Towards Fluent Interactive Data Visualization

Adam Bartonicek

2023-10-12

Contents

1 Introduction

1.1 What do we interact with?

There is a subtle yet profound question in the production and use of interactive data visualizations: *when we interact with a plot, what exactly are we interacting with?* On its face, the question might seem trivial. A person clicking a bar in an interactive barplot may be convinced that they are interacting with the coloured rectangle on the screen, since, by design, that is the "thing" they see change in front of them on the screen. And in some way, this is true - by interacting with the bar, we can affect its graphical attributes: we can change its colour, we can squeeze it/stretch it, and so on. Yet, in another, deeper way, this perception of interacting with a plain geometric object is just an illusion. How so?

The illusion lies in the fact that the geometric object - the coloured rectangle representing the bar - is not meaningful on its own. Instead, the bar is only ever meaningful as a bar within the context of the plot. We can see this easily. If we were to take the coloured rectangle and transpose it onto a blank area of the screen, we would lose some crucial information that the context of the plot provides - it would no longer be a "bar" in the same sense that it was before.

So objects in plots have some additional structure, beyond their simple geometry. That statement should not come as surprising or controversial to people familiar with data visualization. However, it may be more challenging to define in detail what exactly this "structure" is. There are a few ideas we may be able to muster. We know that the geometric objects in a plot are supposed to represent some underlying data. That much is clear - if objects in a graphic do not represent any external information but are instead drawn according to some arbitrary rules, we cannot really, in good conscience, call it a "plot". But data is only a part of the story. When drawing plots, we rarely represent the raw data directly. Instead, we often summarize, aggregate, or transform the data, by applying mathematical functions such as count, sum, mean, log, or the quantile function.

The output of these transformations is what we then represent by the geometric objects. So, when interacting with a bar in an interactive barplot, we are not just interacting with a plain geometric object, we are interacting with a mathematical function, or, more likely, several of them. This is important since functions have properties. The core argument of the present text is that the properties of these functions impose limits on what kinds of visualizations and interactions can be meaningfully composed. Before diving deeper, however, let's first define some key terms and draw a rough sketch of the data visualization process.

1.2 Rough sketch of the data visulization process

To create a data visualization, be it static or interactive, we need several key ingredients: data, summaries, scales/coordinate systems, and geometric objects. First of all, as was mentioned above, every data visualization needs to be built on top of some underlying data. We can represent this as a set of some arbitrary units of information D. Data in the wild usually comes with more structure than that - for example, we often encounter data stored in a tabular (or "tidy" [CITE]) format, stratified by rows and columns. In that case, we could substitute D by the set of rows R, the set of columns C, or the set of cell values $R \times C$ (where $\times$ indicates the cartesian product). However, for the purpose of this description, we do not have to assume any special structure and just speak of the data units $d \in D$.

Secondly, the set of data units D is transformed into a set of collections of summaries S via some function α. It may be the case that α is one-to-one (bijection), in which case there is one unit of data for every summary. Examples of this include the typical scatterplot, in which α is the identity function. However, more often, α is many-to-one (surjection), which means that each summary may be composed of multiple units of data. This is the case for the typical barplot, histogram, density plot, or violin plot. When α is many-to-one, it will typically reduce the cardinality of the set at hand such that $|S| \leq |D|$ (i.e. in a typical barplot, there will be fewer bars than there are rows of the data, unless each row has a unique level of the categorical variable). This is done by stratifying on one or more variables which may either come from the data directly (as in the case of a barplot or a treemap) or may themselves be a summary of the data (as in the case of

histogram bins). Importantly also, each collection of summaries $s \in S$ may (and usually will) hold multiple values, produced by a different constituent function each - for example, the collection s for a single boxplot "box" will consist of a median, first and third quartile, the minimum and maximum of some variable, and the outlier values, for a given level of some stratifying variable (which itself will also be an element of s). Finally, the output of these constituent functions may also depend on some external parameters, such as anchor and binwidth in a histogram, which may be either directly supplied by the visualization creator or inferred from the data via some heuristic.

Thirdly, each collection of summaries $s \in S$ needs to be translated from the data- (or summary-) coordinates to graphical coordinates/attributes $g \in G$, via a function β. This means that each summary value gets mapped or "scaled" to a graphical attribute via a constituent scaling function. Note that this mapping preserves cardinality - there are as many collections of graphical attributes as there are collections of summaries, $|G| = |S|$. For numeric/continuous summaries, scales typically come in the form of linear transformations, such that the minimum and maximum of the data are mapped near the minimum and maximum of the plotting region, respectively. Continuous scales may also provide non-linear trasnformations such as log-transformation or binning, and the values may even get mapped to discrete graphical attributes. Likewise, discrete summaries may get translated to either continuous or discrete graphical attributes. There are also coordinate systems, which may further translate values from multiple scales simultaneously, e.g. by taking values in cartesian (usual) coordinates and translating them to polar coordinates. Either way, β can be viewed as one function, representing the composition of scales and coordinate systems.

Finally, the collections of graphical attributes/coordinates $g \in G$ are drawn as geometric objects inside the plotting region, which we can represent as the set of pixels P. While the act of drawing does take the collections of graphical coordinates $g \in G$ as inputs, it does not simply return an output for each input (like a mathematical function would), but instead mutates the state of the graphical device via a side effect γ^*, i.e. changing the colour values of pixels. In other words, how the graphic ends up looking may depend on the order in which we draw things, for example, some geometric objects may end up being plotted over others, and so γ^* is not a simple mapping from G to P. As such, we cannot call γ^* a true mathematical function. The geometric objects may be simple, such as points, lines, or bars, or compound, such as a boxplot or pointrange. Importantly, each attribute necessary to draw the geometric object, such as x- and y-position, width, height, area, etc... needs to be present in the corresponding g.

The whole process can be summarized as follows:

$$D \overset{\alpha}{\to} S \overset{\beta}{\to} G \overset{\gamma^*}{\Rightarrow} P$$

Or, equivalently:

$$\text{(data)} \overset{\text{summarize}}{\longrightarrow} \text{(summaries)} \overset{\text{translate/encode}}{\longrightarrow} \text{(graph. coordinates)} \overset{\text{draw}^*}{\Rightarrow} \text{(graph. device state)}$$

The above should be fairly non-controversial description of how a data visualization is produced, and applies equally well to static as well as interactive visualizations.

1.3 Why summaries matter

In some treatments of data visualization, the summarizing step ($\alpha : D \to S$) is de-emphasized, in one of two ways. In the first case, the data is already presumed to arrive pre-summarized (i.e. $S = D$, for example we can draw a barplot from a pre-summarized table of counts), and data visualization is just about encoding the data into graphical coordinates. This framing feels especially natural when considering plots which show a 1-to-1 mapping (bijection) between the data and the geometric objects, such as the scatterplot or the lineplot/time-series chart. Indeed, in these plots, the summarizing function is identity, and as such *can* be ignored. In the second case, the computation of summaries is considered, however, it is absorbed into the translation step, such that $\alpha = \beta$ (e.g., a histogram may be thought of as directly translating its underlying

variable into bin coordinates). Both of these approaches produce a tight coupling between the summaries and graphical coordinates.

We can give a much richer account of visualizations by treating both the summarizing and the translating step as first class citizens. Crucially, some plots may encode the same summaries through different graphical attributes and geometric objects; other plots may compute different summaries but display them via the same means. As an example of the former, please inspect the histogram and spineplot in Figure 1. Histograms and spineplots use the same summaries, i.e. binned counts of some underlying continuous variable x. However, histograms encodes the bin breaks into rectangle boundaries along the x-axis, and the counts as the top rectangle boundary along the y-axis, whereas spineplot encodes the counts into both x- and y-axis rectangle boundaries, with both the x- and the y- dimensions stacked and the y-dimension additionally scaled to 1, and the bin breaks are encoded as x-axis labels. As an example of the latter, scatterplot and bubbleplot both use points/circles to display their underlying summaries, however, whereas the summary for scatterplot is identity (or at least bijective), the summary for bubbleplot involves binning along x- and y- axis.

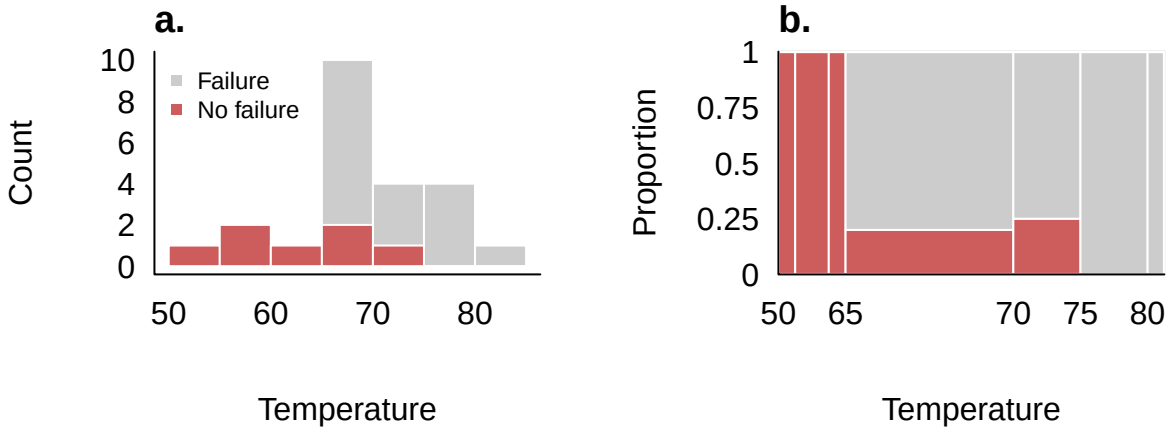


Figure 1: Histogram and spineplot use the same summaries but encode them in different ways. a) In histogram, the x-axis display bin breaks and y-axis displays count, stacked across groups (failure/no failure). b) In spineplot, the x-axis and y-axis both display count; the x-axis shows count stacked across bins, whereas the y-axis shows count stacked across groups and scaled by the total bin count (such that the total bin height = 1). The bin breaks are displayed as x-axis labels (however, the underlying summary is still stacked count).

1.4 A couple of caveats

There are also a few important caveats to the sketch of the data visualization process outlined above. Firstly, while not always the case, *order* can be important. Within each $s \in S$ and $g \in G$, some summaries may be ordered lists of values, as in the case of the lineplot (i.e. lines need to be drawn by connecting a series of points in the correct order). Secondly, the collections in S (and in turn, in G) may themselves be ordered and form a hierarchical or tree-like structure. This is particularly relevant in the case of stacking. For example, when drawing a stacked barplot, we need to stack the graphical coordinate values representing the sub-bars on top of each other, such that each sub-bar is stacked within the appropriate parent-bar and in the correct order (e.g. such that sub-bars belonging to group 2 always go on top group 1 sub-bars). If it is the case that order matters for either α or β, then we cannot call them true functions either, since each of the elements in their domain needs to be "aware" of the other elements, or of what has happened before.

Finally, the data limits (minimum and maximum) and values that are used for scales are often derived from

S rather than from the raw data (e.g. the upper y-axis limit in barplot or histogram is the highest count across bins). Further, the limits may come from a higher level of hierarchy than the summaries that are actually being drawn - for example, in a stacked barplot, for the upper y-axis limit we need to know the count (height) of the tallest *whole* bar but do not need to know the counts within the stacked sub-bars (since these are, by definition, smaller or equal to the whole bar).

This is where the fundamental differences between static and interactive visualizations become very relevant. In static visualizations, all computation is done only once, before the plot is rendered, and so the issues of order, hierarchical structure of the summaries, and the tracking of axis limits are less important. Interactive visualizations, on the other hand, need to reactively respond to the user's input, and some computations may need to run many times within a single second. As a result, it is imperative to organize the process in such a way that we do only as little work as is necessary.

2 Interactivity and Hierarchy

As was briefly mentioned in the previous section, interactive graphics come with their own set of considerations around hierarchy and efficiency. Specifically, computations which result from interaction may need to occur repeatedly within short time frames.

3 The Problem of Statistical Summaries

There is an even deeper issue when it comes to interactivity and statistical summaries of the data. Specifically, not every statistical summary "works" equally well - instead, some summaries may be better than other. Let's first illustrate the problem with an example.

Linked brushing or highlighting is one of the most popular types of interactive features used in interactive data visualizations [CITE]. It allows the user to select objects (such as points or bars) within one plot by e.g. clicking or clicking-and-dragging, and the corresponding cases (rows of the data) are then highlighted in all other plots. Its usefulness comes from the fact that allows the user to rapidly "drill-down" [CITE] and explore the summaries that would result from subsetting different rows of the data, within the context of the entire dataset.

Now, let's imagine we have three interactive plots: a classical scatterplot, a barplot of summarizing the count of cases within the levels of some categorical variable x, and a barplot summarizing the mean of some other variable y, within levels of the same categorical variable x. The plots are linked such that they allow for linked brushing/highlighting. Intuitively, it might seem that the barplot of counts and the barplot of means are equally valid/useful representations of the underlying data. However, if we consider these plots in the context of linked brushing, few subtle-yet-fundamental differences become apparent.

3.1 Empty selections

Firstly, as is shown in Figure 2, how do we draw an empty selection? In the case of counts, we have a meaningful default value - zero - as in "the number of cases in an empty set is 0". When we seen an absence of a bar in a barplot, we know that the count for the corresponding level of the stratifying variables.

However, there is no similar default value for means: the mean of an empty set is not defined. We could just not draw the bar representing the empty selection, however, that decouples the statistical summary from the visual representation: the absence of a bar may now indicate that *either* no cases are selected *or* that some cases are selected and their mean is equal to the lower y-axis limit.

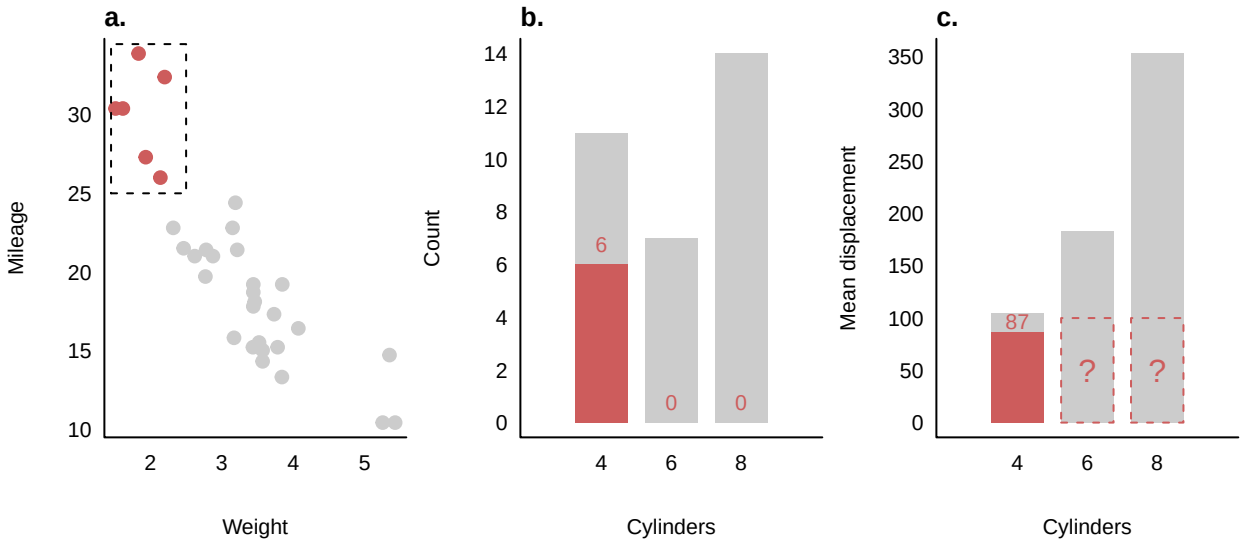


Figure 2: The problem of representing empty selection. a) An illustration of selection by linked brushing. b) In the barplot of counts, the count within an empty selection (red) is zero and so an absence of a bar accurately represents a count of zero. c) In the barplot of means, the mean of an empty selection is not defined. Absence of a bar could indicate that either no cases are selected or some cases are selected and their mean is equal to the lower y-axis limit.

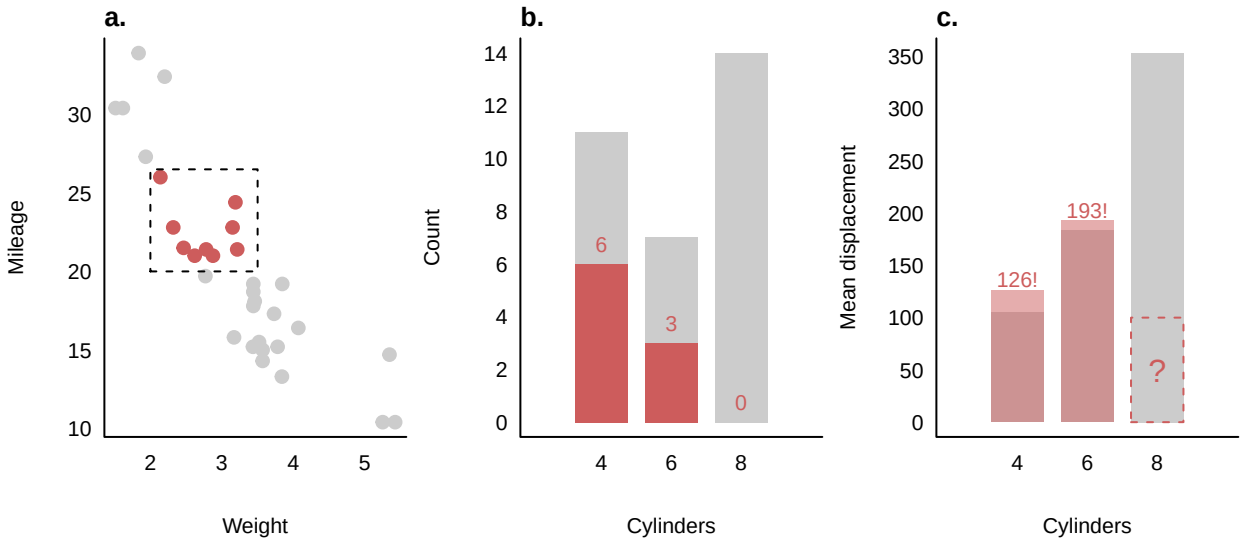


Figure 3: The relationship between selection vs. the whole. a) An illustration of selection by linked brushing. b) In the barplot of counts (middle), the count within a selection (red) is always less than or equal to the count within the whole and the outline of the bars does not change. c) In the barplot of means, the mean of a selection can be greater and so the outline of the bars will change in response to user input.

3.2 Part vs. the whole

Second, as shown in Figure 3, how does the summary on the selection relate to the summary on the whole? In the case of the barplot of counts, the height of the sub-bar is always less than or equal ($\leq$) to the height of the whole bar (because so is the count). Thus, we can always draw the sub-bar over the whole bar, and the whole bar act as a stable visual reference - the outline of the whole bar will remain the same no matter which cases of the data are selected. In fact, we can either draw the whole bar (as shown in grey in Figure 3) and draw the selected sub-bar (red) over it, both starting from the y-axis origin (0), or we can draw a red sub-bar and the "leftover" grey-sub bar stacked on top of it, starting from the top y-coordinate of the selection bar and with **height** = (count of the whole − count of the selection). The resulting plots will be identical, visually.

The barplot of means, does not share these nice properties. Specifically, the "sub-bars" can be taller than the whole bar (because the mean of a subset can be greater than the mean of the original set). As a result, if we draw the selection sub-bars on top of the whole bar, the whole bar may become completely obscured by it (as shown in Figure 3). We could choose to draw the selection sub-bars as semitransparent, or draw the the bars side-by-side instead of on top of each other (dodging), however, the question then remains how to display the non-selected (grey) cases - do we draw the "whole" bar that remains the same height throughout interaction, or the "leftover" bar whose height changes with the selection? Also, if we choose to draw the bars side-by-side, will the whole bar be initially wide and shrink in response to selection to accommodate the selection bars, or will it be narrow from the initial render? Finally, if the user brushes the side-by-side bars, will they be able to select individual bars or will brushing one select all of them?

3.3 Combining parts

Finally, as displayed in Figure 4, when multiple selections/groups are present, how do we combine them together? Again, in the case of the barplot of counts, there is an idiosyncratic way to do this: we can stack the counts across the selection groups and the corresponding bars on top of each other, and the height of the resulting bar will be identical to the count of the whole. And, as in the case of a single selection group, we can either draw the bars over each other, starting from the y-axis origin, or draw sub-bars stacked on top of each other, each starting where the last left off, and the resulting plots will be identical.

In the case of barplot of means, there is no meaningful way to combine the selections. If we were to stack the bars on top of each other, the resulting statistic (i.e. the sum of the group means) would not be meaningful. Worse yet, if we were to take the mean of the group means, the resulting statistic will almost surely be different from the mean of the whole: the mean of the group means $\neq$ the grand mean. And again, the grand mean may be less than any one of the group means. Since we cannot meaningfully combine the statistics, we could draw the bars side-by-side, but again, we run into considerations about how to render the base group, the bar width, and the selection.

3.4 Some statistics are better than others

To summarize, counts, as implemented in a typical barplot, provide:

a. An unambiguous way to display empty selections (absence of a bar is a count of 0)
b. A stable visual reference (the count within a sub-bar $\leq$ the count within the whole bar)
c. A way to combine the statistics together (the sum of the sub-bar counts = the count of whole bar).

Means do not share these nice properties: mean of an empty selection is not defined, the mean of a selection is not always subordinate to the mean of the whole, and the mean of group means is different from the grand mean. Importantly, these properties or lack thereof are not tied to any specific graphic (e.g. a barplot) but are instead tied to the underlying statistic (count/mean).

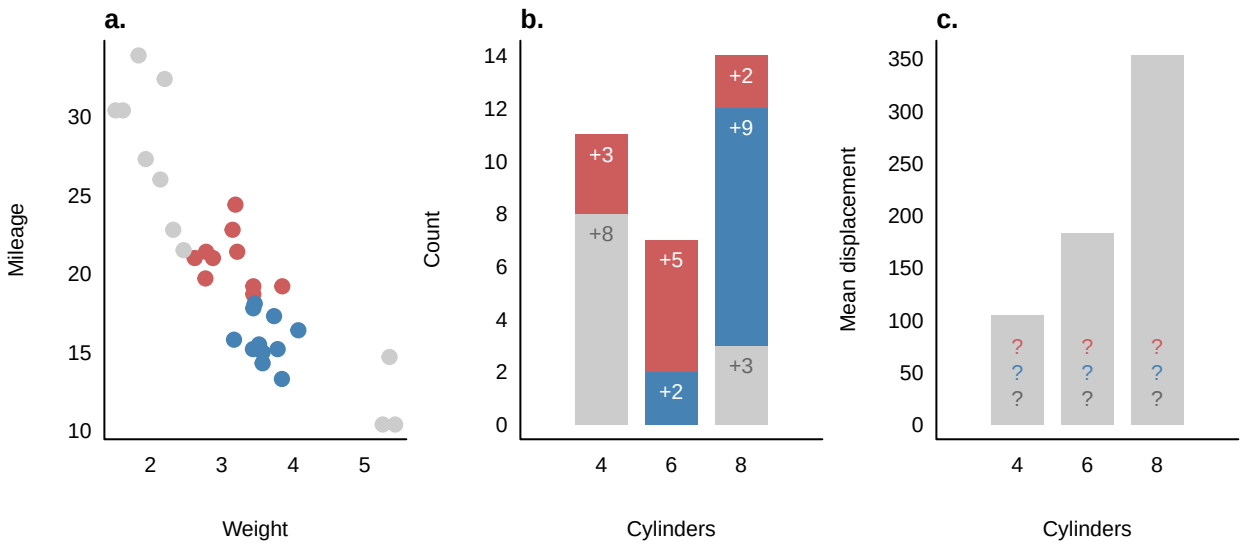


Figure 4: Combining selections. a) An illustration of selection by linked brushing, with two groups. b) In the barplot of counts, statistics can be stacked on top of each other such that the count of the stacked bar is identical to the count of the whole bar (i.e. one that would result from no selections). c) In the barplot of means, no such procedure for combining statistics exists.

There are, of course, ways to display means with linked brushing/selection, and some have been described above. The key point, however, is that to display means, more decisions need to be made in order to produce a coherent interactive visualization. Consequently, since no choice is more "natural" than any other, a person who interacts with a linked barplot of means for the first time may be surprised in how the plot behaves. For the barplot of counts, on the other hand, there *are* such natural solutions readily available, and this may explain why barplots of counts are a popular type of plot in systems which implement linked brushing.

One might also wonder if these problems are just a quirk of linked brushing. However, first of all, the problems are not unique to interactive visualizations but arise in static plots as well, with stacking. Second, while they are perhaps the most prominent with linked brushing, we could run into them with any other interactive feature that deals with summary statistics on several parts of the data. If, for example, on top of linked brushing, we wanted to implement a pop-up window that displays summary statistics within a given object as text, we would still have to contend with the problem of what to do in the case of the mean of an empty selection. Here, there is perhaps a simple solution in displaying an `NA` or an empty string for the mean of empty selection. However, we should point out that by doing this, we have to go outside of the type of non-empty selections (real number/float). This may not be a problem in whatever statistical software we are using to render the plots. However, there is still something more natural about counts having the default value (0) be an element of the same type as all other values (integer).

Returning to the nice properties of counts, are counts uniquely "good" in this regard? The short answer is "no". For example, sums or products (of values ≥ 1) conform to these same nice properties of counts. In other words, in the context of linked brushing, a stacked barplot will behave equally well if it display sums (not surprising) or products (perhaps somewhat surprising?). In fact, there is whole a class of mathematical functions, or more precisely mathematical objects, that share these nice properties. To discuss these, let's first lay the groundwork with some relevant theory.

4 Few Relevant Bits of Category Theory

4.1 Functions

A function is a mapping between two sets. More specifically, given the set of sources S (also called the *domain*) and the set of possible targets T (also called the *codomain*), we can think of a function as a subset $F \subseteq S \times T$ of valid source-target pairs (s,t), such that for every $s \in S$ in there exists a unique $t \in T$ with $(s,t) \in F$. The function then can be thought of as "selecting" a t for any valid s it is given.

If a function f covers all of its codomain, i.e. for all $t \in T$ there exists a $s \in S$ such that $f(s) = t$, then it is a *surjective* or *onto* function. If no two sources lead to the same target, i.e. for $s_1, s_2 \in S$, if $f(s_1) = t$ and $f(s_2) = t$, then $s_1 = s_2$, then it is an *injective* or *one-to-one* function. Also, for any given subset of targets, we can ask about the subset of sources or *pre-image* that could have produced them, i.e. for $T_i \subseteq T$ we can define $f^{-1}(T_i) = \{s \in S | f(s) \in T_i\}$. Finally, functions can be composed together: if we have two functions $f : X \to Y$ and $g : Y \to Z$, we can combine them into new function $h = g \circ f$ such that $h : X \to Z$, i.e. $h(x) = g(f(x))$.

4.2 Partitions

One useful thing we can do with functions is to form partitions. Specifically, given some arbitrary set A, we can assign every element a label from a set of part labels P via a surjective function $f : A \to P$. Conversely, we can then take any part label $p \in P$ and recover the corresponding subset of A by pulling out its pre-image: $f^{-1}(p) = A_p \subseteq A$. We can use this to define partitions in another way, without reference to f: a partition of A consists of a set of part labels P, such that, for all $p \in P$, there is a non-empty subset A_p and:

$$A = \bigcup_{p \in P} A_p \qquad \text{and} \qquad \text{if } p \neq q, \text{ then } A_p \cap A_q = \varnothing$$

I.e. the parts A_p jointly cover the entirety of A and parts cannot share any elements.

We can rank partitions by their coarseness. That is, for any set A, the coarsest partition is one with only one part label $P = \{1\}$, such that each element of A gets assigned 1 as label. Conversely, the finest partition is one where each element gets assigned its own unique part label, such that $|A| = |P|$. Finally, given two partitions, we can form a finer (or at least as fine) partition by taking their intersection, i.e. by taking the set of all unique pairs of labels that co-occur for any $a \in A$ as the new part labels. For example, if $A = \{1,2,3\}$ and partition 1 assigns part labels $P_1 = \{x, y\}$ to A such that $f_1(1) = x,\ f(2) = x,\ f(3) = y$, and partition 2 assigns part labels $P_2 = \{\rho, \sigma\}$ such that $f_2(1) = \rho,\ f_2(2) = \sigma,\ f_2(3) = \sigma$, then the intersection partition will have part labels $P_3 = \{(x,\rho), (x,\sigma), (y,\sigma)\}$ such that $f_3(1) = (x,\rho),\ f_3(2) = (x,\sigma),\ f_3(3) = (y,\sigma)$.

4.3 Preorders

A preorder is a set X equipped with a binary relation $\leq$ that conforms to two simple properties:

1. $x \leq x$ for all $x \in X$ (reflexivity)
2. if $x \leq y$ and $y \leq z$, then $x \leq z$, for all $x, y, z \in X$ (transitivity)

Simply speaking, this means that between any two elements in X, there either is a relation and the elements relate (one element is somehow "less than or equal" to the other), or the two elements do not relate.

An example of a preorder is the family tree, with the underlying set being the set of family members: $X = \{\textbf{daughter}, \textbf{son}, \textbf{mother}, \textbf{father}, \textbf{grandmother}, ...\}$ and the binary relation being ancestry or familial relation. Thus, for example, **daughter** $\leq$ **father**, since the daughter is related to the father, and **father** $\leq$ **father**, since a person is related to themselves. However, there is no relation ($\leq$) between **father** and

mother since they are not related. Finally, since **daughter** $\leq$ **father** and **father** $\leq$ **grandmother**, then, by reflexivity, **daughter** $\leq$ **grandmother**.

We can further restrict preorders by imposing additional properties, such as:

3. If $x \leq y$ and $y \leq x$, then $x = y$ (anti-symmetry)
4. Either $x \leq y$ or $y \leq x$ (comparability)

If a preorder conforms to 3., we speak of a partially ordered set or *poset*. If it conforms to both 3. and 4., then it is a *total order*.

4.4 Monoids

A monoid is a tuple $(M, e, \otimes)$ consisting of:

a. A set of objects M
b. A neutral element e called the *monoidal unit*
c. A binary function $\otimes : M \times M \to M$ called the *monoidal product*

Such that:

1. $m \otimes e = e \otimes m = m$ for all $m \in M$ (unitality)
2. $m_1 \otimes (m_2 \otimes m_3) = (m_1 \otimes m_2) \otimes m_3 = m_1 \otimes m_2 \otimes m_3$ (associativity)

In simple terms, monoids encapsulate the idea that *the whole is exactly the "sum" of its parts* (where "sum" can be replaced by the monoidal product). Specifically, we have some elements and a way to combine them, and when we combine the same elements, no matter where we put the brackets we always get the same result (i.e. something like "the order does not matter", although that is not precisely right, more on that later). Finally, we have some neutral element that when combined with an element yields back the same element.

For example, take summation on natural numbers, $(\mathbb{N}, 0, +)$:

$$1 + 0 = 0 + 1 = 1 \qquad \text{(unitality)}$$
$$1 + (2 + 3) = (1 + 2) + 3 = 1 + 2 + 3 \qquad \text{(associativity)}$$

Likewise, products of real numbers $(\mathbb{R}, 1, \times)$ are also a monoid, $(\mathbb{R}, 1, *)$, and so is multiplication of $n \times n$ square matrices $(\mathbf{M}_{n \in \mathbb{Z}}, \mathbf{I}, \cdot)$, where $\mathbf{I}$ is the identity matrix and $\cdot$ stands for an infix operator that is usually omitted. As a counterexample, exponentiation does not meet the definition of a monoid, since it is not associative: $x^{(y^z)} \neq (x^y)^z$.

We can impose further restrictions on monoids, e.g.:

3. $m_1 \otimes m_2 = m_2 \otimes m_1$ for all $m \in M$ (commutativity)

Both commutativity and associativity can both be viewed as a kind of "order does not matter" rule, however, they are fundamentally different. Let's imagine our set of objects consists of three wires of different colours $\{\mathbf{red}, \mathbf{green}, \mathbf{blue}\}$ and the monoidal product consists of connecting wires. Let's also imagine that the **red** wire is connected to a power source and the **blue** wire is connected to a lightbulb, and the blue wire amplifies the current from the power source such that it is enough to power the light bulb. To turn on the lightbulb, we need to connect **red** $\to$ **green** and **green** $\to$ **blue**. The time order in which we connect the three wires does not matter: we can connect **green** $\to$ **blue** first and **red** $\to$ **green** second or vice versa, either way we get the same result (lightbulb turns on). However, the spatial order in which we connect the wires *does* matter: if we connect **red** $\to$ **blue**, then the current will not be enough to power the lightbulb. Hence, the operation is associative (temporal order does not matter) but not commutative (spatial order does matter).

5 The Model

Armed now with some rudimentary theory, we can attempt to describe the data visualization process in more detail.

5.1 Data and Partitions

We again start with the data D which is just some set of values that may or may not be structured. Ultimately, our goal is to try and learn something new from the data. To this end, we want to create a visualization that will show the information from the data through graphical elements, primarily geometric objects such as points, lines, or areas (we may also use, for example, text/symbols).

First of all, we need to think about how the information from the data will be distributed among the objects. To start with the simplest case, sometimes we might want all information in the data to go into a single geometric object. This may be a perfectly good way to display information about the entire data set. Examples of plots with a single geometric object may include time series plot (line), density plot (density polygon), or radar chart (pentagon/other polygon).

However, often, we want to compare and contrast different parts of the data. And this is where we have to start to think more deeply about how information should be allocated. For example, it would not be very useful to draw multiple identical geometric objects representing the same information from the data - if all of the objects looked the same, we could learn the same information from any one of them and so it is redundant to have more than one. Instead, we want each object to represent different aspects of the data and so we need to think about slicing the data up along some axis or axes.

Often, the axes along which we can slice the data can already be present in its structure. For example, a lot of data comes organized in a tabular $R \times C$ format where R is the set of rows and C is the set of columns (and $R \times C$ are the cells), and so we could use any of these three sets as the basis for drawing our geometric objects. This is shown in Figure 5. Notice that, when slicing along rows R, the number of objects (points) is equal to their cardinality: **# objects** $= |R|$. Likewise, when slicing along columns, **# objects** $= |C|$, and when slicing along cells, **# objects** $= |R \times C|$.

However, there is no reason to limit ourselves to slicing along axes that are already present in the structure of the data - we can also create new axes. For example, when computing the statistics underlying the classical barplot, we can slice along the levels of some discrete variable x. Likewise, when drawing a histogram or a heatmap, we can bin some continuous variable(s) and slice along the bins. If we are making an interactive visualization, we can make the slicing axis responsive to user input, for example, we might let the user change the histogram bin width and anchor interactively.

Now, when slicing along any given axis or axes, a question remains *how* to slice. There are two key points to mention. Firstly, no matter how we perform the slicing, every unit of data $d \in D$ should probably end up in *some* slice, i.e. our slicing operation should not discard or "throw away" any data. This seems obvious - we want the visualization to be an accurate and complete representation of the data, and as such we cannot simply leave any unit of data out arbitrarily. Secondly, we also probably want each unit of the data to end up in *one slice only*. This point is a bit more subtle. Suppose, for example, that we were tasked with plotting data for a company, and we were given as data a set of **employees** and a set of **managers**, with every manager in **managers** also being included in the set of **employees** (since they are also an employee of a company). Then, we could use the two sets as the two "slices" of that data and display the size/count of each set via the height of a bar in a barplot (such that the **employees** bar would be at least as tall as the **managers** bar, if all employees were managers). However, while it is certainly possible to create this type of plot, it unnecessarily duplicates information, and any questions we might want to answer with it might be better served by other plots. For example, if we wanted to compare the number of **managers** vs. **non-manager employees**, we would be off drawing each as a separate bar. If we wanted to see what proportion of the total workforce are **managers**, we could draw one stacked bar with **non-manager employees** stacked on top of **managers**.

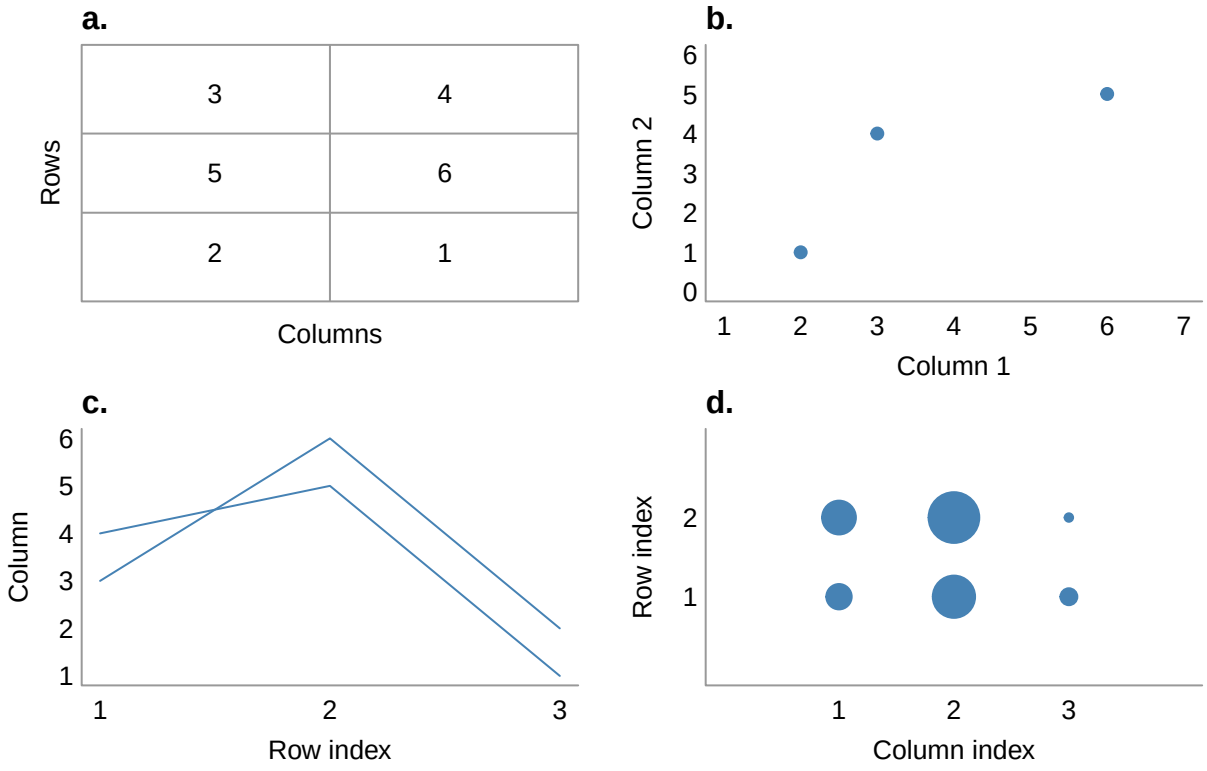


Figure 5: Three different ways of partitioning a tabular data set. a) The underlying data set, represented as a matrix with 3 rows and 2 columns. b) In scatterplot, we can slice along the rows and draw points at the x- and y-position given by the columns. c) In lineplot, we can slice along the columns & draw lines by connecting points given by the corresponding row values. d) In a dotplot, we can slice along both rows and columns and draw points at given row and column index, with point size given by the corresponding cell value.

So, when slicing data, we want to make sure every unit of data ends up in one slice and no two units end up in the same slice. One might notice that this description of slicing corresponds precisely to the mathematical definition of a partition (Section 4.2). That is, the data is split into parts and each unit of data is assigned to one part and one part only. This way of thinking about slicing the data is useful since we can make use of the dual nature of the definition of partition: we can either think of the partition as a function that assigns each data unit a part label $p \in P$, or as the set of part labels $p \in P$ where each has a corresponding subset of the data D_p, such that the union of the subsets recovers the original data set, $\bigcup_{p \in P} D_p = D$, and intersection of any two subsets is empty: $D_p \cap D_q = \varnothing$, for$p \neq q$.

For illustration, when implementing a partition in a programming language, we could implement a partition either as an array of part labels where the array index corresponds to row index (e.g. $[A, A, B, A, B, C, \ldots]$) or as a dictionary of row indices (represented as either arrays or sets), indexed by part labels (e.g. $\{A : [0, 1, 3, \ldots], B : [2, 4, \ldots], C : [5, \ldots]\}$). Either implementation may be more or less useful in different scenarios. For example, when computing some statistics for each part $p \in \{A, B, C\}$, the array representation may be preferable since we have to loop through all rows of the data anyway (looping through an array is $O(n)$ operation, looping through all k sub-arrays in the dictionary is also $O(n)$ but with some additional overhead for setting up the k loops). If instead we wanted to select all indices belonging to part A, the dictionary representation will provide a more efficient $O(1)$ access, instead of having iterate over the entire array to pick out A labels ($O(n)$). Thus, the data structure which we will use to implement the partitioning should reflect this duality, perhaps by internally implementing both the array and the dictionary representation.

Finally, as with the mathematical definition, we can also form finer partitions from coarser ones by taking taking their intersection. Importantly, we can also think of taking the intersection of two partitions as hierarchically nesting one partition within the other. By starting with the coarsest partition, i.e. one that assigns each row of the data to the same part, and then progressively nesting factors within one another, we can build up a hierarchy of partitions. For example, if our data includes the variables $\mathbf{gender} = \{\mathbf{male}, \mathbf{male}, \mathbf{female}, \mathbf{male}, \mathbf{female} \ldots\}$ and $\mathbf{group} = \{\mathbf{A}, \mathbf{B}, \mathbf{A}, \mathbf{B}, \mathbf{B}, \ldots\}$, we might build up a hierarchy of partitions with labels P_1, P_2, and P_3 where P_1 is the set of part labels for the whole dataset $= \{(\,)\}$, $P_2 = labels_{\mathbf{gender}} = \{(\mathbf{male}), (\mathbf{female})\}$, and $P_3 = labels_{\mathbf{gender} \times \mathbf{group}} = \{(\mathbf{male}, \mathbf{A}), (\mathbf{male}, \mathbf{B}), \ldots\}$.

5.2 Computing Statistics

Now that the data has been split into parts, within possibly hierarchical/nested partitions, we need to think about computing some summary statistics on each part that we will then use to define the graphical attributes (length, area, angle,...) of the corresponding geometric objects. To do this, we will need to iterate across the data units $d \in D$ and reduce them into k parts, where k represents the cardinality of the partition at hand, $k = |P|$. Crucially, these statistics are not yet graphical attributes themselves, but will be turned into graphical coordinates either via a simple translation or by further computation.

5.3 Create Partitions: Partition

First, we need to split the data into partitions. We can do this directly by treating some variables in the data as sets of part labels or *factors*, and indeed this will be the typical use case for discrete variables. Alternatively, we can also turn continuous variables into factors by e.g. binning or truncation, as in the case of a histogram or a heatmap. Either way, a convenient way to represent these factors is as tuples $(indices, labels, metadata)$, where $indices$ is an integer vector of length n (= the number of rows in the data) and values $j = 1, \ldots, k$, with ith index of value j assigning the ith row of data to jth part, $labels$ is a dictionary assigning each jth part a collection of part labels consisting of a set of key-value pairs (e.g. "level", "binMin", "binMax"), and $metadata$ being additional information shared across parts, also represented as key-value pairs (e.g. the list of all bin breaks within a histogram).

5.4 Compute Statistics: Reduce

Second, for each jth part in a partition J, we need to compute a collection of summary statistics $s_j \in S$. If the summarizing functions that are used adhere to the monoidal contract (e.g. count, sum, product), this can be done in a single pass through the data or a *reduce* function, by looping through n rows of the data and for each ith row updating the jth part, where j is the corresponding ith factor index. If the summary statistic is not monoidal, we may be still able to compute auxiliary summary statistics that may be later used (in the *map* step) to compute the non-monoidal summary statistic, although breaking the monoidal contract in this way does come with the disadvantage of the representations no longer being guaranteed to be consistent.

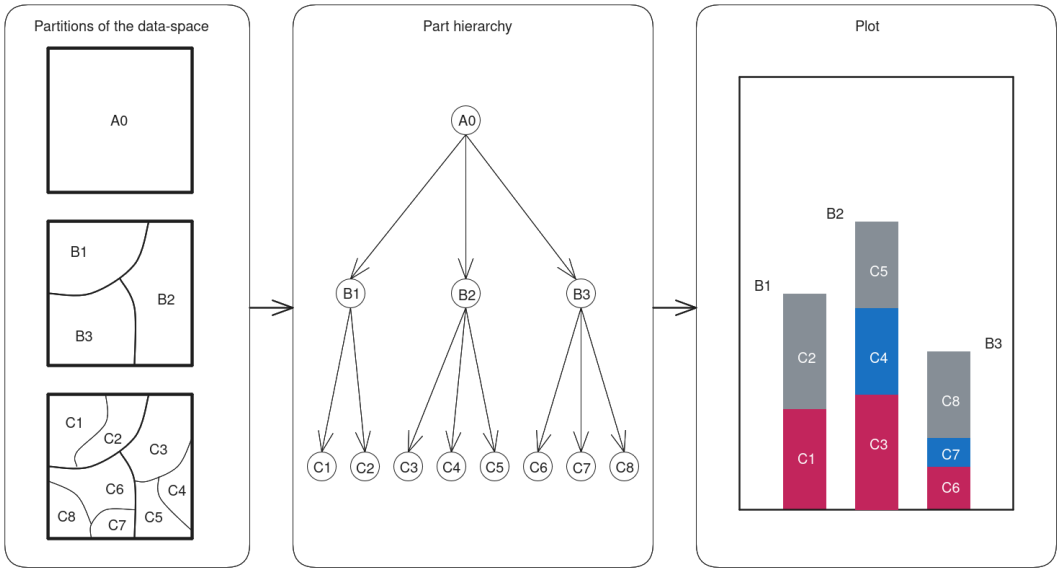
5.5 Translate Statistics to Coordinates: Map

Third, we need to translate each collection of summary statistics $s_j \in S$ into a collection of graphical coordinates $g_j \in G$.

5.6 Combine Coordinates: Stack

6 Something else

Symmetric monoidal preorders also come with some4 computational advantages. Firstly, these summaries are guaranteed to be computable within a single pass through the data, and can also be updated online if new data arrives, without having to refer to the old. This does not necessarily make them superior to other summaries, such as mean or variance, which can also often be computed by collecting multiple constituent summaries in a single pass through the data and them combining them together in one step after (such as sum and count for the mean), and for which online algorithms also often exist too. However, the advantage is that every symmetric monoidal preorder is automatically single pass computable and online - we will never need to look for an algorithm. Secondly, if our plot implements reactive axis limits, since the count within the selection is always guaranteed to be less that or equal to the count within the whole, we only need to keep track of the counts on the whole. For example, in the case of the barplot of counts, we know that the upper y-axis limit will always be equal to the height of the tallest whole bar, and we can safely ignore the sub-bars. If there are N levels of the categorical variable and K selection groups, there will be N whole bars and $N \times K$ sub-bars. This difference might unimportant if N and K are small (as they are likely to be, in the case of the average barplot). However, if there are, for example, two categorical variables with N, and M levels, respectively, as in the case of a fluctuation diagram/bubbleplot/treemap, and both N and M have many levels, then having to iterate through all $N \times M \times K$ sub-bar summaries every time an interaction happens might become prohibitive.


Partitions of the data-space
A0
B1
B2
B3
C1
C2
C3
C4
C5
C6
C7
C8
Part hierarchy
A0
B1
B2
B3
C1
C2
C3
C4
C5
C6
C7
C8
Plot
B1
B2
B3
C1
C2
C3
C4
C5
C6
C7
C8