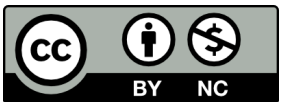
Business 4720

Data Visualization in R and Python

Joerg Evermann



Learning Goals

After reading this chapter, you should be able to:

- Explain different purposes for information visualization.
- Identify deceptive visualization techniques and avoid such techniques in your own visualizations.
- Understand different types of color palettes and be able to choose a color palette for a given visualization purpose.
- Understand the impact of color vision deficiency and its implications for creating meaningful visualization.
- Select a type of plot that is appropriate for a given purpose.
- Create different types of plots in R and Python, including customization of colors, axes, labels, and titles.

1 Introduction

Data visualization, the practice of transforming information into a visual context, has become an indispensable part of modern data analysis and communication. This field intersects art and science, requiring both creativity and analytical skills to convert complex data sets into comprehensible, insightful visual representations. The motivations for visualizing data are multifaceted. Primarily, it enhances understanding by simplifying complex information, making patterns, trends, and correlations more apparent than they would be in raw data. It also aids in storytelling, where data-driven narratives can be compellingly presented to a broad audience, regardless of their expertise in data analysis.

The purpose of visualization is to simplify, summarize and abstract complex information into an easy to understand format for human consumption, for understanding, persuasion or explanation, or for decision making. Visualizations can help to compare different objects or things, they can help identify trends, patterns, and relationships. In general, visualizations help in understanding data and gaining insight into a domain or pheonomenon.

The recent history of data visualization is marked by rapid advancements fueled by technology. In the last few decades, the advent of powerful computing and sophisticated software tools has revolutionized this field. Where once it was the domain of experts and specialists, data visualization has become accessible to a broader audience. Tools ranging from simple spreadsheet applications to advanced data visualization software have democratized the creation and interpretation of visual data. The rise of big data and machine learning has further escalated the importance of data visualization. As data sets have grown in size and complexity, the need for effective visualization tools has become more pronounced, leading to innovative methods and approaches. Interactive visualizations, real-time data mapping, and the use of virtual and augmented

reality are some of the cutting-edge trends redefining how we see and interact with data today. This evolution continues as we find new ways to visually interpret the vast and growing ocean of data that characterizes the digital age.

Visualization is important because humans are very good at visual pattern recognition. In fact, humans are too good at this, as they tend to also recognize patterns where none exist. This makes it easy to deceive oneself or others with data visualizations. Hence, visualization should normally be undertaken with and supported by statistical data analysis to verify the existence of trends or differences.

Visual Discovery

Visual discovery is the use of interactive visualization tools to uncover patterns, trends, and insights from data. This approach is a crucial aspect of modern data analysis, emphasizing the power of human visual perception. Visual discovery leverages the human brain's innate ability to process visual information rapidly. By translating complex data sets into graphical representations, it enables quicker and more intuitive understanding. Users can spot trends, outliers, differences, and patterns more easily than they could through rows of numbers or text.

Visual discovery is a highly iterative and dynamic process. Analysts rapidly create or change data visualizations, such as charts, graphs, and maps, to explore different aspects of the data. This interactivity allows for real-time exploration and analysis, making it easier to drill down into specifics, zoom out for a broader view or change the perspective one takes in examining data.

Visual discovery may be purely exploratory, without any prior knowledge by the data analyst, or it may seek to confirm or verify the beliefs or hypotheses that the data analyst has formed about the particular domain. However, even this confirmation is never final, but only a way to new insights and exploration. In this process, the analyst explores the data, forms some beliefs or hypotheses based on the exploration, tries to support it with a different visualization, and updates their beliefs or hypotheses based on the later visualization.

Declarative Visualization ("Storytelling")

In contrast to visual discovery, declarative visualization is purpose-driven and aims to provide explanations to a particular audience. It is *not* interactive or dynamic. Visualizations are intended to affirm or support a conclusion and to convince an audience or group of stakeholders. Information is not so much explored, as it is merely presented and explained in visualization. Declarative visualization is used to support decision making and is mainly static.

Operational Visualization (Monitoring)

In operational visualization, graphs and charts are used for supervision or monitoring of the operation of a system. They provide supervisors or controllers with a real-time

view of the state of key system properties and are used to spot situations or trends that require intervention in the system's operation, that is, operational decision making.

Quantitative Messages

Good visualizations are focused on the quantitative message they are intended to convey. For example, to present information about a time-series, that is, time-dependent behaviour of one or more variables, a line chart is a good type of visualization. However, that line chart would not be as useful to convey relative rankings of items or objects. For this purpose, a bar chart may be better suited. On the other hand, to describe part-whole relationships, a pie chart may be useful to show what part of the whole is contributed by its parts. Deviations from a mean or other standard, whether positive or negative, can be easily understood from a bar chart as well. To understand frequency distributions, one might use boxplots or histograms. Boxplots show median values, and measures of the "spread" or variability of the data. Histograms can show a one or two dimensional frequency distribution of values. To understand correlations of variables, a scatterplot is useful, where individual data points are plotted in a two or three dimensional coordinate system, often augmented with statistical information about their relationship. Finally, geographic information may use map data for visualization. This is sometimes called a "cartogram". Points may be overlaid on a map, or areas of a map may be colored or otherwise highlighted. In summary, it is important to consider the message to convey or the insight to be gained from a visualization when selecting the type of graph or chart.

2 Honesty in Visualization

For a number of reasons, it is easy to deceive with misleading visualizations. Humans are prone to see trends or patterns where none exist. A misleading visualization can exploit this propensity to suggest relationships between objects or variables that do not exist. Humans recognize some aspects of visualization better, earlier, and easier than others. For example, humans recognize the length of a line easier than the area of a surface, and recognize variations in color better than they interpret textual labels. A misleading visualization can exploit these cognitive effects to make the interpreter focus on particular, misleading aspects in the visualization. Finally, because visualizations are intended to abstract from the data itself and provide a summary, visualizations may not include sufficient information about the data or its processing to allow the reader to understand what is shown, making it easy to suggest interpretations that are misleading.

Here are some general guidelines for using visualizations:

- Do not deceive the target audience
- Do not diminish or hide relationships or trends
- Do not exaggerate relationships or trends
- Do not obfuscate, confuse, or hide information

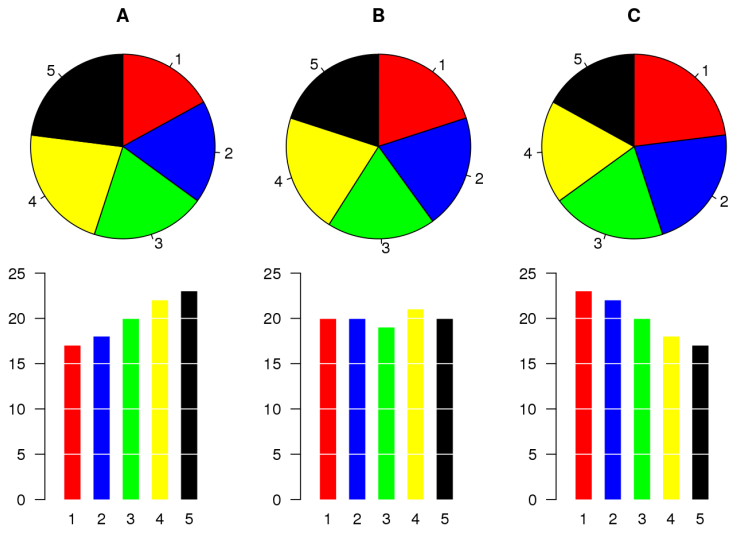
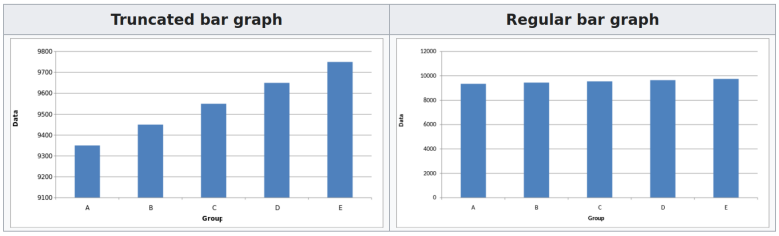


Figure 1: Comparing Pie Charts

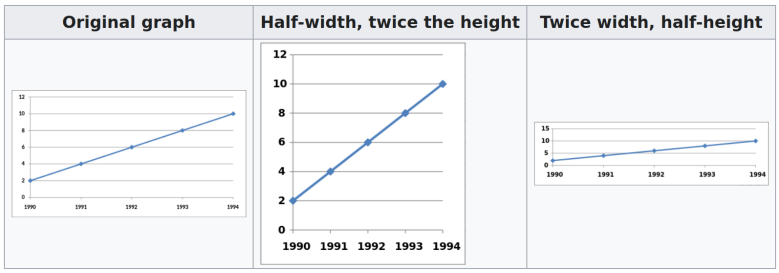
The term ”dark pattern” has been coined to describe the opposite of best practices in a field; in this case practices that are intended to deceive, mislead, or frustrate others. There are many of such dark patterns in visualizations:

- Use an inappropriate graph or chart type to hide or obfuscate relationships or trends. As noted above, different types of graph are suitable to convey different types of messages. An example is shown in Figure 1 that illustrates that a bar or column chart is more useful for comparisons of objects than pie charts, so the use of a pie chart could hide or obfuscate trends that may otherwise be prominent.
- *Graph unrelated data to suggest non-existent relationships.* The viewer of a visualization expects that data that is graphed together has a meaningful relationship. Simply by graphing data together, the analyst suggests a relationship where none may exist.
- *Scale multiple vertical axes to suggest correlations.* Scaling a graph with multiple vertical axes so that lines better align shows visual similarities that are not borne out by the data.
- *Use confusing colors.* For example, a color palette whose perceived color differences do not map linearly to the actual differences in the data (that is, it is not ”*perceptually uniform*”) may be misleading. For another example, using different shades of the same color for values that are very different will visually diminish the difference.
- *Omit summary statistics.* For example, showing only the mean or median values, e.g. in a line or bar chart, omits the uncertainty in the data. It is better to also



https://en.wikipedia.org/wiki/Misleading_graph

Figure 2: Truncated Axes

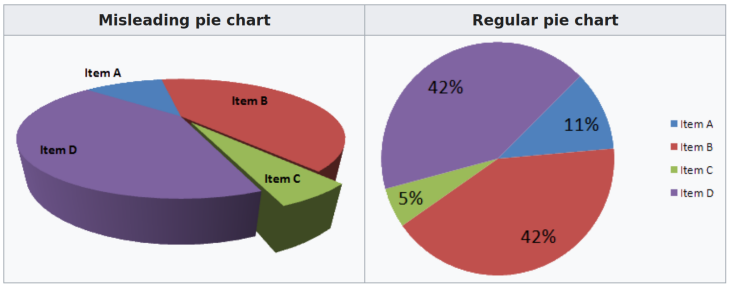


https://en.wikipedia.org/wiki/Misleading_graph

Figure 3: Scaling Axes and Aspect Ratios

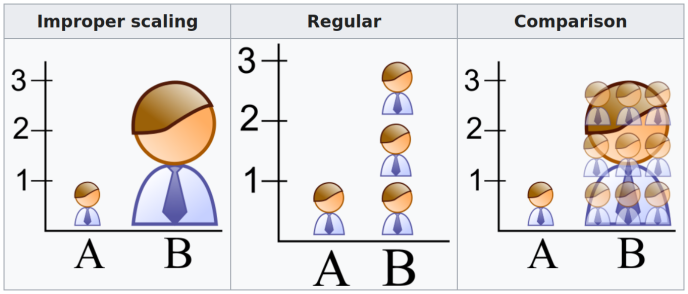
include error bars, information about quartiles or outliers in the chart to show variability or uncertainty, especially when there is significant uncertainty about differences or absolute values.

- *Truncate or scale axes to hide or exaggerate trend.* Truncating or scaling axes leads to increased slopes of lines or perceived differences between points or levels. This exaggerates differences or trends. Figure 2 shows an example of how small differences (right) can be exaggerated (left) in a bar chart. Similarly, Figure 3 shows how scaling or the use of different aspect ratios can be used to visually exaggerate or diminish trends or relationships between variables.
- *Scale in multiple dimensions.* The relative change or difference should be represented by a single dimension only. For example, in a bar or column chart, only the height or length of bar/column should change, not its width or area as well. This issue is often connected to the use of 3-dimensional graphics. While visually appealing, they exaggerate the apparent visual area of a foreground object, as illustrated in Figure 4. A related issue is the use of images in graphs, shown in Figure 5. In the improper scaling, the image is enlarged in two dimensions, visually suggesting a larger difference than there actually exists in the data.
- *Plot cumulative growth to hide trend.* A cumulative trend will always a positive trend, even as the contribution of individual items decreases sharply.
- *Use maps for non-geographic data.* Maps represent geographical area, rather



https://en.wikipedia.org/wiki/Misleading_graph

Figure 4: 3D Pie Charts



https://en.wikipedia.org/wiki/Misleading_graph

Figure 5: Scaling Multiple Dimensions

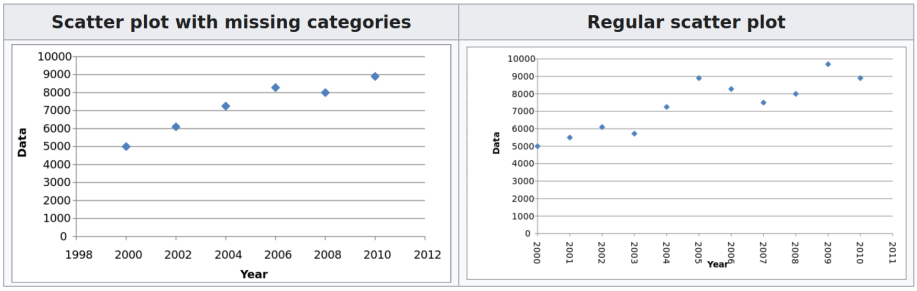
than population or some other variables of interest. For example, coloring a map by voter preference visually overemphasizes thinly populated but large geographic areas.

- *Use incomplete data ("cherry-picking").* This includes examples such as showing only the previous year's data, instead of data for the previous five years to hide a trend, showing quarterly data instead of weekly data to hide volatility, or showing every data for every second month instead of for every month to hide specific data points or trends. Figure 6 shows an example of this dark pattern.
- *Use invalid data.* When data is known to be unreliable, that is, its quality is low, its uncertainty may be high, and it has a large error rate, it is misleading to use it to convey a quantitative message.

The comics in Figure 7, taken from the popular XKCD website[1] shows some of these visualization dark patterns in a humorous way.

In summary, misleading charts and visualizations can be particularly problematic because they exploit the visual nature of human perception, making the deception less noticeable. It is crucial for both creators and consumers of data visualizations to be

[1] All XKCD comics are copyright by their creator (www.xkcd.com) and licensed under CC-BY-NC.



https://en.wikipedia.org/wiki/Misleading_graph

Figure 6: Incomplete Data

aware of these pitfalls and to approach data representation and interpretation with a critical eye.

3 Special Types of Data and Visual Analytics

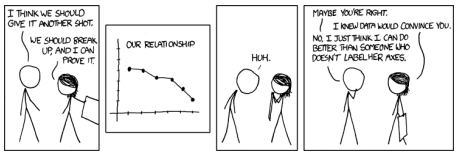
Streaming Data

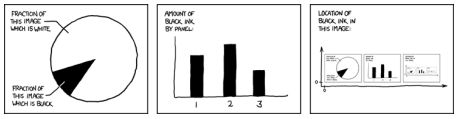
Visualizing streaming data, also known as real-time data visualization, involves the dynamic representation of data that is continuously updated as new data arrives. This type of visualization is essential in contexts where timely and rapid data interpretation is critical, such as in financial trading, or network monitoring.

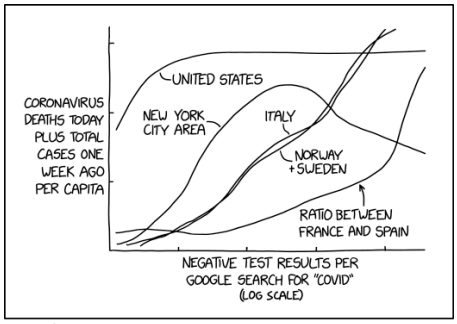
Streaming data presents some specific challenges for visualization. One of the primary challenges is managing the high velocity and volume of streaming data. The system must process and visualize data quickly enough to keep up with the incoming stream. Typically, only a limited window of data is available while older data is discarded. This means that, since the focus is on real-time data, it can be challenging to provide sufficient historical context for users to understand the current data in a broader temporal perspective. Moreover, due to the highly dynamic nature of the data, presenting streaming data in a way that is not overwhelming to the user is challenging. The visualization must strike a balance between providing enough detail and overloading the user with information, in particular information about changes, and also a balance between providing responsive graphs and overloading the user with such rapid changes they lose the ability to understand the data.

Spatial Data

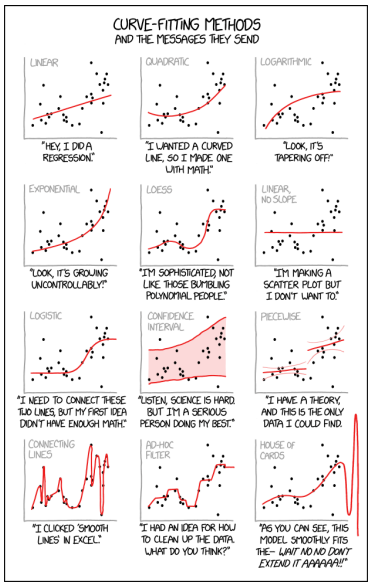
Visualizing geospatial or geographical data involves representing information that has a spatial component on a map or in a spatial context. While this type of visualization can be powerful for revealing patterns and insights related to location and geography, it presents some unique challenges. Geospatial data is often complex and multidimensional, encompassing not only locations but also attributes like time, elevation, population density, and more. Geospatial datasets can be very large, especially with

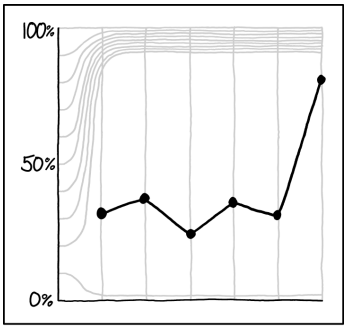




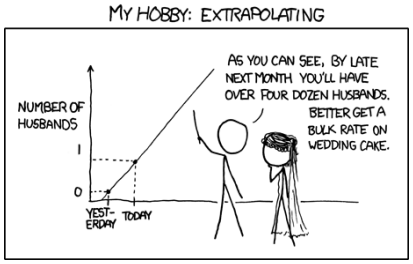


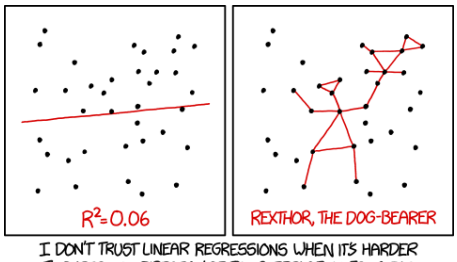
I'M A HUGE FAN OF WEIRD GRAPHS, BUT EVEN I ADMIT SOME OF THESE CORONAVIRUS CHARTS ARE LESS THAN HELPFUL.



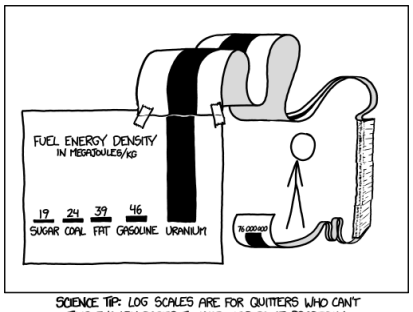


PEOPLE HAVE WISED UP TO THE "CAREFULLY CHOSEN Y-AXIS RANGE" TRICK, SO WE MISLEADING GRAPH MAKERS HAVE HAD TO GET CREATIVE.



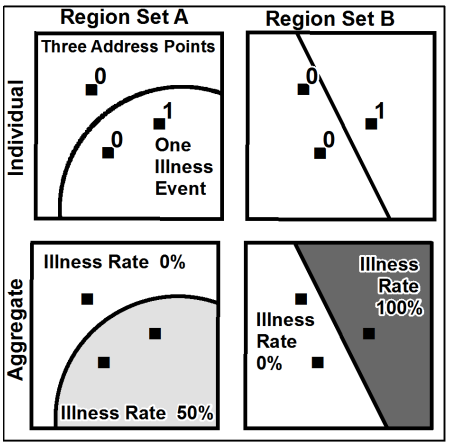


I DON'T TRUST LINEAR REGRESSIONS WHEN IT'S HARDER TO GUESS THE DIRECTION OF THE CORRELATION FROM THE SCATTER PLOT THAN TO FIND NEW CONSTELLATIONS ON IT.



SCIENCE TIP: LOG SCALES ARE FOR QUITTERS WHO CAN'T FIND ENOUGH PAPER TO MAKE THEIR POINT *PROPERLY*.

Figure 7: Misleading Visualizations, Comics by XKCD



https://en.wikipedia.org/wiki/File:Maup_rate_numbers.png

Figure 8: Different types of spatial divisions lead to different interpretations

the advent of satellite imagery, IoT (Internet of Things) sensors, and other sources of big data. Moreover, the granularity of physical space can range from very small areas of a few square meters to very large areas, such as provinces or states. For example, postal-code level data can produce very large data sets, even in small jurisdictions.

A specific problem is the choice of areal unit to use for data analysis or visualization. For example, location data points can be aggregated by counties or districts, by postal code areas, by school districtcs or school intake areas, by police or fire service coverage, or many others. Each of these different areal units will lead to different data summaries and therefore also to different visualizations. Choosing the type of area to use as the basis for visualization can have a large impact on the insights gained or the messages conveyed to the audience. A simple example is shown in Figure 8 that shows how aggregate statistics depend on the type of areal unit or boundary.

Another particular challenge with spatial data is mapping the three-dimensional Earth onto a two-dimensional surface. This mapping inevitably involves some form of projection, which can distort spatial relationships. Choosing an appropriate map projection that minimizes distortion for the specific data and use case is a critical challenge. There are many such projections[2], that distort or leave undistorted various properties such as lengths, areas, or angles. Figure 9 shows some of these issues in a humorous way.

Network and Graph Data

Visualizing network or graph data involves representing entities as nodes and the relationships between them as edges in a graphical format. This type of visualization is crucial for understanding complex structures in various fields like social network analysis, biology, computer science, and more. Typically, nodes are represented as boxes,

[2] https://en.wikipedia.org/wiki/Map_projection

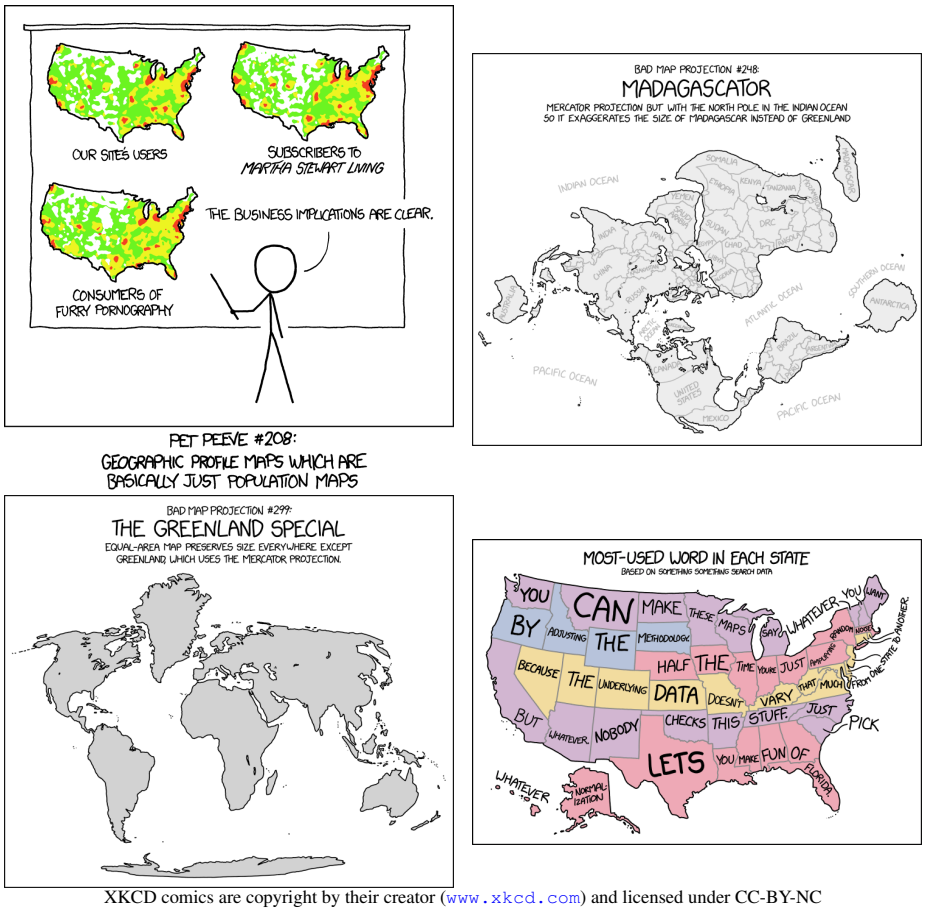
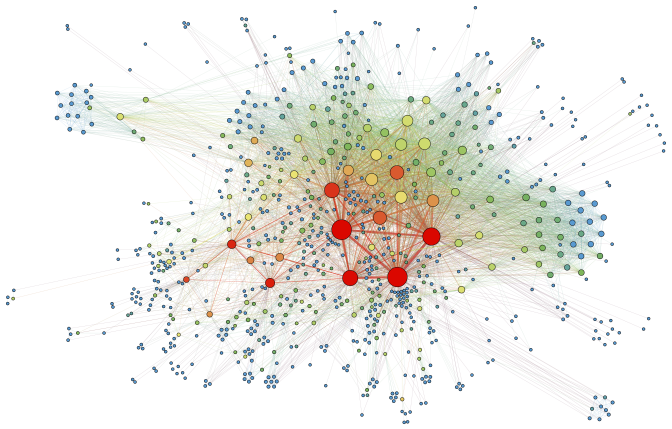


Figure 9: Map Visualization Comics by XKCD

circles, or textual labels, while edges are represented as lines or curves. Directed graphs use arrowheads on lines or curves to indicate the directionality of an edge.

As the number of nodes and edges increases, the visualization can quickly become a tangled mess, making it difficult to discern meaningful patterns or relationships. To effectively explore graph data interactive features like zooming, panning, and highlighting are essential. Graphs may also contain large sets of attributes for nodes and edges. Representing these attributes effectively without cluttering the visualization or overwhelming the viewer is challenging. Techniques like color coding, sizing, or shaping nodes and edges are commonly used techniques, but require careful design.

In densely connected networks, edges can overlap, and nodes can occlude each other, leading to a loss of information and making it difficult to trace relationships or identify individual elements. There exist many different ways to visually layout a graph to make it visually clear and easy to understand.



https://commons.wikimedia.org/wiki/File:SocialNetworkAnalysis.png

Figure 10: Force-directed graph layout example

One of the most commonly-used types of algorithms positions graph vertices based on the physical metaphors of attractive and repulsive forces, for example an imaginary system of physical springs, sometimes called a force-directed graph layout. Adjacent vertices are modelled with an attractive force, while all vertices have a repulsive force. The graph layout algorithm then tries to produce a layout in which an overall energy function is minimized. Figure 10 shows an example of such a graph layout.

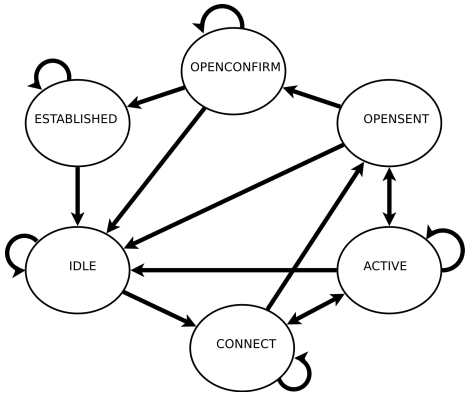
Another commonly used graph layout algorithm is the simple circular layout, where nodes are arranged equidistantly around a circle with edges drawn as lines or arrows. Figure 11 shows an example this type of graph layout.

In an arc diagram, as shown in Figure 12, the nodes are arranged on a straight line while edges are drawn as semicircles between nodes. In this layout, it is important to arrange the nodes to minimize the number of crossings of edge semicircles.

A common type of layout for directed and acyclic graphs is the layered graph, typically layed out from top to bottom or from left to right. The layout begins at the root node or nodes, and increments the layer for each edge between adjacent nodes, as shown in Figure 13.

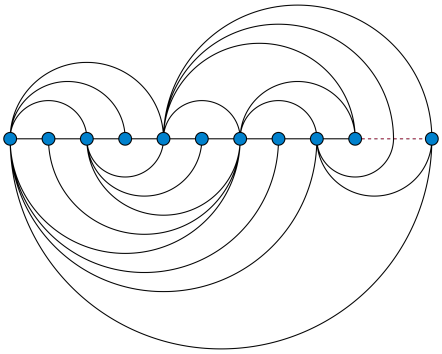
4 Color Palettes

The use of color in data visualization is crucial, serving not only to enhance the visual appeal of a graphic but also to improve its clarity and interpretability. Color choices in data visualizations, determined by the selected color palettes, play a significant role in distinguishing different data points or categories, setting the tone of the presentation (for example, formal versus informal presentations), ensuring accessibility for viewers with color vision deficiencies, and enhancing the overall aesthetic appeal. Desirable



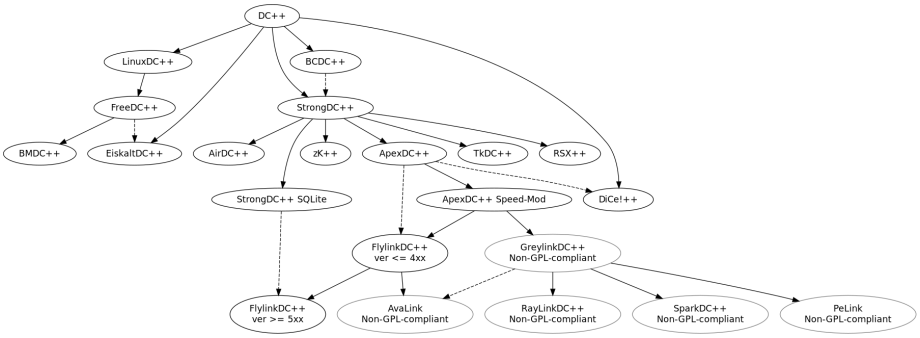
https://commons.wikimedia.org/wiki/File:BGP_FSM_3.svg

Figure 11: Circular graph layout example



https://commons.wikimedia.org/wiki/File:Goldner-Harary-linear.svg

Figure 12: Arc graph layout example



https://commons.wikimedia.org/wiki/File:DC%2B%2B_derivatives.svg

Figure 13: Layered graph layout example

characteristics of color palettes are:

- *Range of Values*: Colorful palettes are required when many different values have to be represented and distinguished.
- *Perceptual Unformity*: The relative perceived differences between colors in the palette should mirror the relative differences in the data values represented by the colors.
- *Robustness to Color Vision Deficiency*: Colour vision deficiency (CFD), colloquially called ”color blindness” impacts almost 10% of the population and must therefore be a consideration when choosing color palettes so that the data visualization can be properly perceived and interpreted by everyone.
- *Consistency*: When using multiple plots, their color palette should be the same or at least consistent so as not to cause confusion in interpretation and require less effort for understanding by the reader.
- *Aesthetic Appeal*: Finally, a colour palette should also be ”pretty”.

Types of Color Palettes

Color palettes can be distinguished by the number of colours they use, and whether the colors span a continuous color space or are a discrete set.

Sequential color palettes

Sequential color palettes, like the one in Figure 14a, use a single color and vary the hue or depth of the color. They are best used for data that has an inherent order, as they clearly show progression or gradation. However, they are not suitable for data that lacks a natural ordering. The *monochromatic color palette* is a special case of a sequential palette. This may be suitable when it is likely that the output will be printed on media without the use of color.

Diverging color palettes

Diverging color palettes, like the one in Figure 14c use two colors as anchors and use gradations either through white, as in Figure 14c, or through black. They are ideal for emphasizing deviations from a median or mean value, or for highlighting extremes on either side of a critical midpoint. However, these palettes may be misleading if used for data without a meaningful center.

Spectral color palettes

Spectral color palettes, like the one in Figure 14d use a variety of different colors without any implicit ordering. They are used to represent discrete categories without inherent ordering, and are useful for differentiating distinct groups of data. The downside is that they can become confusing with too many categories.

(a) Sequential color palette

(b) Monochromatic color palette

(c) Diverging color palette

(d) Spectral color palette

Figure 14: Types of Color Palettes

Sequential and diverging color palettes may be *discrete*, like the ones shown in Figure 14, or *continuous*, while spectral palettes are always discrete.

Color Vision Deficiency

The human eye contains three different types of color receptor cells, called ”S-cones” that perceive the color blue, ”M-cones” that perceive the color green, and ”L-cones” that perceive the color red. Color vision deficiency (CVD) is a biological impairment where some color receptor cells in the eye are missing, less frequent, or their function

is diminished. In *protanopia*, the S-cones are missing or impaired, in *deuteranopia*, the M-cones are missing or impaired, and in *tritanopia*, the L-cones are impaired. When all are missing or non-functional, one speaks of *monochromatism*. CVD is a fairly common disability, afflicting approximately 1 in 12 men and 1 in 200 women, with an overall incidence rate in Canada of more than 5%.

To show the different types of color deficiencies, consider the images in Figure 15. Figure 15a shows the original image as it is perceived by a person who does not suffer from CVD. The remaining four images show how the photo appears to persons with different types of CVD.

Realizing the prevalence and the effects of CVD means that the color palette that is chosen for data visualization should be interpretable for and lead to the same interpretation even for readers with CVD. For example, the Viridis color palette available in many visualization software packages was designed with CVD readability in mind. Compare the popular "Color Brewer Paired" palette in Figure 16 to the Viridis palette in Figure 17. The figures show that the Viridis palette is readable and interpretable with any CVD condition, whereas the Paired palette is not because some colors cannot be distinguished under various CVD conditions.

In summary, the thoughtful application of color in data visualization is not merely an artistic decision but a strategic one. It influences how effectively the data is communicated and understood, ensuring that visualizations are not only informative and accurate, but also inclusive and engaging to a diverse audience.

(a) Original Image (MUN Faculty of Education Class Room)

(b) Monochromatism

(c) Protanopia

(d) Deuteranopia

(e) Tritanopia

Figure 15: Simulated Color Vision Deficiencies

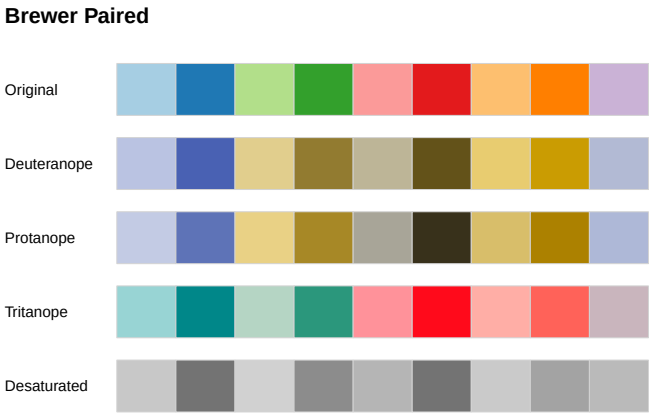


Figure 16: Colourbrewer ”Paired” Colour Palette

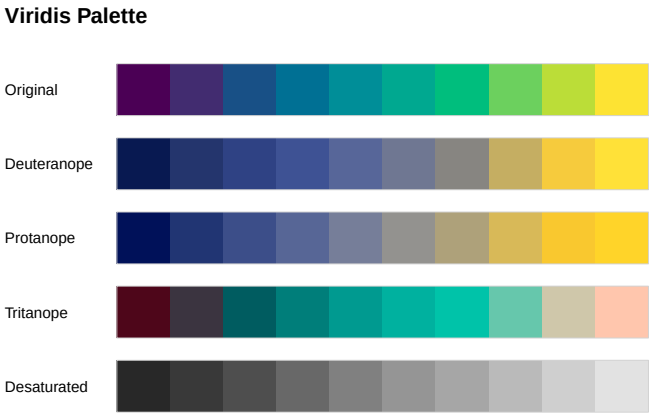


Figure 17: Viridis Colour Palette

5 Common Types of Plots

Depending on the number of variables to visualize, whether they are discrete or continuous, and the quantitative message to convey, different types of plots may be chosen. While the list of plot types presented here is not comprehensive, and new ways of visualizing data are constantly being invented, these are widely used plot types that are available in most visualization software packages and can be created with little effort.

- Plots for One Variable
  - Continuous
    - **Area**: Degree of change over time, or relationship of parts to aggregate
    - **Density, Dot, Frequency, Histogram**: Show frequency distribution of data
  - Discrete
    - **Bar**: Connections among individual things, compare items of different groups
    - **Pie**: Relationships of parts to aggregate
- Plots for Two Variables
  - Both Continuous
    - **Point**: Connections among numeric values, show multiple groups of data
    - **Lines, Local Regression**: Relationships/correlations among multiple data series or over time
    - **Text / Label**: Frequency of labels in content/document
  - One Discrete, One Continuous
    - **Column**: Correlations among things or information changes over time
    - **Box, Dot, Violin**: Compare distributions between many groups, display spread and skew of data
  - Both Discrete
    - **Points/Counts**: Magnitude of counts
    - **Jitter**: Plots of data points
  - Distributions of Two Variables
    - **Bin2D, Density2D, Hex**: Shows frequency of values over two continuous variables
- Plots for Three Variables
  - Continuous

        * **Contour, Raster and Tile**: Shows relationships among three data series

- Visualizing Errors and Uncertainty
  - Give a general idea of how precise a value is, or how far a value might be from the true value
  - Typically used to augment a given visualization
  - Common Visualization Styles:
    * Crossbar
    * Errorbar
    * Range (line, point)

6 Graphics Libraries and Frameworks

R

The R software system offers several powerful data visualization packages, each with unique features and strengths. Among the most prominent are ggplot2, Plotly for R, ggvis, and Shiny, which collectively cater to a wide range of visualization needs.

At the forefront is *ggplot2*, a package based on the Grammar of Graphics, which provides a coherent system for describing and building graphs. Its strength lies in its ability to create complex, multi-layered graphics with a syntax that is both powerful and expressive. ggplot2's approach allows users to build plots layer by layer, making it easier to handle and modify the components of a graphic. Its extensive customization options and the ability to handle a wide variety of graphical forms make it popular for static graphics.

Plotly for R integrates the functionality of the Plotly JavaScript library into R, enabling the creation of interactive, web-based graphs. This package extends the interactive capabilities of R visualizations, allowing users to produce graphics that can be zoomed, panned, and hovered over to reveal additional information. Its integration with R makes it a popular choice for adding an interactive element to data presentations, bridging the gap between static and dynamic visualizations.

ggvis, another package in the R visualization landscape, combines the concepts of ggplot2 with the interactivity of the web. It is designed to integrate well with R's reactive programming package, Shiny, and the dplyr package, enabling a smooth workflow for interactive data exploration. ggvis focuses on web-based, interactive visualizations, providing a syntax similar to ggplot2 but with additional capabilities to interactively change the data display and explore data in real-time.

Shiny, distinct from the traditional visualization packages, is an R package for building interactive web applications. It allows users to turn their analyses into interactive web

applications without requiring HTML, CSS, or JavaScript knowledge. Shiny applications have the power to not only display complex visualizations but also to interact with the user, making it possible to dynamically change the data, the types of plots, filters, and other aspects of the visualization based on user input. This interactivity makes Shiny particularly useful for creating data dashboards, where users need to explore and interact with data in a flexible manner.

Together, these packages provide R users with a comprehensive toolkit for creating static and interactive visualizations. From detailed and layered static plots with ggplot2 to dynamic, user-driven applications with Shiny, the R ecosystem enables a vast array of data visualization possibilities, catering to both simple and complex, interactive data exploration and presentation needs.

Python

The Python programming environment also offers a rich landscape of data visualization packages, each tailored to different needs and preferences.

Matplotlib is the foundational library for data visualization in Python, offering a wide array of functionalities to create static, animated, and interactive plots. It is highly customizable and capable of creating virtually any type of chart or graph. The versatility of Matplotlib allows for detailed control over plot elements, but this can also lead to more complex code for intricate visualizations.

Seaborn builds on Matplotlib and simplifies the creation of beautiful, informative statistical graphics. It integrates closely with Pandas, a data manipulation library in Python, and provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn's strength lies in its ability to create complex visualizations like heatmaps, time series, and violin plots with relatively straightforward commands.

Plotnine is inspired by R's ggplot2 library and brings the Grammar of Graphics to Python. It offers a similar layer-based approach to visualization, making it a familiar choice for users transitioning from R to Python. Plotnine is particularly effective for creating complex, multi-layered graphics with a syntax that emphasizes the declarative nature of the visualization process.

Plotly Express is a high-level interface for the Plotly library, designed to make it easy to create complex, interactive, and beautifully rendered visualizations. It offers a simple syntax for creating a wide variety of chart types and is particularly adept at handling large and complex datasets. Plotly Express's strength lies in its integration with Dash, another Plotly product, for building interactive web applications.

Plotly Graph Objects is the lower-level interface of the Plotly library, providing more granular control over the visualization elements. It's ideal for users who need to create highly customized visualizations or who require fine-tuning beyond what Plotly Express offers.

Plotly Dash is a framework for building interactive web applications with Python (and R and Julia). Dash is unique in its ability to create richly interactive, web-based data

visualizations and dashboards without requiring advanced knowledge of web development. It integrates seamlessly with Plotly's suite, allowing for the creation of sophisticated data visualization interfaces.

Bokeh, another prominent Python library, excels in creating interactive and real-time streaming visualizations. It is particularly well-suited for web-based dashboards and applications, offering both simplicity in creating complex interactive plots and the power to handle streaming datasets.

In summary, Python's ecosystem for data visualization is diverse and robust, ranging from Matplotlib's comprehensive capabilities for static plots to the interactive and web-based functionalities of Plotly and Bokeh. Each library offers unique strengths, whether it be in creating complex statistical visualizations, interactive web applications, or real-time data streams, catering to a wide range of data visualization needs and preferences.

JavaScript/Web

JavaScript, being the standard language of web development, boasts several powerful data visualization libraries that are integral for creating interactive and dynamic visualizations on the web. Among these, D3.js, Chart.js, and Google Charts are particularly noteworthy, each with their unique capabilities and strengths.

D3.js stands out as the most sophisticated and flexible JavaScript library for data visualization. Its core strength lies in its ability to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. D3 allows for extremely detailed and sophisticated visualizations by giving developers direct control over the SVG or HTML output. This level of control enables the creation of complex, interactive, and highly customizable visualizations. However, this power comes with a steep learning curve and can be overkill for simpler visualizations.

Chart.js is a more lightweight and user-friendly alternative, specifically designed for creating simple yet beautiful and interactive charts. It uses HTML5 Canvas for rendering, which makes it efficient in terms of performance. Chart.js supports a variety of chart types, including bar, line, pie, radar, and more, all of which are responsive and mobile-ready by default. Its simplicity and ease of use make it a popular choice for developers who need to implement standard charts quickly and without the complexity of D3.js.

Google Charts provides an even simpler way of incorporating charts into web pages. It offers a wide array of chart types and is particularly known for its integration with other Google services, like Google Spreadsheets. Google Charts is designed to be easy to use, and it handles a lot of the heavy lifting behind the scenes, such as drawing the charts, which makes it an appealing option for users who prefer a more straightforward and less code-intensive approach. The downside is that it offers less customization compared to D3.js and is reliant on external Google services, which might raise privacy concerns or issues with data control.

Each of these libraries serves different needs within the web development and data visualization community. D3.js is ideal for creating complex, interactive visualizations

where control and customization are paramount. Chart.js offers a balance between simplicity and functionality, suitable for standard web-based charts. Google Charts, with its ease of use and integration with Google products, is excellent for straightforward visualizations where ease of implementation is a priority. The choice among these libraries largely depends on the specific requirements of the project, the complexity of the visualizations needed, and the developer's proficiency with JavaScript and web technologies.

7 Mapping Data to Plot Elements

Creating a basic visualization in two dimensions, such as bar chart, a line chart, or a bubble chart, means that data variables or data series must be mapped to visualization elements. This is the core of the visualization task, and the most fundamental choice the data analyst has to make. Table 1 shows plot elements that data variables can be mapped to. In principle, a different data variable can be mapped to each of these, resulting in potentially being able to represent more than a dozen variables in one diagram. However, in practice, the number of concurrent variables to represent should be limited to no more than 3, in order for the visualization to remain interpretable and not to require too much cognitive effort on the part of the reader. The next sections show examples of mapping different variables to different plot element.

X, Y, Z axes
Colour (of points, lines, areas, shapes)
Transparency ("alpha")
Patterns (within areas, shapes)
Size, Weight/Width (of points and lines)
Shape, Style (of points and lines)

Table 1: Plot elements that can be mapped to data variables

8 Visualization in R using ggplot2

This section provides an introduction to data visualization using the ggplot2 library in R. The example dataset for this section is the Fuel Consumption Ratings for battery electric vehicles, provided the Government of Canada through its Open Government Portal[3]. At the time of writing, the dataset was last updated on October 10, 2023. The dataset contains the variables shown in Table 2.

Reading and preprocessing the data is straightforward in R, shown in the following code block:

[3] https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64

Column	Data Type	Definition
Make	Discrete	Manufacturer
Model	Discrete	Model name
Year	Numeric	Model year
Category	Discrete	Small, Midsize, Large, Pickup, SUV, Station Wagon, etc.
City	Numeric	Consumption in l/100km equiv.
Hwy	Numeric	Consumption in l/100km equiv.
Comb	Numeric	Consumption in l/100km equiv.
Range	Numeric	Driving range in km

Table 2: Fuel efficiency data set variables

```
# Load tidyverse package
library(tidyverse)

# Read the data set to a Tibble
data <- read_csv('https://evermann.ca/busi4720/fuel.csv')

# Ensure vehicle category is a factor (categorical)
data$Fuel <- as.factor(data$Fuel)
```

Next, load the required graphics libraries. A number of extensions to the core ggplot2 library have been developed to provide additional capabilities, such as radar plots, pattern fills, providing more control over scales and axes, etc.

```
library(ggplot2)
library(ggpattern)
library(ggstream)
library(ggsci)
library(scales)
library(ggrepel)
library(ggradar)
```

The core `ggplot()` function can be used in a dplyr pipeline and accepts the processed data tibble. The core argument to `ggplot()` is the "*aesthetic*" that maps plot elements to data variables. The actual plots themselves are then added through the use of various "`geoms`". Such geoms respresent commonly used plot types. The geoms "inherit" the aesthetic specified in `ggplot()` and can add to it by including more variables mapped to different plot elements. More than one geom can be added to a plot, allowing the analyst to overlay plot types or combine plots for multiple data series or data sets. The final graph can be saved in a variety of different image formats. The following examples are intentionally kept simple and are not intended to show the full capabilities

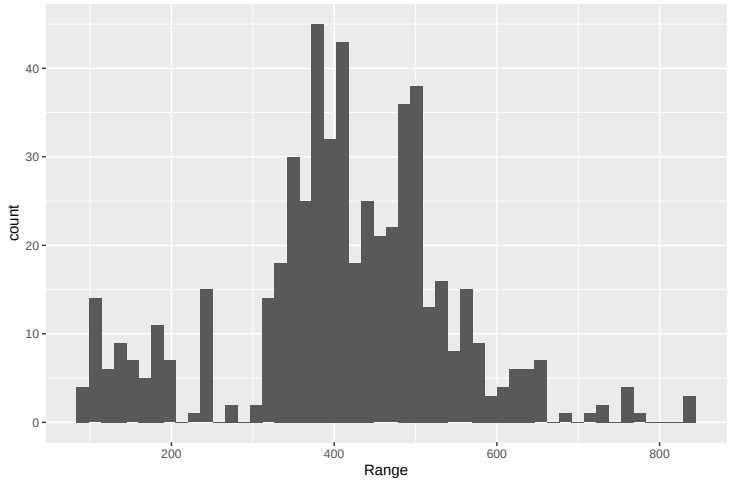
of the ggplot package. Many of the functions used below have a multitude of options to customize the output and a wide variety of ways, but this section focuses on the default visualizations they produce.

The first example below introduces the histogram geom. Histograms show the count of values in a certain range. The `ggplot()` function's aesthetic maps the "Range" variable of the data tibble to the x axis of the plot. The argument to the `geom_histogram()` function indcates that 50 bins should be formed, i.e. the data is divided in 50 separate regions for counting and plotting.

```
data |>
  ggplot(aes(x=Range)) +
    geom_histogram(bins=50)
```

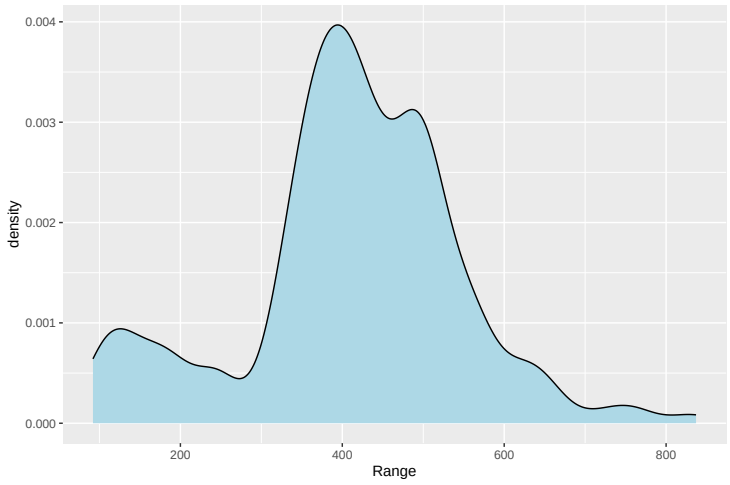
The `ggsave()` function saves the last plot in a file with the specified height and width. The function attempts to automatically determine the output format, such as JPEG or PNG raster images, PDF files, etc. from the file name ending. See the manual (`?ggsave`) for further options.

```
ggsave("histogram.pdf", height=5, width=7.5, units='in')
```



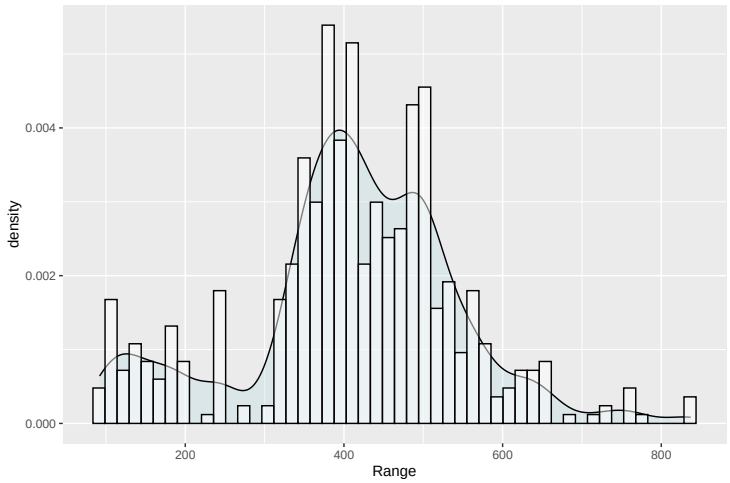
A density plot using the `geom_density()` function, is similar to a histogram in that it indicates the frequency of values. However, a density plot shows a continuous *probability distribution* of the data values, and as such the vertical axis is limited in range between 0 and 1. The `kernel` option for the `geom_density` geom specifies how the line is smoothed to produce the curve in the plot below.

```
data |>
  ggplot(aes(Range)) +
    geom_density(kernel='gaussian', fill='lightblue')
```



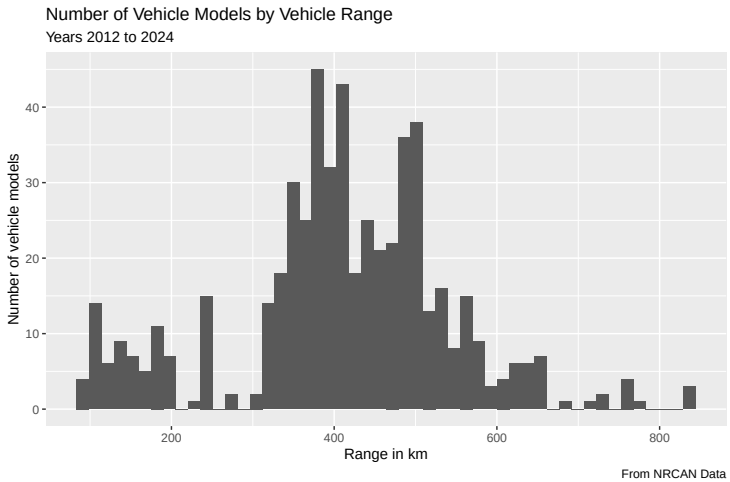
The next example shows how histograms and density plots can be combined simply by adding them to the plot using the "+" sign. Both geoms specify an `alpha` option that determines their transparency. This allows both to be seen. Note also that the y axis of the histogram is adjusted to match the values calculated and visualized by the density geom.

```
data |> ggplot(aes(Range)) +
    geom_density(kernel='gaussian',
        alpha=0.25, fill='lightblue') +
    geom_histogram(aes(y=after_stat(density)), bins=50,
        alpha=0.5, fill='white', color='black')
```



A plot can be labelled. The ggplot2 package offers labels for the x and the y axes, as well as a title, a subtitle, and a caption for the figure. By default, the x and y axis labels are the data columns mapped to them. Labelling is done by adding the `labs` geom to the plot.

```
data |>  ggplot(aes(x=Range)) +
    geom_histogram(bins=50) +
    labs(x = 'Range in km',
         y = 'Number of vehicle models',
         title='Number of Vehicle Models by Vehicle Range',
         subtitle='Years 2012 to 2024',
         caption='From NRCAN Data')
```



Hands-On Exercise

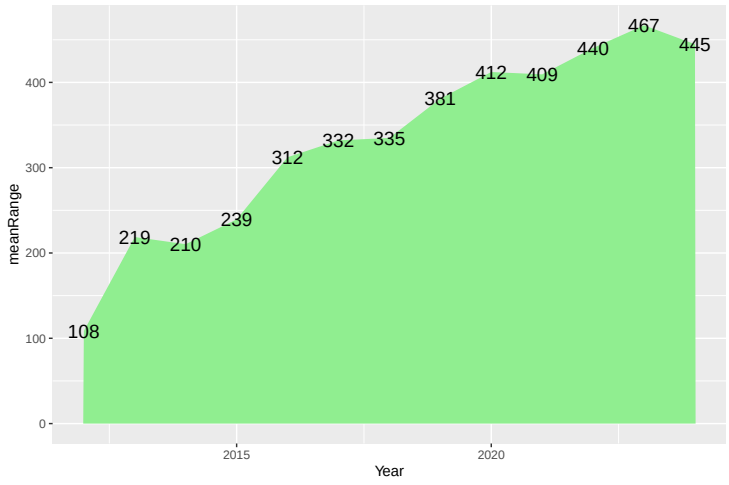
1. Read the EV fuel efficiency data set into R
2. Create a blue histogram of highway fuel efficiency with 10 bins.
3. Add labels for the axes, and add a title

Tips:

- Use the `read_csv()` function from the tidyverse library
- The column name is `Hwy`
- Use the `geom_histogram` geom
- Use the `bins=...` option
- Use the `fill='...'` option
- Use the `labs` geom for labels

An area plot is essentially a line plot where the area under the line is filled. The following example first uses dplyr functions to compute a summary statistic (the mean vehicle range by year), and then pipes that into the `ggplot()` function. The call to the `ggplot()` function is typically the last element in such a data processing pipeline. `geom_text()` is a way to add annotations to the data. This geom uses its own `label` aesthetic, as shown in the example below.

```
data %>%
  group_by(Year) %>%
  summarize(meanRange = mean(Range)) %>%
  ungroup() %>%
  ggplot(aes(Year, meanRange)) +
    geom_area(fill='lightgreen') +
    geom_text(aes(label=round(meanRange)), size=5)
```



The next example shows a column chart. The data needs to be in "long" format, so dplyr functions are used to create suitable summary statistics and then reshape the data.

```
col.data <- data %>%
   group_by(Year) %>%
   summarize(meanCity = mean(City), meanHwy = mean(Hwy)) %>%
   ungroup() %>%
   pivot_longer(
     cols=c('meanCity', 'meanHwy'),
     names_to='metric',
     values_to='consumption')
```

The two boxes below illustrate what the `pivot_longer()` function does. The first data tibble contains the summary statistics prepared by `group_by()` and `summarize()`. The two summaries are in two separate colums. This is called a "wide" format.

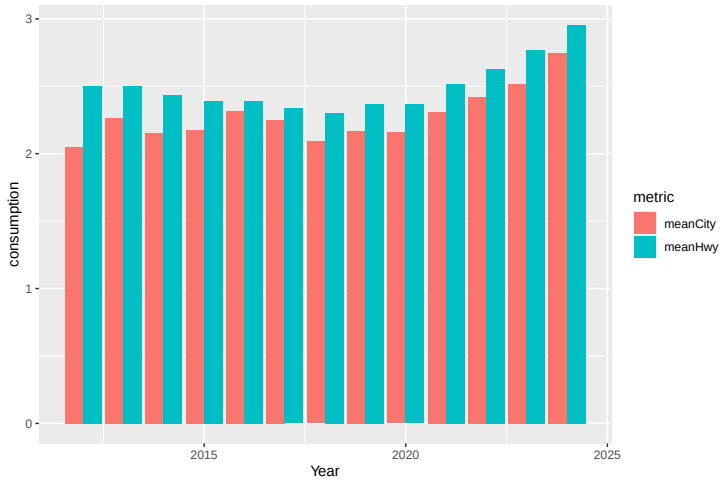
```
   Year meanCity meanHwy
  <dbl>    <dbl>   <dbl>
1  2012     2.05    2.5
2  2013     2.27    2.5
3  2014     2.16    2.43
```

The data tibble in the box below shows the result after the `pivot_longer()` function is applied. Each of the two specified columns is transformed into a spearate row entry. The column names are transformed to be values of the new column "metric" and the column values are moved to the new column "consumption". This format is called a "long" format.

```
   Year metric   consumption
  <dbl> <chr>          <dbl>
1  2012 meanCity        2.05
2  2012 meanHwy         2.5
3  2013 meanCity        2.27
4  2013 meanHwy         2.5
5  2014 meanCity        2.16
6  2014 meanHwy         2.43
```

Producing the column chart itself is simple with the `geom_col()` geom. The variable "metric" is mapped to the "fill" element of the plot, that is, the color with which columns are filled. The `position='dodge'` argument to the `geom_col()` function indicates that columns are located next to each other, instead of being stacked on top of each other.

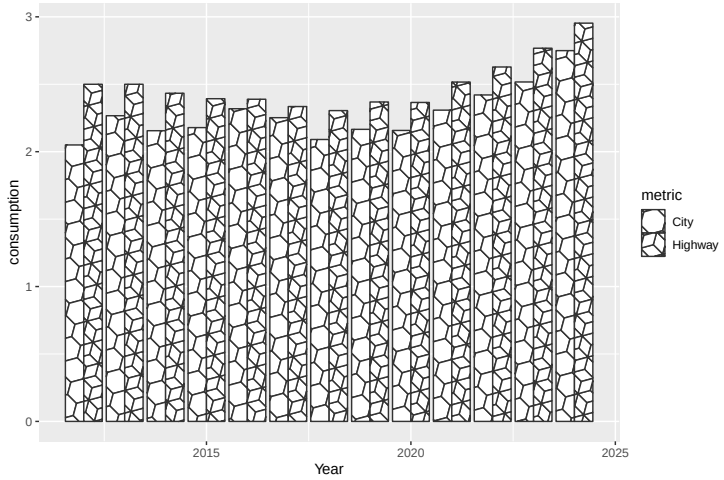
```
col.data |>
   ggplot(aes(x=Year, y=consumption, fill=metric)) +
      geom_col(position='dodge')
```



When it is clear that a plot is likely to be printed in black and white, it may be useful to omit the use of colours and instead use different fill patterns. The `ggpattern` package provides the `geom_col_pattern` geom. As shown in the code below, the aesthetics for this geom can map data values to different aspects of a fill pattern, such as the pattern type and the pattern angle. In the example below, the metric variable is mapped to the pattern type. The other options to the geom specify how the patterns are applied (angle, fill, etc.).

This example also customizes the scale for the "pattern" geom using `scale_pattern_type_manual()` to provide values for the different pattern types and to provide labels for the two data series.

```
col.data |>
  ggplot(aes(x=Year, y=consumption)) +
    geom_col_pattern(aes(pattern_type=metric),
          pattern='polygon_tiling', pattern_angle=45,
          pattern_fill='white', position='dodge') +
    scale_pattern_type_manual(
       values = c('hexagonal', 'rhombille'),
       labels=c("City", "Highway"))
```



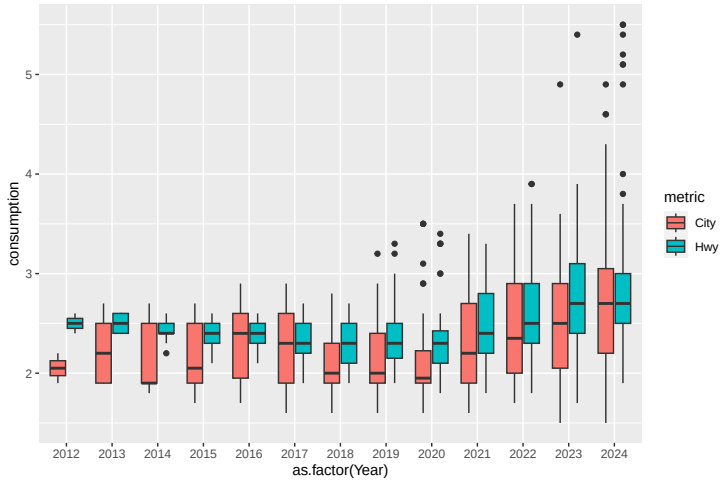
A box plot, also known as a box-and-whisker plot, is a way of displaying the distribution of data based on 5 summary statistics: lower bounds, first quartile (Q1), median, third quartile (Q3), and upper bounds. A box plot therefore provides a visual summary of the spread, central tendency, and symmetry of the data. Boxplots contain the following elements:

- *The Box:*'indexBox (in Box plot) The bottom and top edges of the box represent the first quartile (Q1, the 25th percentile) and the third quartile (Q3, the 75th percentile), respectively. The height of the box therefore describes the interquartile range (IQR), i.e. the distance between the first and third quartiles.
- *The Median:* The line inside the box denotes the median (the 50th percentile) of the dataset. By comparing the placement of the median line to the first and third quartiles, one can judge whether the data is skewed upwards or downwards.
- *Whiskers:* Extending from the box are lines called whiskers, indicating upper and lower bounds. Typically, the lower whisker extends to the smallest data point greater than $Q1 - 1.5 * IQR$ and the upper whisker extends to the largest data point less than $Q3 + 1.5 * IQR$.
- *Outliers:* Data points that fall outside of the whiskers are often considered outliers and may be plotted as individual points.

Box plots are particularly useful for displaying the distribution of data, comparing multiple distributions, and identifying outliers.

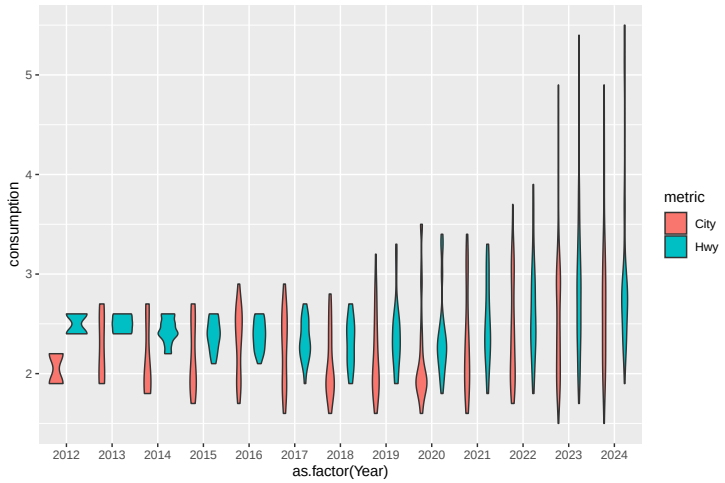
The following example of a box plot also requires reshaping the data from "wide" to "long" format using the `pivot_longer()` function, as was done for the column plot above.

```
data |>
  pivot_longer(cols=c('City', 'Hwy'),
               names_to='metric',
               values_to='consumption') |>
  ggplot(aes(x=as.factor(Year), y=consumption, fill=metric)) +
    geom_boxplot()
```



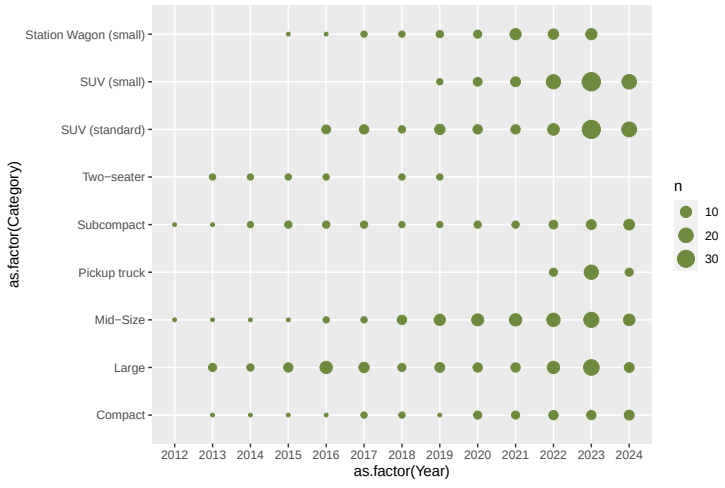
Violin plots are another way to visualize the spread and distribution of the data. Their width is determined by the frequency or distribution of data points. One can think of them as "sideways symmetric density plots". The following example introduces the `geom_violin` geom. In contrast to the box plot, the violing plot shows a density. For comparison to the box plots above, the example below plots the same data as the box plot example.

```
data |>
  pivot_longer(cols=c('City', 'Hwy'),
               names_to='metric',
               values_to='consumption') |>
  ggplot(aes(x=as.factor(Year), y=consumption, fill=metric)) +
    geom_violin()
```



A count plot is useful to show the count of data values as the size of a point. The message conveyed by this plot is similar to that of a histogram. Whereas the histogram shows the frequency or counts for one variable, a count plot does this for two variable. In the following example, the point size is determined by the count of values in each combination of ”Year” and ”Category”. All points have the same color. Additionally, the example shows customized axis labels for the discrete y axis.

```
data %>%
ggplot(aes(as.factor(Year), as.factor(Category))) +
    geom_count(color='darkolivegreen4') +
    scale_y_discrete(
      labels=c('Compact', 'Large', 'Mid-Size', 'Pickup truck',
               'Subcompact', 'Two-seater', 'SUV (standard)',
               'SUV (small)', 'Station Wagon (small)'))
```



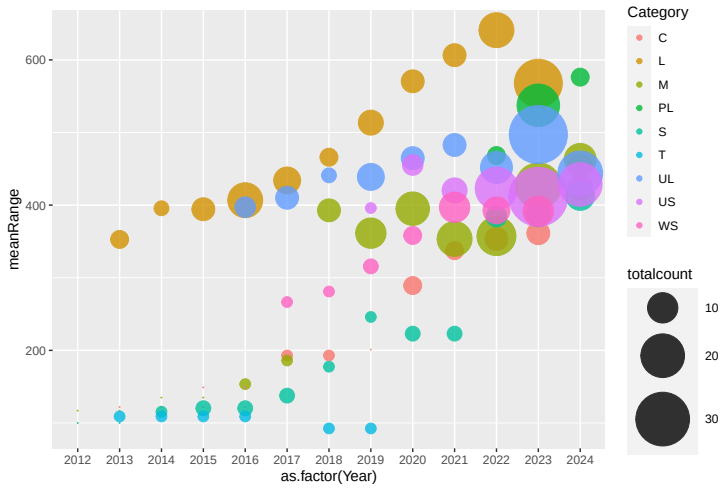
A similar message can be conveyed with a jitter plot, which shows all data points. ”Jitter” describes slightly, randomly changing the position of plot elements (points) so they do not overlap. This generates a point ”cloud” in the following example, where the size of the ”cloud” is used analogously to the size of the point. Visually, the following plot achieves a similar goal as the previous dot plot. Note that the same variable is mapped to both the x axis as well as the color element. Because of this, the plot omits the guides (that is, the legend) for the color because this information is shown in the x axis labels.

```
data |>
  filter(Year >= 2020) |>
  ggplot(aes(x=as.factor(Year), y=as.factor(Category),
             color=as.factor(Year))) +
    geom_jitter(width=0.2, height=0.2) +
    guides(color='none')
```

A points plot, sometimes called a bubble chart, generalizes the count plot. Whereas the count plot uses the number of data values to determine the size of the point, the points plot allows one to map a variable to the point size.

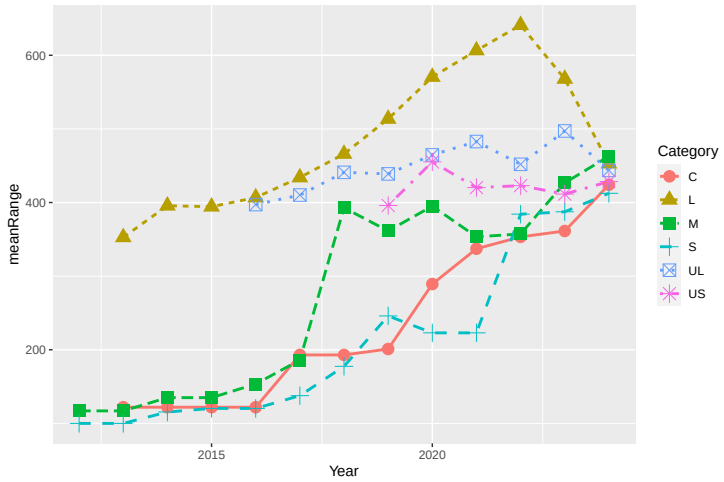
The following example summarizes grouped data and maps four variables to plot elements, pushing the limits of interpretability of the plot. The total count of observations for each year and vehicle category is assigned to the point size, the vehicle category to the point colour, and the year and mean vehicle range to the x and y axis respectively. The points are made slightly transparent (`alpha=0.8`) so they remain visible when they overlap, and their size is scaled to be between 0 and 20. Note that the legend contains information both for the size as well as the colour of the points.

```
data |>
  group_by(Year, Category) |>
  summarize(totalcount=n(), meanRange=mean(Range)) |>
  ungroup () |>
  ggplot(aes(x=as.factor(Year), y=meanRange,
             size=totalcount, color=Category)) +
    geom_point(alpha=0.8) +
    scale_size_continuous(range=c(0, 20))
```



The next example uses two geoms: `geom_line()` to show a line plot and `geom_point()` to also include the data points themselves. While visually not very informative in this case, the example illustrates an aesthetic that maps variables to five different plot elements. Vehicle category is mapped to three different plot elements, the colour (of both points and lines), the shape of a point, and the style or type of the line.

```
data |>
  filter(Category %in% c('C','L','M','S','US','UL')) |>
  group_by(Year, Category) |>
  summarize(meanRange = mean(Range)) |>
  ungroup() |>
  ggplot(aes(x=Year, y=meanRange,
             color=Category,
             shape=Category,
             linetype=Category)) +
    geom_line(size=1) +
    geom_point(size=4)
```



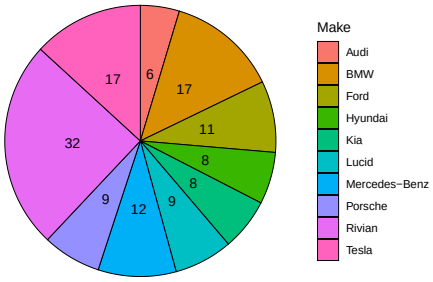
Pie charts are useful to show how different values add up to the whole. A pie chart is produced in ggplot2 by taking a stacked bar chart, and ”bending” it by plotting it on a polar coordinate system.

The following example uses the `coord_polar()` function to specify a coordinate system where the ”y” axis is mapped to the angle of rotation, `direction=-1` indicates clock-wise rotation and `start=0` indicates to begin the chart at the top of the ”pie”.

The `geom_text()` geom is used to specify labels and compute their position in the pie chart. It provides its own aesthetic for the label's color and position. Note that it assumes a stacked bar chart so that `position_stack(vjust=0.5)` positions the label vertically in the center of the area. When plotted in the polar coordinate system, this translates to the label centered in the pie slice.

Finally, the `theme_void()` function removes the usual chart elements like minor and major lines, the grey background, and axis ticks.

```
data |>
  filter(Year==2023) |>
  group_by(Make) |>
  summarize(totalcount = n()) |>
  filter(totalcount >= 5) |>
  ungroup() |>
  ggplot(aes(x='', y=totalcount, fill=Make)) +
    geom_bar(stat='identity', color='black', size=0.25, width=1) +
    coord_polar('y', direction=-1, start=0) +
    geom_text(aes(label=ifelse(totalcount >= 5, totalcount,'')),
                  position = position_stack(vjust=0.5)) +
    theme_void()
```



A radar plot, sometimes called a spiderweb plot, is useful to show a comparison of different objects on a range of variables. In R, the radar plot is produced by the `ggradar` library and requires its data in "wide" format. Rather than a single column that provides values for different categories, the values for each category must be provided in their own column. The following example computes summary statistics grouped by vehicle makes, and filters the data to retain only a few vehicle makes. All variables except the vehicle make are then scaled to a range between 0 and 1, i.e. standardized, using `mutate_at()` with the `rescale` function.

```
radardata <- data |>
  filter(Year == 2023) |> group_by(Make) |>
  summarize(meanCity = 1/mean(City),
            meanHwy = 1/mean(Hwy),
            meanRange = mean(Range)/100,
            nModels = n()) |>
  filter(nModels >= 5) |> ungroup() |>
  select(-nModels) |>
  mutate(across(-Make, rescale))
```

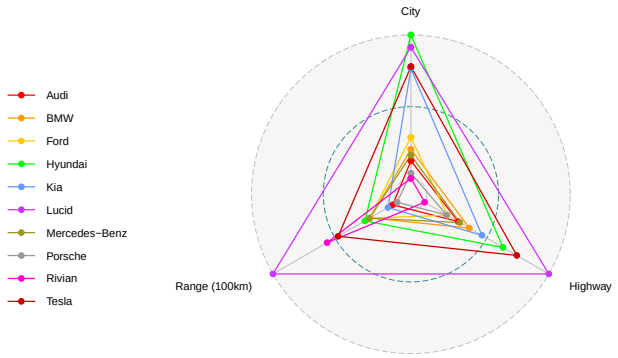
The box below shows an excerpt of the resulting data tibble. Note the "wide" format and the scaled, standardized values.

```
# A tibble: 10 x 4
   Make          meanCity meanHwy meanRange
   <chr>            <dbl>   <dbl>     <dbl>
 1 Audi             0.122   0.270    0.0357
 2 BMW              0.202   0.360    0.219
 3 Ford             0.287   0.182    0.220
 4 Hyundai          1       0.630    0.260
```

The radar plot does not require an aesthetic specification because it is based on the

number of columns in the data frame or tibble that is provided. Note the change of colour palette in the example below:

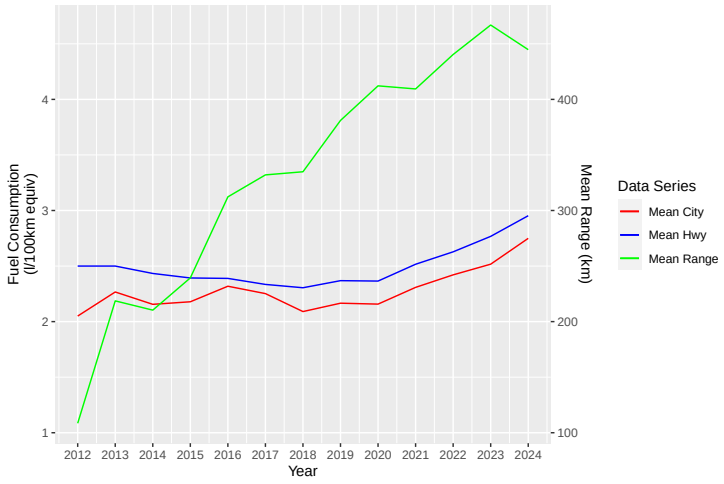
```
radardata |>
  ggradar(
    axis.labels=c('City', 'Highway', 'Range'),
    values.radar='',
    group.line.width=0.75,
    group.point.size=3) +
  scale_color_ucscgb()
```



Sometimes, it is useful to compare trends of variables that use different scales. However, beware of the potential for misuse; that is, it is easy to visually suggest correlations where none exist. The following example first groups the data by year and computes mean fuel consumption and range values for the plot. The plot includes three line plots (`geom_line()`) and specifies a secondary y axis using `sec.axis=sec_axis(...)`. In ggplot2, the secondary axis cannot be arbitrary but must be a scaled version of the primary axis. In this example, it is scaled by multiplying by one hundred using the formulat `.*100`. Accordingly the data is provided by dividing by a hundred using `mutate()`.

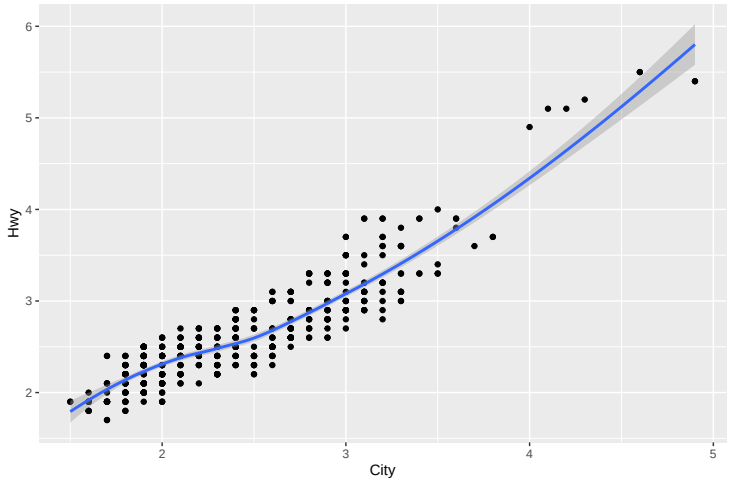
Colour values are set explicitly using the `scale_color_manual()` function rather than the `geom_line` geoms so that they appear in the legend of the plot. The labels for the continuous x axis are set explicitly to include labels for all years.

```
data |>
   group_by(Year) |>
   summarize(meanCity = mean(City),
             meanHwy = mean(Hwy),
             meanRange = mean(Range)/100) |>
   ungroup() |>
ggplot(aes(x=Year)) +
  geom_line(aes(y=meanCity, color='Mean City')) +
  geom_line(aes(y=meanHwy, color='Mean Hwy')) +
  geom_line(aes(y=meanRange, color='Mean Range')) +
  scale_color_manual(name='Data Series',
     values=c('Mean City' = 'red',
              'Mean Hwy' = 'blue',
              'Mean Range' = 'green')) +
  scale_y_continuous(labels=scales::comma,
      name="Fuel Consumption\n(l/100km equiv)",
      sec.axis=sec_axis(~ .*100,
                        labels=scales::comma,
                        name="Mean Range (km)")) +
  scale_x_continuous(breaks=seq(from=2012,to=2024,by=1))
```



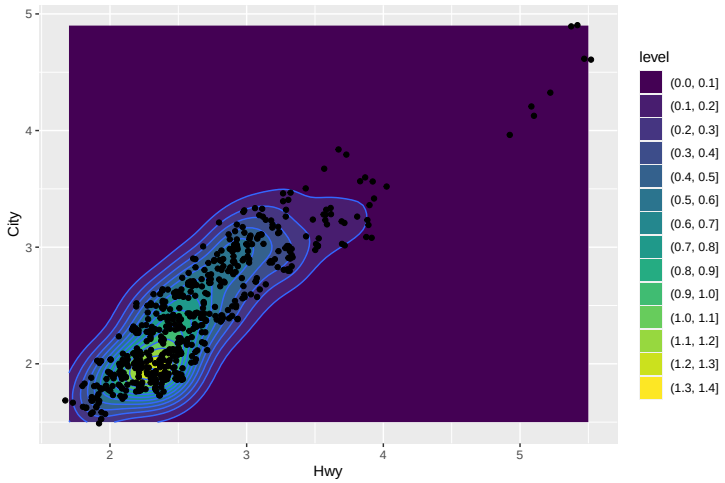
So called ”trendlines” can be added to plots easily with the `geom_smooth` geom. Different options to compute the trendlines exist, but the most frequently used one is the local polynomial regression, where the slope of the line is determined by a regression that uses data points in the vicinity of the line, weighted by their proximity. As with any regression, there is uncertainty around the estimated slope parameters (standard deviation or variance) and this uncertainty can be visualized as well, shown in the plot below by the gray area around the trendline. Note that the uncertainty is greater in areas where there are fewer data points, as would be expected.

```
data |>
  ggplot(aes(City, Hwy)) +
    geom_point() +
    geom_smooth()
```



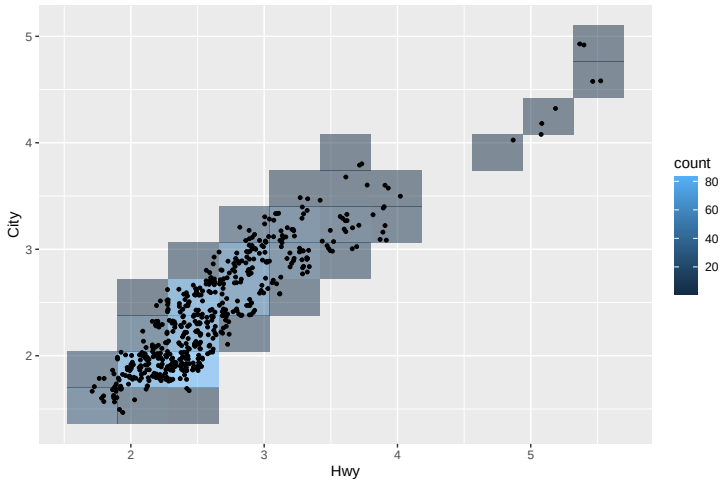
The one-dimensional density plots seen earlier can be generalized to show the two-dimensional joint probability density (co-distribution) of values of two variables. The following example uses three geoms. The `geom_density_2d_filled()` provides the colour fill for areas of same density. The `geom_density_2d` overlays this with the density lines, and the `geom_point` plots the actual data points, slightly jittered. Note that the order in which the geoms are specified matters so that the fill does not obscure the points or lines. Alternatively, transparency can be used for the colour fill using the option `alpha=0.5` or some suitable value.

```
data |>
  ggplot(aes(x=Hwy, y=City)) +
    geom_density_2d_filled() +
    geom_density_2d() +
    geom_point(position='jitter')
```



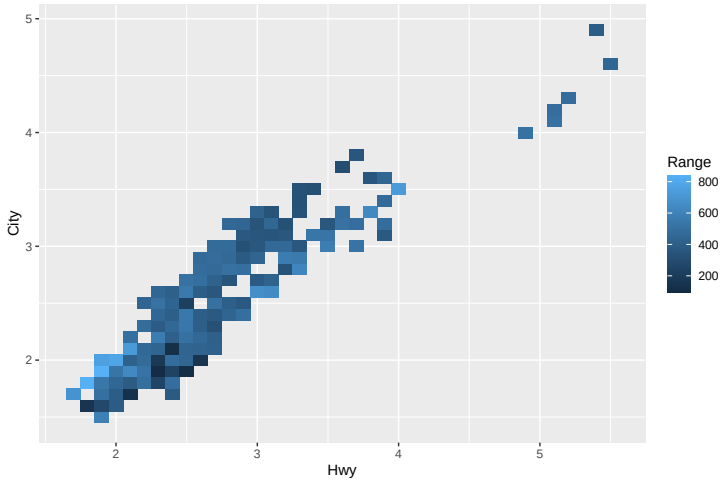
When one is interested in discretizing or "binning" the two variables, one can show the frequencies or counts in a two-dimensional bin plot. Again, individual data points are shown using the `geom_point` geom. The bin plot is essentially a generalization of a 1D histogram to two variables/dimensions and conveys a similar quantitative message. It is also similar to the the point plot above that used point size to indicate the frequency of observations, whereas the bin plot uses colour. However, whereas that point plot could be used for two discrete variables on the x and y axis, the bin plot here is be used for two continuous variables on the x and y axis.

```
data %>%
  ggplot(aes(x=Hwy, y=City)) +
    geom_bin2d(alpha=0.5, bins=10) +
    geom_point(color="black", size=1, position='jitter')
```



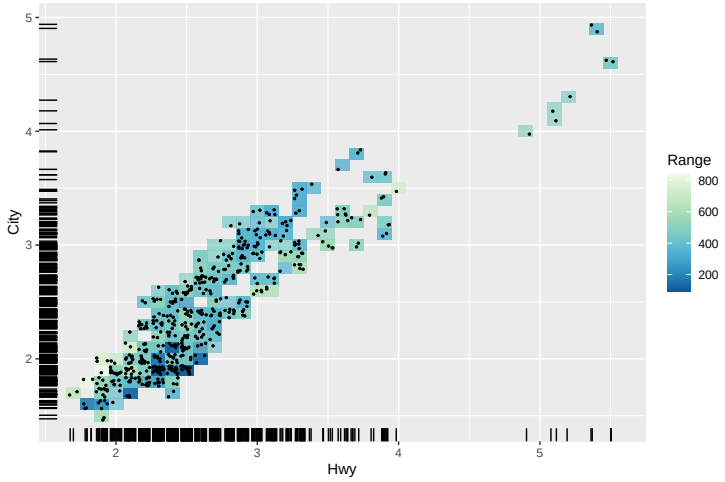
The previous two plots are plots of two variables even though they used the x and y axes and additionally used colour. However, colour was used only to indicate frequency or probability density information. If, instead of count or density, values of a third variable are to be shown, one can use the raster plot using the geom `geom_tile()`. This requires the aesthetic to specify a data variable mapping for the color element of the plot, as in the following example. Again, individual data points are included with the point geom.

```
data |>
  ggplot(aes(x=Hwy, y=City, fill=Range)) +
    geom_tile()
```



The last example shows the use of so-called ”rugs” that show marginal distributions of the plot variables. They can be added to different types of plots, including 3D raster plots as done here, as well as 2D bin and hex plots, or one-dimensional histograms. Rugs are added by using the `geom_rug()` function. Note the use of a non-standard fill colour scale to make the plot ”prettier”.

```
data |>
  ggplot(aes(x=Hwy, y=City, fill=Range)) +
    geom_tile() +
    geom_point(size=0.5, position='jitter') +
    geom_rug(position='jitter') +
    scale_fill_distiller(palette=4, direction=-1)
```



Hands-On Exercises Using the Pagila database files from the previous chapter on data analysis with R, create the following plots using ggplot2/R. Use the appropriate ggplot2 functions to add informative labels for axes, useful legends to the plots, and use suitable color palettes.

1. A histogram and/or density chart of film length by film category
2. A column chart of the mean rental payments for films by film category
3. A scatter plot of total rental payments by week
   - Add a local regression line to this plot
4. A pie or donut chart of rental counts by film rating

9 Visualization in Python using Plotly Express

This section demonstrates visualization in Python using Plotly Express. Plotly Express by default produces web-based, i.e. JavaScript based, interactive plots. On the Python side, the diagram is expressed in more primitive graphical descriptions, serialized in a JSON document, which is sent to the web browser, where the Plotly JavaScript library renders them. Interactivity includes the ability to zoom and pan the plot, and to hover over plot elements to get tooltip overlays, e.g. specific values of points or lines in the plots. Figures can also be saved in a variety of image file formats.

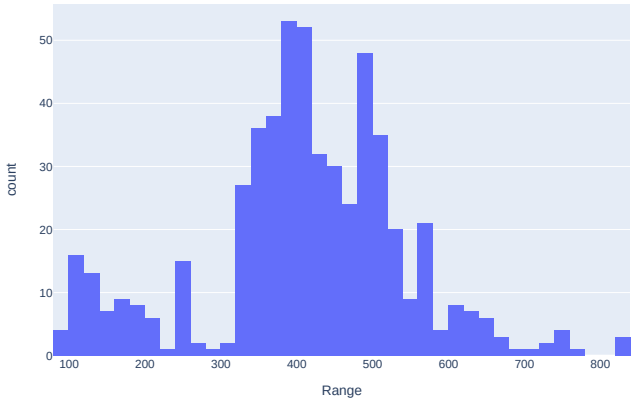
The examples in this section use the same data set as the R examples above, and as much as possible try to provide similar diagrams. As in the previous section, the code is kept simple, relying on default options, to help understanding and does not show the full capabilities or extent of customizability of the Plotly Express package.

The first Python code fragment imports the required packages and loads the data set.

```
import pandas as pd
import plotly.express as px
import plotly.io as pio
pio.kaleido.scope.mathjax = None

# Read data
fuel = pd.read_csv('https://evermann.ca/busi4720/fuel.csv')
```

The `histogram()` function does as its name suggests, and creates a histogram, that is, shows the frequency of values. After creating the figure, it can be shown or written as image to a file using a variety of format. By default, the `show()` function opens the standard web browser to show the figure in an interactive mode.

```
# Create histogram
fig = px.histogram(fuel, x='Range', nbins=50)

# Show histogram, by default show in web browser
fig.show()

# Save figure to image
fig.write_image("px.histogram.pdf", height=500, width=750)
```



The next example creates a column chart, showing different data variables or series. As with the R example in the previous section, the data must be provided in ”long” format. The Pandas function `melt()` is analogous to the `pivot_long()` function in R.

The following Python code block groups the data by Year and calculates the mean city and highway fuel efficiency for each year.

```
data_grouped = \
    data.groupby('Year') \
        .agg(
             meanCity=('City','mean'),
             meanHwy=('Hwy','mean')) \
        .reset_index()

data_long = \
    pd.melt(data_grouped,
            id_vars=['Year'],
            value_vars=['meanCity', 'meanHwy'],
            var_name='metric',
            value_name='consumption')
```

The first code block below shows the result of the grouping and aggregation. The data frame is in "wide" format, with each variable in its own column.

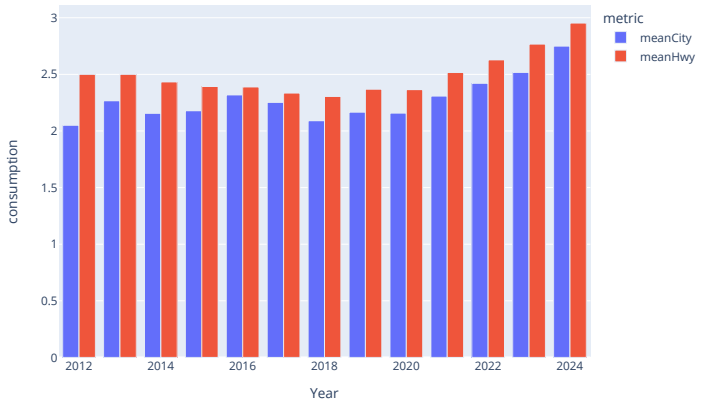
```
>>> data_grouped
    Year  meanCity   meanHwy
0   2012  2.050000  2.500000
1   2013  2.266667  2.500000
2   2014  2.155556  2.433333
3   2015  2.178571  2.392857
4   2016  2.318519  2.388889
```

The `melt()` function transforms the data into a "long" format, shown below. The values for the two values variables are moved to a new columns "consumption" and the names of the columns are moved to a new column "metric".

```
>>> data_long
    Year    metric  consumption
0   2012  meanCity     2.050000
1   2013  meanCity     2.266667
2   2014  meanCity     2.155556
3   2015  meanCity     2.178571
4   2016  meanCity     2.318519
```

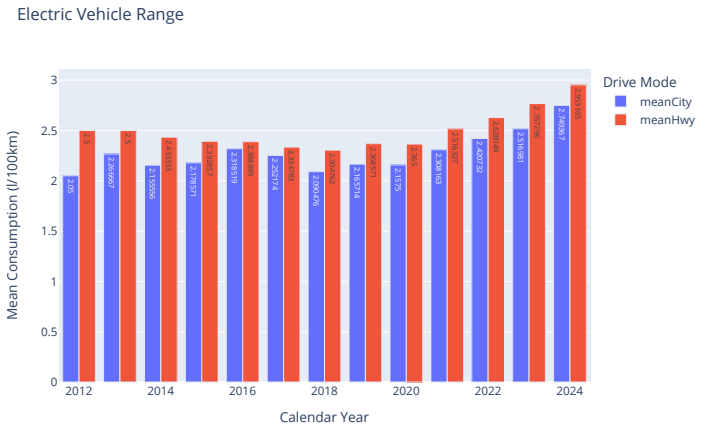
The column chart itself is produced using the `bar()` function. The function option `barmode='group'` indicates that the different data series are shown next to each other, rather than stacked on top of each others.

```
fig = px.bar(data_long,
   x='Year', y='consumption', color='metric',
   barmode='group')
```



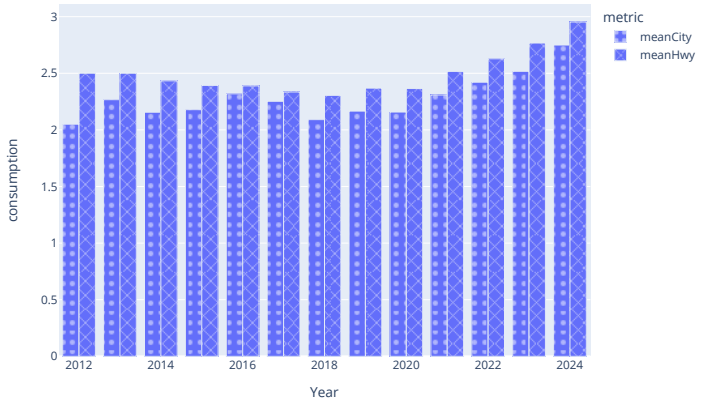
A Plotly Express figure has a title and labels for each axis or graph element that is used (such as colour or pattern). The following example shows the same column plot but adds explicit labels and a figure title using the `labels` and `title` options that are available in all Plotly Express figure creation functions.

```
fig = px.bar(data_long,
  x='Year', y='consumption', color='metric',
  barmode='group',
  text_auto=True,
  title = 'Electric Vehicle Range',
  labels={'consumption': 'Mean Consumption (l/100km)',
          'Year': 'Calendar Year',
          'metric': 'Drive Mode'})
```



When colour is not appropriate, for example because it is known the figure is reproduced or printed in grayscale, patterns can be used to distinguish the different data variables or data series. The following example shows how the `bar()` function in Plotly Express can produce patterned column charts simply by mapping a data variable to the `pattern_shape` graph element. The example also shows how the sequence of patterns that is to be used is specified.

```
fig = px.bar(data_long,
   x='Year', y='consumption', pattern_shape = 'metric',
   pattern_shape_sequence = ['.', 'x', '+', '|', '-', '/'],
   barmode='group')
```



Hands-On Exercise

1. Read the EV fuel efficiency data set into a Pandas data frame
2. Create a histogram of highway fuel efficiency with 25 bins.
3. Add labels for the axes, and add a title

Tips:

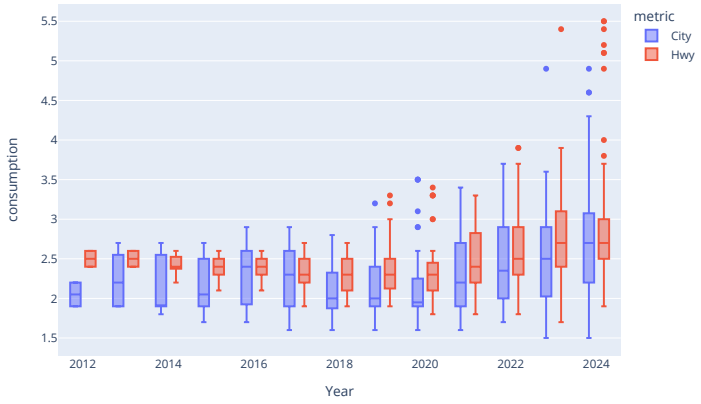
- Use the `pd.read_csv()` function from the tidyverse library
- The column name is `Hwy`
- Use the `px.histogram()` function
- Use the `title=...` option
- Use the `labels=[...]` option

A box plot is created using the `px.box()` function. Box plots show the distribution of values of a variable, indicating the 1st quartile (Q1, 25th quantile) and the 3rd quartile (Q3, 75th quantile) by the extent of the box, with the median (50th quantile) shown as a line across the box. The whiskers extend to $1.5 \times IQR$ above and below the 1st

and 3rd quartile, where $IQR = Q3 - Q1$. Outliers beyond the whiskers are typically shown as individual points.

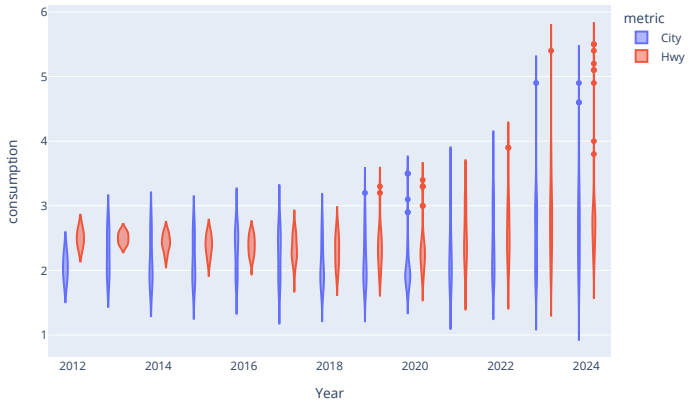
The data for a box plot must be provided in "long" format and the example below again uses the `melt()` function in Pandas to reshape the data appropriately. The box plot itself is created using the `box()` function in Plotly Express.

```
data_long = pd.melt(data,
    id_vars=['Year'],
    value_vars=['City', 'Hwy'],
    var_name='metric',
    value_name='consumption')
fig = px.box(data_long, x='Year', y='consumption', color='metric')
```



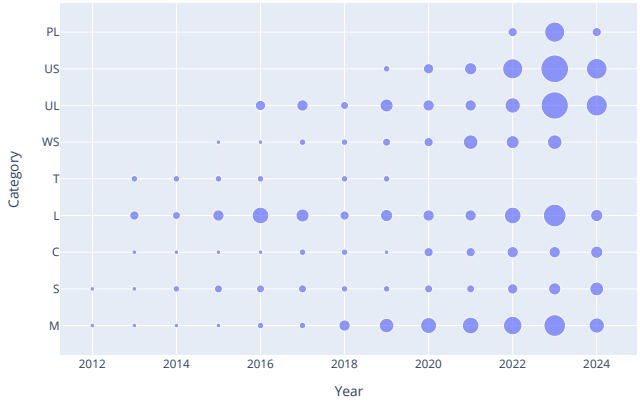
A violin plot is constructed similarly to a box plot. Using the same "long" data shape, the following example shows a violin plot. Violin plots also communicate the distribution of the data, but do not provide the summary information that box plots provide. Instead, they can be read as a "sideways density" plot indicating the probability distribution of the data values.

```
data_long = pd.melt(data,
    id_vars=['Year'],
    value_vars=['City', 'Hwy'],
    var_name='metric',
    value_name='consumption')
fig = px.violin(data_long, x='Year', y='consumption', color='metric')
```



A count plot communicates the number of observations in the size of a point. For the following example, the data frame is first grouped by year and vehicle category, and the size (that is, the count of values) of each group is recorded in a new column "counts". This transformed data frame is then used for a scatter plot that maps the "counts" variable to the size of the points in the `scatter()` function.

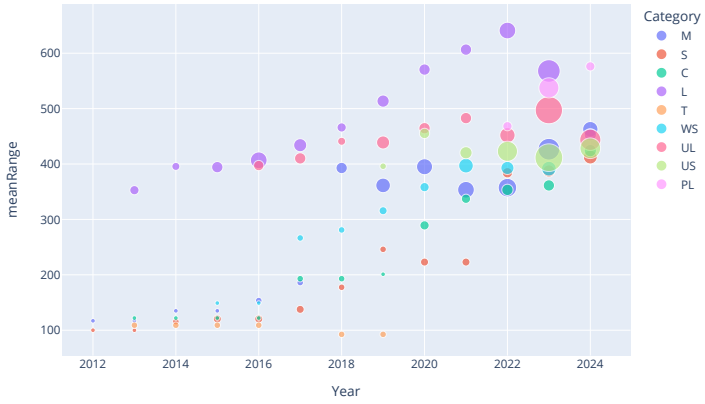
```
count_data = data \
   .groupby(['Year', 'Category']) \
   .agg(counts = ('Range', 'size')) \
   .reset_index()

fig = px.scatter(count_data, x='Year', y='Category', size='counts')
```



The following points plot is another example of the use of the `scatter()` function,

which adds a fourth variable to the plot: The vehicle category is mapped to the color plot element. Grouping and aggregtation now produces two aggregates, the count and the mean vehicle range.

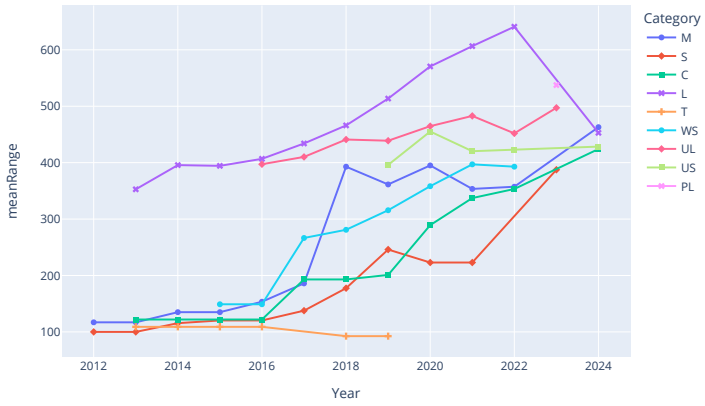
```
grouped_data = data \
    .groupby(['Year', 'Category']) \
    .agg(totalcount=('Range', 'size'),
         meanRange=('Range', 'mean')) \
    .reset_index()

fig = px.scatter(grouped_data,
           x='Year', y='meanRange',
           size='totalcount', color='Category')
```



A line plot shows can show trends or progression of data values over time or some other ordered metric. The following example computes mean vehicle range for different years and vehicle categories, and then selects a subset of categories to keep the resulting figure clear and interpretable.

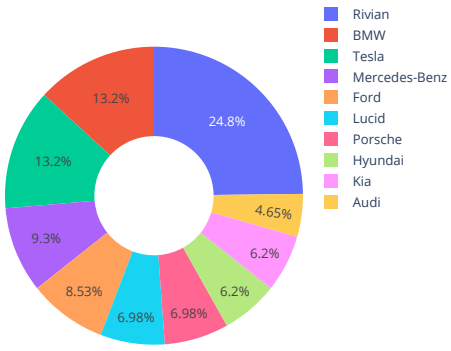
The line plot includes points as well, specified by the option `markers=True` and mapping the vehicle category variable to the `symbol` plot element.

```
filtered_data =
    data.groupby(['Year', 'Category']) \
       .agg(meanRange=('Range', 'mean')) \
       .reset_index() \
       [data['Category'].isin(['C','L','M','S','US','UL'])]

fig = px.line(filtered_data,
    x='Year', y='meanRange', color='Category',
    symbol='Category', markers=True)
```



Pie charts show the contribution of individual components or aspects to a whole. In Plotly Express they are created using the `px.pie()` function. The following example calculates the number of vehicle models for each make for the year 2023. The results are reduced to those makes with more than 5 models in order to limit the number of pie slices. The option `hole=0.4` transforms the pie chart into a donut chart by leaving a hole of the specified radius in the center.

```
data_pie = \
  data[data['Year'] == 2023] \
    .groupby('Make') \
    .agg(totalcount=('Model', 'size')) \
    .reset_index() \
    .query('totalcount >= 5')

fig = px.pie(data_pie,
             names='Make', values='totalcount',
             hole=0.4)
```

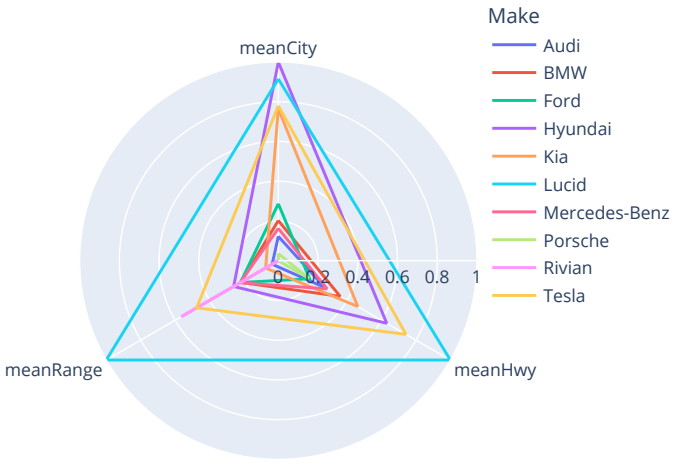


A radar plot can be used to compare observations on multiple variables. In Plotly Express, it is created as a line plot on a polar coordinate system, using the `px.line_polar()` function.

The following example first groups the data for the year 2023 by vehicle make and computes aggregate statistics for the comparison of vehicle makes. The grouped data is then filtered to reduce the amount of information shown in the radar plot in order to keep it readable and understandable.

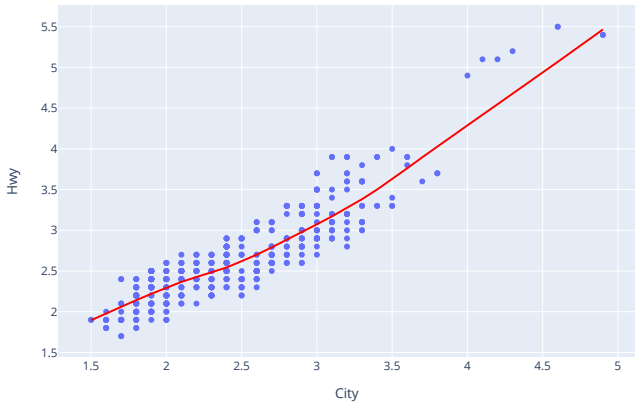
The aggregate values are then scaled to values between 0 and 1, using the `MinMaxScaler` from the `sklearn` package. Finally, the data frame is reshaped into ”long” format using `melt()` where the values are moved to a column `value` and the variable names are moved to a column ”variable”. The radar plot maps the value column to the radius (that is, how far from the center) and the variable column to the angle around the plot (theta). The vehicle Make is mapped to colour and lines are closed.

```
from sklearn.preprocessing import MinMaxScaler

grouped = data \
  .query('Year == 2023') \
  .groupby('Make') \
  .agg(
      Cty=('City',lambda x: 1/x.mean()),
      Hwy=('Hwy',lambda x: 1/x.mean()),
      Rng=('Range',lambda x: x.mean()/100),
      nModels=('Make','size')) \
  .query('nModels >= 5')

grouped[['Cty', 'Hwy', 'Rng']] = \
  MinMaxScaler().fit_transform(
    grouped[['Cty', 'Hwy', 'Rng']])

melted = grouped \
  .reset_index() \
  .melt(id_vars='Make',
        value_vars=['Cty', 'Hwy', 'Rng'],
        value_name = 'value',
        var_name = 'variable')

fig = px.line_polar(melted,
     r='value', theta='variable',
     color='Make', line_close=True)
```



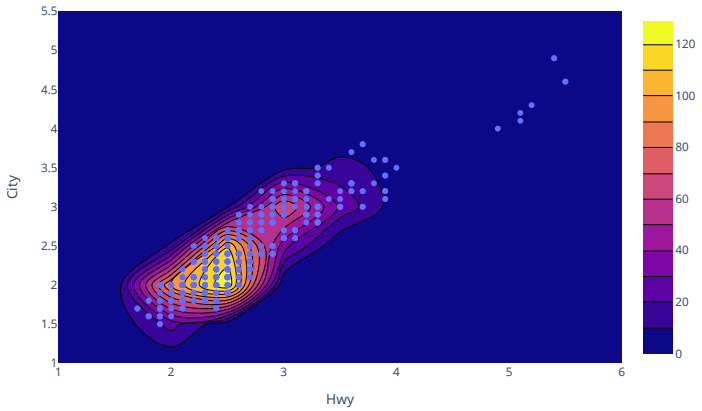
A scatter plot can be created with the `px.scatter()` function which can include trendlines computed using different methods. The most common trendline uses local weighted regression estimation for smoothing (LOWESS) and is specified with the `trendline='lowess'` argument. Note that the LOWESS trendline is NOT a global (linear) regression function but is computed locally. The example below produces a scatterplot of City and Highway fuel efficiency that includes a red LOWESS trendline.

```
fig = px.scatter(data,
    x='City', y='Hwy',
    trendline='lowess',
    trendline_color_override='red')
```



Two-dimensional histograms are analogous to one-dimensional histograms and show the count or frequency of data values against two other variables, as shown in the following example. Because Plotly Express does not provide a suitable function, the example begins with a scatter plot of the data points, and then adds a ”trace”, that is, a data series from the Plotly Graph Objects package.

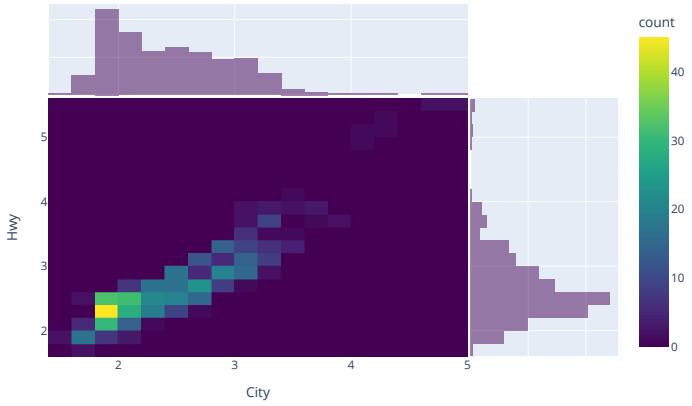
```
import plotly.graph_objects as go

fig = px.scatter(data, x='Hwy', y='City')

fig.add_trace(
  go.Histogram2dContour(x=data['Hwy'], y=data['City']))
```



The final example is somewhat more complex and introduces ”marginal” plots that

accompany a one- or two-dimensional plot. This example produces a two-dimensional density plot ("heatmap") and then uses the `marginal_x` and `marginal_y` options to add histograms for each marginal distribution along the x and y axis. Other useful options for marginal plots are "rug" and "box" which add a projections of data values or a box plot as marginals.

```
fig = px.density_heatmap(data,
  x = 'City', y = 'Hwy',
  nbinsx=20, nbinsy=20,
  marginal_x='histogram', marginal_y='histogram',
  color_continuous_scale = px.colors.sequential.Viridis)
```



Hands-On Exercises

Use the Pagila film rentals data from https://evermann.ca/busi4720/rentals.csv

1. Read the data into a Pandas data frame using `read_csv()`
2. Create a box plot of the rental payment amounts for films by rating
3. Create a violin plot of the rental payment amounts for films by rating
   - Compare the information conveyed by a box plot and a violin plot. What are the commonalities and what are the differences?

Hands-On Exercises

Use the Pagila film rentals data from https://evermann.ca/busi4720/rentals.csv

1. Read the data into a Pandas data frame using `read_csv()`
2. Use `drop_duplicates()` to drop duplicates of the film titles
    - Because each film may have been rented multiple times.
3. Produce a *histogram* of counts of films by rating

Tip:

- Use the `columns` and `shape` properties or the `describe()` method of a data frame to examine it.

Hands-On Exercises

Use the Pagila film rentals data from https://evermann.ca/busi4720/rentals.csv

1. Read the data into a Pandas data frame using `read_csv()`
2. Create a data frame with the mean rental payments for films by rating
3. Generate a bar chart of the mean rental payments for films by rating
4. Generate a pie or donut chart of rental counts by film rating

Tips:

- Use the `groupby()` function to group the data
- Use the `mean()` function to find the mean for a data frame column

10 Dashboards

Dashboards are sets of plots that show related information. They are typically customizable by the user of the dashboard or consumer of the information. Often, they are automatically updated, possibly in real time when required. Dashboards provide information of key metrics in a simple and easy to understand way to support decision making. Typical dashboards in a business might be one of key financial indicators for the CFA, key production measures for the COO, real-time machine data for the machine operator, etc.

Figure 18 shows an example of a dashboard for Canadian federal procurement contracts data. Such a dashboard may be useful for the procurement managers. The dashboard shows a selection box to select the year for which the data is to be shown, and a radio box that allows the user to decide whether to view the number of contracts or the dollar value of contracts. These two interactive elements determine the information shown on *all* plots of the dashboard. In other words, when the dashboard user changes their selection, all plots are updated.

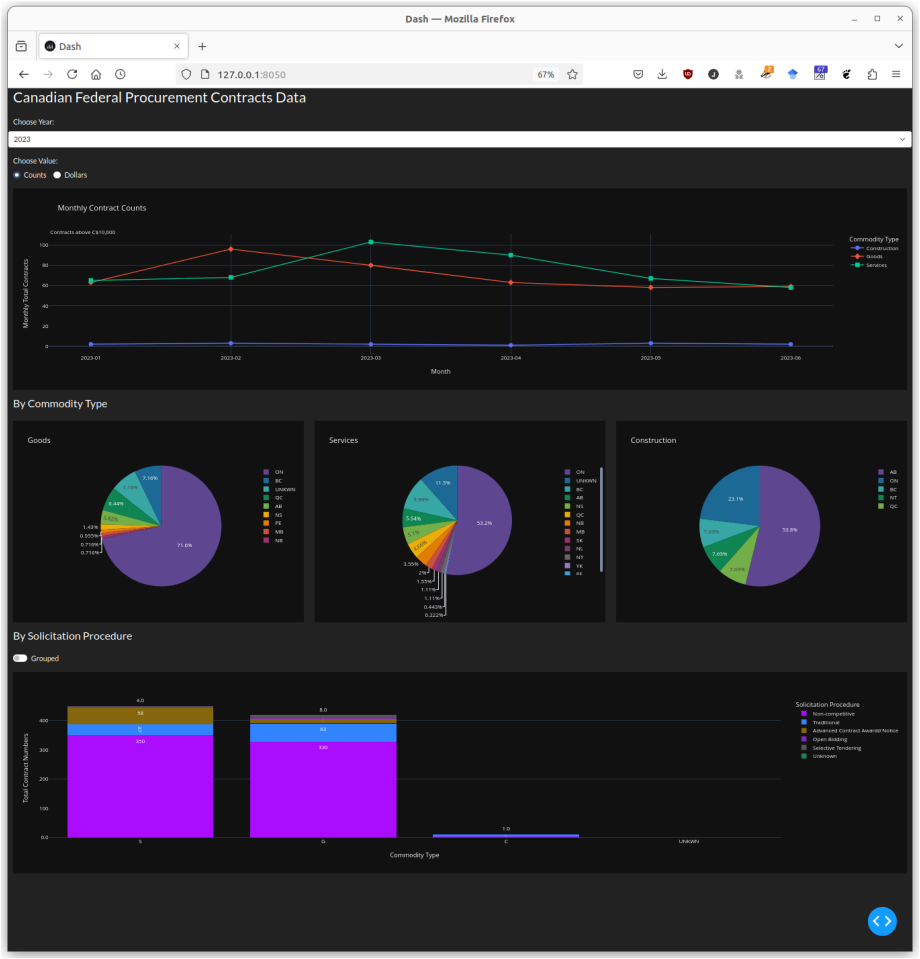


Figure 18: Example dashboard

The first plot is a line plot that spans the entire width of the dashboard. This is followed by another heading, followed by three pie charts next to each other. The next row is another header, followed by a binary choice box. A final column chart spans the last row of the dashboard.

This section briefly describes how interactive dashboards can be built using the Plotly Dash package for Python. Because Plotly Express figures are interactive, browser-based already, they lend themselves to being included in dashboards.

The first step is to import the required packages and components. The `dash` package provides core elements, dash bootstrap components provide the visual layout of the dashboard, and plotly express provides the figures or graphs that are part of the dashboard.

```
from dash import Dash, html, dcc, callback, Output, Input
import dash_bootstrap_components as dbc
import plotly.express as px
```

Next, the visual layout of the dashboard is specified, as shown in the following Python code block. The main layout is a container (`dbc.Container`) that contains a list of rows (`dbc.Row`). Each row contains a list of elements, such as HTML elements (headers (`html.H3` or paragraphs `html.P`), or interactive elements such as selection boxes (`dbc.Select`), radio buttons (`dbc.RadioItems`) and their labels (`dbc.Label`). Importantly, every interactive component has an `id` attribute that uniquely identifies it.

The following code block shows the beginning of the layout container definition, with the first row containing an HTML header element, and the second row containing selection boxes, radio items, and their labels. Compare the layout specification to the dashboard appearance in Figure 18.

```
app.layout = dbc.Container([
  dbc.Row([
    html.H3('Canadian Federal Procurement Contracts Data')]),
  dbc.Row([
    html.P(),
    dbc.Label('Choose Year: '),
    dbc.Select(
      id='year-selection',
      options=[{"label": x, "value": x} for x in years],
      value=years[-1]),
    html.P(),
    dbc.Label('Choose Value: '),
    dbc.RadioItems(
      options=[{"label": "Counts", "value": 0},
               {"label": "Dollars", "value": 1}],
      value=0, inline = True,
      id = 'value-selection'),
    html.P() ]),
```

The next code block below continues the layout container specification. It defines another row with a single column (`dbc.Col`) that spans the entire width of 12 units. Columns in turn contain lists of elements. The column in the example below contains a list with a single `dcc.Graph` element, which in turn has a `figure` and `id` attribute. This graph element is a placeholder for a Plotly Express figure, in this case the line plot that is shown in Figure 18.

Another row follows that contains a list of three HTML elements. The next row contains a list of three columns (`dbc.Col`) that each contain a list of a single `dcc.Graph` element as a placeholder for the three pie charts that are located next to each other in Figure 18. The layout code for the remaining dashboard elements in Figure 18 are not shown but are similar to these.

```
# Continued from previous snippet
   dbc.Row([
        dbc.Col([
            dcc.Graph(figure={}, id='line')
        ], width=12),
    ]),
    dbc.Row([
        html.P(),
        html.H4('By Commodity Type'),
        html.P()
    ]),
    dbc.Row([
        dbc.Col([
            dcc.Graph(figure={}, id='pie1')
        ], width=4),
        dbc.Col([
            dcc.Graph(figure={}, id='pie2')
        ], width=4),
        dbc.Col([
            dcc.Graph(figure={}, id='pie3')
        ], width=4),
    ]),
```

In order to generate the figures, each `dcc.Graph` element in the layout container has a corresponding update function. The code block below shows the `update_line` function for the line chart. The update function creates a figure using the Plotly Express commands illustrated in the previous section.

Importantly, each such update function has an associated "callback" specification. The callback has an `Output` element that specifies the component in the dashboard layout that receives the output of the function, that is, the figure. Additionally, the `callback` specifies a list of `Input` components that determine the inputs to the update function as values from interactive elements in the dashboard layout.

The example below specifies that the `update_line` function receives the values of the `year-selection` and the `value-selection` elements in the layout, see the `id` of those components in the above code block. With these values, the update function can produce an appropriate figure. The output of the `update_line` function (a figure) is shown in the element with id `line` in the dashboard layout.

```
@callback(
    Output(component_id='line', component_property='figure'),
    [
        Input(component_id='year-selection',
              component_property='value'),
        Input(component_id='value-selection',
              component_property='value')
    ]
)
def update_line(year_chosen, value_chosen):
    # Figure creation code, for example
    # fig = px.line(...)
    return fig
```

This callback mechanism links the interactive elements, such as radio buttons, selection boxes, value sliders, and others to the figures and plots that make up the dashboard. Whenever the dashboard user changes the value of any interactive element, the dashboard identifies which components receive those values as inputs and calls the corresponding update functions to produce new figures. Those figures are then placed into their graph object placeholders in the dashboard.

The final step is to run the dashboard app. This is accomplished with the following line of code:

```
app.run()
```

Resources

Complete implementation of the dashboard example is available at https://evermann.ca/busi4720/dashboard.py

11 Review Questions

The following review questions are intended to check your understanding of the material on visualization.

1. Explain the significance of data visualization in modern data analysis and communication.
2. How does data visualization blend artistic creativity with analytical skills?
3. List and explain the main reasons why data visualization is used.
4. What is visual discovery in the context of data visualization?
5. Contrast declarative visualization with visual discovery in terms of their purpose and interactivity.
6. Define operational visualization and its role in monitoring and decision making.

7. Explain the importance of focusing on quantitative messages in visualization. Provide examples of how different types of graphs or charts convey different types of data or relationships.
8. Discuss some of the challenges or pitfalls that can occur in data visualization, especially regarding pattern recognition and data interpretation.
9. Explain how the choice of a specific type of data visualization depends on the message or insight that needs to be conveyed.
10. What are "dark patterns" in the context of data visualization? Provide examples of common dark patterns used to deceive or mislead viewers.
11. How can cognitive biases be exploited in creating misleading data visualizations?
12. Explain how scaling and truncating axes in graphs can mislead the viewer. Provide examples.
13. How can the choice of an inappropriate graph type lead to misleading conclusions? Give specific examples.
14. Describe how the use of color in data visualization can be misleading. What are the best practices in choosing colors for visualizations?
15. Discuss the problems associated with using 3D elements or images in graphs. How can these elements distort the data representation?
16. Describe the unique challenges of visualizing streaming or real-time data. How do these challenges impact the design of such visualizations?
17. What are the specific challenges of visualizing network or graph data? How do these challenges influence the choice of visualization techniques?
18. Describe the different types of graph layouts (force-directed, circular, arc, layered) and their use cases. What are the benefits and drawbacks of each layout?
19. Why are interactive features like zooming, panning, and highlighting important in graph visualizations, especially for large datasets?
20. List and explain the criteria for assessing the quality of a graph visualization. Why are these criteria important?
21. Discuss the challenges associated with projecting three-dimensional Earth onto a two-dimensional surface in map visualizations. How do different projections affect the representation of spatial data?
22. Discuss the techniques used to represent attributes of nodes and edges in network visualizations. How can these techniques enhance or hinder the understanding of the network?
23. Explain how different areal units (e.g., counties, postal codes, districts) can impact the interpretation of geospatial data visualizations.
24. Explain why color choice is crucial in data visualizations and list the desirable characteristics of color palettes.
25. Describe sequential color palettes and discuss their appropriate use cases. Provide an example where a sequential palette is suitable.
26. What are diverging color palettes and when are they most effectively used in data visualization? Illustrate with an example.
27. Explain spectral color palettes and their application in visualizing data. Discuss the potential drawbacks of using spectral palettes.
28. Discuss the importance of considering color vision deficiency (CVD) in choosing color palettes for data visualizations.

29. How do the different types of color vision deficiencies (e.g., protanopia, deuteranopia, tritanopia) affect the perception of colors in data visualizations?
30. Define and discuss the importance of perceptual uniformity in color palettes. How does it impact the interpretation of data?
31. What are monochromatic color palettes and in what situations might they be preferred?
32. What is a box plot and what are the key summary statistics it displays?
33. Explain the concept of the interquartile range (IQR) in a box plot. How is it calculated and what does it represent?
34. Describe the significance of the median line in a box plot. How can the median line's placement provide insights into data skewness?
35. What do the whiskers in a box plot represent? Explain the common method for determining their length.
36. How are outliers represented in a box plot? What criteria is typically used to classify a data point as an outlier in this context?
37. How can you determine if a dataset is symmetric or skewed based on its box plot?
38. Compare and contrast box plots and histograms. In what scenarios might one be preferred over the other?
39. Compare and contrast box plots and violin plots. In what scenarios might one be preferred over the other?