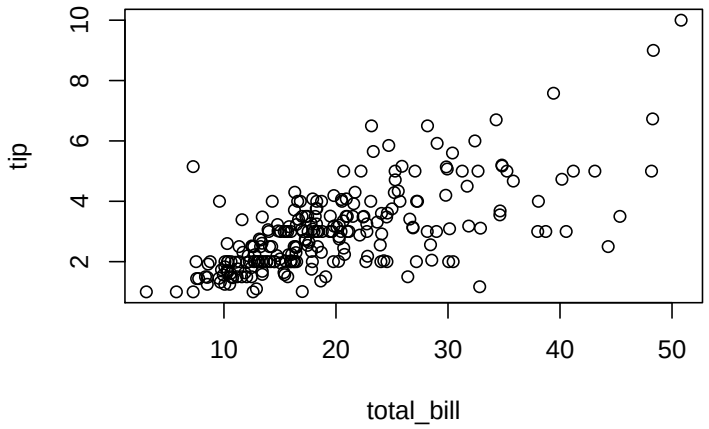


Figure 2.4. Stock R scatterplot using the `graphics` package is printed immediately.

```
# nothing is stored, as plot is already displayed
g
## NULL
```

`lattice` graphics meets in between `ggplot2` and `graphics`, as all layers must be supplied at plot creation, but the plot is not displayed until print time. Two prior `ggplot2` examples, Figure 2.1 and Figure 2.3, both displayed the ability to delay the printing of the `ggplot2` plot object. `lattice` graphics can also delay the display of the plot object as shown in Figure 2.5. At print time, both `lattice` and `ggplot2` convert their internal plot objects to be displayed using the `grid` package. The `grid` package does not implement full statistical plots, but rather it implements a R plotting framework to be used by other packages like `ggplot2` and `lattice`.

```
l <- lattice::xyplot(tip ~ total_bill, data = tips)
# display plot
l
```

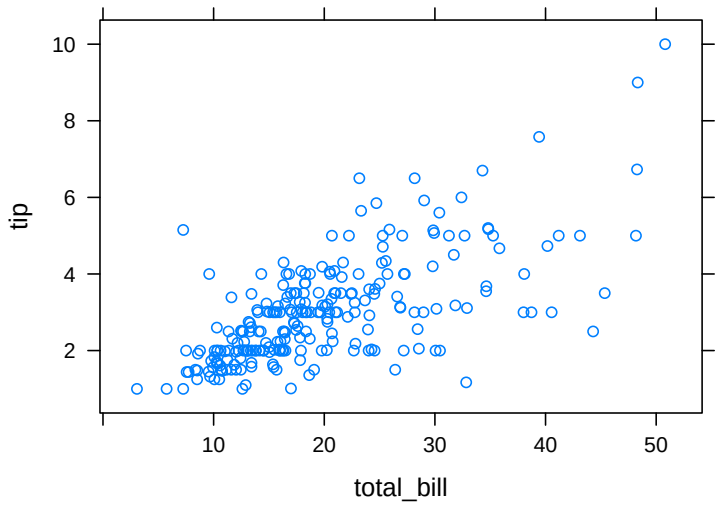


Figure 2.5. `lattice` scatterplot displaying using the `grid` framework.

Finally, after executing the `graphics` and `lattice` examples, we can add a point layer to the minimal `ggplot2` plot example in Figure 2.6.

```
# layer added after plot inception
minimal_points <- minimal_plot + geom_point()
# displays plot
minimal_points
```

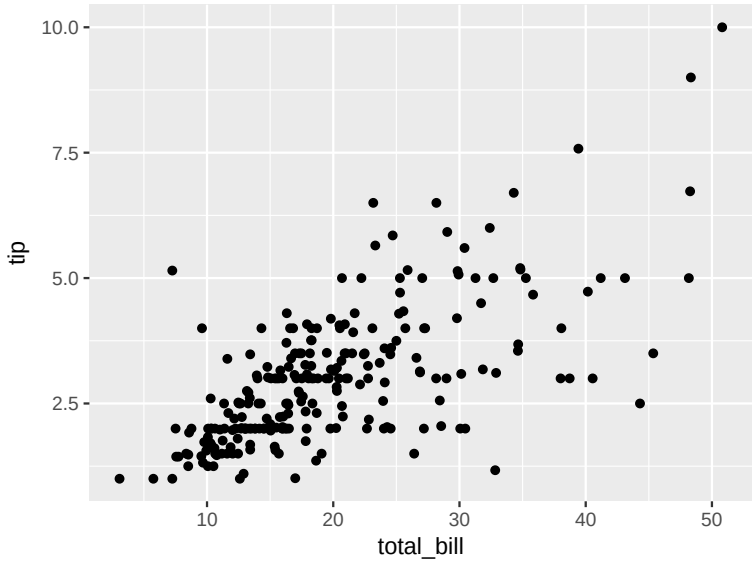


Figure 2.6. Adding a 'point' layer to the base `ggplot2` plot object created in an earlier code chunk.

Modularization of the `ggplot2` code allows for customization of each layer added to the plot. This leads to a smaller, more consistent interface for each layer function.

2.2 Facets

Winston Chang, maintainer of `ggplot2`, explains facet'ing as "[ploting] subsets of data into separate panels" [17]. This is achieved using existing conditioning variables in the supplied data set. For each conditioning combination, a panel is produced. This technique commonly refered to as small multiples [11]. The same style of plot is displayed, but each plot is constructed from an independent subset of the data. Typically, only one or two conditioning variables are used, but any number of variables may be used when creating small multiples or faceting a `ggplot2` plot.

Facets are useful when looking at the interaction of conditioning variables. Once all existing combinations of the conditioning variables have been made, subsets of the data are displayed in each of the panels with the strip (panel label) of the panel displaying the conditioning variable information. Missing combinations can either be dropped or display an empty panel. `ggplot2`'s wrapping facet will drop missing combinations by default, and `ggplot2`'s faceted grid will display an empty panel for missing combinations. In `ggplot2`, all variables are considered discrete when used as conditioning variables.

2.2.1 Facet wrap

There are two types of faceting in `ggplot2`: facet wrap and facet grid. Facet wrap displays each panel starting from the top row to the bottom row and left to right within each row. The number of rows and / or columns can be specified to ease the guessing work made by `ggplot2`. If no facet row or column counts are supplied, `ggplot2` uses `grDevices` [1] algorithm, `n2mfrow(n)` , to determine a sensible number of rows and columns. Figure 2.7 contains no faceting variables, while Figure 2.8 and Figure 2.9 condition on one and two variables respectively.

```
p <- ggplot(tips, aes(total_bill, tip)) + geom_point()
p +
  labs(title = "No Facets")
```

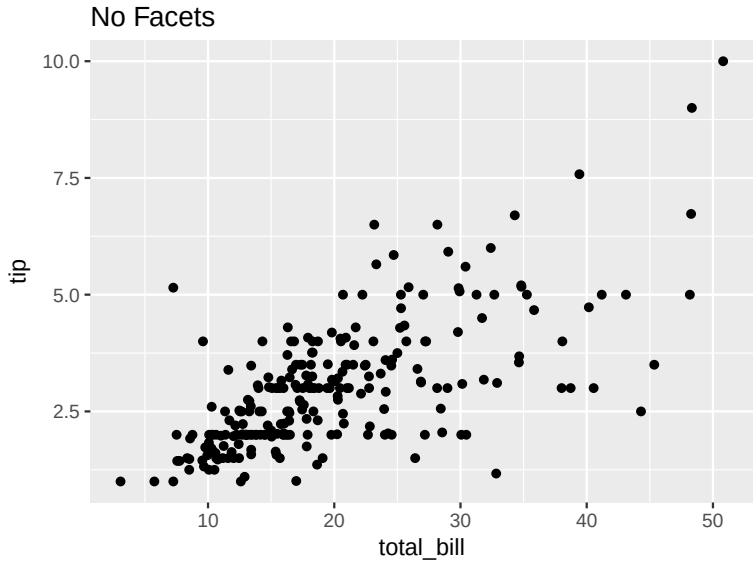


Figure 2.7. *tip* vs *total_bill* with facets or conditioning variables.

```
p +
  facet_wrap(~ day, labeller = label_both) +
  labs(title = "Facet Wrap (~ day)")
```

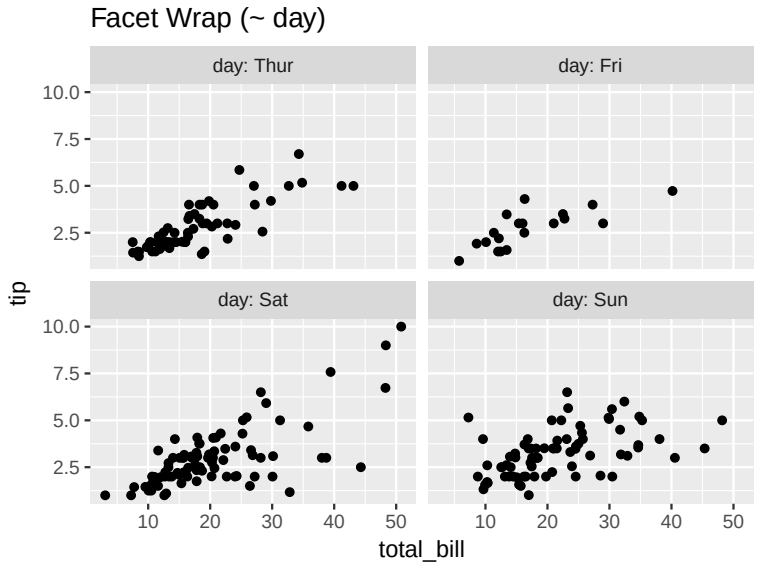


Figure 2.8. *tip* vs *total_bill* faceted by *day*. Each panel belongs to a given day in the data.

```
p +
  facet_wrap(~ day + time, labeller = label_both) +
  labs(title = "Facet Wrap (~ day + time)")
```

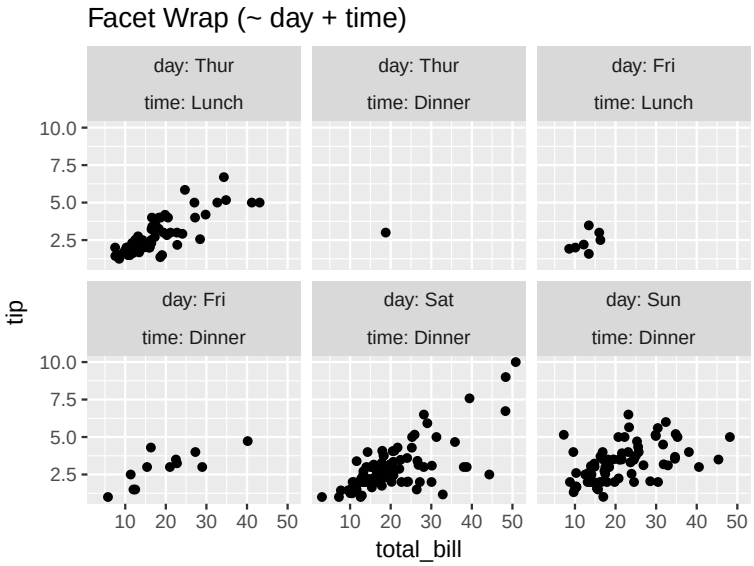


Figure 2.9. *tip* vs *total_bill* faceted by all existing *day* and *time* combinations.

2.2.2 Facet grid

Facet grid forces a two dimensional, matrix-like layout when faceting a plot. The layout involves two sets of conditioning variables: the X conditioning variables and the Y conditioning variables. These two sets of conditioning variables pre-determine how many rows (Y) and columns (X) are presented.

```
p +
  facet_grid(time ~ day, labeller = label_both) +
  labs(title = "Facet Grid (time ~ day)")
```

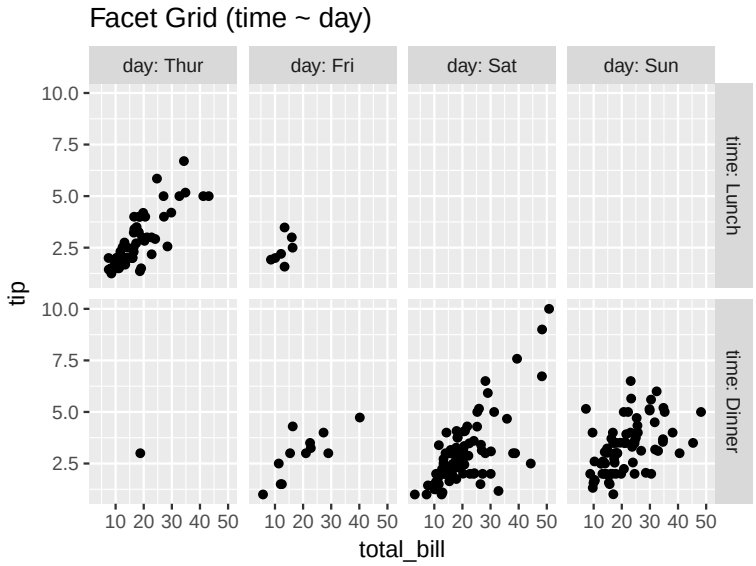


Figure 2.10. *tip* vs *total_bill* faceted in a grid pattern with *time* representing each row and *day* representing each column.

Figure 2.10 displays *time* on the rows (Y) and *day* on the columns (X). The `facet_grid` formula follows the R `stats::lm` linear model formula of `y ~ x`. Multiple variables can be used in either the X position or the Y position. This allows the user to display a column combination set against another column combination set. Using multiple conditioning variables within facet grid in Figure 2.11, both Male and Female smokers during Sunday Dinner tip value does not follow a positive linear trend as the total bill increases.

```
p +
  geom_smooth(method = lm, se = FALSE) +
  facet_grid(sex + smoker ~ time + day, labeller = label_both)
```

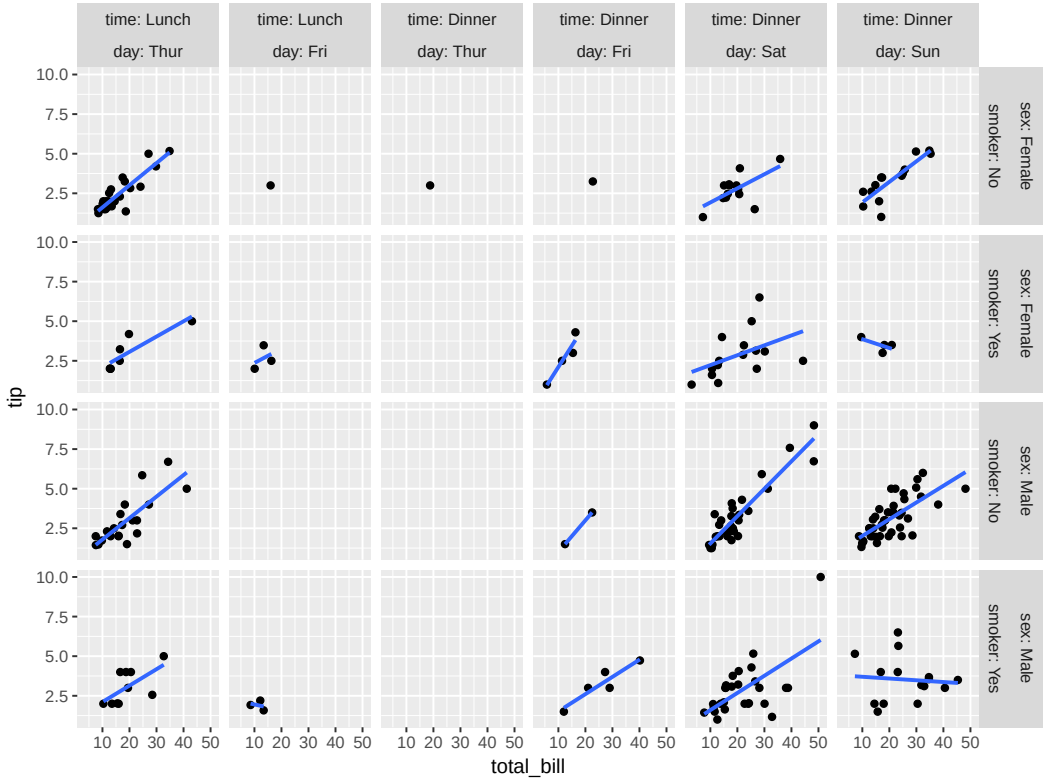


Figure 2.11. *tip* vs *total_bill* faceted in a grid pattern with *sex* and *smoker* combinations representing each row and *time* and *day* combinations representing each column.

2.3 Plot matrix

`ggduo` is continuation of prior work in the `GGally` R package. In this section, I will cover how a pairs plot is different than a small multiple, background on the scatterplot matrix and generalized pairs plot matrix, and the `ggmatrix` object that generalized the plot matrix used in `ggduo`.

2.3.1 Pairs plot

As mentioned earlier in the chapter, `ggplot2` is designed to allow plots to be built layer by layer and not rendered until a final print command is executed. `ggplot2` plot objects has some implementation rules it follows when faceting a plot:

1. Each faceted panel must share the same original X and Y columns.
2. Each faceted panel's data must be independent of all other panels in the same plot.
3. Each plot must be created with the same layers.

Following these rules allows for the implementation of small multiples. Small multiples are repetitions of plots with identical layers, but each plot is comprised of different data. Small multiples were popularized by *Edward Tufte* in his 1983 paper Visual Display of Quantitative Information [18]. The `facet_wrap` and `facet_grid` functions will produce a multi-panel output in the form of small multiples.

A pairs plot, originally referred to as a "draftsman's plot" [19], violates all three principles of a small multiple:

1. Each panel is comprised of a different X and Y combination.
2. Each panel shares the same underlying data.
3. Each panel may be created with different layers that better suited for the data types.

The original scatterplot matrix was only displayed using a scatterplot for each sub panel. *The Generalized Pairs Plot* by Emerson et. al., suggested that a pairs plot should not be restricted to just continuous scatterplots. A pairs plot should be generalized to allow for discrete data to be displayed as well as continuous data. A scatterplot matrix is not appropriate for discrete data. In a discrete data versus discrete data plot, all of the data points would be overplotted onto the unique discrete

combinations. This overplotting renders the discrete only pairs plot combinations uninterpretable. The bottom right quadrant of the pairs plot in Figure 2.12 display eight unique points between *smoker* and *day*, but do not convey that they really contain 244 points.

```
graphics::pairs(tips[c("total_bill", "tip", "smoker", "day")])
```

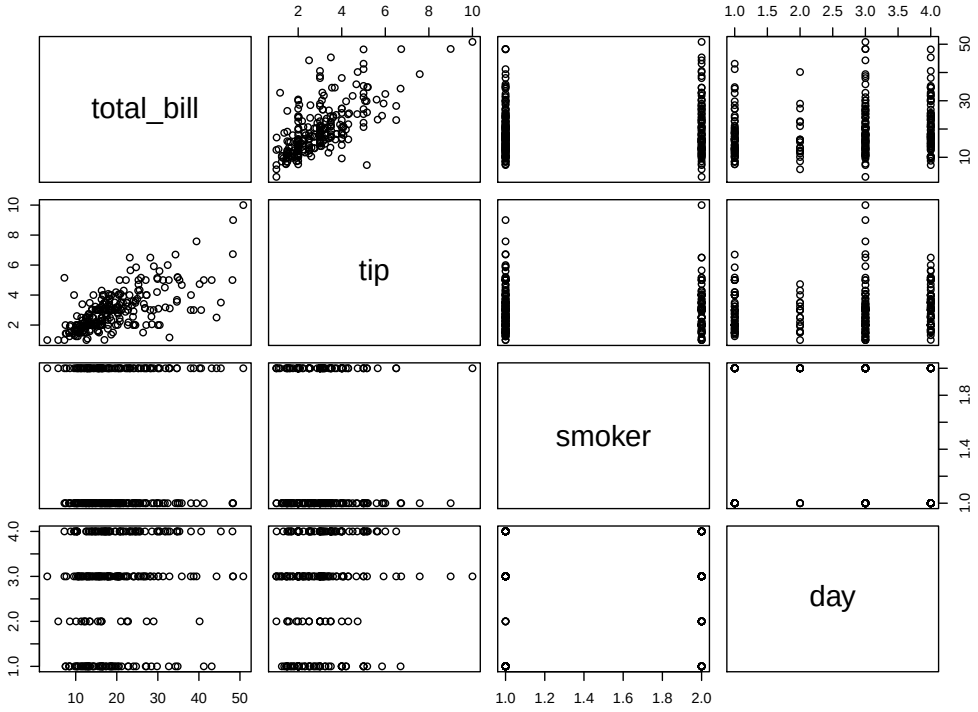


Figure 2.12. Stock R graphics scatterplot matrix displaying the `tips` data set.

Emerson et.al. discussed using a generalized pairs plot to handle the mix of variable types by providing different plotting layers for different data types. The generalized pairs plot displays the same collection of data using different axes but allows for a mix of plotting methods with both continuous and discrete plot axes. This addresses the issue of overplotting in discrete columns when displaying data in a scatterplot matrix. Figure 2.13 and Figure 2.14 are two plotting methods to handle the discrete data overplotting issue. The `gpairs` [20] R package is printed using `lattice` graphics (Figure 2.13), while the `GGally` R package prints using `ggplot2` graphics (Figure 2.14).

```
gpairs::gpairs(tips[, c("total_bill", "tip", "smoker", "day")])
```

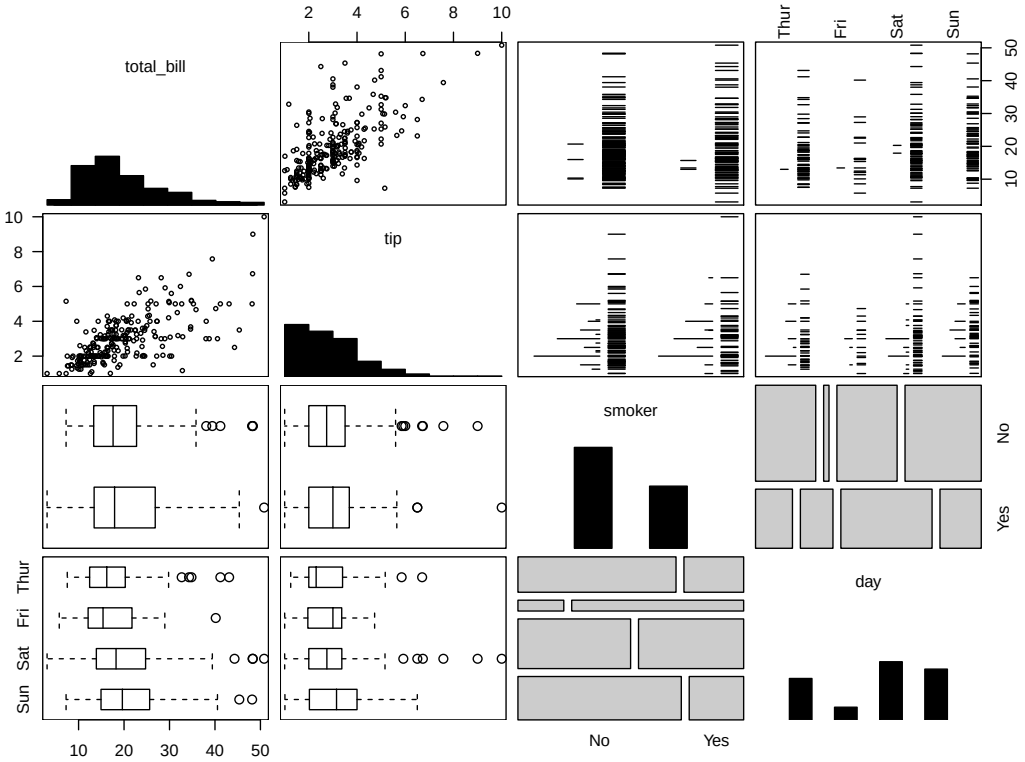


Figure 2.13. `gpairs` plot matrix from the `gpairs` R package displaying the `tips` data set.

```
pm <- ggpairs(tips, c("total_bill", "tip", "smoker", "day"))
pm
```

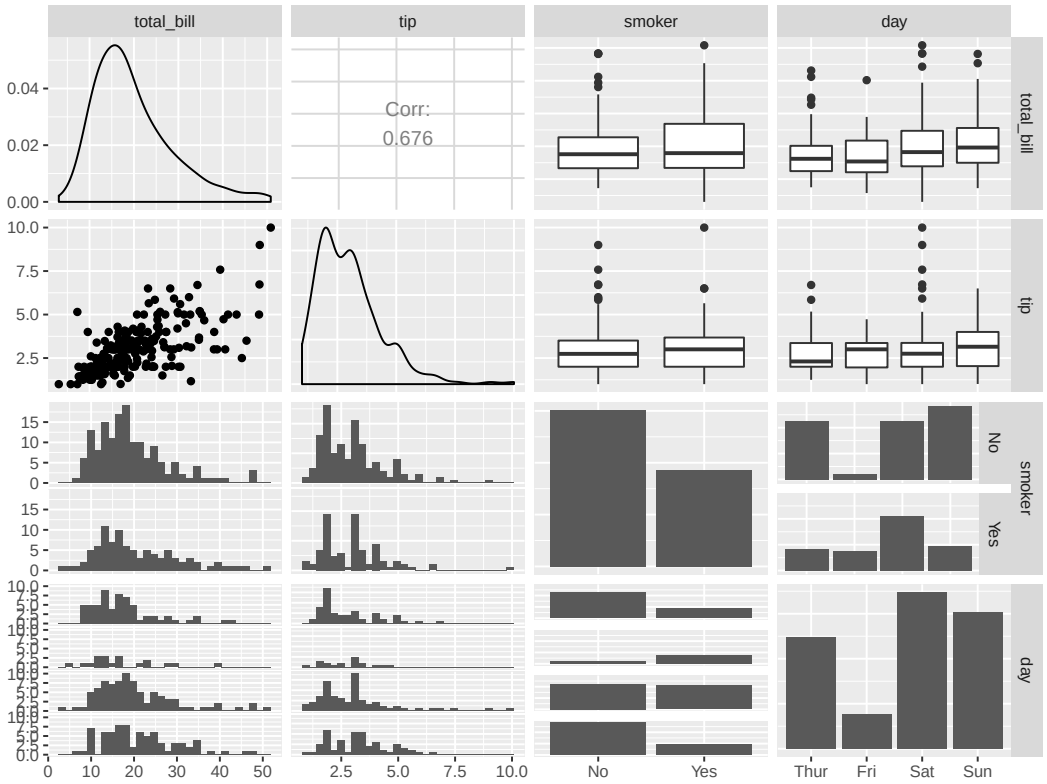


Figure 2.14. `ggpairs` plot matrix from the `GGally` R package displaying the `tips` data set.

`gpairs::gpairs` was originally built to handle the generalization of the pairs plot to include discrete data. Unfortunately, this is written using `lattice` graphics and does not utilize the `ggplot2` framework. `GGally::ggpairs` was originally written as an independent port of of `gpairs` to the `ggplot2` framework but with portability and modularization kept in mind.

`GGally`'s `ggpairs` addressed the portability and modularization by

1. making each sub plot an independently functioning `ggplot2` plot. Figure 2.15 displays how a sub plot may be retrieved.

2. allowing each sub plot to be replaced after the initial plot matrix is created. Figure 2.16 creates a new plot that is placed inside the matrix in Figure 2.17.

3. and not displaying the plot matrix until the `print` command is executed. Similar to earlier `ggplot2` examples, the displaying of a `GGally` plot matrix is delayed until print time as in Figure 2.17.

```
# retrieve the second row, first column sub plot
#  of the tips plot matrix
pm[2,1]
```

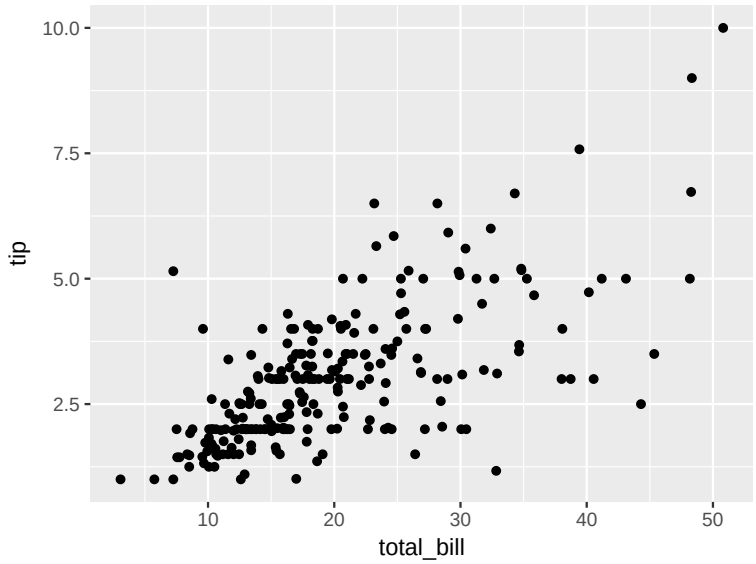


Figure 2.15. Full `ggplot2` plot object from the second row and first column of a `ggmatrix` plot matrix.

```
replacement_plot <- ggally_text(
  "Replacement\nPlot",
  aes(color = "red"),
  size = 6
)
replacement_plot
```

Figure 2.16. Replacement plot displaying “Replacement Plot” in red.

```
# insert the new plot into the second row, first column
#  of the tips plot matrix
pm[2,1] <- replacement_plot
# display the updated plot matrix
pm
```

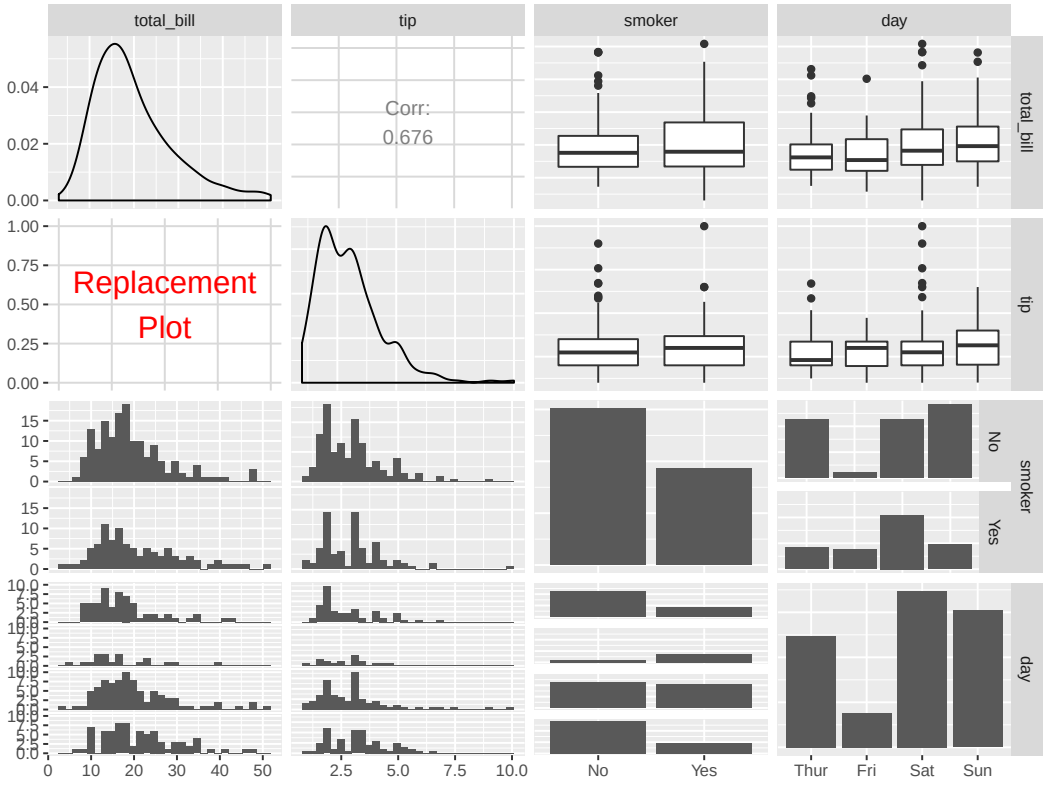


Figure 2.17. The replacement plot is placed in the second row and first column. The updated plot matrix is displayed.

2.3.2 `ggmatrix`

The `ggmatrix` object is used to handle two main situations: handle mixed plot scales and contain the necessary information for displaying a plot matrix.

Plot scales

`ggplot2` prevents discrete scales from being mixed with continuous scales. Two different scales for the same axis is not possible in a multi panel `ggplot2` plot as `ggplot2` is built on the small multiple princple of displaying similar scales in every faceted panel. Only one scale type is used per ggmatrix panel. This keeps the original 'per panel' logic intact.

Produce two related mixed-axes plots in the same graphic could only be achieved using the `gridExtra` [21] or `grid` [1] R packages before `ggmatrix`. Both of these existing methods behaved like a side by side, fully printed plots, rather than a native plot matrix.

```
p1 <- pm[1,4]
p2 <- pm[2,4]
gridExtra::grid.arrange(p1, p2, ncol = 1)
```

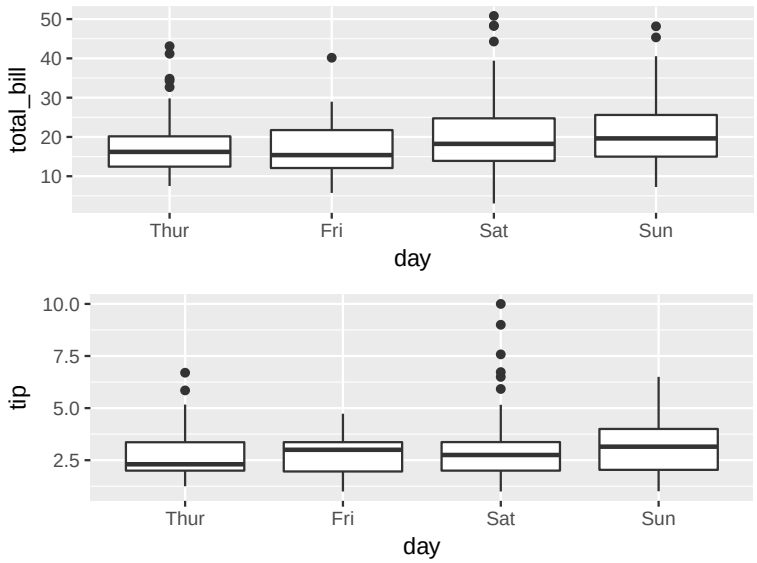


Figure 2.18. Two fully displayed `ggplot2` plot objects arranged using `gridExtra`. Duplicate axes and labels are present. The X axis does not align as the plots are treated independently. It appears as two independent plots in one display.

`ggmatrix` allows for arbitrary scales per panel. When used properly, independently produced sub plots deliver common axes within each row and column. This displays a cohesive and interpretable plot matrix structure.

Display information

The necessary information includes the number of rows and columns, label information, `ggplot2` display theme information, how plot strips should be displayed, default plot data, and individual plot information.

```
str(pm)
##
## Custom str.ggmatrix output:
## To view original object use 'str(pm, raw = TRUE)'
##
## List of 19
##  $ data               :'data.frame': 244 obs. of  7 variables:
##   ..$ total_bill: num [1:244] 17 10.3 21 23.7 24.6 ...
##   ..$ tip       : num [1:244] 1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##   ..$ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##   ..$ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ day       : Factor w/ 4 levels "Thur","Fri","Sat",..: 4 4 4 4 4 4 4 4 4 4 ...
##   ..$ time      : Factor w/ 2 levels "Lunch","Dinner": 2 2 2 2 2 2 2 2 2 2 ...
##   ..$ size      : int [1:244] 2 3 3 2 4 4 2 4 2 2 ...
##  $ plots              :List of 16
##   ..$ : chr "PM; aes: c(x = total_bill); fn: {wrap: 'ggally_densityDiag'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = tip, y = total_bill); fn: {wrap: 'ggally_cor'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = smoker, y = total_bill); fn: {wrap: 'ggally_box_no_facet'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = day, y = total_bill); fn: {wrap: 'ggally_box_no_facet'}; gg: FALSE"
##   ..$ : chr "PM; ggplot2 object; mapping: c()"
##   ..$ : chr "PM; aes: c(x = tip); fn: {wrap: 'ggally_densityDiag'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = smoker, y = tip); fn: {wrap: 'ggally_box_no_facet'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = day, y = tip); fn: {wrap: 'ggally_box_no_facet'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = total_bill, y = smoker); fn: {wrap: 'ggally_facethist'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = tip, y = smoker); fn: {wrap: 'ggally_facethist'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = smoker); fn: {wrap: 'ggally_barDiag'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = day, y = smoker); fn: {wrap: 'ggally_facetbar'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = total_bill, y = day); fn: {wrap: 'ggally_facethist'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = tip, y = day); fn: {wrap: 'ggally_facethist'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = smoker, y = day); fn: {wrap: 'ggally_facetbar'}; gg: FALSE"
##   ..$ : chr "PM; aes: c(x = day); fn: {wrap: 'ggally_barDiag'}; gg: FALSE"
##  $ title              : NULL
##  $ xlab               : NULL
##  $ ylab               : NULL
##  $ showStrips         : NULL
##  $ xAxisLabels        : chr [1:4] "total_bill" "tip" "smoker" "day"
##  $ yAxisLabels        : chr [1:4] "total_bill" "tip" "smoker" "day"
##  $ showXAxisPlotLabels: logi TRUE
##  $ showYAxisPlotLabels: logi TRUE
##  $ labeller           : chr "label_value"
##  $ switch             : NULL
##  $ xProportions       : NULL
##  $ yProportions       : NULL
##  $ legend             : NULL
##  $ gg                 : NULL
##  $ nrow               : int 4
```

```
##  $ ncol               : int 4
##  $ byrow              : logi TRUE
##  - attr(*, "_class")= chr [1:2] "gg" "ggmatrix"
```

For memory optimization, each plot is stored as a function that will produce the `ggplot2` plot at print time. To do so, the data set provided at a `ggmatrix` inception sets a default data set to be used at print time. If a new sub plot is stored after inception, the fully defined `ggplot2` object is stored directly.

Similar to `ggplot2`'s function `ggplot_gtable`, `ggmatrix_gtable` produces display only information that the `grid` graphics framework uses to display the plot. To retrieve all necessary display information, `ggmatrix_gtable` executes `ggplot_gtable` for each sub plot individually, then extracts the required plotting display information (plotting area and possibly the panel strips) from each sub plot. These sub plot panels are then placed inside the final plot matrix.

The resulting plot matrix is displayed with the exact same styling as `ggplot2`'s `facet_grid`. This returns all the display theme styling and display constraints back onto `ggplot2`. By displaying the `ggmatrix` as a `facet_grid` in `ggplot2`, titles, legends, and other common plot artifacts are able to be natively displayed. Figure 2.19 displays a legend on the right side of the plot matrix using the legend from the sub plot at position $(3, 1)$. Figure 2.20 uses the `ggplot2` theme functionality to move the legend to beneath the plot matrix panels.

```
# color the plots according to smoker
# display legend from the 3rd row, 1st column plot
pm <- ggpairs(
  tips, c("total_bill", "tip", "smoker", "day"),
  mapping = aes(color = smoker),
  legend = c(3,1)
)
pm
```

Figure 2.19. `ggmatrix` displaying a color legend on the right (default) side of the plot matrix.

```
# display legend on bottom using ggplot2 theme
pm + ggplot2::theme(legend.position = "bottom")
```

Figure 2.20. `ggmatrix` moving the legend to the bottom using `ggplot2`'s `theme` function.

2.4 `ggduo`: Plot matrix for two-grouped data

A pairs plot is defined as displaying every column of the data against every other column in the data. This is effective in full data exploration. If a data set has columns A, B, and C, $3^2 = 9$ combinations produced in a corresponding pairs plot: $A : A$, $A : B$, $A : C$, $B : A$, $B : B$, etc.. A pairs plot matrix can be generalized one step further by pairing two column groups against each other.

For the example just described, column set $\{A, B, C\}$ is paired against the column set $\{A, B, C\}$. As expected, this produces a pairs plot. However, this generalization allows us to produce a plot matrix of the combinations of $\{A, B, C\}$ and $\{D, E\}$, or

any combination of two column sets. Using the same underlying `ggmatrix` functionality, `ggduo` produces plot combinations of two-grouped data.

As a quick example, let us look at the Australian students' fifth grade *Math*, *Reading*, and *Science* scores against their *gender* and how many hours of *homework* each student completed each week in Figure 2.21.

```
ggduo(
  australia_PISA2012,
  c("gender", "homework"),
  c("PV5MATH", "PV5READ", "PV5SCIE")
)
```

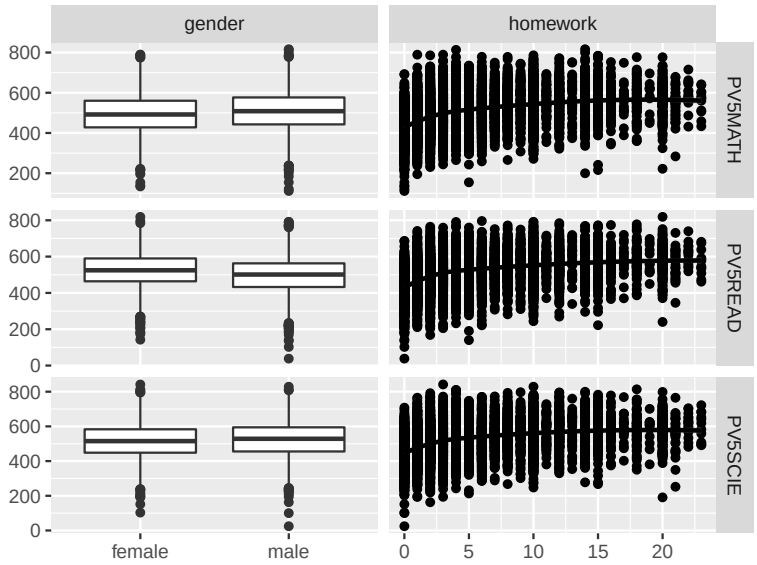


Figure 2.21. 5^{th} grade Australian student scholasitc scores vs their *sex* and hours of weekly homework.

Comparing these two groups if we are interested in knowing if there is any difference in any score output given a child's sex and homework time. A `ggpairs` plot would create within and identity combinations as well as the between combinations of the `ggduo` plot, which is undesired in this situation.

2.4.1 Column types

`ggduo` inspects and displays the data columns according to their variable type: continuous or discrete. There are three plot type groups that can be made from these two options:

- continuous vs. continuous. A scatterplot is a continuous vs. continuous plot.
- continuous vs. discrete. A box plot or grouped box plots is a continuous vs. discrete plot.
- and discrete vs. discrete. A ratio plot, or a plot that can display the `tips`'s *smoker* vs. *day* (eight combinations of 244 records) is a discrete vs. discrete plot.

`ggduo`'s default plotting behavior for continuous vs. continuous, or 'continuous plot', is to produce a scatterplot with a loess smooth curve displayed on top of the points. The default plotting behavior for discrete vs. discrete, or 'discrete plot', is to summarize the data and display it as a ratio plot. A ratio plot displays rectangles whose size is proportional to the counts of the value combination in both the X and Y direction.

The third group, continuous vs. discrete, is refered to as a 'combination plot'. `ggduo` makes a distinction between the two possible combination plots: continuous vs. discrete (vertical combination plot) and discrete vs. continuous (horizontal combination plot). By default, `ggduo` displays grouped histograms for a horizontal combination plot and grouped box plots for a vertical combination plot as in Figure 2.23. `ggpairs` is able to handle the combination plot differently for the upper and lower triangle using the the `upper` and `lower` arguments as in Figure 2.22. Unlike `ggpairs`, a distiction between a horizontal and vertical combination plot is made as there are no upper and lower triangle plot matrix sections in a `ggduo` plot as in Figure 2.24.

```
ggpairs(
  tips,
  c("total_bill", "smoker", "day", "tip"),
  upper = list(combo = "denstrip"),
  lower = list(combo = "facetdensity")
)
```

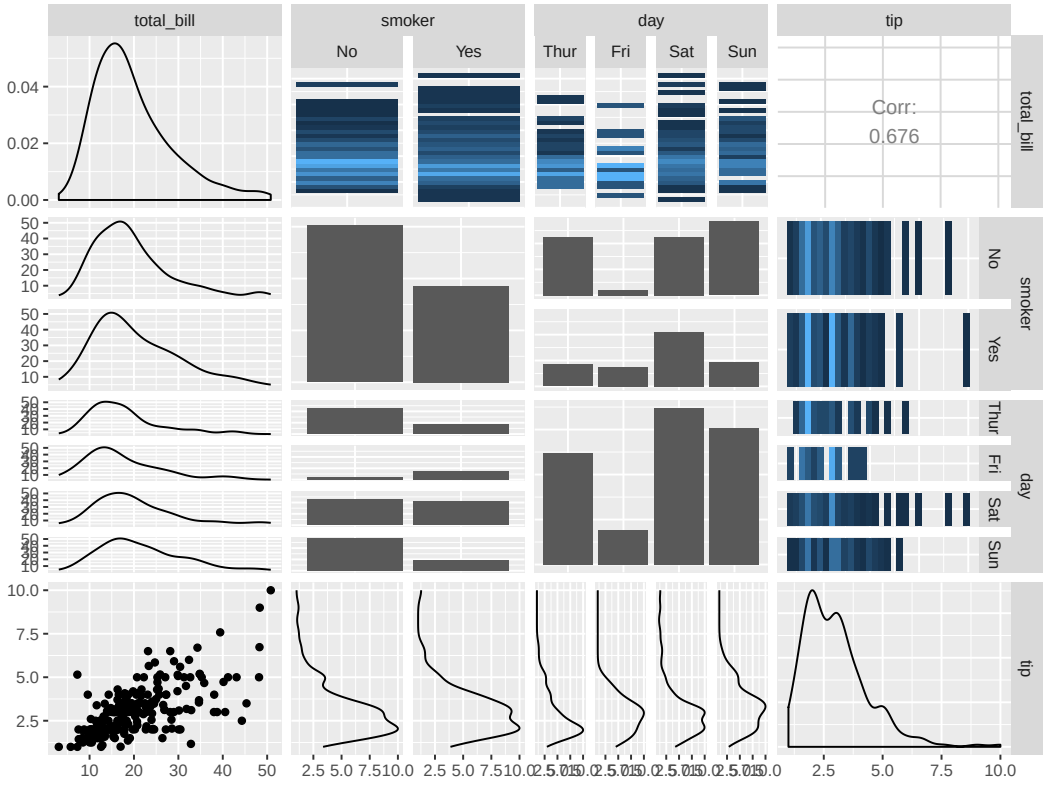


Figure 2.22. Altering the `ggpairs` upper and lower `combo` plot types.

```
ggduo(
  tips,
  c("total_bill", "smoker", "day"),
  c("tip", "time")
)
```

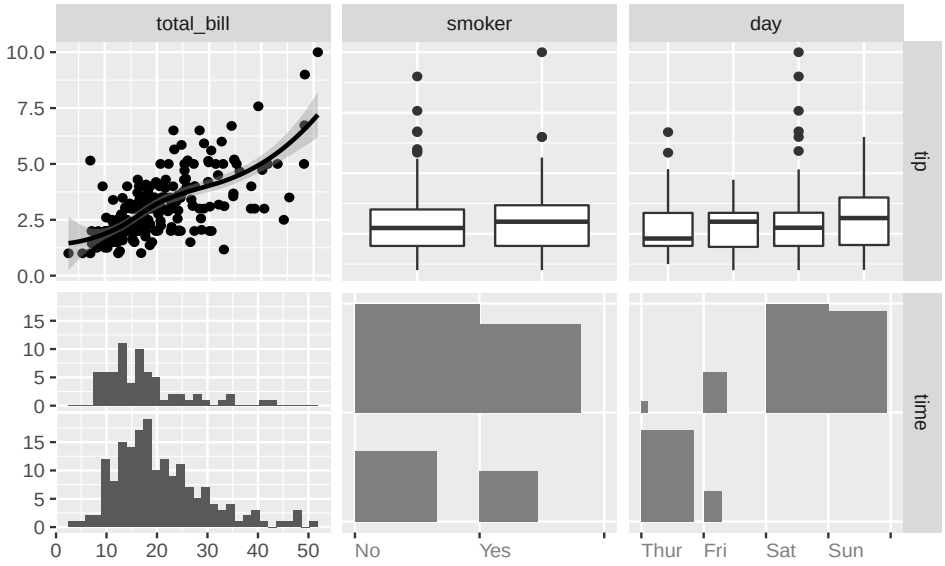


Figure 2.23. two-grouped plot matrix using `ggduo` with no upper or lower triangle areas.

```
ggduo(
  tips,
  c("total_bill", "smoker", "day"),
  c("tip", "time"),
  types = list(
    comboVertical = "denstrip",
    comboHorizontal = "facetdensity"
  )
)
```

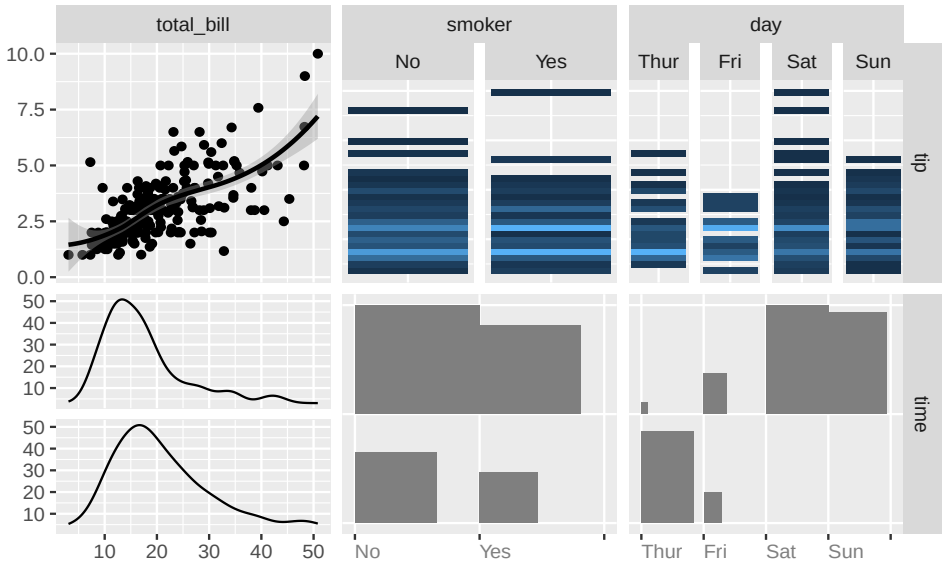


Figure 2.24. Updating the combination types in a `ggduo` plot matrix.

2.4.2 User defined functions

There are many plotting functions are provided by the `GGally` package, however they are not all encompassing. The user may supply their own plotting functions for each panel type. This allows for complete customization of every panel. For example, a violin plot is a combination style plot that is not included by default. A user may create their own function that uses the function call API of `function(data, mapping, ...) { ... }`. A sample custom function example is provided below in Figure 2.25.

```
my_violin <- function(data, mapping, ...) {
  ggplot(data = data, mapping = mapping) +
    geom_violin(...)
}
ggduo(
  tips,
  c("total_bill", "smoker", "day"),
  c("tip", "time"),
  types = list(
    comboVertical = my_violin
  )
)
```

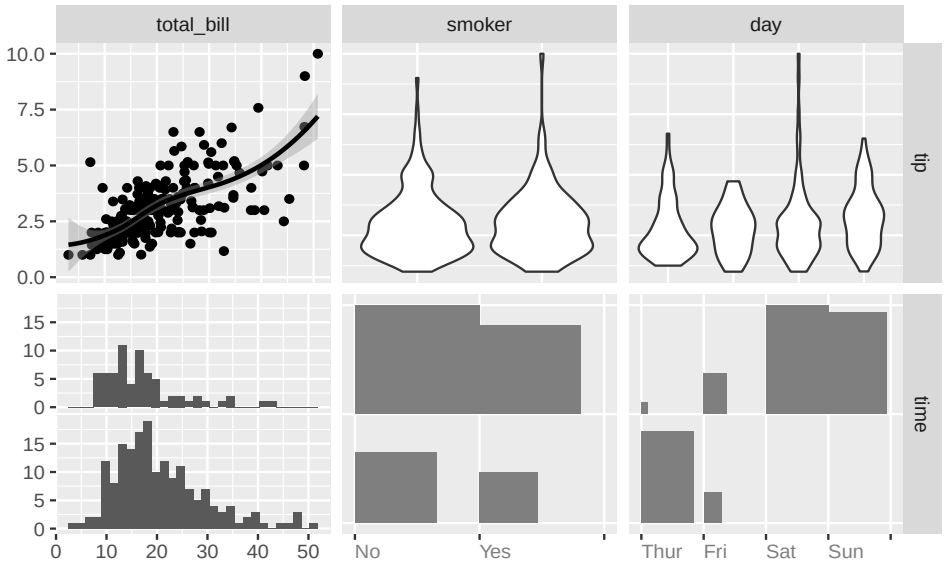


Figure 2.25. Use a custom function to display a plot within a `ggduo` plot matrix.

2.5 `ggduo` in practice

2.5.1 Canonical correlation analysis

Canonical correlation analysis (CCA) is a method to analyize the correlation between two matrices [22]. CCA can be directly displayed with `ggduo`. Before `ggduo`, canonical correlation analysis did not have a cohesive plotting mechanism to visually display the associations of two sets of mixed type variables. Examples used

`ggpairs` to display all pairs of columns when only a subset of combinations are needed. `ggpairs` is a well suited candidate for *within* correlation for the explanatory variables and the response variables. Whereas `ggduo` can be used to check the correlation *between* the explanatory and response variables.

Figure 2.26 is an altered example from the UCLA Intstitute for Digital Research and Education [23]. The website provided an example using `ggpairs` to analyse the *within* corrleation. Continuing their example, we can use `ggduo` to check the *between* correlation. The `psychademic` data consists of 600 records of three psychological variables, four academic variables, and each student's gender. The psychological variables are treated as the response while the academic variables and gender are treated as explantory variables. Figure 2.26 uses `ggduo` to check the correlation between the two sets of columns.

```
ggduo(
  psychademic,
  c("motivation", "locus_of_control", "self_concept"),
  c("read", "write", "math", "science", "sex"),
  showStrips = FALSE,
  types = list(continuous = "smooth_lm")
) +
  labs(
    title = "Between Academic and Psychological Variable Correlation Analysis",
    x = "Psychological",
    y = "Academic"
  )
```

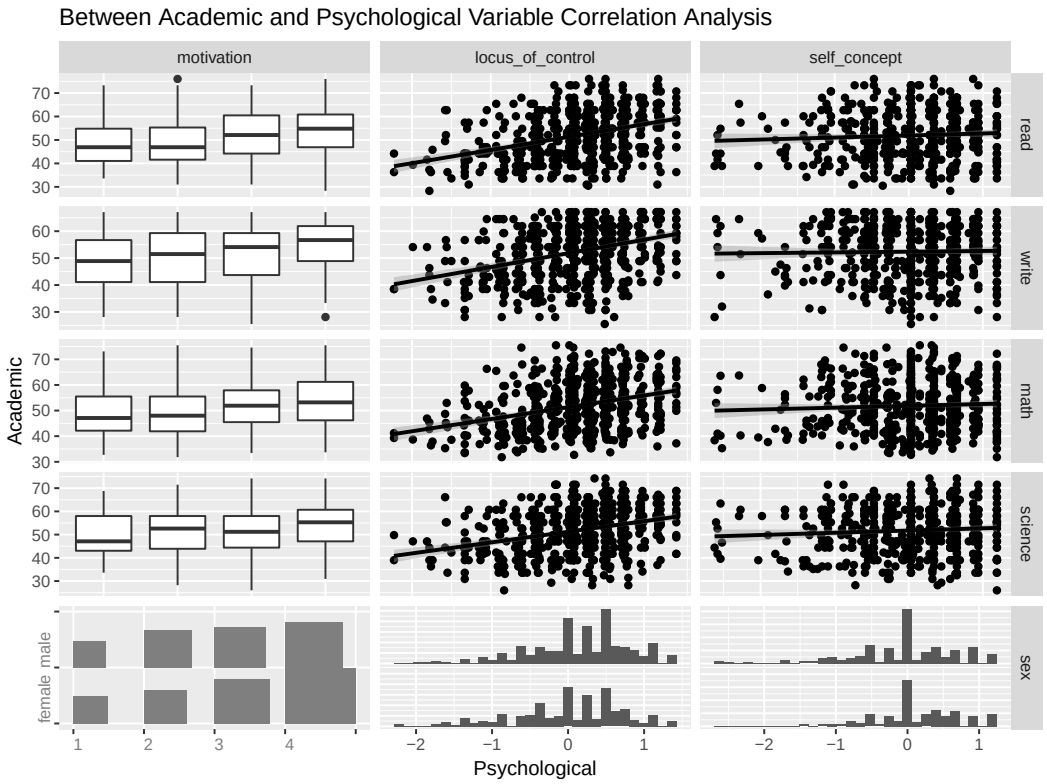


Figure 2.26. `ggduo` plot matrix displaying academic varaibles against psychological varaibles. Continuous vs. continuous plots are displayed with a linear model.

2.5.2 Multiple time series analysis

A multiple time series plot displays the time axis on the X axis with multiple columns on the Y axis. The `stats` [1] package contains the `ts.plot` function that allows for multiple time series to be printed in a single panel sharing the same axes. Displaying the data on the same vertical axis does not make sense for most situations as is shown in Figure 2.27. This is showcased when looking at the first 6 months of half-hourly recorded elctricity demand for Victoria, Australia, in 2014. The electricity demand and temperature should not be displayed on the same scales, as they do not have similar units. Including the $WorkDay$ boolean value does not add to the understanding of the data in the plot.

```
stats::ts.plot(fpp2::elecdemand)
```

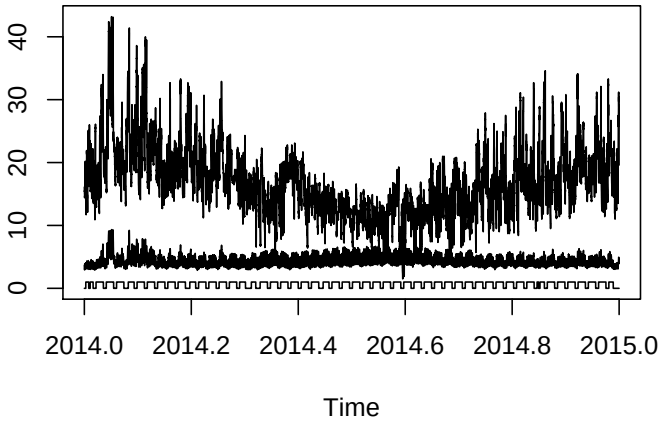


Figure 2.27. Stock `stats` R package time series plot. All variables are displayed on the same axis.

By splitting the multiple time series plot along the Y axis, we can display multiple panels with different Y axes that share the same X axis using `ggduo`. This can be done with the function `ggts` which wraps to `ggduo`. The X column label is turned off by default and has an X label of 'time'. An extra column of the counts of above or below median demand has been added to display mixed Y axes in Figure 2.28.

```
ggts(
  elec_median,
  mapping = aes(color = WorkDay),
  columnsX = "Time",
  columnsY = c("Demand", "Temperature", "HighUsage"),
  columnLabelsY = c(
    "Demand",
    "Temperature",
    "Demand Counts\nAbove Median  |  Below Median"
  ),
  legend = c(3,1),
  showStrips = FALSE,
  types = list(
    comboHorizontal = wrap("facethist", binwidth = 1),
    continuous = wrap("smooth_loess", alpha = 0.1)
```

```
  )
)
```

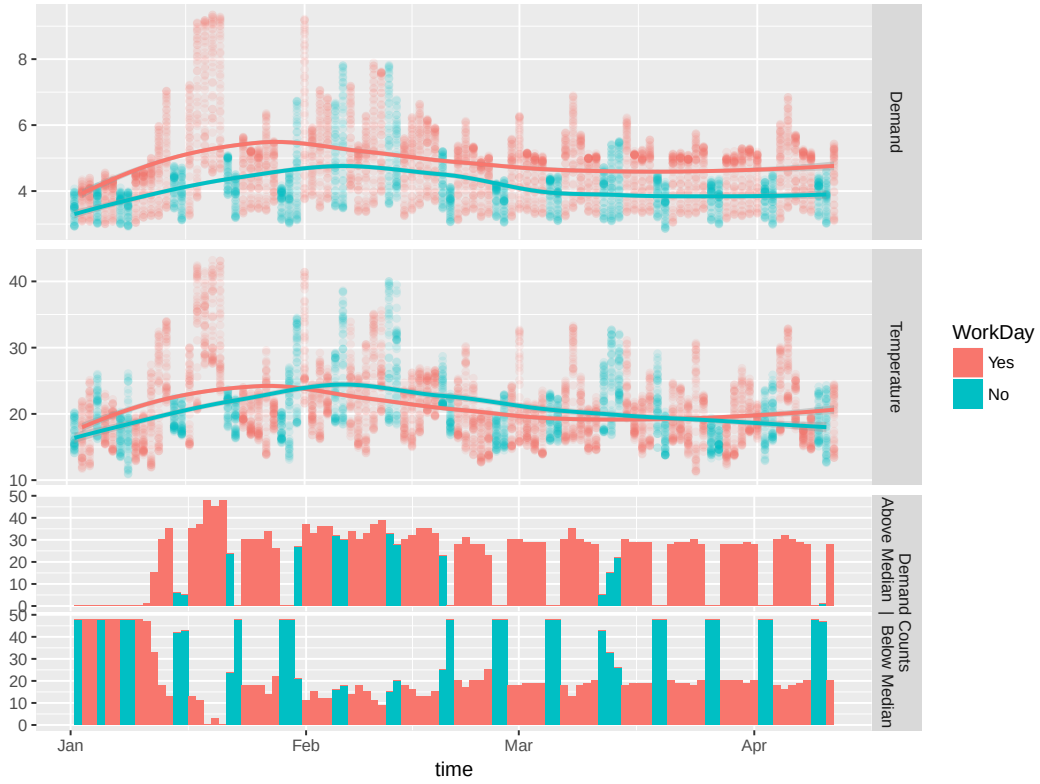


Figure 2.28. Same `elec_median` data, but the data is displayed using the `ggts` function which calls `ggduo`.

2.5.3 Multiple regression diagnostics

With the basis of `ggduo` displaying each row of the data in every panel with different functions, `ggduo` quickly extends to model diagnostics. There are many readily available diagnostics that can be calculated for each row of explanatory data. By default, `ggnostic` (a function that displays a `ggduo` plot matrix) looks at the residuals, leave one out sigma value, leverage points, and Cook's Distance against all model predictor variables. Each piece of diagnostic information is plotted against every explanatory variables used in the model.

Using the flea data set from the `GGally` package, we fit a model to determine the size of the flea's head. Using `stats::step` to determine a good fitting model, the default model diagnostics are displayed against *species*, $tars1$, $tars2$, and *aede*. The model diagnostics are displayed in Figure 2.29 below. Residual panels contain dashed 95% confidence interval lines and a solid line at the expect 0 value. Leave one out sigma value panels display a solid line for the current model's sigma value. Leverage point (diagonal of the hat matrix) panels are centered around the solid, expected line at p/n and have a dashed, *significance* line at $2 * p/n$ [24]. Finally, the Cook's distance panel has a grey dashed, *significance* line at $F_{p,n-p}(0.5)$ [24]. Each solid line corresponds to the expected value and each dashed line corresponds to a *significgance* cuttoff value. The asterisks in the X axis strips correspond to the significance of an anova F test.

```
flea_model <- step(lm(head ~ ., data = flea), trace = FALSE)
ggnostic(flea_model)
```

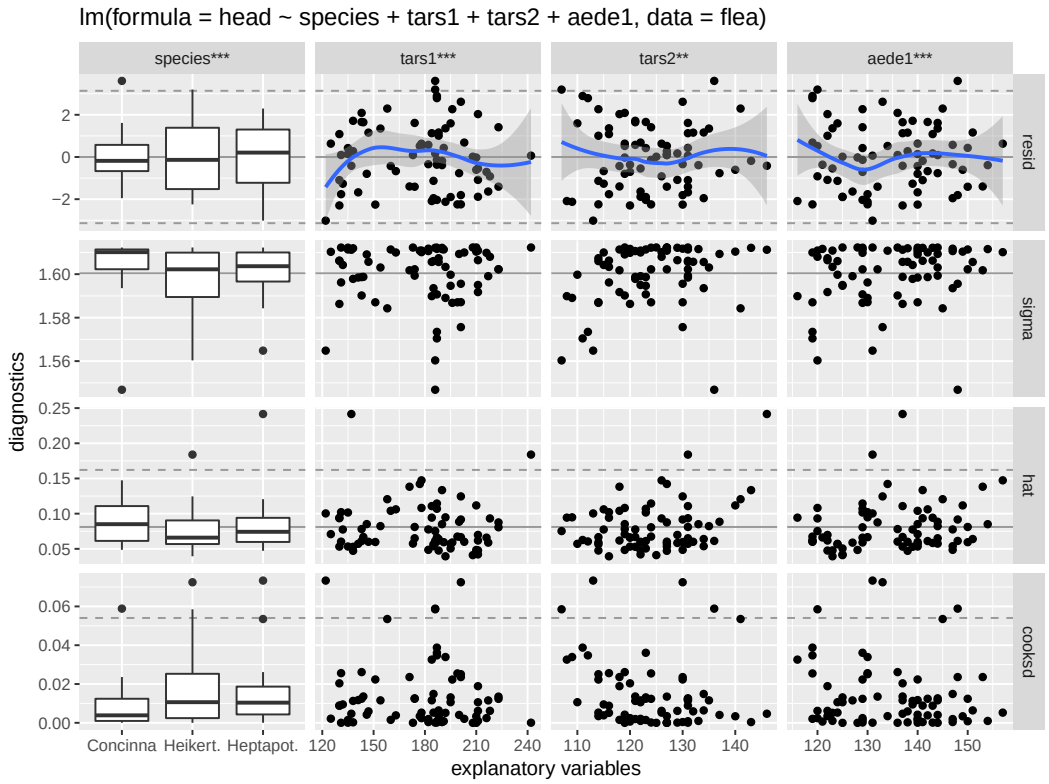


Figure 2.29. Linear model diagnostics for a model predicting the maximal head with in millimeters.

The model diagnostics can be extended further by coloring and grouping according to *species* and displaying the fitted values as in Figure 2.30.

```
ggnostic(
  flea_model,
  mapping = aes(color = species),
  columnsY = c(
    "head", ".fitted", ".se.fit", ".resid", ".std.resid", ".hat", ".cooksd"
  ),
  continuous = list(
    default = ggally_smooth_lm
  ),
  combo = list(
    default = wrap(ggally_box_no_facet, outlier.shape = 21),
    .fitted = wrap(ggally_box_no_facet, outlier.shape = 21),
    .se.fit = wrap(ggally_nostic_se_fit, outlier.shape = 21),
    .resid = wrap(ggally_nostic_resid, outlier.shape = 21),
    .std.resid = wrap(ggally_nostic_std_resid, outlier.shape = 21),
    .hat = wrap(ggally_nostic_hat, outlier.shape = 21),
    .cooksd = wrap(ggally_nostic_cooksd, outlier.shape = 21)
  )
)
```

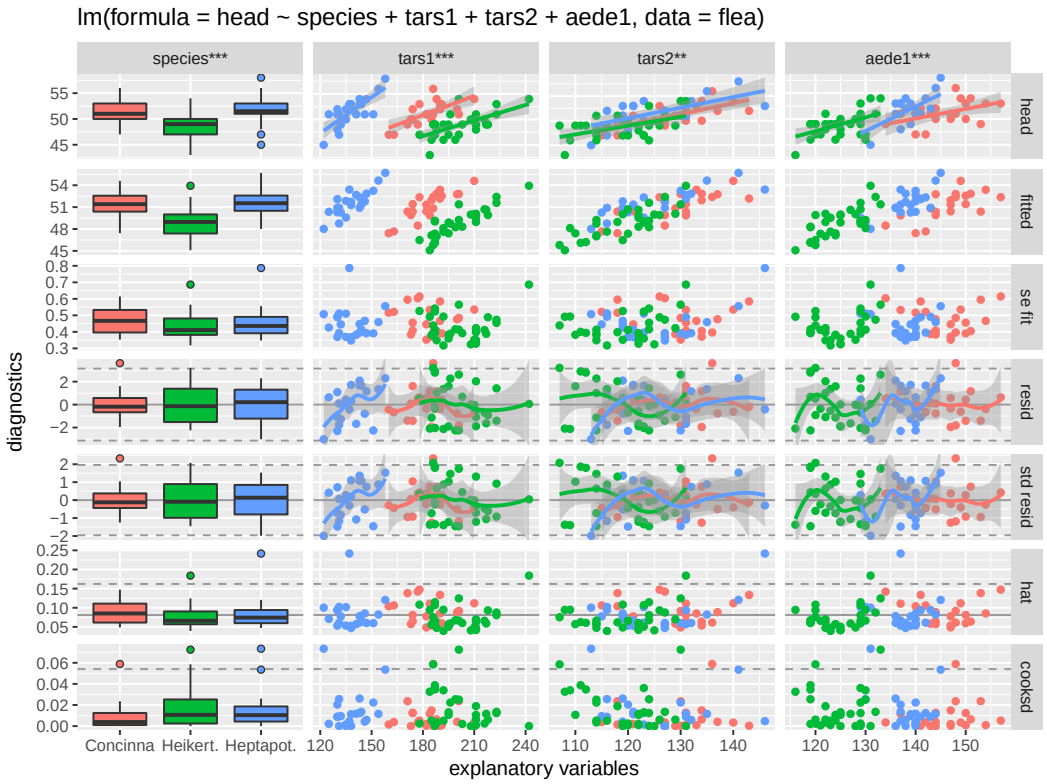


Figure 2.30. Each plot is colored according to *species*. Extra row diagnostics (*head*, *fitted*, and *sefit*) are added to the dianostic plot matrix.

2.6 Summary

`ggduo` generalizes the two group plot matrix by allowing multiple plot types within each panel of the plot matrix. Similar to `ggplot2`, `ggduo` is programatically extended with the `ggts` and `ggnostic` functions to produce plot matrices for different contexts. `ggnostic` showcases `ggduo`'s ability to accept user defined functions to display data for each type of scale combination: continuous, discrete, and combination. `ggduo` is built upon the modular `ggmatrix` allowing sub plots to be retrieved and replaced on command. This modularity and ability to display complex plots with `ggduo` enables users to explore the same data within different sub plots in a cohesive matrix display.

3. AUTOCOGS: METRICS ENABLING DETAILED INTERACTIVE DATA VISUALIZATION

3.1 `trelliscopejs`

`trelliscopejs` is an R package used to visualize data with many conditioning combinations. `trelliscopejs` conditions on columns within the data set and displays a plot, image, or HTML object for each plot in an independent display panel. However, `trelliscopejs`is built to more panels from conditioning combinations than current plotting architectures.

`trelliscopejs` is built to handle more panels than the current plotting frameworks in R. `ggplot2`'s `facet_wrap` can feasibly hold 10s - 100s of displays in one plot. Since `ggplot2` does not paginate (print over multiple pages), the size of the computer screen limits how many panels may be displayed. A core feautre of `lattice` is its ability to paginate panels of a plot. This allows the number of panels to scale 100-1000 times. `lattice` outputs can be saved to PDF and manually inspected page by page. This method is efficient in detecting small visual differences between plots due to the strucutre of small multiples. However, there is a limit to how many plots a human can manually ingest. If 1000 pages were visually inspected at a rate of two pages per second, it would take over 8 minutes to manually flip through each page. Pagination does not scale with large data when there are millions of pages to flip through.

3.1.1 Data size

"Big Data" is a great buzzword, but a poor definition. It is commonly used in different contexts with different meanings. For this chapter, I will define the usage

of different sizes of data in this section. Will define three main sizes of data: Small Data, Medium Data, and Large Data.

Small Data (Memory Data) consists of in memory data only. This includes `data.frames` in R and Excel files. Small data excels at very fast response time when retrieving information. The major disadvantage to Small Data is the size is limited to the amount of memory on a machine. `data.frames` can only get as big as memory can handle. Current machines configurations allow for hundreds of gigabytes of memory.

Medium Data (Disk Data) extends the capabilities of the memory to the storage capacity of the computer. Data is read to and from disk using memory as a buffer. Hard drives today can store multiple terabytes of information. However, retrieving data is much slower as data must be read into memory to be processed. The gain in size comes at a cost of speed.

Finally, Large Data (Cluster Data) is data that is spread across multiple machines. Many machines may be used in a cluster to house Large Data. Large data is the slowest in response time, as data is communicated between machines for calculations. How the data is stored on each machine is up to the data base architecture. Typically each machine stores Medium Data locally, but functions as a cohesive unit globally.

Each class of data balances speed and size to achieve the final goal. These definitions allow for exponential advancement in computing power according to Moore's Law [25] as time advances.

3.1.2 Computation

The split-apply-combine [26] approach for data computation is applicable for all three types of data. As the name states, there are three main steps: split the data, apply a function to the data subsets, and combine the function results. These three steps may be scaled as necessary given computational powers.

1. Split. Data is conditioned on some identifying, or conditioning, columns. This can include the row number (each row is treated uniquely) or may include many existing columns in the data set. Like faceting in `ggplot2`, all conditioning values are considered discrete values. Once the conditioning columns have been selected, the data is split into groups where the conditioning values match.

2. Apply. Once the data frame has been split into independent subsets, a function is applied to each subset. The same function will be applied to all subsets and a similar result will be returned from each functino execution.

3. Combine. With similarly shaped results from each subset, the results will be combined into a final result for further analysis. The uniformity in the result shape makes result combination easy to achieve.

The R package `plyr` [26] implemented the split-apply-combine approach for many kinds of data shapes: `array`, `list`, `vector`, and `data.frame`. `dplyr` [27] has many specific routines to interact with a `data.frame`. Examples in this chapter will be using the `dplyr` package functions.

```
library(dplyr)
library(gapminder)
gapminder
## # A tibble: 1,704 x 6
##        country continent  year lifeExp      pop gdpPercap
##         <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan      Asia  1952  28.801  8425333  779.4453
##  2 Afghanistan      Asia  1957  30.332  9240934  820.8530
##  3 Afghanistan      Asia  1962  31.997 10267083  853.1007
##  4 Afghanistan      Asia  1967  34.020 11537966  836.1971
##  5 Afghanistan      Asia  1972  36.088 13079460  739.9811
##  6 Afghanistan      Asia  1977  38.438 14880372  786.1134
##  7 Afghanistan      Asia  1982  39.854 12881816  978.0114
##  8 Afghanistan      Asia  1987  40.822 13867957  852.3959
##  9 Afghanistan      Asia  1992  41.674 16317921  649.3414
## 10 Afghanistan      Asia  1997  41.763 22227415  635.3414
## # ... with 1,694 more rows
gapminder %>%
  # group by unique country and continent combinations
```

```
  group_by(country, continent) %>%
  # calculate the maximum lifeExp for each combination
  summarise(
    max_lifeExp = max(lifeExp)
  ) %>%
  # print the data.frame
  print() %>%
  # display a histogram plot of the maximum life expectancies
  ggplot(aes(max_lifeExp, fill = continent)) +
    geom_histogram(binwidth = 1)
## # A tibble: 142 x 3
## # Groups:   country [?]
##       country continent max_lifeExp
##        <fctr>    <fctr>       <dbl>
##  1 Afghanistan      Asia      43.828
##  2     Albania    Europe      76.423
##  3     Algeria    Africa      72.301
##  4      Angola    Africa      42.731
##  5   Argentina  Americas      75.320
##  6   Australia   Oceania      81.235
##  7     Austria    Europe      79.829
##  8     Bahrain      Asia      75.635
##  9  Bangladesh      Asia      64.062
## 10     Belgium    Europe      79.441
## # ... with 132 more rows
```

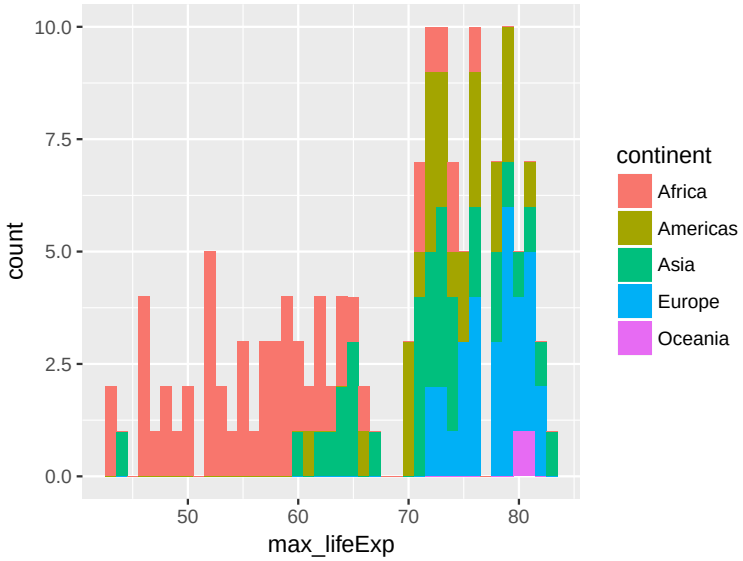


Figure 3.1. Each country's maximum life expectancy value displayed as a histogram with each color representing a continent.

The split-apply-combine paradigm applies to each data size type.

1. Small, In Memory Data: Can use the `plyr` package for computation.

2. Medium, On Disk Data: The R package `dplyr` can be used to connect to a MySQL database stored on disk. Results are executed within the MySQL environment, but returned to the R execution environment. Many other data bases can be connected to R to handle medium sized data.

3. Large, Distributed Data: The R package `Rhipe` [28] or `sparklyr` [29] can be used to execute R commands across multiple compute nodes in a cluster.

Each package implements the split-apply-combine approach to data computation using computational tools built for each scenario. Small data is processed using R. Medium data is processed in a database that is built to handle information larger than memory can hold and results are returned to R. Finally, Large data is executed in the distributed environment and results are stored in the distributed environment. If memory allows, distributed results may be returned to R.

3.1.3 Summary statistics

The `gapminder` data set is an "Excerpt of the Gapminder data on life expectancy, GDP per capita, and population by country" [30]. The 142 countries have data from 1952 to 2007. Figure 3.1 finds the maximum life expectancy for the 142 countries. They are then displayed in a plot colored according to the country's continent. A lot of information may be gleaned from the maximum life expectancy summary plot in Figure 3.1, but a summary plot does not tell the full story of each country's life expectancy over time.

```
gapminder %>%
  filter(country %in% c("Japan", "Switzerland")) %>%
  ggplot(aes(year, lifeExp)) + geom_line() + facet_wrap(~ country) +
    ylim(20, 85) + labs(title = "Two countries with a high life expectancy")
```

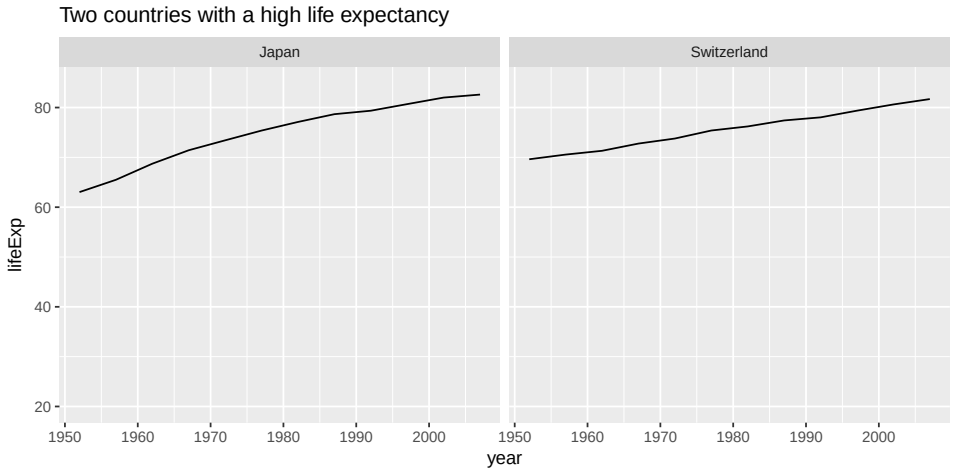


Figure 3.2. Both higher life expectancy countries display linear model trends over time.

```
gapminder %>%
  filter(country %in% c("Afghanistan", "Rwanda")) %>%
  ggplot(aes(year, lifeExp)) + geom_line() + facet_wrap(~ country) +
    ylim(20, 85) + labs(title = "Two countries with lower life expectancy")
```

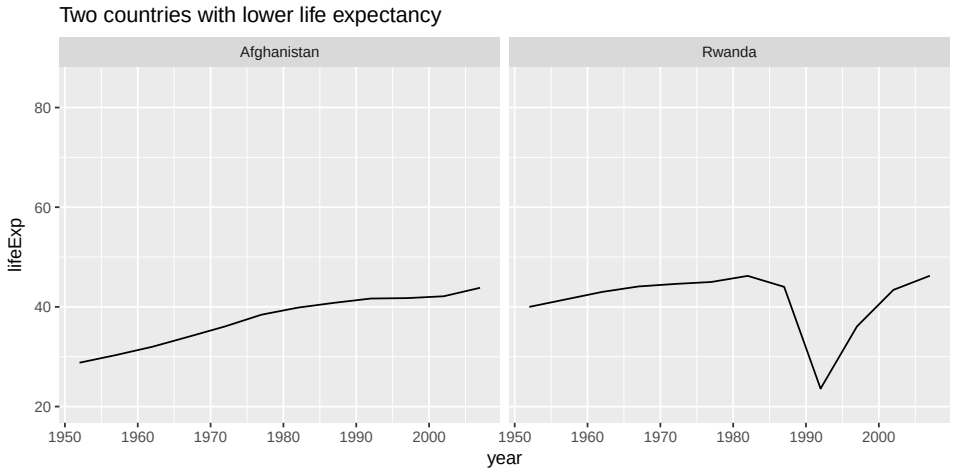


Figure 3.3. Lower life expectancy countries may not always display linear model trends over time.

Figure 3.2 displays two longer living countries, Japan and Switzerland. Japan has a higher maximum life expectancy, but Switzerland had a higher starting life expectancy. Figure 3.3 displays two lower life expectancy countries, Afghanistan and Rwanda. Afghanistan has a lower maximum life expectancy, but steadily increases

over time. Rwanda's life expectancy steadily increased until the 1980's when it dips and recoveres by the 2000's.

Summary statistics are great in interpreting information using less data. However, summary statistics, by their nature, do not convey the full data story and are complimented by data visualization.

`trelliscopejs` allows users to plot full plot detail while allowing users to change how many panels are displayed on the screen at one time, sort the panel ordering, and filter to a smaller subset of panels. `trelliscopejs` achieves these actions by obtaining a plot for every conditioning combination and suplimentary metrics for each plot.

Using the `gapminder` data set in Figure **??**, life expectancy is explored over time with the supplimentary metrics of minimum and maximum life expectancy.

```
gapminder %>%
  group_by(country, continent) %>%
  # condense the data
  tidyr::nest() %>%
  print() ->
gapminder_condensed
## # A tibble: 142 x 3
##        country continent              data
##         <fctr>    <fctr>            <list>
##  1 Afghanistan      Asia <tibble [12 x 4]>
##  2     Albania    Europe <tibble [12 x 4]>
##  3     Algeria    Africa <tibble [12 x 4]>
##  4      Angola    Africa <tibble [12 x 4]>
##  5   Argentina  Americas <tibble [12 x 4]>
##  6   Australia   Oceania <tibble [12 x 4]>
##  7     Austria    Europe <tibble [12 x 4]>
##  8     Bahrain      Asia <tibble [12 x 4]>
##  9  Bangladesh      Asia <tibble [12 x 4]>
## 10     Belgium    Europe <tibble [12 x 4]>
## # ... with 132 more rows
gapminder_condensed %>%
  # add metrics and plots for every conditioning combination
  mutate(
    min_lifeExp = purrr::map_dbl(data, function(dt) min(dt$lifeExp)),
    max_lifeExp = purrr::map_dbl(data, function(dt) max(dt$lifeExp)),
    panel = trelliscopejs::map_plot(data, function(dt) {
      # display a line plot of X:year, Y:life expectancy
```

```
      ggplot(dt, aes(year, lifeExp)) + geom_line() + ylim(20, 85)
    })
  ) %>%
  # remove the condensed data
  select(-data) %>%
  print() ->
gap_trellis
## # A tibble: 142 x 5
##        country continent min_lifeExp max_lifeExp    panel
##         <fctr>    <fctr>       <dbl>       <dbl>   <list>
##  1 Afghanistan      Asia      28.801      43.828 <S3: gg>
##  2     Albania    Europe      55.230      76.423 <S3: gg>
##  3     Algeria    Africa      43.077      72.301 <S3: gg>
##  4      Angola    Africa      30.015      42.731 <S3: gg>
##  5   Argentina  Americas      62.485      75.320 <S3: gg>
##  6   Australia   Oceania      69.120      81.235 <S3: gg>
##  7     Austria    Europe      66.800      79.829 <S3: gg>
##  8     Bahrain      Asia      50.939      75.635 <S3: gg>
##  9  Bangladesh      Asia      37.484      64.062 <S3: gg>
## 10     Belgium    Europe      68.000      79.441 <S3: gg>
## # ... with 132 more rows
```

```
# display the plots and metrics in trelliscopejs
gap_trellis %>% trelljs("gapminder")
## Error in file(con, "r"):  cannot open the connection
```

The `trelliscopejs` HTML widget in Figure ?? displays three rows and five columns of panels. There are `142` panels in total, making 10 pages of panels in total. While this example does not display millions of panels, it does convey the capabilities of the HTML widget. Icons on the left, as in Figure 3.1.3, open foldout displays for panel layout control, turning panel labels on and off, filtering panels using metrics, and panel sorting.

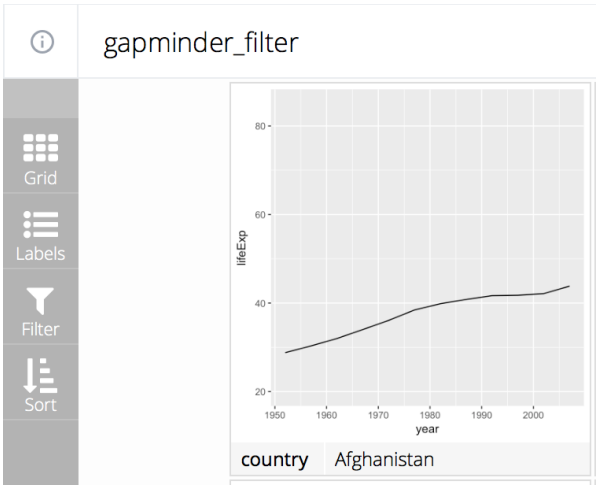


Figure 3.4. The sidebar on the left side of a `trelliscopejs` HTML widget can be opened for panel layout control, displaying panel labels, filtering panels, and sorting panels.

3.2 Cognostics

Displaying panels alone has already been solved with `ggplot2` and `lattice`. Scaling panels beyond `lattice`'s limits is still limited without the use of sorting and filtering the panels. `trelliscopejs`'s power is leveraging subset metrics to organize the panels. These subset metrics are called *cognostics* [12]. Cognostics are univariate statistics calculated for every independent subset of the conditioned data.

Cognostics can be simple summary statistics such as `mean` or `median`, or can be meta data information such as a URL or census information for a conditioned county.

Tukey and Tukey first proposed calculating univariate metrics for scatterplots called scagnostics [12] as a way to describe a scatterplot. Wilkinson et. al. [31] implemented Tukeys' scagnostic definitions in the R package `scagnostics` [32]. Scagnostics can be repurposed as cognostics when applied to every panel containing a scatterplot. These cognostic groupings may then be shown, filtered, and sorted accross the different subset panels.

In `trelliscopejs`, cognostics are displayed as two types of metrics: continuous or discrete. Continuous valued cognostics are filtered using open or closed ranges. Figure 3.2 shows an open range using the `gapminder` country panels that contain a maximum life expectancy value less than or equal to 70 years of age. A closed range example would have both a *from* and a *to* in the selection range, i.e. within 50 to 65 years of age. Currently, `trelliscopejs` does not support more than one range selection for each cognostic variable.

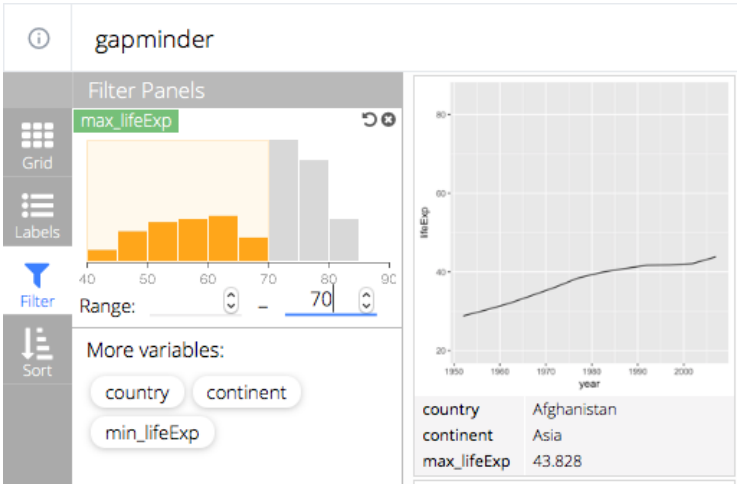


Figure 3.5. A cropped view of `trelliscopejs` filtering on countries whose maximum life expectancy is lower than 70 years old.

With `trelliscopejs`, discrete values are handled using regular expressions or by manually selecting values. When using regular expressions, matching values are displayed immediately. Figure 3.2 displays the immedate results of regular expression on a continent. The immediate feedback confirms whether the regular expression was successful or needs to be updated.

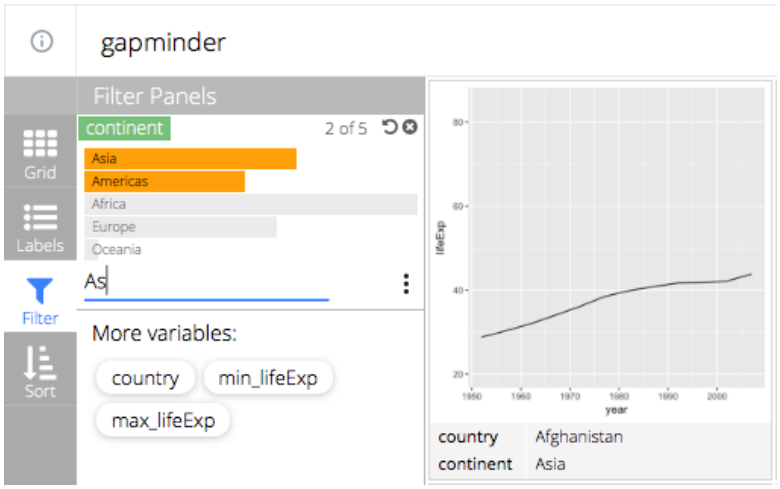


Figure 3.6. A cropped view of `trelliscopejs` filtering on continent who matches the regular expression "as".

Multiple filters are a part of the data analysis process and are embraced in `trelliscopejs`. All of the cognostic filters are applied as a collective, logical *and*.

3.3 Automatic cognostics for data visualization

From a data plotting perspective, we should be able to utilize what is already being displayed in the plot to sort and filter the panels of a `trelliscopejs` widget. This plot information is not readily available as it is calculated within each plotting mechanism.

Using the prior `trelliscopejs` example in Figure **??**, the minimum and maximum values for each country's life expectancy were calculated manually. While the prior example only retrieved the minimum and maximum life expectancy values for each panel, the median and mean values may be of interest as well. These statistics are only looking at the Y variable. There are many more statistics involving both the Y variable and the X variable. In Figure **??**, the X and Y covariance and correlation are added. A linear model (`geom_smooth(method = "lm")`) is also added to visually detect linear trend deviations.

```
gapminder_condensed %>%
  mutate(
    # add metrics
    min_lifeExp = purrr::map_dbl(data, function(dt) min(dt$lifeExp)),
    mean_lifeExp = purrr::map_dbl(data, function(dt) mean(dt$lifeExp)),
    median_lifeExp = purrr::map_dbl(data, function(dt) median(dt$lifeExp)),
    max_lifeExp = purrr::map_dbl(data, function(dt) max(dt$lifeExp)),
    cov = purrr::map_dbl(data, function(dt) cov(dt$year, dt$lifeExp)),
    corr = purrr::map_dbl(data, function(dt) cor(dt$year, dt$lifeExp)),

    # add panel
    panel = trelliscopejs::map_plot(data, function(dt) {
      # display a line plot of X:year, Y:life expectancy
      ggplot(dt, aes(year, lifeExp)) +
        geom_smooth(method = "lm") + # add a linear model
        geom_line() +
        ylim(20, 85)
    })
  ) %>%
  # remove the condensed data
  select(-data) %>%
  print() ->
gap_trellis_plus
## # A tibble: 142 x 9
##        country continent min_lifeExp mean_lifeExp median_lifeExp
##         <fctr>    <fctr>       <dbl>        <dbl>          <dbl>
##  1 Afghanistan      Asia      28.801     37.47883        39.1460
##  2     Albania    Europe      55.230     68.43292        69.6750
##  3     Algeria    Africa      43.077     59.03017        59.6910
##  4      Angola    Africa      30.015     37.88350        39.6945
##  5   Argentina  Americas      62.485     69.06042        69.2115
##  6   Australia   Oceania      69.120     74.66292        74.1150
##  7     Austria    Europe      66.800     73.10325        72.6750
##  8     Bahrain      Asia      50.939     65.60567        67.3225
##  9  Bangladesh      Asia      37.484     49.83408        48.4660
## 10     Belgium    Europe      68.000     73.64175        73.3650
## # ... with 132 more rows, and 4 more variables: max_lifeExp <dbl>,
## #   cov <dbl>, corr <dbl>, panel <list>
gap_trellis_plus %>% trelljs("gapminder_plus")
## Error in file(con, "r"):  cannot open the connection
```

The amount of work to retrieve information that can be readily seen or calculated from the visual display quickly increases. At first, there were two values used. Now there are six values used to explain just the Y data and the $\{X, Y\}$ combination. None of the added metrics explain the linear model added to each panel. As the amount

of layers increase, the number of cognostics needed to explain the plot layer will also increase. Given each plotting panel already contains many statistical cognostics, each panel should be leveraged to generate cognostics automatically.

3.3.1 Linear model example

Using the panel column only, we will automatically derive many cognostics from each panel using the `autocogs` R package.

```
gapminder %>%
  group_by(country, continent) %>%
  tidyr::nest() %>%
  mutate(
    panel = trelliscopejs::map_plot(data, function(dt) {
      # display a line plot of X:year, Y:life expectancy
      ggplot(dt, aes(year, lifeExp)) +
        geom_smooth(method = "lm") +
        geom_line() +
        ylim(20, 85)
    })
  ) %>%
  select(-data) %>% # remove the condensed data
  print() ->
gap_panel
## # A tibble: 142 x 3
##        country continent    panel
##         <fctr>    <fctr>   <list>
##  1 Afghanistan      Asia <S3: gg>
##  2     Albania    Europe <S3: gg>
##  3     Algeria    Africa <S3: gg>
##  4      Angola    Africa <S3: gg>
##  5   Argentina  Americas <S3: gg>
##  6   Australia   Oceania <S3: gg>
##  7     Austria    Europe <S3: gg>
##  8     Bahrain      Asia <S3: gg>
##  9  Bangladesh      Asia <S3: gg>
## 10     Belgium    Europe <S3: gg>
## # ... with 132 more rows
autocogs::add_panel_cogs(gap_panel)
## # A tibble: 142 x 9
##        country continent    panel        `_smooth`             `_lm`
##         <fctr>    <fctr>   <list>           <list>             <list>
##  1 Afghanistan      Asia <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
```

```
##  2     Albania    Europe <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  3     Algeria    Africa <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  4      Angola    Africa <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  5   Argentina  Americas <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  6   Australia   Oceania <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  7     Austria    Europe <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  8     Bahrain      Asia <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
##  9  Bangladesh      Asia <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
## 10     Belgium    Europe <S3: gg> <tibble [1 x 3]> <tibble [1 x 19]>
## # ... with 132 more rows, and 4 more variables: `_x` <list>,
## #   `_y` <list>, `_bivar` <list>, `_n` <list>
```

For each panel, `2 + 19 + 5 + 5 + 2 + 5 = 38` cognostics were auto generated to aid in exploration of the panels.

- Ten of the cognostics calculate information individual column information. (`_x` , `_y`)
- Two cognostics calculate continuous bivariate metrics. (`_bivar`)
- Five cognostics calculate metrics involving the number of points and their availbility. (`_n`)
- Two cognostics calculate metrics for the smooth line added to each panel. (`_smooth`)
- Nineteen cognostics are calculated for the linear model applied to the panel. These metrics will be dicussed in greater detail later. (`_lm`)

Each univariate grouping is labeled to provide context as to what is calculated and which variables were used.

3.3.2 Framework

`autocogs` package is built to provide a consistent framework for calculating cognostics independent of the class of the plotting object supplied. Ideally, it should work for all major layer-based visualization packages (such as `ggplot2`, `rbokeh` [8],

and `plotly` [9]) and produce the same result for similar plot displayed in the different plotting packages. Currently, only `ggplot2` hooks have been installed, but `rbokeh` and `plotly` can be added using the publically available functions in the `autocogs` package.

By definition, each subset panel within a `trelliscopejs` widget will contain the same plotting layers, but with different data. Addressing each layer will produce the same group of output cognostics but with different values. There is a "one to many" mapping from plot to plot layers and a "one to many" mapping from each plot layer to the cognostic groups. The same final cognostics may be produced from many different layers as shown in Figure 3.3.2.

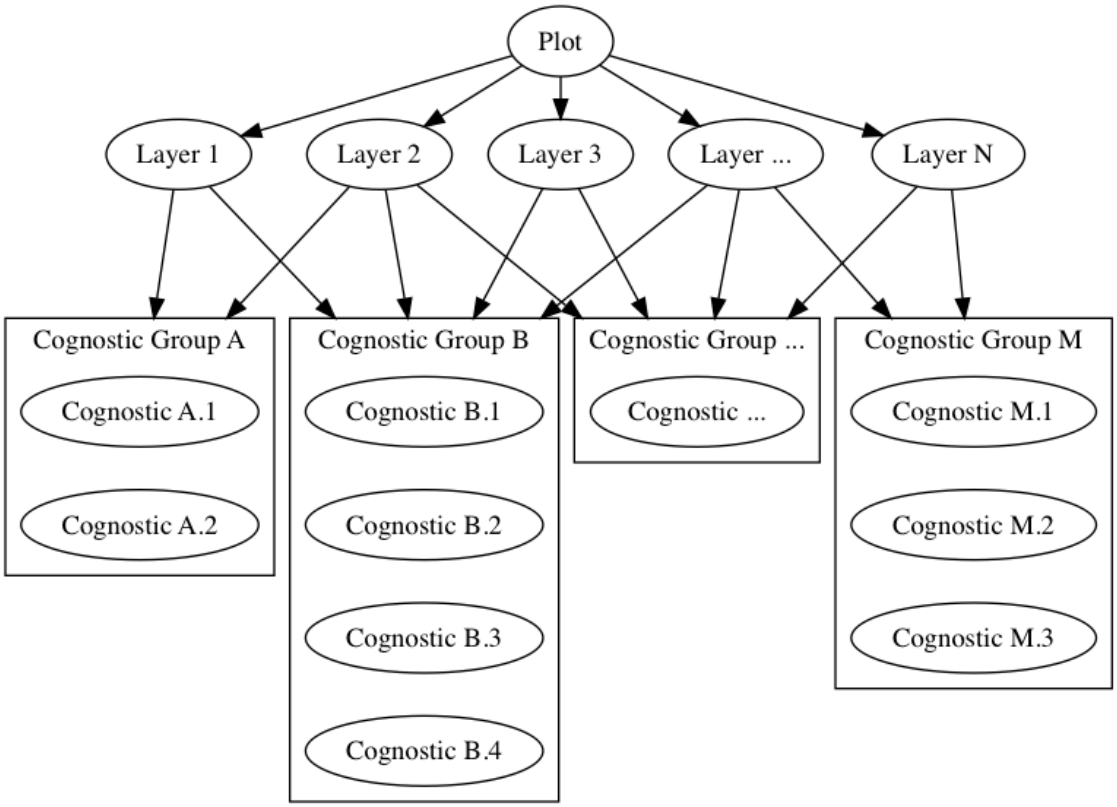


Figure 3.7. Theoretical framework of how multiple cognostic groups can be connected to multiple plot layers.

Once all the cognostics have been produced for a given plot, the cognostic groups are reduced to keep only the unique cognostic name and value combinations. Using Figure 3.3.2, the cognostic groups $\{A, B, ..., M\}$ will only be returned once.

3.3.3 Cognostic groups

Each set of cognostics is returned in a cognostic group. This is similar to scagnostics. All scagnostics pertain to scatterplots. Likewise, the number of non-NA X, non-NA Y, and non-NA X and Y points pertain to non-NA counts of the displayed data. These groups can be extended to each type of statistical display: box plot, histogram, linear model, etc.

The cognostics produced in the `autocogs` `gapminder` linear model example in subsection 3.3.1 were:

- `_smooth`: Two cognostics pertaining to a "smooth" line being added to the panel.
- `_lm`: Nineteen cognostics pertaining to the linear model line added to the panel.
- `_x, _y` Five cognostics pertaining only to the X and Y values respectively.
- `_bivar` Two cognostics pertaining only to both the X and Y values.
- `_n` Five cognostics pertaining only to counts of X and Y values.

Each cognostic group column contains a single row `data.frame` nested in each cell. Nesting data structures as list-columns [33] is considered an advanced R technique, but by nesting the cognostic `data.frame`s in each cell, `autocogs` maintains the tidy data input with tidy data cognostics. Tidy data is defined to have variables in each column, observations for each row and each cell contains a value. The value does not need to conform to atomic values which allows for complex structures as long as the "tidy data" rules are maintained as shown in Figure 3.3.3.

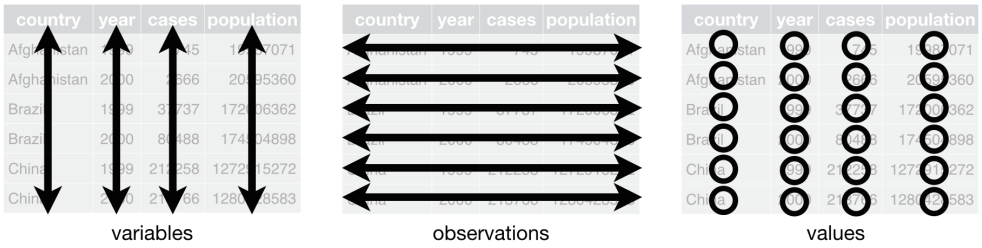


Figure 3.8. Figure courtesy of [33]. There are three rules to tidy data: columns contain variables, rows contain observations, and cells contain values.

3.4 Cognostic groups

There are three main types of cognostic groups: Univariate, Bivariate, and Counts. Each type of cognostic group will explain in detail their corresponding cognostic groups in detail throughout this section.

3.4.1 Univariate

The univariate cognostic groups examples will use the same data set of the maximum life expectancy of the `gapminder` data set.

```
gapminder %>%
  group_by(country, continent) %>%
  summarise(lifeExp = max(lifeExp)) %>%
  print() ->
gap_max
## # A tibble: 142 x 3
## # Groups:   country [?]
##        country continent lifeExp
##         <fctr>    <fctr>   <dbl>
##  1 Afghanistan      Asia  43.828
##  2     Albania    Europe  76.423
##  3     Algeria    Africa  72.301
##  4      Angola    Africa  42.731
##  5   Argentina  Americas  75.320
##  6   Australia   Oceania  81.235
##  7     Austria    Europe  79.829
```

```
##  8     Bahrain      Asia  75.635
##  9  Bangladesh      Asia  64.062
## 10     Belgium    Europe  79.441
## # ... with 132 more rows
```

Univariate Continuous Cognostics

Univariate continuous cognostics utilize the standard statistical calculations: minimum, maximum, mean, median, and variance. Each value is quickly interpretable and provides a good starting point when filtering panels within a `trelliscopejs` widget.

```
library(autocogs)
auto_cog("univariate_continuous", gap_max$lifeExp)
## # A tibble: 1 x 5
##      min    max     mean  median      var
##    <dbl>  <dbl>    <dbl>   <dbl>    <dbl>
## 1 42.568 82.603 68.03542 71.9355 116.3098
```

Univariate Discrete Cognostics

Univariate discrete values only have counts and names to determine the cognostics. The minimum and maximum count values are reported, as well as the mean count value. For both the minimum and maximum values, a corresponding name is reported alongside the count. The example below shows that the continent `"Africa"` contains the most countries at `52` and `"Oceania"` contains least amount of countries at `2` countries.

```
auto_cog("univariate_discrete", gap_max$continent)
## # A tibble: 1 x 5
##   min_name count_min count_mean count_max max_name
##      <chr>     <int>      <dbl>     <int>    <chr>
## 1  Oceania         2       28.4        52   Africa
```

Continuous Density Cognostics

Continuous density cognostics revolve around the shape of the density. The maximum density value and its corresponding location are returned for comparison across cognostic calculations. Skew and kurtosis [34] are computed to help distinguish the density shape. The p value of Hartigans' dip test from `diptest` [35] for unimodality is reported as well. To help locate where the maximum density occurs, the maximum density value and location are provided. Finally, the number of clusters may be calculated using the `mclust` [36] [37] R package. Cluster calculations are not included by default due to their slower computational speed.

```
auto_cog("density_continuous", gap_max$lifeExp, clusters = TRUE)
## # A tibble: 1 x 6
##   max_density max_density_location clusters unimodal_p_value
##         <dbl>                <dbl>    <int>            <dbl>
## 1  0.04425023             75.22066        3        0.2994391
## # ... with 2 more variables: skew <dbl>, kurt <dbl>
```

Boxplot Cognostics

Boxplot metrics include lower whisker, Q1, median, Q3, and upper whisker locations. The number of outliers above the boxplot and below the boxplot are also reported.

```
auto_cog("boxplot", gap_max$lifeExp)
## # A tibble: 1 x 7
##   n_outlier_lower lower_whisker       q1  median       q3
##             <int>         <dbl>    <dbl>   <dbl>    <dbl>
## 1               0       59.6945 69.71875 71.9355 74.15225
## # ... with 2 more variables: upper_whisker <dbl>,
## #   n_outlier_upper <int>
```

Quantile Quantile Cognostics

Quantile-Quantile plots display the theoretical distribution quantiles verses the sample quantile points. Two variables to help determine skewness count how many

points are above and below a non robust quantile line. The non robust quantile line is calculated using the 25^{th} and 75^{th} percentiles of sample points, rather than the a robust linear model. This calculation replicates the base R function `stats::qqline` behavior. A p value for the Kolmogorov-Smirnov test is added to the result to determine how close the sample points come from the test distribution. The test distribution defaults to the normal distribution. Finally, a mean squared error from the quantile line is reported for the Quantile-Quantile plot. This value should be comparable to all of the independent samples as each sample should come from the same distribution. Larger mean squared error values help discover outliers in the distribution.

```
auto_cog("quantile_quantile", gap_max$lifeExp)
## # A tibble: 1 x 4
##   points_above points_below ks_test   qq_mse
##          <int>        <int>   <dbl>    <dbl>
## 1           87           55       0 13.78215
```

3.4.2 Bivariate

The cognostic groups below is calculated using the `gapminder` data set where continent equals `"Americas"` (`americas`) or where country equals `"United States"` (`usa`).

```
americas <- gapminder %>% filter(continent == "Americas")
usa <- gapminder %>% filter(country == "United States")
```

Bivariate Continuous Cognostics

Similar to the Univariate Continuous cognostic group, the Bivariate Continuous cognostic group calculates the two standard bivariate summary statistics: covariance and correlation.

```
auto_cog("bivariate_continuous", usa$year, usa$lifeExp)
```

```
## # A tibble: 1 x 2
##   covariance correlation
##        <dbl>       <dbl>
## 1     59.855   0.9929351
```

Scagnostics Cognostics

Scatterplot scagnostics are a pre-existing cognostic group for a continuous bivariate plot. The following scagnostics are explained in more detail in [31].

- Outlying: the proportion of the total edge length due to extremely long edges connected to points of single degree.
- Skewed: the distribution of edge lengths of a minimum spanning tree gives us information about the relative density of points in a scattered configuration.
- Clumpy: the Hartigan and Mohanty RUNT statistic is most easily understood in terms of the single-linkage hierarchical clustering tree called a dendrogram.
- Sparse, the 90% quantile of the edge lengths of the minimum spanning tree.
- Striated: the summation of angles over all adjacent edges of a MST.
- Convex: the ratio of the area of the alpha hull and the area of the convex hull.
- Skinny: the ratio of perimeter to area of a polygon measures.
- Stringy: the ratio of width to length of a network.
- Monotonic: squared Spearman correlation coefficient.

```
auto_cog("scagnostics", americas$year, americas$lifeExp)
## # A tibble: 1 x 9
##     Outlying    Skewed     Clumpy    Sparse  Striated    Convex
##        <dbl>     <dbl>      <dbl>     <dbl>     <dbl>     <dbl>
## 1 0.01443262 0.7695572 0.07343219 0.0799376 0.5539216 0.8622428
## # ... with 3 more variables: Skinny <dbl>, Stringy <dbl>,
## #   Monotonic <dbl>
```

Continuous 2D Density Cognostics

The Continuous 2-Dimensional Density cognostic group reports the maximum density. Similar to the Continuous Density cognostic group, the X and Y location are returned. Cluster count calculations are turned off by default due to slow computation speed.

```
auto_cog(
  "density_2d_continuous",
  americas$year, americas$lifeExp,
  clusters = TRUE
)
## # A tibble: 1 x 4
##   max_density max_density_x max_density_y clusters
##         <dbl>         <dbl>         <dbl>    <int>
## 1  0.00118169      1997.435      71.86533        9
```

Bivariate Step Cognostics

A stepwise plot displays a "stair case" like plot where the connecting line only moves in parallel ot the X or Y axis. The Bivariate Step cognostic group returns the number of steps completed, as well as calculating the Univariate Continuous cognostics for the step width and step height.

```
auto_cog("bivariate_step", usa$year, usa$lifeExp)
## # A tibble: 1 x 11
##   steps min_step_width mean_step_width median_step_width
##   <dbl>          <int>           <dbl>             <int>
## 1    11              5               5                 5
## # ... with 7 more variables: max_step_width <int>,
## #   var_step_width <dbl>, min_step_height <dbl>,
## #   mean_step_height <dbl>, median_step_height <dbl>,
## #   max_step_height <dbl>, var_step_height <dbl>
```

Smooth Line Cognostics

The Smooth Line cognostic group is a baseline for all model based lines added to a plot. The smooth line calculations pair well with the Linear Model and Loess Model

cognostic groups. A mean squared error and the max deviation with its corresponding location are reported.

```
auto_cog("smooth_line", usa$year, usa$lifeExp)
## # A tibble: 1 x 3
##          mse max_deviation max_deviation_location
##        <dbl>         <dbl>                  <int>
## 1 0.05389673       0.45948                   1972
```

Linear Model Cognostics

Linear Model cognostics leverage many existing statistics. Knowing that the model will only be a simple linear model allows `autocogs` to return slope and intercept values and corresponding p values. Many cognostics can be generated about the model fit using known diagnostic methods:

- R^2: fraction of variance explained by the model
- σ: square root of the estimated residual variance
- F-statistic: the linear model's F-statistic and corresponding p value.
- Degrees of freedom: how many degrees of freedom in the model and residuals
- Log-likelihood value: the log likelihood value of the model
- AIC, BIC: Akaike's Information Criterion and Schwarz's Bayesian Criterion
- Deviance: the quality-of-fit statistic of the model

There are a few extra diagnostics that perform extra calculations to conform to the univariate requirement of a cognostic.

- Cook's distance: a combination of each points leverage and residual value. Values larger than $F_{p,n-p}(0.5)$ indicate influential data points in the model [24]. The number of influential data points is reported.

- Influence points: the diagonal of the Hat Matrix is how much influence a point has on the model. Each point is expected to equal $\frac{p}{n} = \frac{1}{n}$, with influential points having a value larger than $\frac{2 * p}{n} = \frac{2}{n}$ [24]. The sum of all influential points is reported.

- Shapiro-Wilk test: when using a linear model, the residuals are assumed to come from a normal distribution. The Shapiro-Wilk test tests the residuals against the normal distribution [24]. The corresponding p value is reported.

- Box Cox power transformation: the lower and upper bounds of the 95% confidence interval of the Box Cox power transformation are reported. This transformation is used to stabilize the variance of a linear model. If the confidence interval contains 0, a ln transformation may be used [24].

```
auto_cog("linear_model", usa$year, usa$lifeExp) %>% as.data.frame()
##   intercept intercept_p_value      beta beta_p_value        r2
## 1 -291.0845      1.254513e-09 0.1841692 1.369788e-10 0.9859202
##       sigma statistic      p_value           df   log_lik      aic
## 1 0.4161339  700.2351 1.369788e-10 1.369788e-10 -5.412356 16.82471
##        bic deviance df_residual n_sig_cooks n_sig_hat resid_shapiro
## 1 18.27943 1.731675          10           0         6     0.9843222
##   bc_lower bc_upper
## 1     -1.3        2
```

Loess Model Cognostics

Similar to the linear model, a simple loess model is calculated and the model diagnostics are reported:

- Supplied parameters: the supplied parameters of `span`, the alpha parameter which controls the degree of smoothing, and `degree`, the polynomial degree used in the loess model, are returned.

- Calculated parameters: the effective number of parameters, `enp`, is returned along with the trace of the hat matrix, `trace.hat`. Finally, the sigma value, `s`, of the loess model variance is returned as a single cognostic.

- Iterations: The number of iterations needed to calculate the model are reported as well.

```
auto_cog("loess_model", usa$year, usa$lifeExp)
## # A tibble: 1 x 6
##        enp         s trace.hat  span degree iterations
##      <dbl>     <dbl>     <dbl> <dbl>  <int>      <int>
## 1 4.540734 0.3147398  5.005914  0.75      2          1
```

3.4.3 Counts

The third cognostic type addresses counts of binned data.

Univariate and Bivariate Count Cognostics

Both the Univariate and Bivariate Count cognostics address how many values are `NA` and not `NA`. Bivariate accounts for both the logical AND and OR of the X and Y values being `NA`.

```
auto_cog("univariate_counts", americas$lifeExp)
## # A tibble: 1 x 2
##       n  n_na
##   <int> <int>
## 1   300     0
auto_cog("bivariate_counts", americas$year, americas$lifeExp)
## # A tibble: 1 x 5
##       n n_both_na n_or_na n_x_na n_y_na
##   <int>     <int>   <int>  <int>  <int>
## 1   300         0       0      0      0
```

Pairwise Counts Cognostics

Pairwise Counts cognostics address how often pieces of information occur. Pairwise Counts look at the combinations of the X and Y variables. The Univariate Continous cognostics are reported on the counts of the pairwise combinations.

```
auto_cog("pairwise_counts", americas$country, round(americas$lifeExp))
## # A tibble: 1 x 6
##     min   max     mean median       var na_count
##   <int> <int>    <dbl>  <int>     <dbl>    <dbl>
## 1     1     4 1.083032      1 0.1198922        0
```

Count Testing Cognostics

There are three types of count testing cognostic groupings. In addition to the Univariate Continuous cognostic information, each cognostic group tests whether or not the variables involved in the plot have any effect on the number of counts. This is calculated using a χ^2 test where $H_0 = Y\ \mu$.

- Histogram Counts Cognostics

  The univariate case calculates the counts of a histogram using a default binwidth of 30 equal bin widths. The default width of 30 matches the default width in `ggplot2`.

```
auto_cog("histogram_counts", americas$lifeExp)

## # A tibble: 1 x 6
##   count_min count_max count_mean count_median count_var        chisq
##       <dbl>     <dbl>      <dbl>        <dbl>     <dbl>        <dbl>
## 1         0        26         10            8  55.24138 4.240986e-20
```

- Square Counts Cognostics

  The bivariate case of the histogram is a 2D square grid. The counts in each of the grid spaces are calculated using a default binwidth of 30 equal intervals along each axis. In the example below, six equally spaced bins along each axis are used.

```
auto_cog(
  "square_counts",
  americas$year, americas$lifeExp,
  bins = 6
)
```

```
## # A tibble: 1 x 6
##   count_min count_max count_mean count_median count_var        chisq
##       <dbl>     <dbl>      <dbl>        <dbl>     <dbl>        <dbl>
## 1         1        29      9.375          6.5  53.79032 1.547182e-22
```

- Hexagon Counts Cognostics

  The second bivariate histogram has a honeycomb like array of hexagons. Like the Square Counts Cognostics, the Hexagon Counts Cognostics default to 30 equally spaced hexagons along each axis. The example below also uses six hexagon shaped bins along each axis.

```
auto_cog("hex_counts", americas$year, americas$lifeExp, bins = 6)
```

```
## # A tibble: 1 x 6
##   count_min count_max count_mean count_median count_var        chisq
##       <dbl>     <dbl>      <dbl>        <dbl>     <dbl>        <dbl>
## 1         1        25   7.894737            6  38.04267 2.131691e-20
```

3.5 `ggplot2` layer matching

`ggplot2` has already been integrated into the `autocogs` R package. This integration contains the mapping of each cognostic group to each `ggplot2` geom layer. Each plotting framework has its own data personality and may display cognostics how it sees fit.

Figure 3.5 is a mapping of each `ggplot2` geom layer to each of the univariate cognostics. Geoms that do not map to any cognostic groups are considered building block geoms and do not produce any cognostics when added as a layer.

In this section, I will explore a simple `ggplot2` histogram and a more advanced example involving a `ggplot2` scatterplot and linear model.

3.5.1 Histogram

First, let us create the data set containing each of the panels. Each panel will display a histogram chart of the life expectancy over time for each continent. Figure 3.10 displays the panel where continent is equal to `"Americas"`.

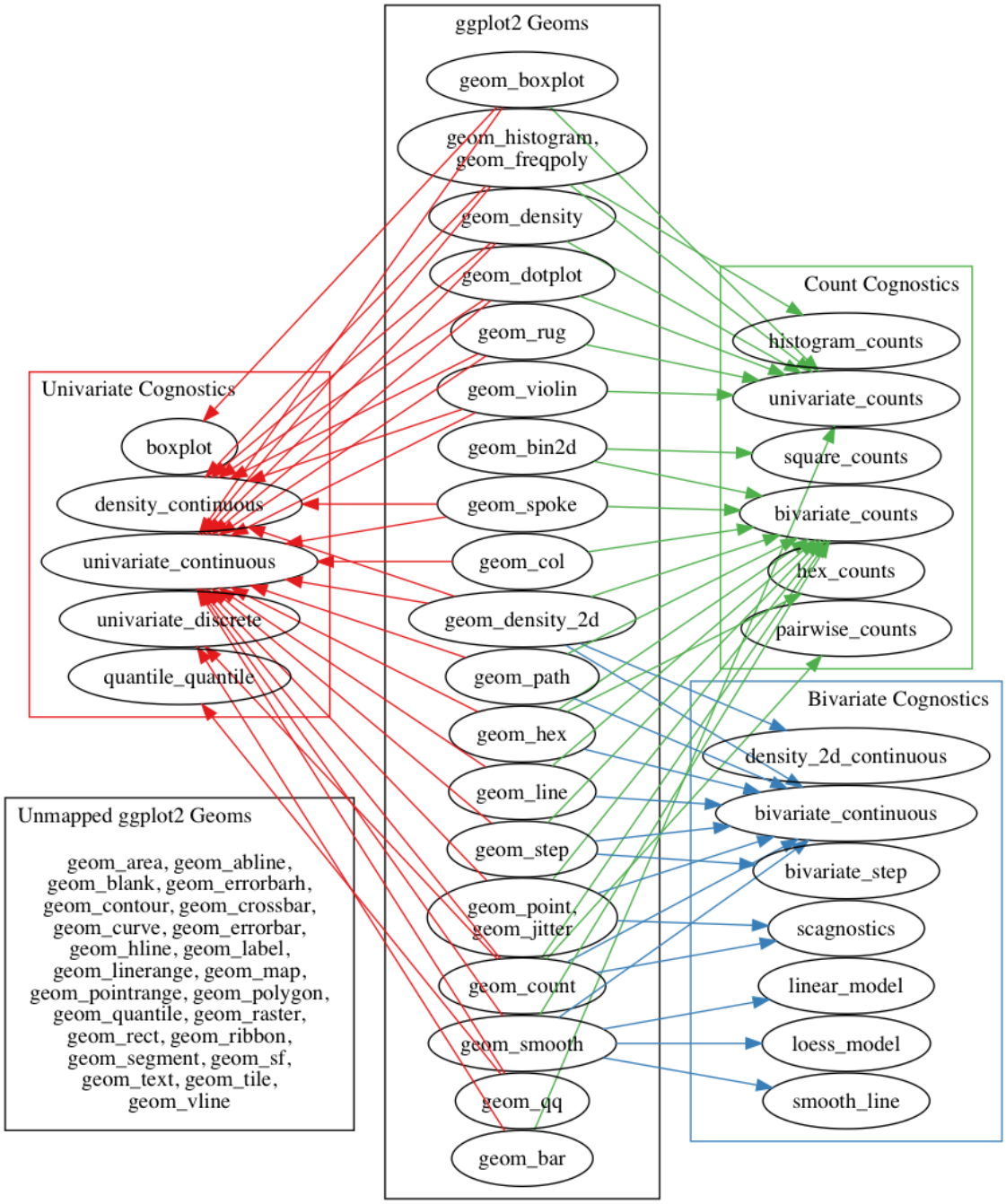


Figure 3.9. Mapping of `ggplot2` geoms to Univariate, Bivariate, and Count cognostic groups

```
gapminder %>%
  group_by(continent) %>%
  do(
    panel = ggplot(., aes(lifeExp)) + geom_histogram(binwidth = 1)
  ) %>%
  print() ->
continent_hists
## Source: local data frame [5 x 2]
## Groups: <by row>
##
## # A tibble: 5 x 2
##   continent    panel
## *    <fctr>   <list>
## 1    Africa <S3: gg>
## 2  Americas <S3: gg>
## 3      Asia <S3: gg>
## 4    Europe <S3: gg>
## 5   Oceania <S3: gg>
americas_pos <- which(continent_hists$continent == "Americas")
continent_hists$panel[[americas_pos]]
```

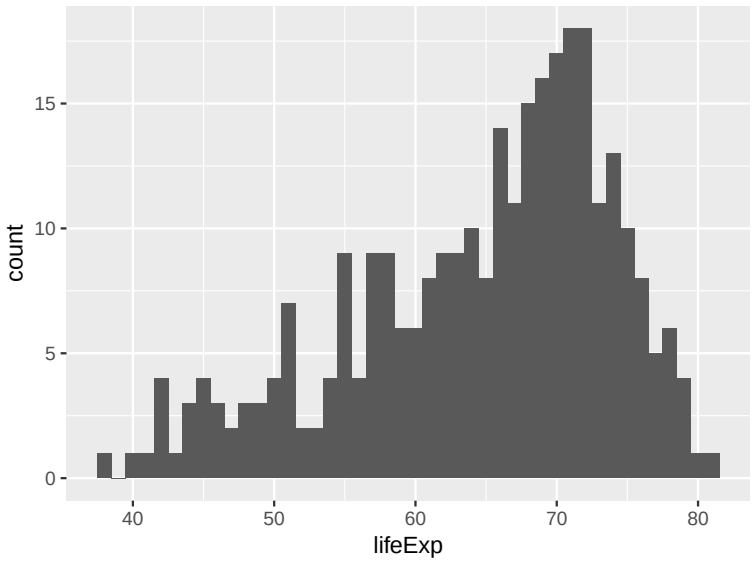


Figure 3.10. The `"Americas"` histogram of life expectancy.

With a nested `data.frame` full of panels, we add the cognostics to the `data.frame`.

```
continent_hists_cogs <- autocogs::add_panel_cogs(continent_hists) %>%
  print()
```

```
## Source: local data frame [5 x 6]
## Groups: <by row>
##
## # A tibble: 5 x 6
##   continent    panel            `_x`     `_density_x`
##      <fctr>   <list>          <list>           <list>
## 1    Africa <S3: gg> <tibble [1 x 5]> <tibble [1 x 6]>
## 2  Americas <S3: gg> <tibble [1 x 5]> <tibble [1 x 6]>
## 3      Asia <S3: gg> <tibble [1 x 5]> <tibble [1 x 6]>
## 4    Europe <S3: gg> <tibble [1 x 5]> <tibble [1 x 6]>
## 5   Oceania <S3: gg> <tibble [1 x 5]> <tibble [1 x 6]>
## # ... with 2 more variables: `_hist_x` <list>, `_n` <list>
as.list(continent_hists_cogs[americas_pos, 3:6])
## $`_x`
## $`_x`[[1]]
## # A tibble: 1 x 5
##      min    max     mean median      var
##    <dbl>  <dbl>    <dbl>  <dbl>    <dbl>
## 1 37.579 80.653 64.65874 67.048 87.33067
##
##
## $`_density_x`
## $`_density_x`[[1]]
## # A tibble: 1 x 6
##   max_density max_density_location clusters unimodal_p_value
##         <dbl>               <dbl>    <lgl>            <dbl>
## 1  0.04992932            70.13661       NA        0.9927842
## # ... with 2 more variables: skew <dbl>, kurt <dbl>
##
##
## $`_hist_x`
## $`_hist_x`[[1]]
## # A tibble: 1 x 6
##   count_min count_max count_mean count_median count_var        chisq
##       <dbl>     <dbl>      <dbl>        <dbl>     <dbl>        <dbl>
## 1         0        18   6.818182            6  25.78013 8.513837e-16
##
##
## $`_n`
## $`_n`[[1]]
## # A tibble: 1 x 2
##       n  n_na
##   <int> <int>
## 1   300     0
```

The above R output contains all cognostic groups that apply to a univariate histogram. For a `ggplot2` histogram, `autocogs` creates:

- `` `_x` `` : univariate continuous cognostics using the X data,
- `` `_density_x` `` : continuous density cognostics using the X data,
- `` `_hist_x` `` : histogram counts cognostics using the X data,
- `` `_n` `` : and univariate count information using the X data.

While this amount of information is a little overwhelming, keep in mind the goal of the `autocogs` R package is to provide as many ways to filter and sort data that normally must be calculated manually. `autocogs` computes cognostics that are suited to each type of visualization layer.

3.5.2 Linear model and scatterplot

In the next example, we will perform a similar workflow, but the panel will contain two layers: points and a linear model line. Each layer is derived from the same original data, but will help produce a different sets of cognostics.

```
gapminder %>%
  group_by(country, continent) %>%
  do(
    panel = ggplot(., aes(year, lifeExp)) +
      geom_point() +
      geom_smooth(method = "lm")
  ) ->
country_model
usa_pos <- which(country_model$country == "United States")
country_model$panel[[usa_pos]]
```

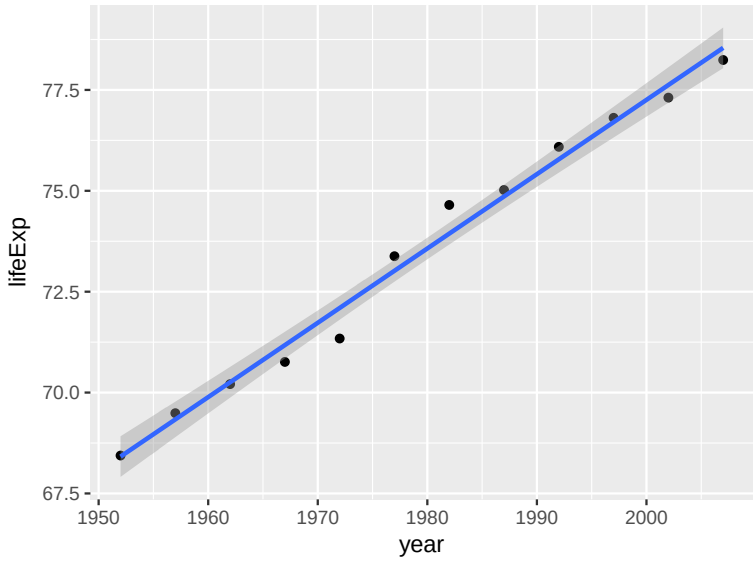


Figure 3.11. The `"United States"` life expectancy over time displayed as a linear model and scatterplot combination.

The `"United States"` panel is displayed in Figure 3.11. The life expectancy has a fairly linear trend that increases over time. Next, we add the cognostics to the panel `data.frame`.

```
country_model_cogs <- autocogs::add_panel_cogs(country_model) %>% print()
## Source: local data frame [142 x 10]
## Groups: <by row>
##
## # A tibble: 142 x 10
##        country continent    panel    `_scagnostic`             `_x`
##         <fctr>    <fctr>   <list>           <list>           <list>
##  1 Afghanistan      Asia <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  2     Albania    Europe <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  3     Algeria    Africa <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  4      Angola    Africa <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  5   Argentina  Americas <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  6   Australia   Oceania <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  7     Austria    Europe <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  8     Bahrain      Asia <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
##  9  Bangladesh      Asia <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
## 10     Belgium    Europe <S3: gg> <tibble [1 x 9]> <tibble [1 x 5]>
## # ... with 132 more rows, and 5 more variables: `_y` <list>,
## #   `_bivar` <list>, `_smooth` <list>, `_lm` <list>, `_n` <list>
as.list(country_model_cogs[usa_pos, 4:10])
## $`_scagnostic`
```

```
## $`_scagnostic`[[1]]
## # A tibble: 1 x 9
##   Outlying    Skewed    Clumpy    Sparse  Striated Convex Skinny
##      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>  <dbl>  <dbl>
## 1        0 0.7430783 0.3112384 0.1636085 0.8181818      0      1
## # ... with 2 more variables: Stringy <dbl>, Monotonic <dbl>
##
##
## $`_x`
## $`_x`[[1]]
## # A tibble: 1 x 5
##     min   max   mean median   var
##   <int> <int>  <dbl>  <dbl> <dbl>
## 1  1952  2007 1979.5 1979.5   325
##
##
## $`_y`
## $`_y`[[1]]
## # A tibble: 1 x 5
##     min    max    mean median      var
##   <dbl>  <dbl>   <dbl>  <dbl>    <dbl>
## 1 68.44 78.242 73.4785 74.015 11.18087
##
##
## $`_bivar`
## $`_bivar`[[1]]
## # A tibble: 1 x 2
##   covariance correlation
##        <dbl>       <dbl>
## 1     59.855   0.9929351
##
##
## $`_smooth`
## $`_smooth`[[1]]
## # A tibble: 1 x 3
##         mse max_deviation max_deviation_location
##       <dbl>         <dbl>                 <int>
## 1 0.1443062     0.7572308                  1972
##
##
## $`_lm`
## $`_lm`[[1]]
## # A tibble: 1 x 19
##   intercept intercept_p_value      beta beta_p_value        r2
##       <dbl>             <dbl>     <dbl>        <dbl>     <dbl>
## 1 -291.0845      1.254513e-09 0.1841692 1.369788e-10 0.9859202
```

```
## # ... with 14 more variables: sigma <dbl>, statistic <dbl>,
## #   p_value <dbl>, df <dbl>, log_lik <dbl>, aic <dbl>, bic <dbl>,
## #   deviance <dbl>, df_residual <int>, n_sig_cooks <int>,
## #   n_sig_hat <int>, resid_shapiro <dbl>, bc_lower <dbl>,
## #   bc_upper <dbl>
##
##
## $`_n`
## $`_n`[[1]]
## # A tibble: 1 x 5
##       n n_both_na n_or_na n_x_na n_y_na
##   <int>     <int>   <int>  <int>  <int>
## 1    12         0       0      0      0
```

As expected, many cognostics were added to the country panel `data.frame`:

- `` `_scagnostic` `` : scatterplot scagnostics cognostics,
- `` `_x` `` : univariate continuous cognostics using the X data,
- `` `_y` `` : univariate continuous cognostics using the Y data,
- `` `_bivar` `` : bivariate continuous cognostics,
- `` `_smooth` `` : smooth line cognostics,
- `` `_lm` `` : linear model cognostics,
- `` `_n` `` : and bivariate count information.

Revisiting the Linear Model Example 3.3.1, we have now exposed the linear model metrics (as well as many other metrics) to be used as cognostics within the `trelliscopejs` widget. `trelliscopejs` will process the nested `data.frame`s as grouped cognostics and display them in the application. Each linear model's R^2 value is now available for sorting. By opening the "Sort" tab in the widget and selecting *ascending* `r2`, all countries will be displayed in increasing order of the R^2 value.

Figure 3.5.2 displays the countries whose life expectancy can not be not explained by a linear model.

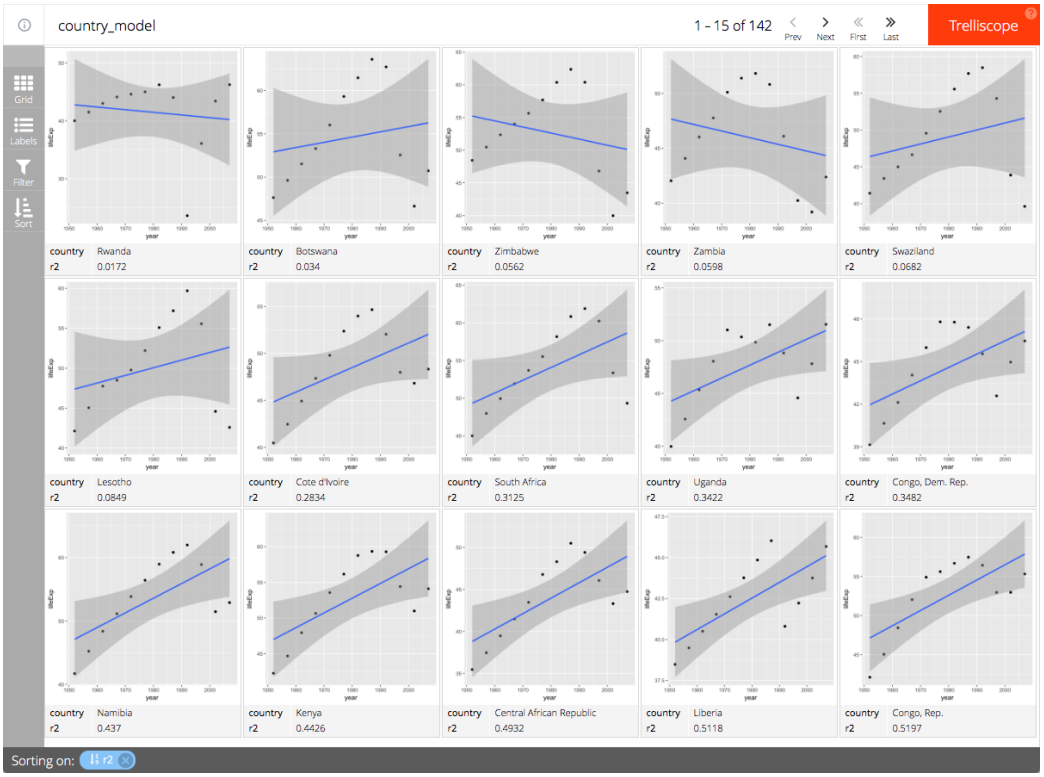


Figure 3.12. A `trelliscopejs` widget of country life expectancy over time where the panel ordering is displayed according to ascending R^2 value of each panels linear model.

In less than ten lines of code we went from a single data set to a full fledge visualization application with multiple cognostics explaining the plot used with each conditioned subset.

3.6 Summary

Automatic cognostics with the `autocogs` R package allow users to enable common univariate metrics that correspond to the plots displayed in a `trelliscopejs` HTML widget. `autocogs` provides users with the ability to retrieve statistical metrics that are displayed or utilized in a plot. Prior examples generated cognostics for simple

histograms to multiple layered, linear model and scatterplot plots. Maintaining a common set of cognostics provides a cohesive cognostic framework to be used by any plotting architecture.

4. GQLR: A GRAPHQL R SERVER IMPLEMENTATION

Since 2012, Facebook has developed GraphQL: "a query language for APIs and a runtime" [38]. GraphQL drastically reduces the number of server requests created by the browser by using a dynamic and nested query structure. Using the Working Draft Specification for GraphQL [39] as guidance, `gqlr` [40] implements a full GraphQL server within R. `gqlr` allows users to supply their own R functions to satisfy the data requirements of a GraphQL query generated by the browser. `gqlr` was originally built to communicate between `trelliscopejs` and an R server session. However, `trelliscopejs`'s development direction has changed since the creation of `gqlr`.

4.1 Application protocol interfaces

Application Protocol Interfaces (APIs) are the fundamental backbone of communication between machines. They enable machines to communicate with each other without needing to know how the responding machine arrives at its answer. There are two conventions for a machine API: simple API and custom API. Each API style has their own advantages and disadvantages. GraphQL was created to address the disadvantages of simple API while keeping the advantages of a custom API at the cost of typing the request.

There are two parts involved in each API communication: the request and response. The request submits a query to the server, and the response returns an answer that should be able to be understood by the requesting entity. The key point to the communication transaction is the request does not need to know how the response calculates the answer.

This layer of abstraction shields the request from knowing the nitty-gritty implementation details of the response. This allows the response to behave like a black box that will only respond to particular requests or queries.

In this chapter, I will only address single response APIs as I am concerned with the request, not the response. A single response API is an API that returns only one response to one answer. I will not discuss streaming APIs in this chapter.

Throughout this section, I will use a calculator, 'Calculator', as my example server. The calculator will be have four internal functions: add, subtract, multiply, and divide. These four internal functions behave just like a regular calculator, but the true implementation of the calculator is hidden.

4.1.1 Simple API

A simple API has a single end point, or place to send a request, for each style of question it knows how to answer. A simple Calculator API would contain four end points: add, subtract, multiply, and divide. Each query routine requires two numeric values as inputs and responds with a single numeric value. To concisely define a RESTful Calculator, we may state the following schema:

```
# GraphQL; Schema
type Calculator {
  add(A: Float, B: Float): Float
  subtract(A: Float, B: Float): Float
  multiply(A: Float, B: Float): Float
  divide(A: Float, B: Float): Float
}
```

Advantages

Without abstracting the function name, the number of request end points match the number of exposed functions. There are no dynamically created functions; everything is static. By fixing all request end points, software systems can be reliably

built against one another. Any software system can build their own logic as to how they solve their particular problems, but each software system will request from the same API.

The ease of use of a simple API has made it very popular with HTTP internet websites with the RESTful API [41]. The four most common functions of HTTP's REST are "GET", "PUT", "POST", and "DELETE".

1. GET: Retrieve the supplied location data only.
2. PUT: Store supplied data the supplied location.
3. POST: Add new data at the supplied location.
4. DELETE: Remove data at the supplied location.

Disadvantages

As is common in practice, databases have many tables with built in relationships. RESTful APIs usually only return information one layer deep.

When looking at the Calculator, it solves a single calculation for each query. To solve a multiple calculation problem, it takes multiple requests to the Calculator API. For example, solving $1+2+3+4+5$ requires 4 requests to the Calculator API; one query for each of the *addition* operations.

```
answer = Calculator::add(1, 2) # 3
answer = Calculator::add(answer, 3) # 6
answer = Calculator::add(answer, 4) # 10
answer = Calculator::add(answer, 5) # 15
```

No matter how the requests are altered, it will require 4 requests to the Calculator. This can become a major disadvantage when required to make many, many requests to the API. If we were to look at a person's friends of their friends, we would need the following Schema and data information:

```
# GraphQL; Schema
type Person {
  id: ID
  name: String
  age: Int
  sex: String
  friends: [ID]
}
```

Each person has an "id", "name", "age", "sex", and a list of person id's for their "friends". While having all of this information is useful, it can bloat the amount of information that is returned. While the current example isn't too big, one could imagine adding a `Person`'s favorite song lyrics to the `Person` object. This would greatly increase the total amount of information returned for each `Person` object.

4.1.2 REST and the Internet

In the case of the internet, however, two major constraints exist. The two major constraints are the number of parallel requests that can be made at one time and the amount of time it takes for your request to reach a responding server.

HTTP 1.1 specification states that "A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy" [42]. In practice, this is a little larger, maxing out at 13 parallel connections to the same host [43]. Limiting the number of server connections reduces the amount of simultaneous requests on the network and improves overall response time.

Once a request is allowed to be made, there are physical limitations on how fast the response can be received. On average, it takes over 200 ms for a request to make a trip to the responding server and back to the user's browser [44].

Using the "friends of my friends" example, let us define each person as having 200 different friends (the median number of friends on Facebook [45]. When using a simple API, we can calculate how many requests are necessary to compute who are the "friends of my friends". It requires $O(n^2)(\sim 200^2 = 40,000)$ independent

requests to the simple API server. If the browser being used is limited to 10 parallel connections, it requires $4000 (= 40,000/10)$ sets of parallel request groups. Each parallel request group takes at least 200ms for a round trip to and from the server. The total amount of time it would require to gather the names of a person's friends of friends is approximately $13\frac{1}{3} minutes = \frac{(200 requests)^2}{10 requests} * \frac{0.200 \frac{seconds}{request}}{60 \frac{seconds}{minute}}$.

The median loading time for a website is about 3 seconds with the average around 5 seconds [46]. Major websites today try to shave milliseconds where ever they can. No major website would allow a load time that is 160 times slower than the average website.

4.1.3 Custom response

A natural response to the "friends of my friends" situation is implement a custom response which directly answers the specific query.

Advantages

Because custom APIs answer specific questions, only one request needs to be sent to the server. The "friends of my friends" example is reduced to a single, 200 ms request. The load time is now reduced from 800 seconds to 0.2 seconds. This answers the "who are my friends of my friends" question in the minimal amount of queries to the server. The custom API achieves the end goal answer with the minimal amount of queries (1) to the server.

Disadvantages

The disadvantages come from the amount of engineering time that is required to implement a custom response for every query need. With a simple query interface, the logic is put on the user who is querying to figure out what to query next. With

a custom response interface, all custom responses must be made before the user can utilize them.

Using the Calculator example, calculating the $1 + 2 + 3 + 4 + 5$ example could be done in a single custom request called "1add2add3add4add5". It would solve the answer in a single request, but the server would have to implement many, many responses for full funtionality of a true Calculator.

With the internet, the custom API response is now tied directly to the requesting website. This creates little separation between the website and the responding API server. By coupling the data server with the requesting service, developments in the requests are slower and more difficult than if a de-coupled, simple API is used. Websites making the requests can not move ahead in development until a new custom response has been enabled by the custom API server.

4.1.4 Balancing act

Let's recap the advantages and disadvantages of the two different styles of APIs.

1. Simple API
    (a) Advantages
        i. Small API
        ii. Easier to implement
    (b) Disadvantages
        i. Many queries are required to solve complex problems
        ii. Many queries causes large time complexity
        iii. Every piece of information is returned for every query
2. Custom Responses
    (a) Advantages
        i. One query, one answer

ii. Fast execution time

(b) Disadvantages

i. All custom API calls need to be implemented

ii. API server is tied to the requests made

Developers balance between minimal engineering time and minimal execution time. Typically the final result falls somewhere in-between, using a custom API for high execution time queries and using a simple API for smaller queries.

4.2 Database storage

Once a request is received by an API, the API must retrieve the data from a database. Databases can store objects in one of two common paradigms: as a relational database or as a key-value database. Relational databases know exactly what kind of object will be returned and can map one table to another with id values. Key-value databases, on the other hand, do not inspect the values of the database. The requirement for a key-value database is that each value is stored at a specific key. Key-value databases are built for speed and scalability over structure and relationships.

While relational databases are already inherently typed (unless purposely stated as an unknown type), well designed key-value databases inherently contain typed values. Each value that is inserted into the database has a known shape and expected response type. If data of an image is stored, audio data should never be returned from that same position. Even if the image had different sizes and formats, it still can be understood as an image.

There are many more comparisons and cost / benefits to every database, however these implementation differences are not apart of the scope of this chapter.

4.3 GraphQL language

GraphQL is a data query language built to unify data APIs. It exists as a execution layer between the requesting user and responding database. This abstraction layer provides many benefits: uniform request and response shape, dynamic queries to handle custom situations, and minimal server requests.

GraphQL is comprised of two main parts, the Request and the Schema.

4.3.1 Schema

GraphQL Schemas are defined using Scalars, Types, Enumerations, Lists, Non-Null types, Interfaces, Unions, and Input types. Each definition is used to define a type or type abstractions that can be used when querying. The Types represent the expected return objects that the database already knows about from the Schema definition. Like most objects, each Object Type will contain fields that point to Scalars or more Types. These Object fields can be queried recursively until a Scalar or Enumeration is reached.

Object type definition

For example, we can setup a Schema for a pet dog.

```
type Dog {
  name: String
  breed: String!
  owners: [Person!]!
}
```

The type definition for a `Dog` is very readable, but has a lot going on.

1. `Dog` is a Object Type definition. It has four fields: `name`, `breed`, `age`, and `owners`. These four fields are the only fields defined for retrieving information from a `Dog`.

2. `String` is a predefined Scalar type definition. This contains the dog's name, i.e. "Clifford". Scalar fields do not contain any sub fields and are considered leafs in the Schema definition tree. Leafs do not contain sub fields for further information retrieval.

3. `String!` represents a Non-null String value. This means that all `Dog` objects will contain the `breed` field and the result will always be a String.

4. `[Person]` represents an array of Person objects that represent the owners of the Dog. By adding the `!` outside the array to form `[Person]!`, it will be guaranteed to return an array for the field `owners` and never a `NULL` value. By adding a `!` to the Person (`[ Person!]!`), the elements inside the position array will never be `NULL`. A length 0 array is still allowed as the `owners` value is not `NULL` and the missing `Person` values are not `NULL`.

4.3.2 Argument and input type definitions

Field definitions can include arguments. These arguments can be simple Scalar definitions or Input Type definitions.

```
# GraphQL; Schema
input ToyInput {
  brand: String
  name: String
  condition: Condition
}
extend type Dog {
  weight(unit: WeightUnit = POUNDS): Float
  does_play_with_toy(toy: ToyInput): Boolean
}
```

Like the R language, all arguments are named arguments, all arguments may be submitted in any order, and default values may be provided. Unlike R, all submitted arguments must have a name and must comply with the argument type. Nothing is

inferred from the arugment's position. Default values may be used in place of missing arguments.

4.3.3 Schema type definition

There are two entry points to a Schema: schema query type and schema mutation type. Every GraphQL Schema definition must have a schema query type and can optionally have a schema mutation type. Both types refer to an object type definition.

```
# GraphQL; Schema
schema {
  query: Dog
  mutation: DogUpdate
}
```

Query types are read only, while mutation types are understood that something will update in the database. A request will have the same shape for both query types and mutation types.

4.3.4 Scalar type definitions

Scalar Types are the leafs of the Schema. Unlike Object Types, Scalars do not have fields to inspect. GraphQL defines the base scalars as a part of the language definition:

1. `Boolean`: *true* or *false*
2. `Integer`: A signed 32-bit integer
3. `Float`: A signed double-precision floating point value
4. `String`: A UTF-8 character sequence
5. `ID`: `ID` performs the same as a `String`, but it is intended to be machine readable only as a unique identifier

New scalars can be defined in a Schema as long as the server running the GraphQL understands how to handle them. Three new Scalars are defined below.

```
# GraphQL; Schema
scalar Date
scalar Binary
scalar Hexadecimal
```

4.3.5 Enumeration type definitions

GraphQL understands a finite category variable. This is similar to a `factor` in R. There is a fixed set of values for every Enum definition. Internally in the GraphQL server, Enum definitions may be stored as Integer values or as an object similar to a Set, but in the GraphQL language, it will be represented as a string with all capitol letters, such as `POUNDS`.

```
# GraphQL; Schema
enum WeightUnit {
  POUNDS
  KILOS
  OUNCES
}
```

Whenever a `WeightUnit` type is expected, only the values of `POUNDS`, `KILOS`, or `OUNCES` may be used. While R supports `factor` values, and full Enumeration class is created in the `gqlr` R package.

4.3.6 Interface type definition

Interfaces are an integral part in abstracting pieces of types of objects. They allow for common fields to be accessed on objects without knowing the true type of the object. All Object Types that inherit an interface must implement all fields of that interface. Any Object Type that implements `Pet`, must implement the two fields `name` and `owners` and return `String` and `[Person!]!` respectively.

```
# GraphQL; Schema
interface Pet {
  name: String
  owners: [Person!]!
}
type Dog implements Pet {
  name: String
  barkVolume: Int
  owners: [Person!]!
}
type Cat implements Pet {
  name: String
  meowVolume: Int
  owners: [Person!]!
}
```

An Object Type may implement extra fields, like `meowVolume` in `Cat`.

Interfaces are useful when queries are made on objects where the exact return type is not known, but a finite set of types exist for the expected value.

```
# GraphQL; Schema
extend type Person {
  pet: [Pet!]! # returns an array of Cat or Dog types
}
```

Object definitions may interface with many Interface definitions.

4.3.7 Union type definition

Unions contain a finite set of Object Types but do not specify common fields. Unions are not allowed to contain other unions or interfaces. `NotAPlant` results being returned could either be a `Person`, `Dog`, or `Cat` type.

```
# GraphQL; Schema
extend type Cat {
  weight(unit: WeightUnit = KILOS): Float
}
union NotAPlant = Person | Dog | Cat
```

4.4 Requests

The following Schema will be used within the Requests section. Classic Disney characters will be used as the data.

```
# GraphQL; Schema
scalar Date
enum WeightUnit {
  POUNDS
  KILOS
  OUNCES
}
type Person {
  name: String!
  weight(unit: WeightUnit = POUNDS): Float
  friends: [Person!]!
  pets: [Pet!]!
}
```

```
interface Pet {
  name: String
  born: Date
  owners: [Person!]!
}
type Dog implements Pet {
  name: String
  barkVolume: String
  born: Date
  owners: [Person!]!
}
type Cat implements Pet {
  name: String
  born: Date
  meowVolume: String
  owners: [Person!]!
}
union SearchResult = Person | Dog | Cat
```

```
type QueryType {
  search_name(name: String!): SearchResult
  person(name: String! = "Mickey Mouse"): Person
  dog(name: String! = "Pluto"): Dog
  cat(name: String! = "Figaro"): Cat
}
type MutationType {
  addToWealth(name: String!, amount: Float): Float
}
schema {
  query: QueryType
  mutation: MutationType
}
```

4.4.1 Queries

Every object in the Schema definition can be queried. A query on an object type must include at least one field. Object Types are not considered leafs in the Schema definition tree as they are guaranteed to have at least one field.

```
# GraphQL; Query
{
  person {
    name
  }
}
```

```
// JSON; GraphQL Response
{
  "data":{
    "person": {
      "name": "Mickey Mouse"
    }}}
```

Queries have the same shape has the result. The only difference occurs when a result is an array of information. Using an `name` argument below, we can look at Jim Dear's pets from the movie "Lady and the Tramp" [47].

```
# GraphQL; Query
{
  person(name: "Jim Dear") {
    name
    pets {
      name
    }}}
```

```
// JSON; GraphQL Response
{
  "data":{
    "person": {
      "name": "Jim Dear",
      "pets": [
        { "name": "Lady" },
        { "name": "Tramp" }
      ]}}}
```

The whole query string is submitted to the GraphQL server. This allows for complex and deeply nested queries in a single request. Using the query below, we can query for all pets owned by the same owners as Duchess the cat [48].

```
# GraphQL; Query
{
  cat(name: "Duchess") {
    owners {
      name
      pets {
        name
      }}}}
```

```
// JSON; GraphQL Response
{
  "data": {
    "cat": {
      "owners": [{
        "name": "Madame Adelaide Bonfamille",
        "pets": [
          {"name": "Duchess"},
          {"name": "Marie"},
          {"name": "Berlioz"},
          {"name": "Toulouse"},
          {"name": "Thomas O'Malley"}
        ]}]}}}
```

Normally the code above would require $O(k * n)$ requests in a simple API, with k owners and n pets belonging to each owner. With GraphQL, this query is resolved in one request and without the need for a custom API to be built. The single GraphQL query needed for the "friends of my friends" example is shown below.

```
# GraphQL; Query
{
  person(name: "Barret") {
    friends {
      name
      friends {
        name
      }}}}
```

Aliases

Aliases can be used to query the same object field multiple times.

```
# GraphQL; Query
{
  duchess: cat(name: "Duchess") {
    name
    meowVolume
  }
  rajah: cat(name: "Rajah") {
    name
    meowVolume
  }}
```

```
// JSON; GraphQL Response
{
  "data": {
    "duchess": {
      "name": "Duchess",
      "meowVolume": 2
    },
    "rajah": {
      "name": "Rajah",
      "meowVolume": 9
    }}}
```

Fragments

For conciseness, the multiple cat query can be made using Fragments. Fragments can be used when repeated fields are called on similar objects.

The same result will occur using the Fragments below.

```
# GraphQL; Query
{
  duchess: cat(name: "Duchess") {
    ...catFields
  }
  rajah: cat(name: "Rajah") {
    ...catFields
  }
}
fragment catFields on Cat {
  name
  meowVolume
}
```

```
// JSON; GraphQL Response
{
  "data": {
    "duchess": {
      "name": "Duchess",
      "meowVolume": 2
    },
    "rajah": {
      "name": "Rajah",
      "meowVolume": 9
    }}}
```

Fragments are very useful in breaking down complex queries into smaller sections.

4.4.2 Mutation

Mutations are the "write" to a database. Mutations also return information to avoid an immediate "read" afterwards. While there is no guarentee that a Query does not alter the database, it is a good practice to distinguish which commands read only and which commands write to the database. The mutation example below adds to the wealth of Scrooge McDuck [49].

```
# GraphQL; Schema
type MutationType {
  # returns total wealth after adding amount
  addToWealth(name: String!, amount: Float): Float
}
```

```
# GraphQL; Mutation
mutation {
  addToWealth(name = "Scrooge", amount = 1000.0)
}
```

```
// JSON; GraphQL Response
{
  "data": {
    "addToWealth": 28800000
  }}
```

4.4.3 Remaining GraphQL language

There are many more intricacies in the GraphQL language that are not in the scope of this paper.

4.5 `gqlr`: A GraphQL R server implementation

`gqlr` is an R package that implements the GraphQL server specification. `gqlr` handles Query and Mutation Requests and returns data in the proper format. It is built upon the next evolution of class definitions in R, `R6` [50].

4.5.1 `R6`

`R6` is a lightweight R package that creates objects that do not follow the particular conventions of R. R known for being "pass by value" language. This means that all

values are copied at the beginning of a function. Like many other languages, R does not have dynamic values for a list object. Values placed in a list in R stay as the same values.

```
my_list <- list(A = TRUE, B = FALSE)
my_list$A
## [1] TRUE
update_A_to_false <- function(x) {
  x$A <- FALSE
  x
}
update_A_to_false(my_list)
## $A
## [1] FALSE
##
## $B
## [1] FALSE
my_list
## $A
## [1] TRUE
##
## $B
## [1] FALSE
```

This R example does not update the value of A to `FALSE` as the value `my_list` was copied at the beginning of the function `update_A_to_false`.

`R6` allows for objects to be altered inside functions that they have been passed to without any changes in assignment method. This is similar to a "pass by reference" coding paradigm. An `R6` object is passed to a function and the function alters the value. The same `R6` object outside of the function is altered as well. This is not expected R behavior.

`R6` is built upon the use of Classes. `R6` classes are similar to Javascript's ES6 classes. There is a constructor, methods, and values for each class. The methods and values can be both private (only able to be seen internally) and public (available to anything). Like Javascript, there is a notion of the "this" value or an object representing itself. `R6` uses the `self` object in this case. Internal object values are retrieved using `value <- self\$key`. The example below defines a new object

`barret` with the name of `"Barret"`. Unlike regular R behavior, the name is changed globally to `"Schloerke"` by calling the function `update_name_to_schloerke`.

```
Minimal <- R6Class("Minimal",
  public = list(
    name = NULL,
    initialize = function(name = NA) {
      self$name <- name
      self$greet()
    },
    greet = function() {
      cat(paste0("Hello, my name is ", self$name, ".\n"))
    }
  )
)
barret <- Minimal$new("Barret")
## Hello, my name is Barret.
barret$name
## [1] "Barret"
update_name_to_schloerke <- function(x) {
  x$name <- "Schloerke"
  invisible(x)
}
update_name_to_schloerke(barret)
barret$name
## [1] "Schloerke"
```

`R6` also allows for dynamic queries. `R6` calls these "active fields". These fields are actually function calls, but appear a regular keys in the object. The active key function can handle a single argument. This argument represents the value of the object being stored. If no value was supplied, then the active key was retrieved, not set.

In the example below, a single active key of *random* will return a uniform value when retrieved and will set the random seed if the *random* key is set. After the key is set to 1234, as expected, the same random values are returned.

```
MinimalActive <- R6Class("MinimalActive",
  active = list(
    random = function(x) {
      if (missing(x)) {
        return(runif(1))
```

```
      }
      set.seed(x)
      TRUE
    }
  )
)
min_active <- MinimalActive$new()
min_active$random
## [1] 0.9829186
min_active$random
## [1] 0.5822385
min_active$random <- 1234
min_active$random
## [1] 0.1137034
min_active$random
## [1] 0.6222994
min_active$random <- 1234
min_active$random # same value as the first random value with seed 1234
## [1] 0.1137034
```

Finally, `R6` allows for inheritance. In the GraphQL's abstract syntax tree, many objects inherit from one another in a directed, acyclic graph structure. `R6`'s class inheritance extends nicely to the abstract syntax tree requirments of GraphQL.

```
ParentClass <- R6Class("ParentClass")
ChildClass <- R6Class("ChildClass", inherit = ParentClass)
child <- ChildClass$new()
class(child)
## [1] "ChildClass"  "ParentClass" "R6"
inherits(child, "ParentClass")
## [1] TRUE
```

In `gqlr`, all active values must inherit the correct class to be allowed to set. An Error will be thrown if a value does not contain the proper inheritance. The example below shows the creation of a named type "Dog". It also shows an attempt at setting the *name* value to a character. This is not allowed as the *name* value only allows objects that inherit the class "Name".

```
(obj <- gqlr:::NamedType$new(name = gqlr:::Name$new(value = "Dog")))
## <graphql definition>
##   | Dog
str(obj$name)
## <Name>
## . value: 'Dog'
```

```
obj$name <- "Dog"
## Error in bad_inherits():  Attempting to set NamedType.name.
## Expected value with class of |Name|.
## Received character
```

`gqlr` uses these active fields to accomplish typed language properties while executing in a untyped language of R. While enforcing typing within R does not happen often, it is required for GraphQL to be implemented.

4.5.2 Execution

At first glance, R and GraphQL seem like an unlikely combination. R is an untyped (dynamically typed) language while GraphQL is a typed language. R is not known for its raw speed and one of GraphQL's goals is to reduce execution time.

However, R is known for its statistical models, statistical graphics, and its very fast iteration speed [51]. `gqlr` is built to help small projects provide proof of concepts and for developers to mock full backend systems locally. Being able to submit the same style of request string for the production server and local development increases the productivity of the web development cycle.

The example below creates a model schema that creates a linear model and a loess model. Both models return the mean squared error (`mse`) for their respective models. The linear model also returns a `GGally`'s `ggnostic` plot which is base64 encoded [52] for data portability. The loess model also returns the effective number of parameters (`enp`).

```
"# GraphQL; Schema
scalar ImageBase64
type LinearModel { mse: Float!, ggnostic: ImageBase64! }
type LoessModel { mse: Float!, enp: Float! }
type Model {
  linear(formula: String, data_name: String): LinearModel
  loess(formula: String, data_name: String): LoessModel
}
schema { query: Model }
" %>%
  gqlr::gqlr_schema(
```

```
    ImageBase64 = function(p, schema) {
      tmp_file <- tempfile(fileext = ".png")
      on.exit(unlink(tmp_file))
      ggplot2::ggsave(tmp_file, p)
      knitr::image_uri(tmp_file) %>%
        # shorten for display purposes
        substr(start = 1, stop = 40) %>%
        paste0("...")
    },
    LinearModel = function(model, schema) {
      list(
        mse = mean(model$residuals ^ 2),
        ggnostic = function(...) {
          GGally::ggnostic(model)
        })},
    LoessModel = function(model, schema) {
      list(
        mse = mean(model$residuals ^ 2),
        enp = model$enp
      )},
    Model = function(ignore, schema) {
      model_ <- function(fn_) {
        function(null, args, schema) {
          formula_ <- as.formula(args$formula)
          data_ <- eval(as.symbol(args$data_name))
          fn_(formula_, data = data_)
        }}
      list(
        linear = model_(stats::lm),
        loess = model_(stats::loess)
      )}) ->
model_schema
```

In about 40 lines of code, a schema definition and execution methods can be implemented. To build a strong typed API and its corresponding implementation in less than 100 lines is not an easy task. With `gqlr`, we are able to do it in less than half the lines of code.

In the implementation, four major R packages are called.

1. `stats` is used for the linear model function and loess function.
2. `GGally` is used to call its model diagnostic plot matrix, `ggnostic`.
3. `ggplot2` is used to save the plot objected created by `GGally`.

4. `knitr` is used for its ability to data64 encode images saved by `ggplot2`.

These packages are built upon many other R packages and displays the extensibility of a schema execution.

To see the Model schema in action, we can execute a request using the classic R data set "iris".

```
'# GraphQL; Query
{
  linear(
    formula: "Petal.Length ~ Petal.Width",
    data_name: "iris"
  ) {
    mse,
    ggnostic
  }
  loess(
    formula: "Petal.Length ~ Petal.Width",
    data_name: "iris"
  ) {
    mse,
    enp
  }
}' %>%
  gqlr::execute_request(schema = model_schema) ->
result
```

```
result
## {
##   "data": {
##     "linear": {
##       "mse": 0.2256,
##       "ggnostic": "..."
##     },
##     "loess": {
##       "mse": 0.1444,
##       "enp": 4.0584
##     }
##   }
## }
```

While R users might not find the query and output very compelling (as they can be done in a regular R session), it is good to remember that any web service with access to the R `Model` schema can execute a similar command and get an answer

that is executed by R. During the request execution, *any* R package can be used. Being able to open the flood gates to R's extensive package list is very powerful. This allows Javascript in the browser and Python programs to retrieve full R plots and R model outputs using the same GraphQL API.

This schema can also be executed in a local GraphQL server session or on a production GraphQL server. Both the local and production servers have the same schema, so each service will return the same shapes. Each querying programming language will have the same query string and receive the same shaped response.

4.5.3 Web service

A simple web server is included in `gqlr`, but extending the web server to other url routes and authentication services are not included. `gqlr` was built to handle GraphQL requests and leave the url routing and authentication to better suited packages. While some may argue for a single, go-to R package, there are many existing URL request handlers and authenticators existing for R.

4.6 Summary

`gqlr` leverages the GraphQL language to effectively and efficiently communicate custom defined queries to and from the server. `gqlr` provides users the ability to rapidly iterate in a local R environment to mock large production-scale data backends. Combining R's fast iteration speed with R's ability to connect to many different existing backend services saves developers time and effort while keeping the data API communication consistent.

5. SUMMARY

In this thesis, I have described

i) `ggduo`, an R function that produces generalized plot matrices for two groups of variables,

ii) `autocogs`, an R package that automatically generates cognostics for a set of plots, and

iii) `gqlr`, an R package which implements the GraphQL data query application protocol interface.

`ggduo` extended the application of the generalized pairs plot to a generalized plot matrix for two-grouped data. This function has direct application to canonical correlation analysis and was extended by `ggnostic` to produce a generalized plot matrix for model diagnostics and `ggts` to produce a generalized plot matrix for time series data. These `ggplot2`-style plot matrices are implemented using the `ggmatrix` plot object in `GGally`.

`autocogs` implemented multiple standard cognostic groups to be automatically produced given the different plot layers of a `ggplot2` visualization. Each layer within a plot is connected to multiple cognostic groups. These sets of cognostics are then leveraged within a `trelliscopejs` HTML widget to aid in its data panel exploration. These cognostics alleviate the user from creating each cognostic value manually in the data set speeding up the data exploration process.

`gqlr` implemented the GraphQL server within the R environment. The GraphQL query language minimized the number of incoming data requests. By decoupling the websites and the stored data, iteration speed is increased in both web site development and data storage development. `gqlr` enabled R users to make use of the GraphQL query language for efficient data extraction and statistical modeling.

5.1 Discussion and Future Work

5.1.1 Interactive data exploration

The scale of data sets in practice is increasing at a rapid rate. `htmlwidgets` [15] has opened a new form of interactive visualization tools for R by leveraging Javascript in the web browser. There are two limitations that quickly occur: the amount of memory provisioned for the widget and the amount of data that is transferred to the web page. Both web browser limitations restrict web pages to host only small-sized (in memory only) data. `crosstalk` [53] currently handles communication between `htmlwidgets` that use small data.

To showcase the impact of memory limitations, let us consider a simple example where we would like to plot two density curves and a scatterplot using the same data set. If the data set is already considered small data, there should be little difficulty calculating and displaying density curves within the web page and displaying all three graphs. Selecting regions of any visualization panel could highlight the selected subset of information in the remaining visualizations. However, if the data is not small data and is too large to fit in memory, it requires an approach that must sacrifice either data quantity or the ability to interact within the browser to produce similar results. If the data *must* be kept in the browser, a sample of the population data could be used. If all visualizations can be rendered outside of the web browser, the difficulty of the data visualization can be outsourced to where the data is located. Neither of these situations allow for native interactive data exploration within the browser on the full, raw data set. An argument could be made for calculating histograms with very small binwidths to generate the density curves and using very small square or hexagonal bins for the scatterplot. While these summary statistics could be made small enough to fit within memory, this solution does not operate directly on the original data set.

5.1.2 Visualization syntax

Two of the latest interactive visualization systems within R have integrated with existing, non-R visualization libraries: `rbokeh` and `plotly`. Both R packages execute commands within the R session to produce a single JSON specification that is understood by the visualization library. Unfortunately, each of the R packages reinvented the wheel when it came to adding layers to their respective plots. `ggplot2` has proved itself to be an effective R package at creating visualizations. However, I would like to see the generation of data visualization objects and the displaying of said visualization objects have integratable plotting routines.

For example, a plot could be created using "`ggplot2` syntax" and displayed using the Bokeh [54] visualization library. The "`ggplot2` syntax" package would calculate

- how data should be displayed,
- the range of each axes and where the breakpoints are,
- what is displayed within each legend, and
- any extra annotation material such as a title or caption.

This plot construction information would then be passed to a thin "display only" data visualization library needed to reproduce the visualization within the respective library. An example implementation is shown below using a theoretical "`ggplot2` syntax" R package `ggsyntax` and a theoretical "display only" packages `display_bokeh` for Bokeh and `display_plotly` for Plotly.

```
# create a plot object
#   add all x,y data as points
#   calculate and add a linear model on x,y data
p <- ggsyntax::setup(example_data, ggsyntax::aes(x, y)) +
  ggsyntax::geom_point() +
  ggsyntax::geom_smooth(method = "lm")

# display using the Bokeh visualization library
display_bokeh::from_ggsyntax(p)
```

```
# display using the Plotly visualization library
display_plotly::from_ggsyntax(p)
```

The same plot object could be utilized by many different data visualization packages within R. This allows users to ingrain a consistent coding behavior when using any visualization library.

It could be argued that after a `ggplot2` plot is built, not drawn,

```
after_built <- ggplot_build(p)
```

the plot object could then be transformed to be used in a different visualization library. While this is not impossible, the currently built `ggplot2` plot object does not readily contain all necessary information to be displayed by another library. Some internal routines are still needed to finalize the plot's production information.

REFERENCES

REFERENCES

[1] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[2] Robert A. Muenchen. The popularity of data science software, Sep 2017.

[3] Nick Diakopoulos and Stephen Cass. Interactive: The top programming languages 2017, Sep 2017.

[4] R Development Core Team. Contributed packages, Sep 2017.

[5] GitHub.com. The world's leading software development platform github, Sep 2017.

[6] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. ISBN 978-0-387-75968-5.

[7] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.

[8] Ryan Hafen and Continuum Analytics, Inc. *rbokeh: R Interface for Bokeh*, 2016. R package version 0.5.0.

[9] Carson Sievert, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Corvellec, and Pedro Despouy. *plotly: Create Interactive Web Graphics via 'plotly.js'*, 2017. R package version 4.7.1.

[10] John W Emerson, Walton A Green, Barret Schloerke, Jason Crowley, Dianne Cook, Heike Hofmann, and Hadley Wickham. The generalized pairs plot. *Journal of Computational and Graphical Statistics*, 22(1):79–91, 2013.

[11] Edward Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, USA, 1990.

[12] J. W. Tukey and P. A. Tukey. Computer Graphics and Exploratory Data Analysis: An Introduction. In *Proc. the Sixth Annual Conference and Exposition: Computer Graphics '85, Vol. III, Technical Sessions*, pages 773–785. Nat. Computer Graphics Association, 1985.

[13] Barret Schloerke. *Automatic Cognostic Calculations*, 2017.

[14] Ryan Hafen and Barret Schloerke. *trelliscopejs: Create Interactive Trelliscope Displays*, 2017. R package version 0.1.9.

[15] Ramnath Vaidyanathan, Yihui Xie, JJ Allaire, Joe Cheng, and Kenton Russell. *htmlwidgets: HTML Widgets for R*, 2017. R package version 0.9.

[16] Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, Jan 2010.

[17] Winston Chang. *R Graphics Cookbook*. O'Reilly Media, Inc., 2013.

[18] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1986.

[19] John M. Chambers, William S. Cleveland, Paul A. Tukey, and Beat Kleiner. *Graphical Methods for Data Analysis*. Duxbury Press, 1983.

[20] John W. Emerson and Walton A. Green. *gpairs: gpairs: The Generalized Pairs Plot*, 2014. R package version 1.2.

[21] Baptiste Auguie. *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2017. R package version 2.3.

[22] Harold Hotelling. Relations Between Two Sets of Variates. *Biometrika*, 28(3/4):321–377, December 1936.

[23] UCLA: Institute for Digital Research and Education. CANONICAL CORRELATION ANALYSIS — R DATA ANALYSIS EXAMPLES, Sep 2017.

[24] Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. *Applied linear statistical models*. The McGraw-Hill/Irwin series operations and decision sciences. McGraw-Hill Irwin, 2005.

[25] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, Apr 1965.

[26] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):129, 2011.

[27] Hadley Wickham, Romain Francois, Lionel Henry, and Kirill Mller. *dplyr: A Grammar of Data Manipulation*, 2017. R package version 0.7.4.

[28] The DeltaRho Project. Analyze and Visualize Large Complex Data in R: Quickstart, 2017.

[29] Javier Luraschi, Kevin Ushey, JJ Allaire, and The Apache Software Foundation. *sparklyr: R Interface to Apache Spark*, 2017. R package version 0.6.3.

[30] Jennifer Bryan. *gapminder: Data from Gapminder*, 2015. R package version 0.2.0.

[31] L. Wilkinson, A. Anand, and R. Grossman. Graph-theoretic scagnostics. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pages 157–164, Oct 2005.

[32] Lee Wilkinson and Anushka Anand. *scagnostics: Compute scagnostics - scatterplot diagnostics*, 2012. R package version 0.2-4.

[33] Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, 1 edition, Jan 2017.

[34] G. Casella and R.L. Berger. *Statistical Inference*. Duxbury advanced series in statistics and decision sciences. Thomson Learning, 2002.

[35] Martin Maechler. *diptest: Hartigan's Dip Test Statistic for Unimodality - Corrected*, 2016. R package version 0.75-7.

[36] Chris Fraley, Adrian E. Raftery, Thomas Brendan Murphy, and Luca Scrucca. *mclust Version 4 for R: Normal Mixture Modeling for Model-Based Clustering, Classification, and Density Estimation*, 2012.

[37] Chris Fraley and Adrian E. Raftery. Model-based clustering, discriminant analysis and density estimation. *Journal of the American Statistical Association*, 97:611–631, 2002.

[38] Facebook Open Source. A query language for your api, Sep 2017.

[39] Facebook. Graphql, Sep 2017.

[40] Barret Schloerke. *gqlr: 'GraphQL' Server in R*, 2017. R package version 0.0.1.

[41] ServiceObjects: Insight On Demand. Why rest is so popular, Sep 2017.

[42] Fielding, et al. Connections, Sep 2017.

[43] Browserscope. Network: Top browsers, Sep 2017.

[44] dstrickler. Internet weather map, Sep 2017.

[45] Steven Mazie. *Do You Have Too Many Facebook Friends?*, Sep 2017.

[46] John Stevens. *How Slow is Too Slow in 2016?*, Sep 2017.

[47] Fandom. Jim Dear, Sep 2017.

[48] Fandom. *Madame Adelaide Bonfamille*, Sep 2017.

[49] Fandom. *Scrooge McDuck*, Sep 2017.

[50] Winston Chang. *R6: Classes with Reference Semantics*, 2017. R package version 2.2.2.

[51] R Development Core Team. What is R?, Sep 2017.

[52] Simon Josefsson. The base16, base32, and base64 data encodings. 2006.

[53] Joe Cheng. *crosstalk: Inter-Widget Interactivity for HTML Widgets*, 2016. R package version 1.0.0.

[54] Anaconda. *Welcome to Bokeh*, 2017.

[55] Barret Schloerke, Jason Crowley, Di Cook, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Joseph Larmarange. *GGally: Extension to 'ggplot2'*, 2017. R package version 1.3.2.

[56] Barret Schloerke, Jason Crowley, Di Cook, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Joseph Larmarange. *GGally: Extension to 'ggplot2'*, 2017.

[57] Hadley Wickham and Chang Winston. ggplot2, 2017.

[58] Barret Schloerke. *gqlr: 'GraphQL' Server in R*, 2017.

[59] Ramnath Vaidyanathan, Yihui Xie, JJ Allaire, Joe Cheng, and Kenton Russell. *htmlwidgets: HTML Widgets for R*, 2017.

[60] Plotly. *plotly: Visualize Data, Together*, 2017.

[61] Carson Sievert, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Corvellec, and Pedro Despouy. *plotly: Create Interactive Web Graphics via 'plotly.js'*, 2017.

[62] Ryan Hafen and Continuum Analytics, Inc. *rbokeh: R Interface for Bokeh*, 2017.

[63] Ryan Hafen and Barret Schloerke. trelliscopejs: Create interactive trelliscope displays, 2017.

APPENDICES

A. R PACKAGE DESCRIPTIONS

The following alphabetically sorted package descriptions are excerpts from their *DESCRIPTION* files.

`autocogs`: "Automatic Cognostic Summaries"

"Automatically calculates cognostic groups for plot objects and list column plot objects. Results are returned in a nested data frame." [13]

`GGally`: "Extension to `ggplot2`"

"The R package `ggplot2` is a plotting system based on the grammar of graphics. `GGally` extends `ggplot2` by adding several functions to reduce the complexity of combining geometric objects with transformed data. Some of these functions include a pairwise plot matrix, a two group pairwise plot matrix, a parallel coordinates plot, a survival plot, and several functions to plot networks." [55] [56]

`ggplot2`: "Create elegant data visualisations using 'The Grammar of Graphics'"

"`ggplot2` is a system for declaratively creating graphics, based on *The Grammar of Graphics*. You provide the data, tell `ggplot2` how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details." [57] [7]

`gqlr`: **"GraphQL server in R"**

"Server implementation of 'GraphQL' [39], a query language created by Facebook for describing data requirements on complex application data models. Visit `http://graphql.org` [38] to learn more about 'GraphQL'." [40] [58]

`lattice`: **"Trellis graphics for R"**

"A powerful and elegant high-level data visualization system inspired by Trellis graphics, with an emphasis on multivariate data. Lattice is sufficient for typical graphics needs, and is also flexible enough to handle most non-standard requirements." [6]

`htmlwidgets`: **"HTML Widgets for R"**

"A framework for creating HTML widgets that render in various contexts including the R console, 'R Markdown' documents, and 'Shiny' web applications." [15] [59]

`plotly`: **"Create Interactive Web Graphics via 'plotly.js'"**

"Easily translate `ggplot2` graphs to an interactive web-based version and / or create custom web-based visualizations directly from R. Once uploaded to a 'plotly' account [60], 'plotly' graphs (and the data behind them) can be viewed and modified in a web browser." [9] [61]

`rbokeh`: **"R interface for Bokeh"**

"A native R plotting library that provides a flexible declarative interface for creating interactive web-based graphics, backed by the Bokeh visualization library `http://bokeh.pydata.org/`." [8] [62]

`trelliscopejs`: "Create interactive Trelliscope displays"

"Trelliscope is a scalable, flexible, interactive approach to visualizing data. This package provides methods that make it easy to create a Trelliscope display specification for `trelliscopejs`. High-level functions are provided for creating displays from within `dplyr` or `ggplot2` workflows. Low-level functions are also provided for creating new interfaces." [63]

B. DATA SETS

Restaurant tips

```
reshape::tips %>% as_data_frame()
```

```
## # A tibble: 244 x 7
##    total_bill   tip    sex smoker    day   time  size
##  *      <dbl> <dbl> <fctr> <fctr> <fctr> <fctr> <int>
##  1      16.99  1.01 Female     No    Sun Dinner     2
##  2      10.34  1.66   Male     No    Sun Dinner     3
##  3      21.01  3.50   Male     No    Sun Dinner     3
##  4      23.68  3.31   Male     No    Sun Dinner     2
##  5      24.59  3.61 Female     No    Sun Dinner     4
##  6      25.29  4.71   Male     No    Sun Dinner     4
##  7       8.77  2.00   Male     No    Sun Dinner     2
##  8      26.88  3.12   Male     No    Sun Dinner     4
##  9      15.04  1.96   Male     No    Sun Dinner     2
## 10      14.78  3.23   Male     No    Sun Dinner     2
## # ... with 234 more rows
```

Psychological and academic data

```
psychademic %>% as_data_frame()
```

```
## # A tibble: 600 x 8
##    locus_of_control self_concept motivation  read write  math
##               <dbl>        <dbl>      <chr> <dbl> <dbl> <dbl>
##  1            -0.84        -0.24          4  54.8  64.5  44.5
##  2            -0.38        -0.47          3  62.7  43.7  44.7
##  3             0.89         0.59          3  60.6  56.7  70.5
##  4             0.71         0.28          3  62.7  56.7  54.7
##  5            -0.64         0.03          4  41.6  46.3  38.4
##  6             1.11         0.90          2  62.7  64.5  61.4
##  7             0.06         0.03          3  41.6  39.1  56.3
##  8            -0.91        -0.59          3  44.2  39.1  46.3
##  9             0.45         0.03          4  62.7  51.5  54.4
## 10             0.00         0.03          3  62.7  64.5  38.3
## # ... with 590 more rows, and 2 more variables: science <dbl>,
## #   sex <chr>
```

Election demand

```
fpp2::elecdemand %>% head()
```

```
## Time Series:
## Start = c(2014, 1)
## End = c(2014, 6)
## Frequency = 17520
##        Demand WorkDay Temperature
## 2014 3.698171       0        16.1
## 2014 3.426123       0        16.0
## 2014 3.295835       0        15.6
## 2014 3.166052       0        15.4
## 2014 3.071107       0        15.4
## 2014 2.999543       0        15.5
```

```
fpp2::elecdemand %>%
  as_data_frame() %>%
  mutate(
    WorkDay = factor(c("No", "Yes")[WorkDay + 1], levels = c("Yes", "No")),
    # Time = zoo::as.Date(zoo::as.Date(time(elecdemand)))
    Year = 2014,
    Day = rep(1:365, each = 48),
    HighUsage = c("below", "above")[(Demand > median(Demand)) + 1]
  ) %>%
  filter(Day <= 100) %>%
  mutate(Time = as.Date(Day, origin = "2014-01-01")) ->
elec_median

elec_median
```

```
## # A tibble: 4,800 x 7
##      Demand WorkDay Temperature  Year   Day HighUsage       Time
##       <dbl>  <fctr>       <dbl> <dbl> <int>     <chr>     <date>
##  1 3.698171      No        16.1  2014     1     below 2014-01-02
##  2 3.426123      No        16.0  2014     1     below 2014-01-02
##  3 3.295835      No        15.6  2014     1     below 2014-01-02
##  4 3.166052      No        15.4  2014     1     below 2014-01-02
##  5 3.071107      No        15.4  2014     1     below 2014-01-02
##  6 2.999543      No        15.5  2014     1     below 2014-01-02
##  7 2.955342      No        15.3  2014     1     below 2014-01-02
##  8 2.927419      No        15.1  2014     1     below 2014-01-02
##  9 2.934816      No        15.1  2014     1     below 2014-01-02
## 10 2.932894      No        15.1  2014     1     below 2014-01-02
## # ... with 4,790 more rows
```

Flea

```
GGally::flea %>% as_data_frame()
```

```
## # A tibble: 74 x 7
##     species tars1 tars2  head aede1 aede2 aede3
##      <fctr> <int> <int> <int> <int> <int> <int>
##  1 Concinna   191   131    53   150    15   104
##  2 Concinna   185   134    50   147    13   105
##  3 Concinna   200   137    52   144    14   102
##  4 Concinna   173   127    50   144    16    97
##  5 Concinna   171   118    49   153    13   106
##  6 Concinna   160   118    47   140    15    99
##  7 Concinna   188   134    54   151    14    98
##  8 Concinna   186   129    51   143    14   110
##  9 Concinna   174   131    52   144    14   116
## 10 Concinna   163   115    47   142    15    95
## # ... with 64 more rows
```

Gapminder

```
gapminder::gapminder %>% as_data_frame()
```

```
## # A tibble: 1,704 x 6
##        country continent  year lifeExp      pop gdpPercap
##         <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan      Asia  1952  28.801  8425333  779.4453
##  2 Afghanistan      Asia  1957  30.332  9240934  820.8530
##  3 Afghanistan      Asia  1962  31.997 10267083  853.1007
##  4 Afghanistan      Asia  1967  34.020 11537966  836.1971
##  5 Afghanistan      Asia  1972  36.088 13079460  739.9811
##  6 Afghanistan      Asia  1977  38.438 14880372  786.1134
##  7 Afghanistan      Asia  1982  39.854 12881816  978.0114
##  8 Afghanistan      Asia  1987  40.822 13867957  852.3959
##  9 Afghanistan      Asia  1992  41.674 16317921  649.3414
## 10 Afghanistan      Asia  1997  41.763 22227415  635.3414
## # ... with 1,694 more rows
```

Iris flower

```
iris %>% as_data_frame()
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
##  1          5.1         3.5          1.4         0.2  setosa
##  2          4.9         3.0          1.4         0.2  setosa
##  3          4.7         3.2          1.3         0.2  setosa
##  4          4.6         3.1          1.5         0.2  setosa
##  5          5.0         3.6          1.4         0.2  setosa
##  6          5.4         3.9          1.7         0.4  setosa
##  7          4.6         3.4          1.4         0.3  setosa
##  8          5.0         3.4          1.5         0.2  setosa
##  9          4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## # ... with 140 more rows
```

C. R PACKAGES

The following sections contain publically exported functions where I created or have made significant contributions.

C.1 `GGally`

- `+.gg`: Modify a ggmatrix object by adding an `ggplot2` object to all plots
  This operator allows you to add `ggplot2` objects to a `ggmatrix` object.
- `broomify`: Broomify a model
  `broom::augment` a model and add `broom::glance` and `broom::tidy` output as attributes. X and Y variables are also added.
- `find_plot_type`: Find Plot Types
  Retrieves the type of plot for the specific columns.
- `fn_switch`: Function switch
  Function that allows you to call different functions based upon an aesthetic variable value.
- `ggally_barDiag`: Plots the Bar Plots Along the Diagonal
  Plots the bar plots along the diagonal of a `ggpairs` plot.
- `ggally_blank`, `ggally_blankDiag`: Blank Plot
  Draws nothing.
- `ggally_box`, `ggally_box_no_facet`: Box Plot
  Make a box plot with a given data set. `ggally_box_no_facet` will be a single panel plot, while `ggally_box` will be a faceted plot.
- `ggally_cor`: Correlation from the Scatterplot
  Estimate the correlation from the provided data.

- `ggally_density`: Scatter Density Plot

  Produce a scatter density plot from a provided data.

- `ggally_densityDiag`: Density Plot Along the Diagonal

  Produce density plot along the diagonal.

- `ggally_denstrip`: Tile Plot with Facets

  Facet a tile plot using the provided data.

- `ggally_dot`, `ggally_dot_no_facet`: Dot Plot with Facets

  Facet a dot plot using the provided data.

- `ggally_facetbar`: Bar Plot with Facets

  X variables are plotted using `ggplot2::geom_bar` and faceted by the Y variable.

- `ggally_facetdensity`: Density Plot with Facets

  Facet a density plot using the provided data.

- `ggally_facethist`: Histogram Plot with Facets

  Facet a histogram plot using the provided data.

- `ggally_na`, `ggally_naDiag`: `NA` plot

  Draws a large `NA` in the middle of the plotting area. This plot is useful when all X or Y data is `NA`.

- `ggally_nostic_cooksd`: `ggnostic` - Cooks Distance

  A function to display `stats::cooks.distance`.

- `ggally_nostic_hat`: `ggnostic` - Leverage Points

  A function to display `stats::influence`s hat information against a given explanatory variable.

- `ggally_nostic_line`: `ggnostic` - Background Line with Geom

  If a non-null linePosition value is given, a line will be drawn before the given `continuous_geom` or `combo_geom` is added to the plot.

- `ggally_nostic_resid` : `ggnostic` - Residuals

  If non-null p value and sigma values are given, confidence interval lines will be added to the plot at the specified p value percentiles of a $N(0, \sigma)$ distribution.

- `ggally_nostic_se_fit` : `ggnostic` - Fitted Value Standard Error

  A function to display `stats::predict` s standard errors.

- `ggally_nostic_sigma` : `ggnostic` - Leave One Out Model Sigma

  A function to display `stats::influence` s sigma value.

- `ggally_nostic_std_resid` : `ggnostic` - Standardized Residuals

  If non-null p value and sigma values are given, confidence interval lines will be added to the plot at the specified p value locations of a $N(0, 1)$ distribution.

- `ggally_points` : Scatterplot

  Produces a scatterplot using the provided data.

- `ggally_ratio` : Mosaic Plot

  Produces a mosaic plot using fluctuation.

- `ggally_smooth` , `ggally_smooth_loess` , `ggally_smooth_lm` : Scatterplot with Smoothing

  Produces a smoothed line on top of a scatterplot.

- `ggally_text` : Text Plot

  Display text in the middle of a plot while maintaining a background scales.

- `ggduo` : A `ggplot2` Generalized Pairs Plot for Two Columns Sets of a `data.frame`

  Make a matrix of plots with a given data set with two different column sets.

- `ggfacet` : Single `ggplot2` Plot Matrix with `facet_grid`

  Produce a single `ggplot2` object using `ggplot2::facet_grid` .

- `gglegend` : Legend of Plot Function

  Only display the legend of a plot. Use this function to retrieve the only legend.

- `ggmatrix` : A`ggplot2`Plot Matrix

  Make a generic plot matrix of `ggplot2` plots.

- `ggmatrix_gtable`: Compute the `ggmatrix` `gtable`
  This functino builds all plots necessary for displaying the plot matrix and stores them in a `ggplot2` plot `gtable`.
- `ggnostic`: Statistical Model Diagnostics Plot Matrix
  Display commonly known linear model diagnostics against model predictor variables in a `ggmatrix`.
- `ggpairs`: A `ggplot2`Generalized Pairs Plot
  Produce plots of all variable combinations with different plot types for the upper triangle, lower triangle, and diagonal of the plot matrix.
- `ggts`: Multiple Time Series
  `GGally`implementation of ts.plot. Wraps around the ggduo function and removes the column strips.
- `grab_legend`: Extract a `ggplot2` Legend
  Extract the legend of a `ggplot2` object to be drawn at a later time.
- `print.ggmatrix`: Print a `ggmatrix` object
  Print method altered from `ggplot2:::print.ggplot` to accomodate a ggmatrix object
- `v1_ggmatrix_theme`: Original `ggmatrix` Layout `theme`
  Modify a ggmatrix object by adding an ggplot2 object to all plots
- `wrap`, `wrapp`, `wrap_fn_with_params`, `wrap_fn_with_param_arg`: Wrap a Function with Different Parameter Values
  Wraps a function with the supplied parameters to force different default behavior. This is useful for functions that are supplied to `ggpairs`. It allows you to change the behavior of one function, rather than creating multiple functions with different parameter settings.

C.2 `autocogs`

- `add_cog_group`: Add a Cognostic Group

  Add a new cognostic to be used when calculating automatic cognostics.

- `add_layer_cogs`: Add Plot Layer Cognostics

  Add a new set of cognostic groups for a given plot layer. If the plot layer is found, the corresponding cognostic groups will be calculated.

- `auto_cog`: Cognostic Group Function

  Calculate an automatic cognostic function given a cognostic group name.

- `cog_desc`: Cognostic and Description

  Add a description to a cognostic.

- `cog_group_df`: Cognostic Group `data.frame`

  Make a cognostics group data frame to be passed into `add_layer_cogs`

- `field_info`: Field Type Info

- `layer_count`: Number of Layers in Plot

  Retrieve the number of layers in a given plot

- `layer_info`: Plot Layer Information List

  Retrieve the data and parameter information for all layers of a plot.

- `panel_cogs`, `add_panel_cogs`: Calculate Panel Cognostics

  Return or concatinate panel cognostics. For each panel (plot) in the panel column, cognostics will be calculated for each panel. The result will be returned in a nested `tibble::tibble`.

- `plot_class`: Plot Class

  First class of the plot object. Exception is ggplot2 as many objects are of class gg.

C.3 trelliscopejs

- `+.gg` ggplot2Add Method

  Add method for `gg` / `facet_trelliscope`.

- `as_cognostics` As Cognostics

  Cast a data frame as a cognostics `data.frame`.

- `facet_trelliscope` `trelliscopejs` Faceting

  Facet using a `trelliscopejs` wigit. This function uses `ggplot2::facet_wrap` and `ggplot2::facet_grid` like syntax.

- `print.facet_trelliscope` Print a `facet_trelliscope` Object

  Prints a `trelliscopejs` wigit by saving the necessary files to disk. Like `ggplot2`, this allows for all plot alterations to be executed independently before print time.

- `trelliscope` Create a `trelliscopejs` Display

  Creates a `trelliscopejs`display by writing all necessary files to disk.

C.4 gqlr

- `as_R6`: As R6

  Debug method that strips all `gqlr` classes and assigns the class as 'R6'

- `ErrorList`: ErrorList

  Handles all errors that occur during query validation. This object is returned from execute request function (`ans <- execute_request(query, schema)`) under the field `error_list` (`ans$error_list`).

- `execute_request`: Execute GraphQL server response

  Executes a GraphQL server request with the provided request.

- `gqlr_schema`: Create Schema definitions

  Creates a Schema object from the defined GraphQL string and inserts the provided descriptions, resolve methods, and `resolve_type` methods into the appropriate place.

- `parse_ast`: Parse AST

  This is a helper function for Scalars. Given a particular kind and a resolve function, it produces a function that will only parse values of a particular kind.

- `Schema`: GraphQL Schema object

  Manages a GraphQL schema definition. A Schema can add more GraphQL type definitions, assist in determining definition types, retrieve particular definitions, and can combine with other schema definitions.

  Typically, Schema class objects are created using `gqlr_schema`. Creating a `Schema$new()` object should be reserved for when multiple Schema objects are combined.

- `Schema`: Run basic GraphQL server

  Run a basic GraphQL server with the jug package. This server is provided to show basic interaction with GraphQL. The server will run until the function execution is canceled.

VITA

VITA

Barret Schloerke was born in Ames, Iowa in 1988. He worked as a software engineer for the small startup Metmarkets in between earning his Bachelor of Science degree at Iowa State University in 2010, and a Master of Science degree in Mathematical Statistics from Purdue University 2014. His academic interests include data visualization, statistical computing, exploratory data analysis, large data, and computational statistics.