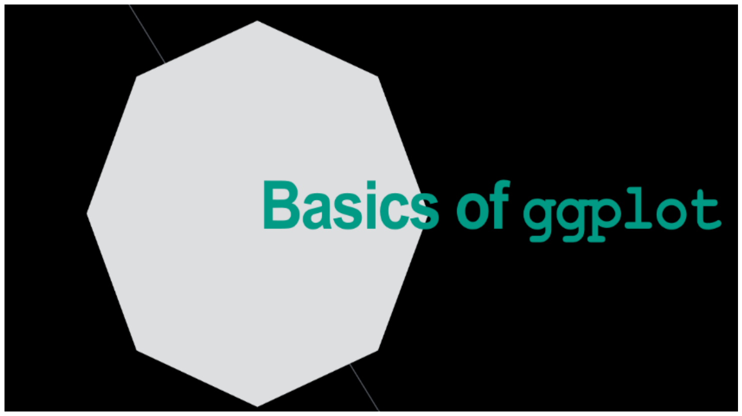
IT137IU: Data Analysis
Lab#6/Assignment#6: Data Visualization

Introduction



In this lab you will acquire the skills needed to visualize data in R. The objectives of the guide are as follows:

1. Get familiar with ggplot.
2. Learn how to use ggplot to visualize data.

1. Basics of ggplot

In this section, we will work towards a first plot with ggplot. It will be a scatter plot for the avocado price data.

The following paragraphs introduce the key concepts of ggplot:

- **incremental composition**: adding elements or changing attributes of a plot incrementally
- **convenience functions & defaults**: a closer look at high-level convenience functions (like geom_point) and what they actually do
- **layers**: seeing how layers are stacked when we call, e.g. different geom_ functions in sequence
- **grouping**: what happens when we use grouping information (e.g., for color, shape or in facets)

The section finishes with a first full example of a plot that has different layers, uses grouping, and customizes a few other things.

To get started, let's first load the (preprocessed) avocado data set used for plotting:

```
avocado_data <- read.csv("avocado.csv")
```

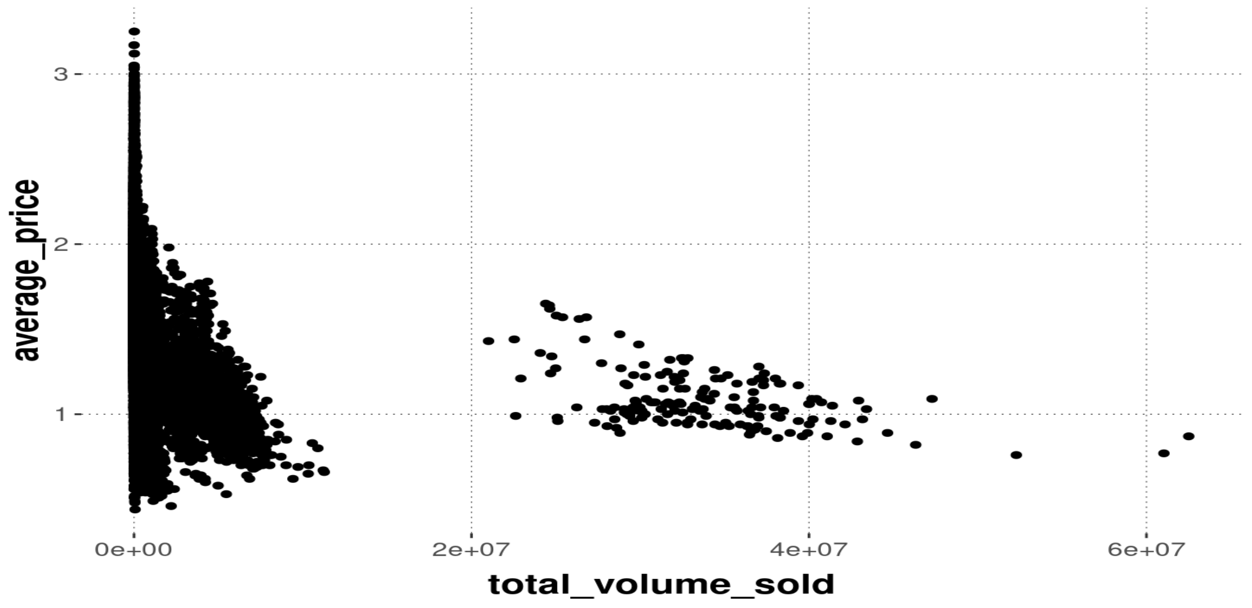
a. Incremental composition of a plot

The “gg” in the package name `ggplot` is short for “grammar of graphs”. It provides functions for describing scientific data plots in a compositional manner, i.e., for dealing with different recurrent elements in a plot in an additive way. As a result of this approach, we will use the symbol `+` to *add* more and more elements (or to override the implicit defaults in previously evoked elements) to build a plot. For example, we can obtain a scatter plot for the avocado price data simply by first calling the function `ggplot`, which just creates an empty plot:

```
incrementally_built_plot <- ggplot()
```

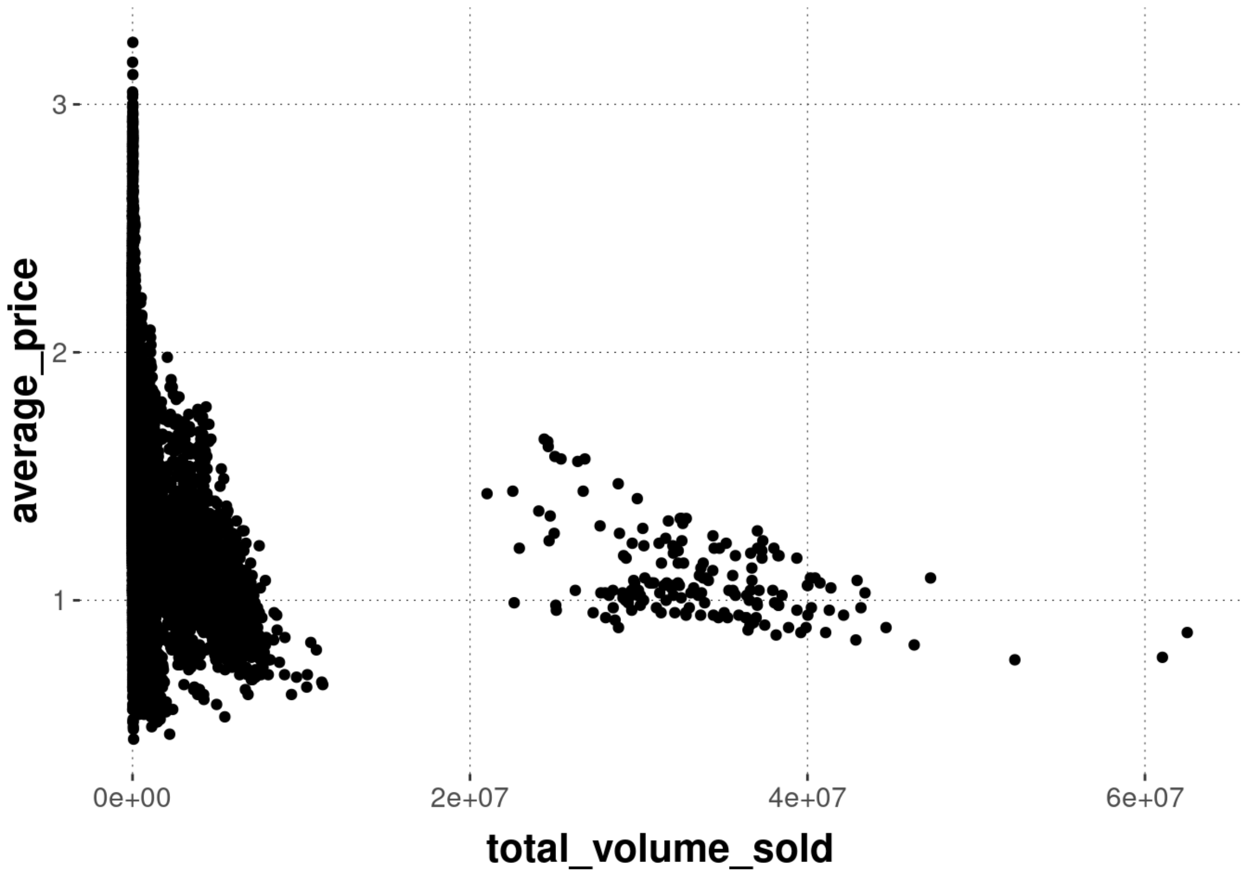
But we can add some stuff. Don’t get hung up on the details right now, just notice that we use `+` to add stuff to our plot

```
incrementally_built_plot +
  # add a geom of type `point` (=> scatter plot)
  geom_point(
    # what data to use
    data = avocado_data,
    # supply a mapping (in the form of an 'aesthetic' (see below))
    mapping = aes(
      # which variable to map onto the x-axis
      x = total_volume_sold,
      # which variable to map onto the y-axis
      y = average_price
      )
  )
```



You see that the function `geom_point` is what makes the points appear. You tell it which data to use and which mapping of variables from the data set to elements in the plot you like. That's it, at least to begin with.
We can also supply the information about the data to use and the aesthetic mapping in the `ggplot` function call. Doing so will make this information the default for any subsequently added layer. Notice also that the `data` argument in function `ggplot` is the first argument, so we will frequently make use of piping, like in the following code which is equivalent to the previous in terms of output:

```
avocado_data %>%
  ggplot(aes(x = total_volume_sold, y = average_price)) +
  geom_point()
```



b. Elements in the layered grammar of graphs

Let's take a step back. Actually, the function `geom_point` is a convenience function that does a lot of things automatically for us. It helps to understand subsequent code if we peek under the hood at least for a brief moment initially, if only to just realize where some of the terminology in and around the "grammar of graphs" comes from.

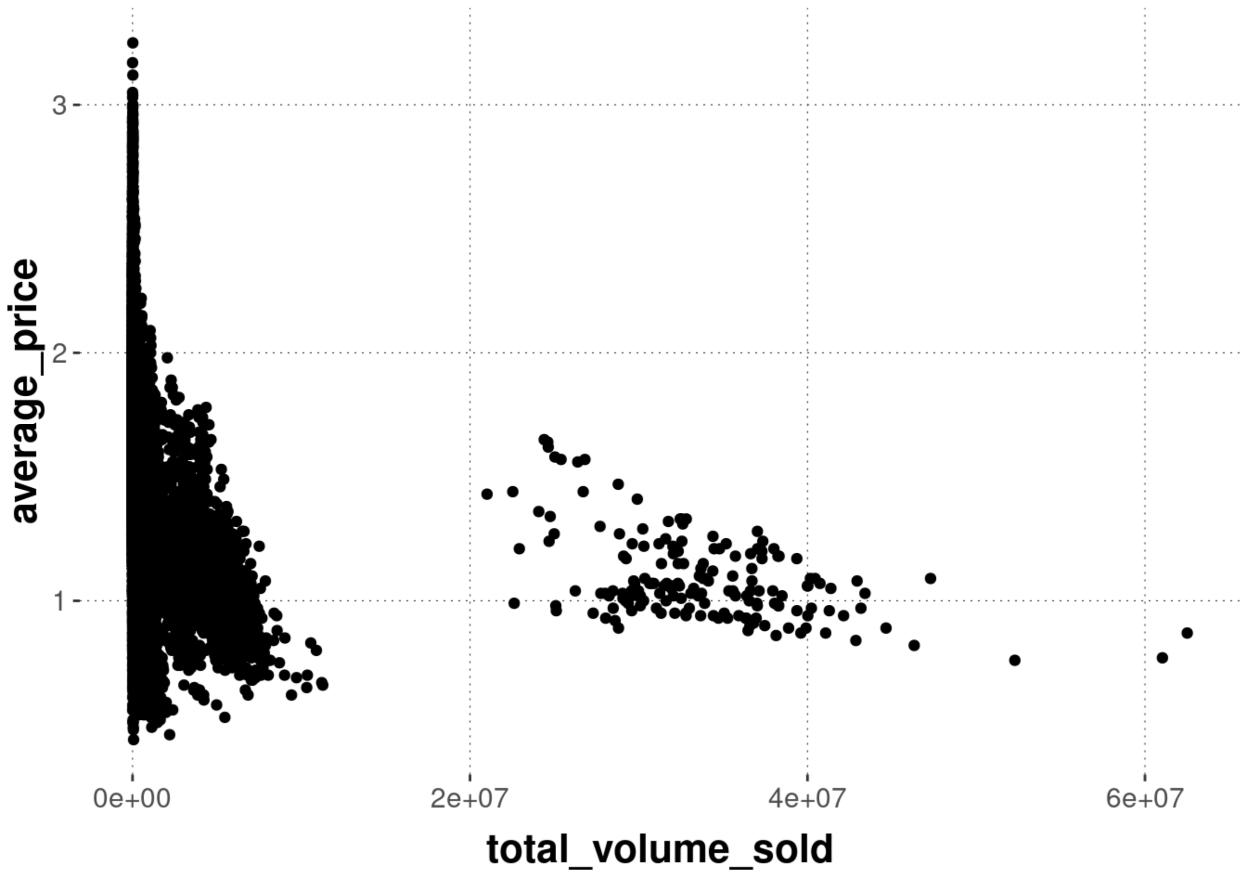
The `ggplot` package defines a **layered grammar of graphs** (Wickham 2010). This is a structured description language for plots (relevant for data science). It uses a smart system of defaults so that it suffices to often just call a convenience wrapper like `geom_point`. But underneath, there is the possibility of tinkering with (almost?) all of the (layered) elements and changing the defaults if need be.
The process of mapping data onto a visualization essentially follows this route:

```
data -> statistical transformation -> geom. object -> aesthetics
```

You supply (tidy) data. The data is then transformed (e.g., by computing a summary statistic) in some way or another. This could just be an "identity map" in which case you will visualize the data exactly as it is. The resulting data representation is mapped onto some spatial (geometric) appearance, like a line, a dot, or a geometric shape. Finally, there is room to alter the specific aesthetics of this mapping from data to visual object, like adjusting the size or the color of a geometric object, possibly depending on some other properties it has (e.g., whether it is an observation for a conventional or an organically grown avocado).

To make explicit the steps which are implicitly carried out by `geom_point` in the example above, here is a fully verbose but output-equivalent sequence of commands that builds the same plot by defining all the basic components manually:

```
avocado_data %>%
ggplot() +
  # plot consists of layers (more on this soon)
  layer(
    # how to map columns onto ingredients in the plot
    mapping = aes(x = total_volume_sold, y = average_price),
    # what statistical transformation should be used? - here: none
    stat = "identity",
    # how should the transformed data be visually represented? - here: as
points
    geom = "point",
    # should we tinker in any other way with the positioning of each
element?
    # - here: no, thank you!
    position = "identity"
  ) +
  # x and y axes are non-transformed continuous
  scale_x_continuous() +
  scale_y_continuous() +
  # we use a cartesian coordinate system (not a polar or a geographical
map)
  coord_cartesian()
```

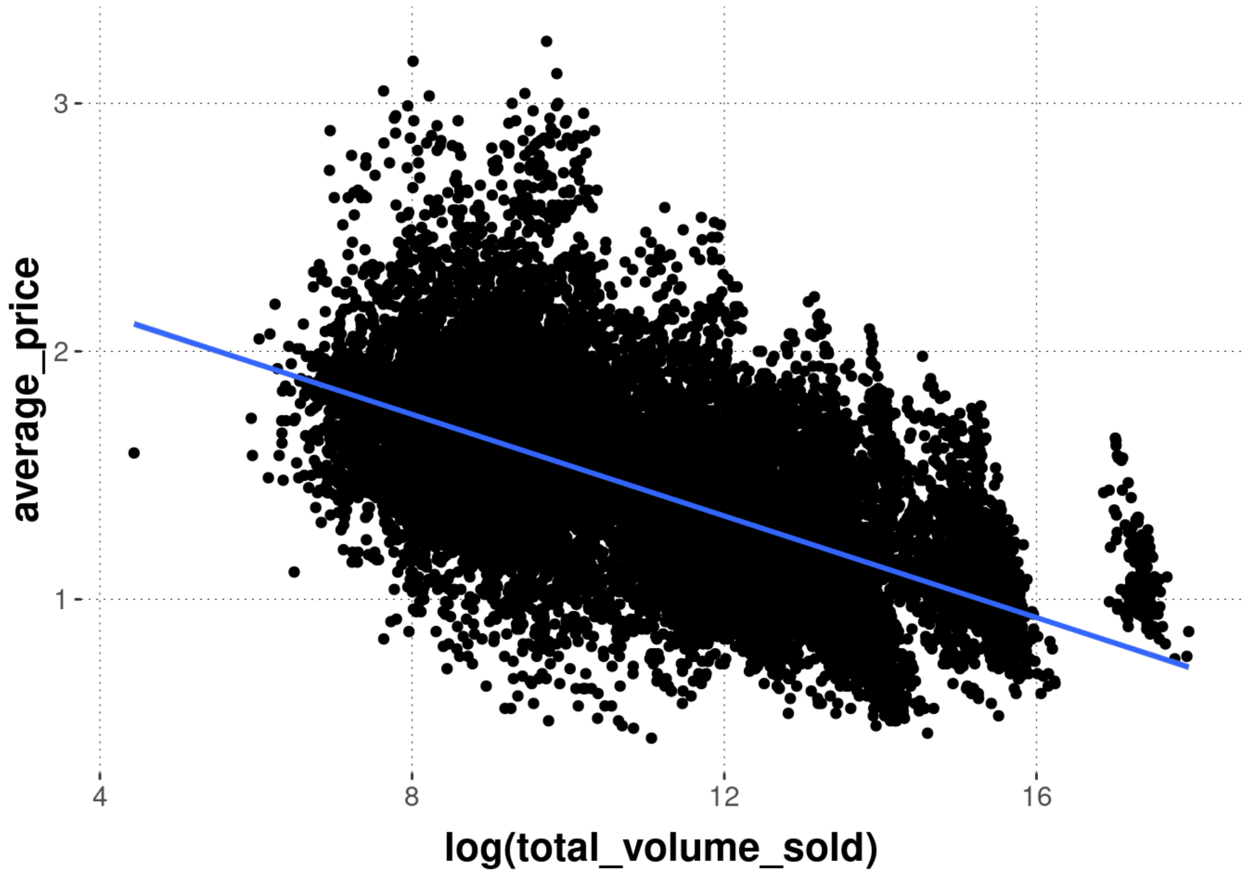


In this explicit call, we still need to specify the data and the mapping (which variable to map onto which axis). But we need to specify much more. We tell `ggplot` that we want standard (e.g., not log-transformed) axes. We also tell it that our axes are continuous, that the data should not be transformed and that the visual shape (= geom) to which the data is to be mapped is a point (hence the name `geom_point`).
It is not important to understand all of these components right now. It is important to have seen them once, and to understand that `geom_point` is a wrapper around this call which assumes reasonable defaults (such as non-transformed axes, points for representation etc.).

c. Layers and groups

`ggplot` is the "grammar of *layered* graphs". Plots are compositionally built by combining different layers, if need be. For example, we can use another function from the `geom_` family of functions to display a different visualization derived from the same data on top of our previous scatter plot

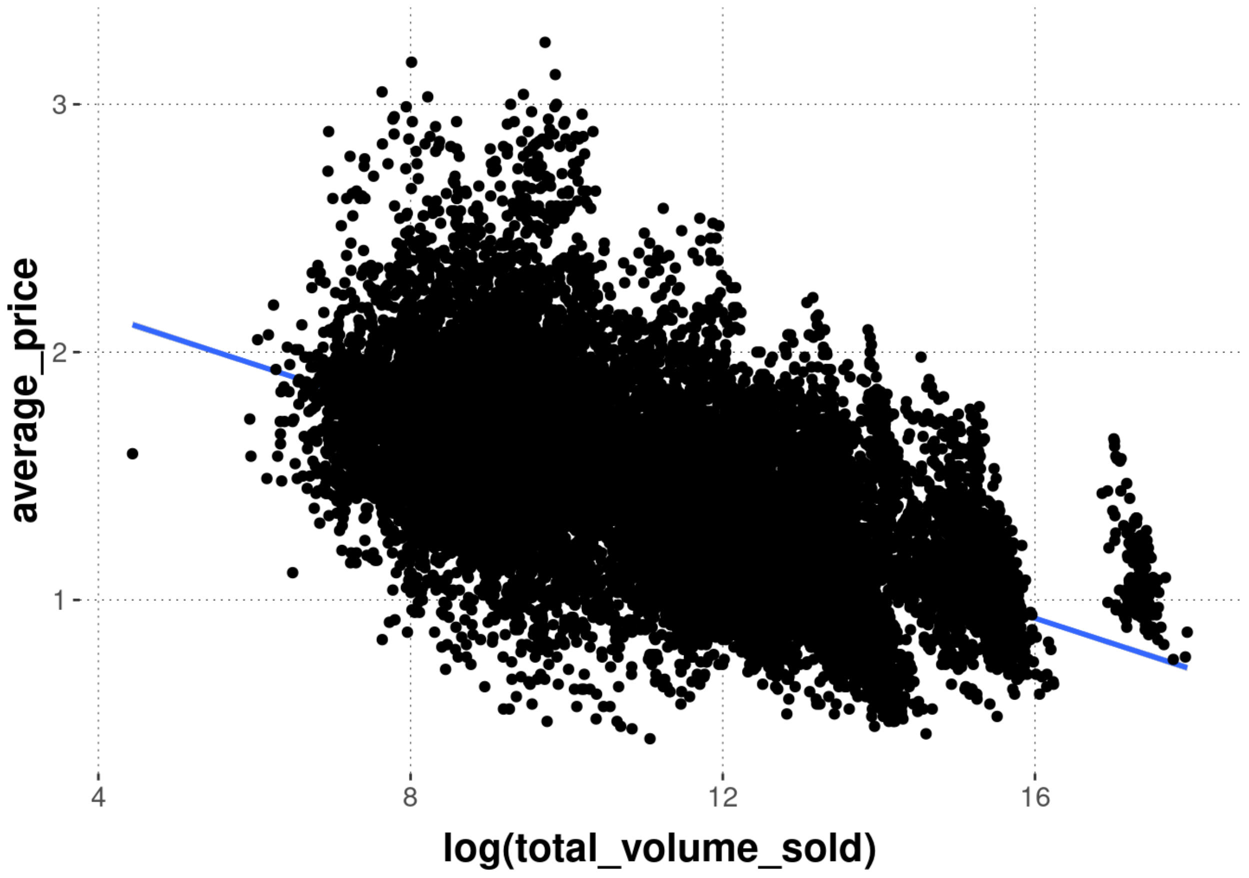
```
avocado_data %>%
  ggplot(
    mapping = aes(
      # notice that we use the log (try without it to understand why)
      x = log(total_volume_sold),
      y = average_price
    )
  ) +
  # add a scatter plot
  geom_point() +
  # add a linear regression line
  geom_smooth(method = "lm")
```



Notice that layering is really sequential. To see this, just check what happens when we reverse the calls of the `geom_` functions in the previous example:

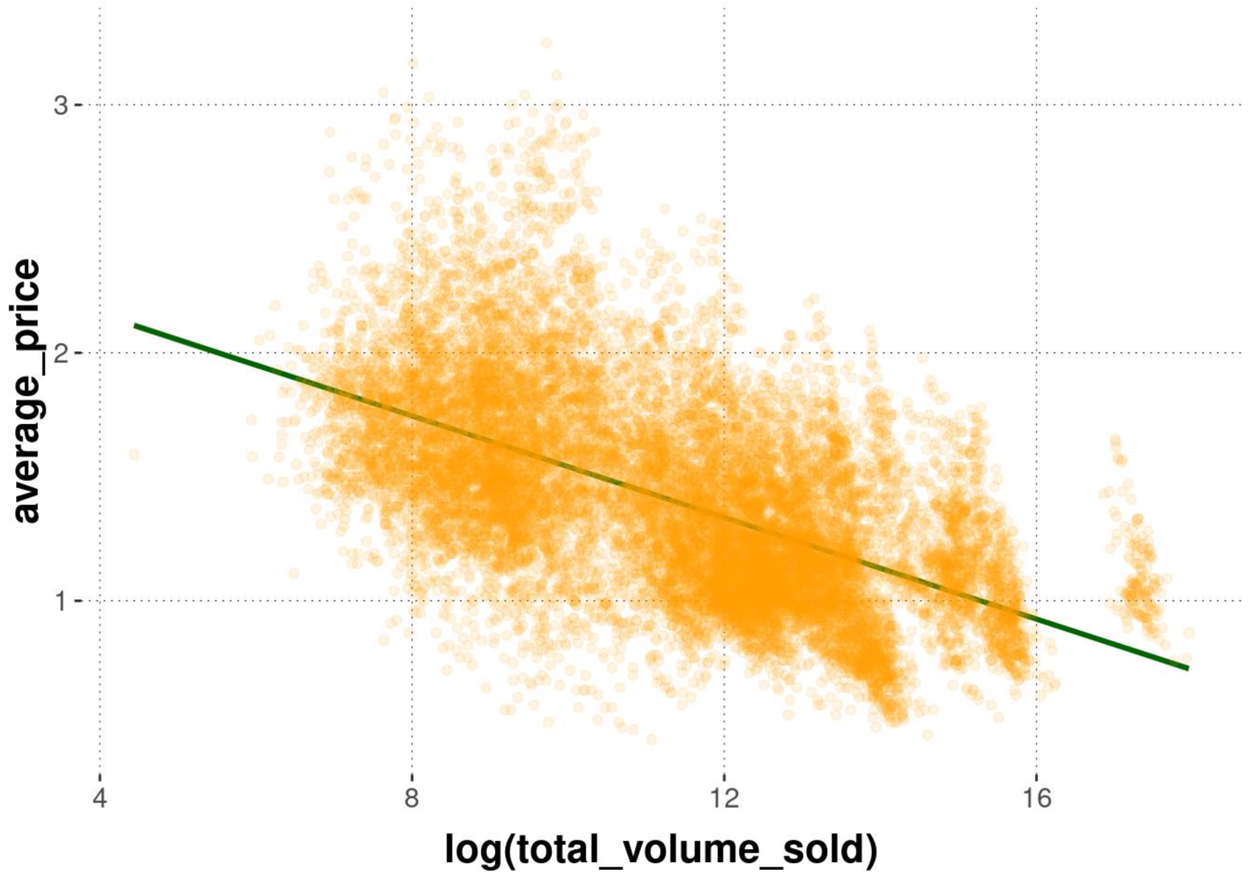
```
avocado_data %>%
  ggplot(
    mapping = aes(
      # notice that we use the log (try without it to understand why)
      x = log(total_volume_sold),
      y = average_price
    )
```

```r
  ) +
  # FIRST: add a linear regression line
  geom_smooth(method = "lm") +
  # THEN: add a scatter plot
  geom_point()
```



If you want lower layers to be visible behind layers added later, one possibility is to tinker with opacity, via the `alpha` parameter. Notice that the example below also changes the colors. The result is quite toxic, but at least you see the line underneath the semi-transparent points.

```r
avocado_data %>%
  ggplot(
    mapping = aes(
      # notice that we use the log (try without it to understand why)
      x = log(total_volume_sold),
      y = average_price
    )
  ) +
  # FIRST: add a linear regression line
  geom_smooth(method = "lm", color = "darkgreen") +
  # THEN: add a scatter plot
  geom_point(alpha = 0.1, color = "orange")
```

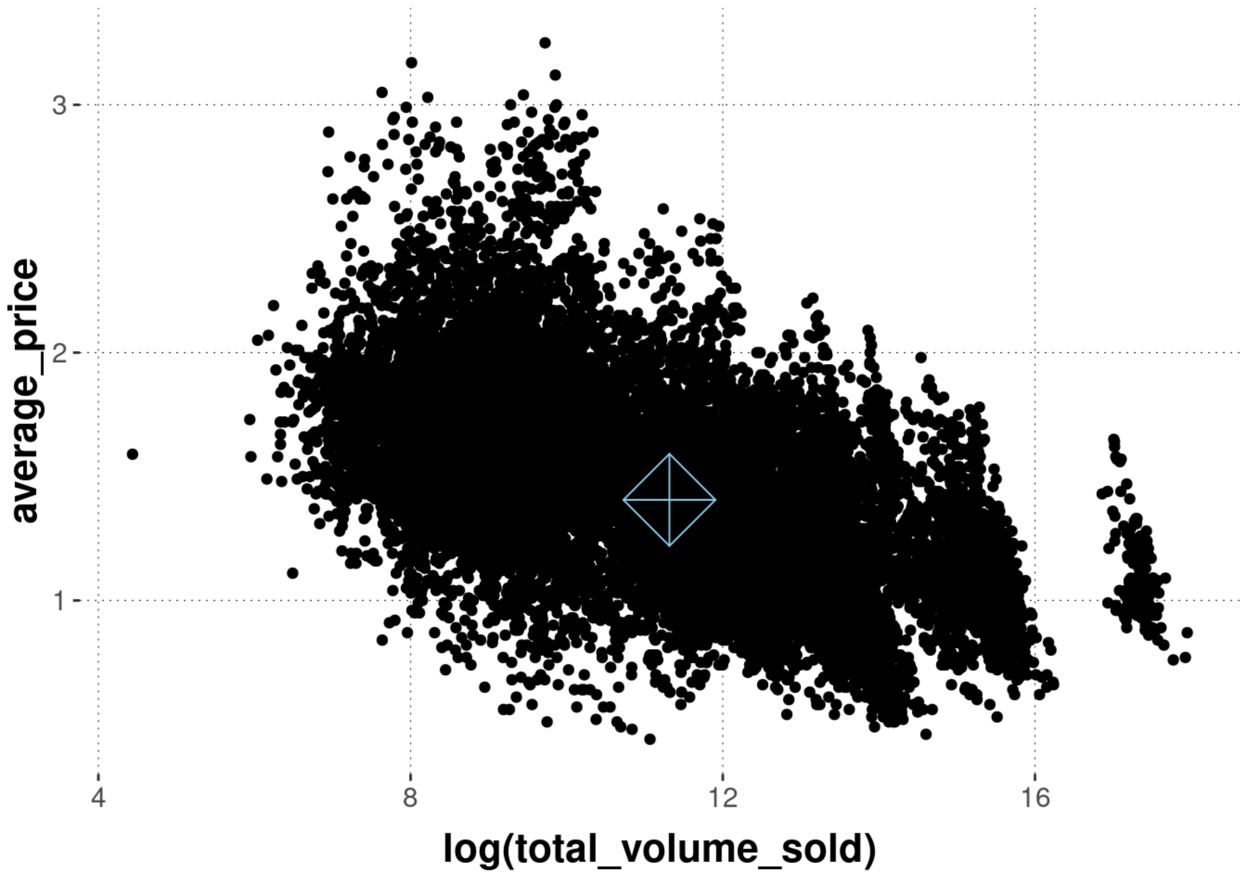


The aesthetics defined in the initial call to `ggplot` are global defaults for all layers to follow, unless they are overwritten. This also holds for the data supplied to `ggplot`. For example, we can create a second layer using another call to `geom_point` from a second data set (e.g., with a summary statistic), like so:

```
# create a small tibble with the means of both
#   variables of interest
avocado_data_means <-
  avocado_data %>%
  summarize(
    mean_volume = mean(log(total_volume_sold)),
    mean_price  = mean(average_price)
  )
avocado_data_means
```

```
## # A tibble: 1 × 2
##   mean_volume mean_price
##         <dbl>      <dbl>
## 1        11.3       1.41
```

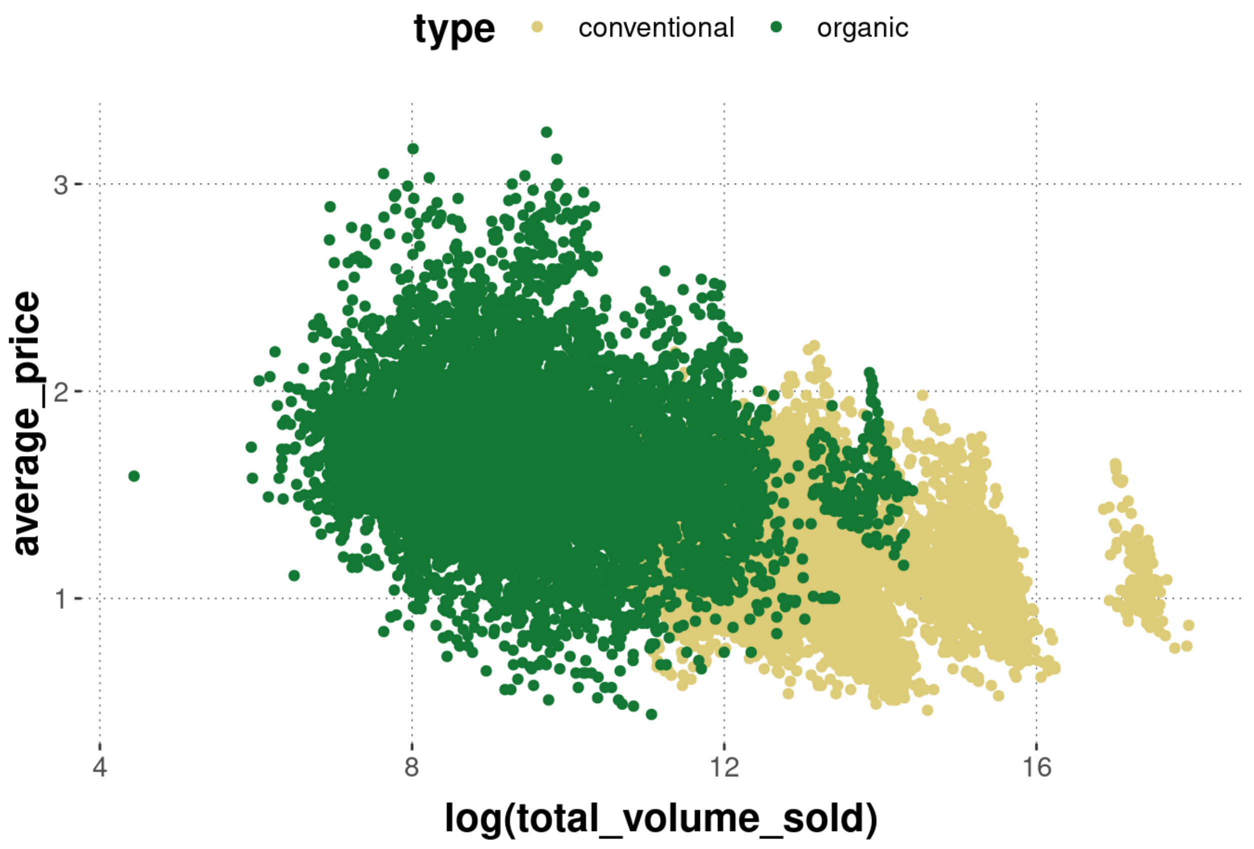
```
avocado_data %>%
  ggplot(
    aes(x = log(total_volume_sold),
        y = average_price)
  ) +
  # first layer uses globally declared data & mapping
  geom_point() +
  # second layer uses different data set & mapping
  geom_point(
    data = avocado_data_means,
    mapping = aes(
      x = mean_volume,
      y = mean_price
    ),
    # change shape of element to display (see below)
    shape = 9,
    # change size of element to display
    size = 12,
    color = "skyblue"
  )
```



d. Grouping

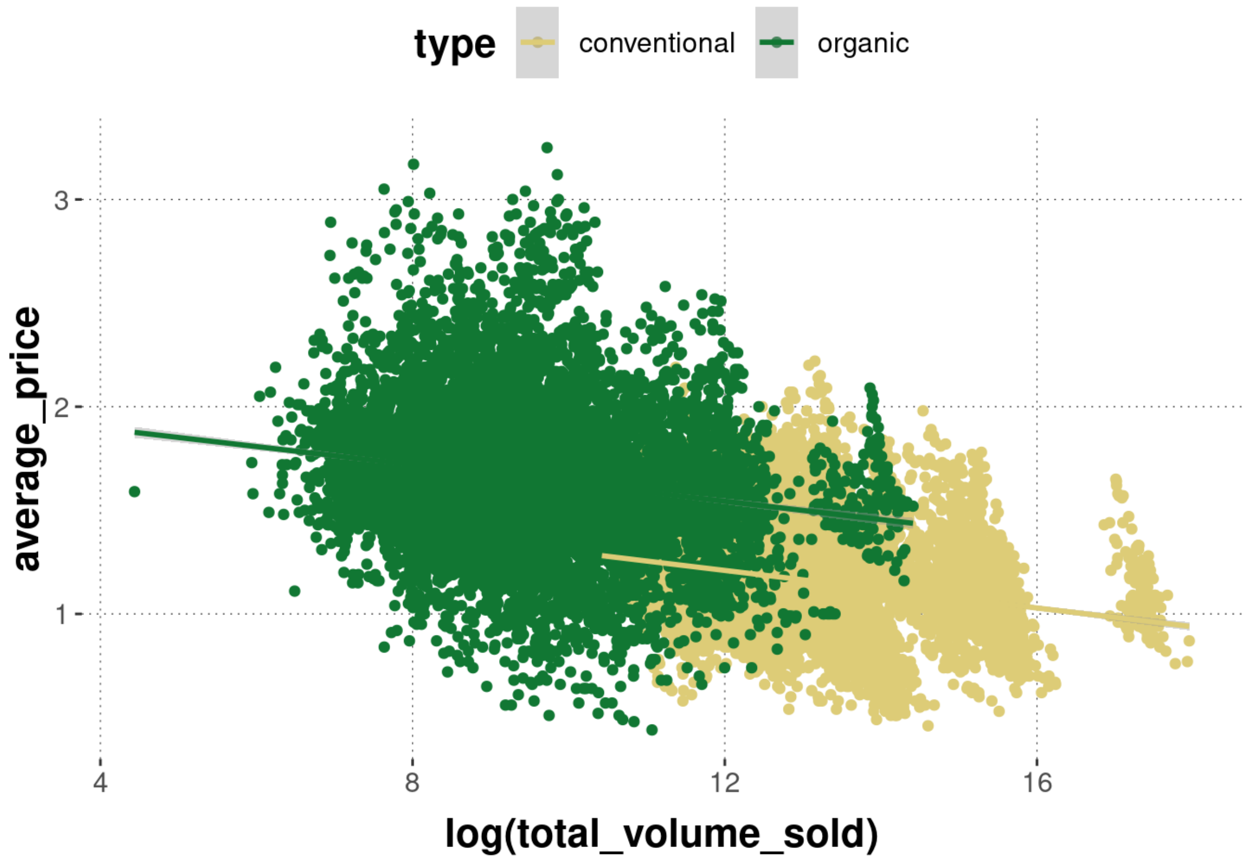
Categorical distinction is frequently important in data analysis. Just think of the different combinations of factor levels in a factorial design, or the difference between conventionally grown and organically grown avocados. `ggplot` understands grouping very well and acts on appropriately, if you tell it to in the right way.
Grouping can be relevant for different aspects of a plot: the color of points or lines, their shape, or even whether to plot everything together or separately. For instance, we might want to display different types of avocados in a different color. We can do this like so:

```
avocado_data %>%
  ggplot(
    aes(
      x = log(total_volume_sold),
      y = average_price,
      # use a different color for each type of avocado
      color = type
    )
  ) +
  geom_point()
```



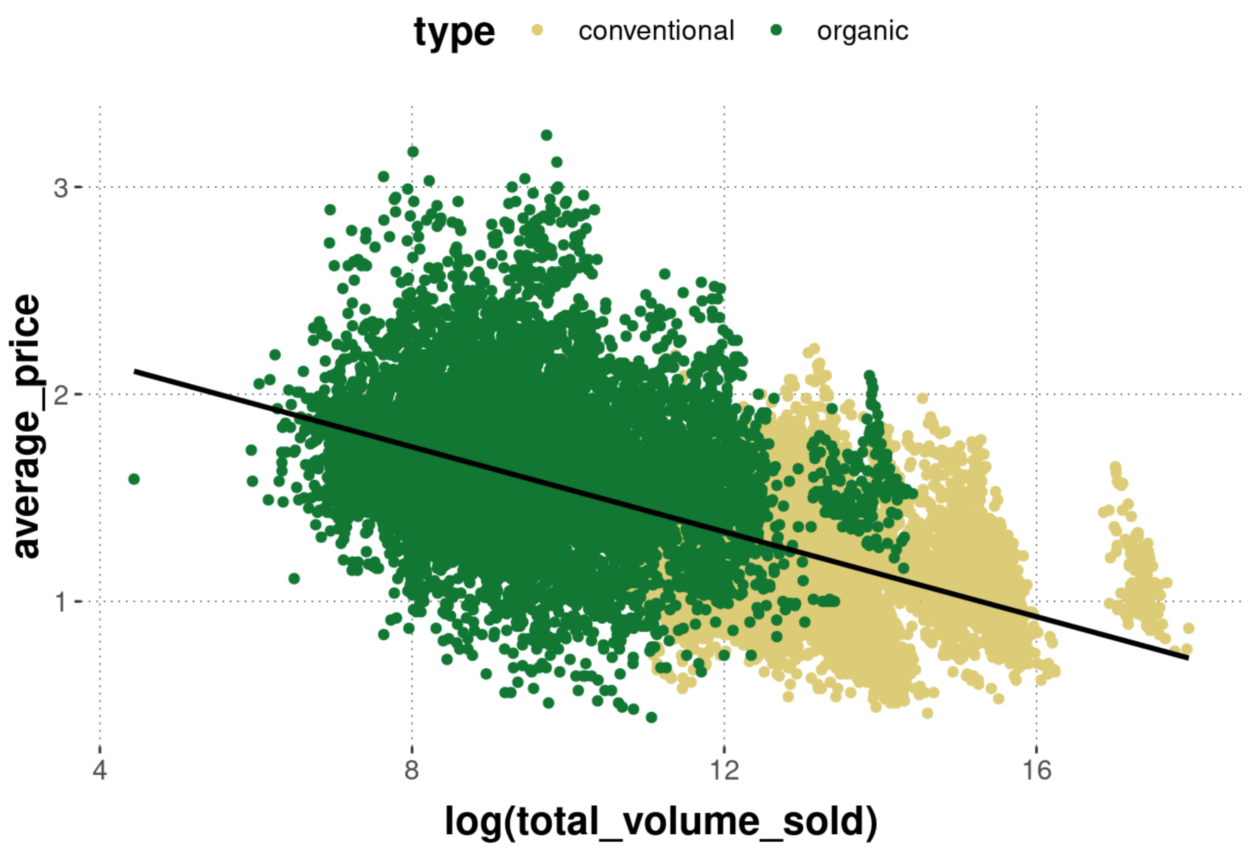
Notice that we added the grouping information inside of `aes` to the call of `ggplot`. This way the grouping is the global default for the whole plot. Check what happens when we then add another layer, like `geom_smooth:`

```
avocado_data %>%
  ggplot(
    aes(
      x = log(total_volume_sold),
      y = average_price,
      # use a different color for each type of avocado
      color = type
    )
  ) +
  geom_point() +
  geom_smooth(method = "lm")
```



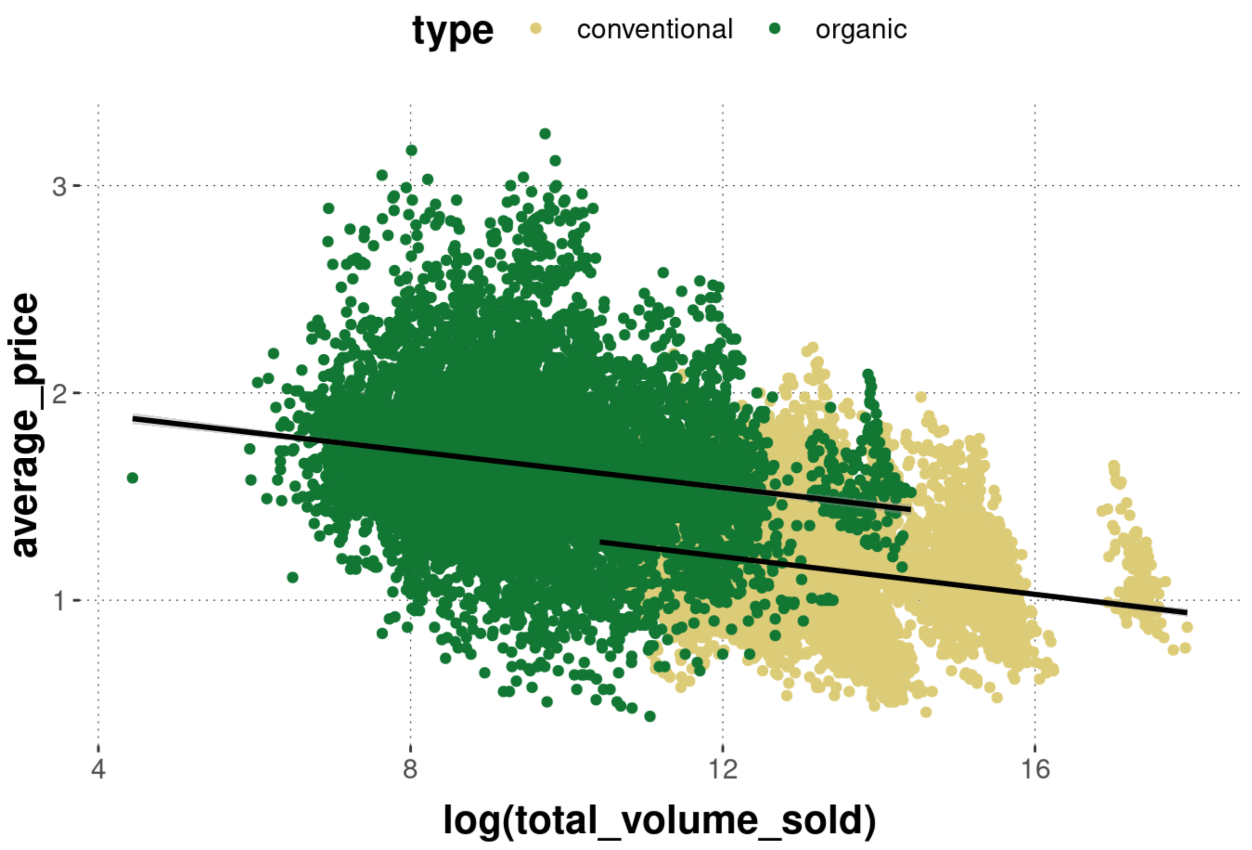
The regression lines will also be shown in the colors of the underlying scatter plot. We can change this by overwriting the `color` attribute locally, but then we lose the grouping information:

```
avocado_data %>%
  ggplot(
    aes(
      x = log(total_volume_sold),
      y = average_price,
      # use a different color for each type of avocado
      color = type
    )
  ) +
  geom_point() +
  geom_smooth(method = "lm", color = "black")
```



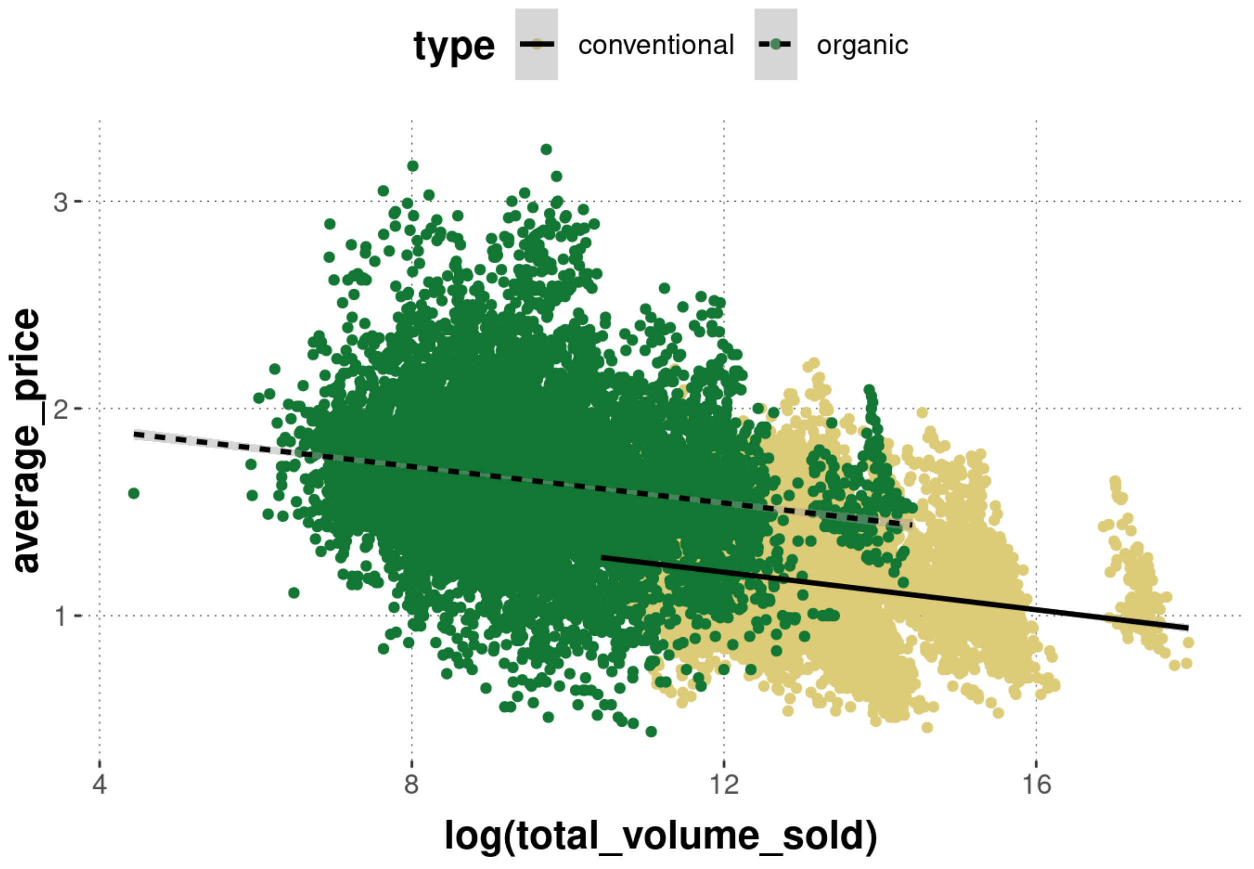
To retrieve the grouping information, we can change the explicit keyword `group` (which just treats data from the relevant factor levels differently without directly changing their appearance):

```
avocado_data %>%
  ggplot(
    aes(
      x = log(total_volume_sold),
      y = average_price,
      # use a different color for each type of avocado
      color = type
    )
  ) +
  geom_point() +
  geom_smooth(
    # tell the smoother to deal with avocados types separately
    aes(group = type),
    method = "lm",
    color = "black"
  )
```



Finally, we see that the lines are not uniquely associable with the avocado type, so we can also change the regression line's `shape` attribute conditional on avocado type:

```
avocado_data %>%
  ggplot(
    aes(
      x = log(total_volume_sold),
      y = average_price,
      # use a different color for each type of avocado
      color = type
    )
  ) +
  geom_point() +
  geom_smooth(
    # tell the smoother to deal with avocados types separately
    aes(group = type, linetype = type),
    method = "lm",
    color = "black"
  )
```

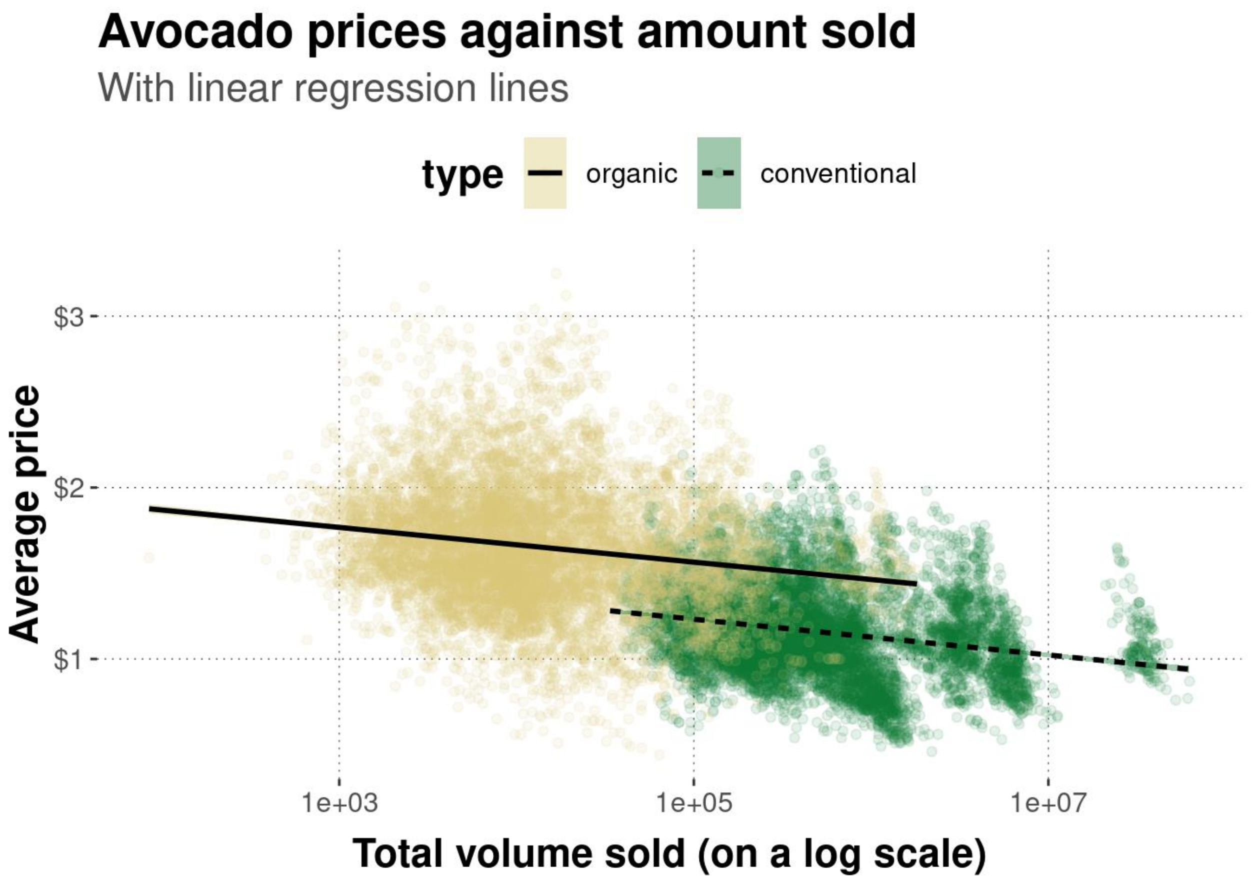


e. Example of a customized plot

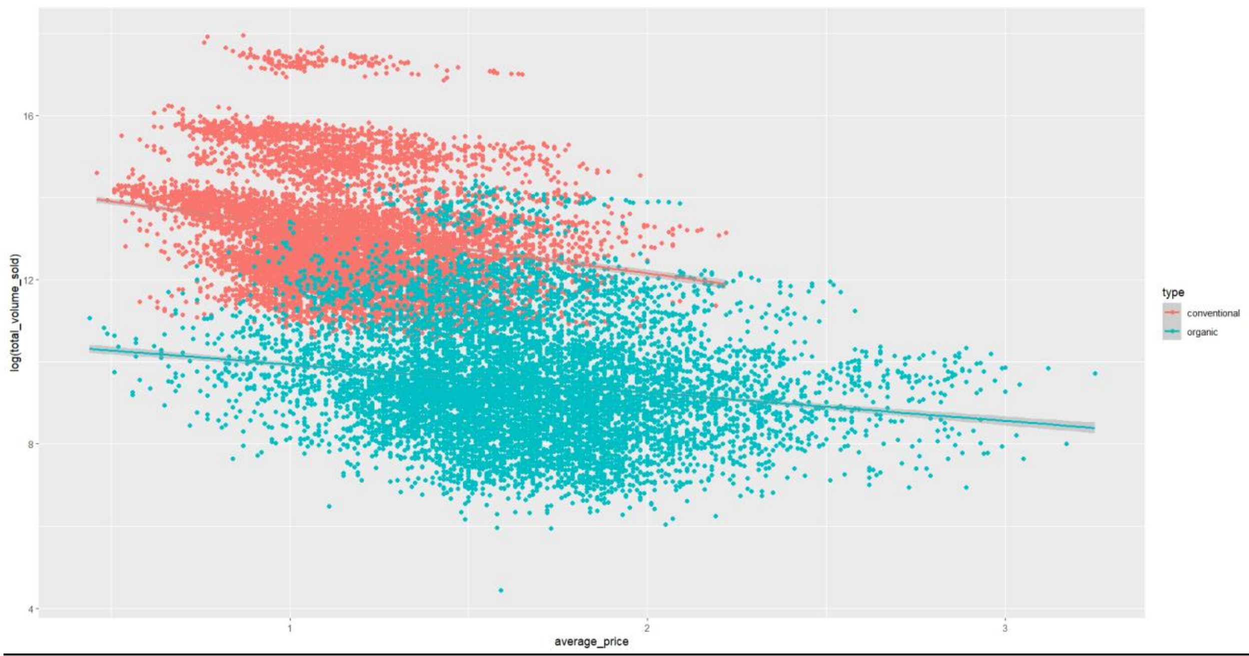
If done with the proper mind and heart, plots intended to share (and to communicate a point, following the idea of hypothesis-driven visualization) will usually require a lot of tweaking.

To nevertheless get a feeling of where the journey is going, at least roughly, here is an example of a plot of the avocado data which is much more tweaked and honed. No claim is intended regarding the false idea that this plot is in any sense optimal. There is not even a clear hypothesis or point to communicate. This just showcases some functionality. Notice, for instance, that this plot uses two layers, invoked by `geom_point` which shows the scatter plot of points and `geom_smooth` which layers on top the point cloud regression lines (one for each level in the grouping variable).

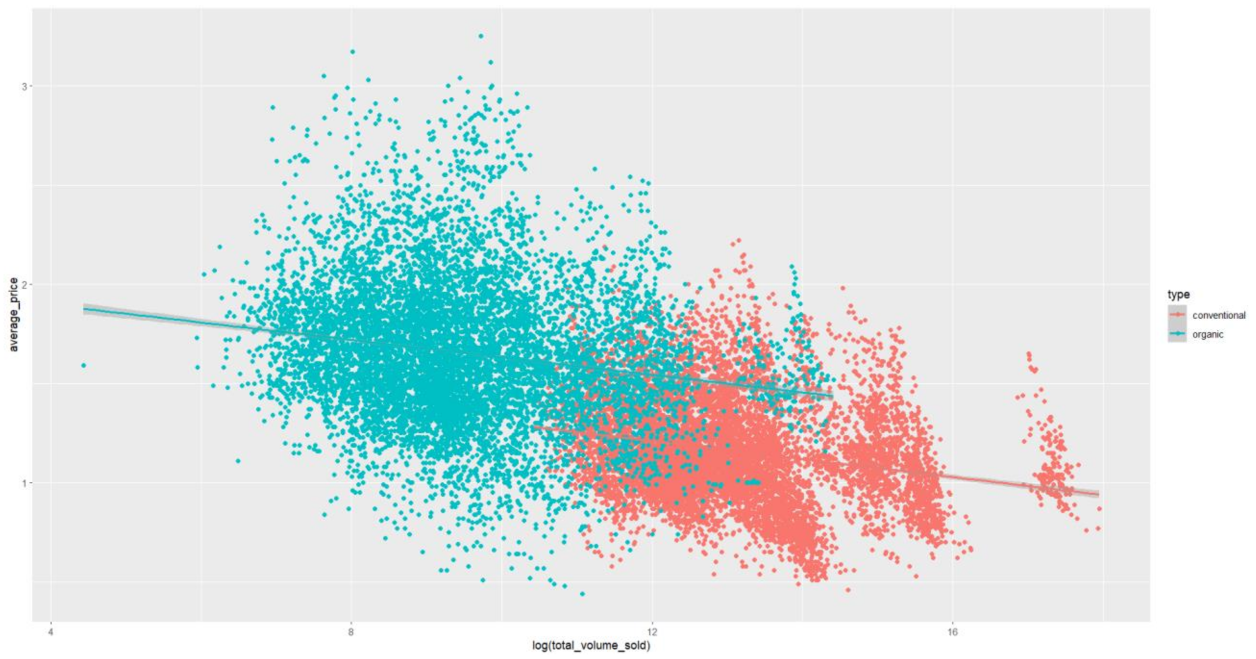
```r
# pipe data set into function `ggplot`
avocado_data %>%
  # reverse factor level so that horizontal legend entries align with
  # the majority of observations of each group in the plot
  mutate(
    type = fct_rev(type)
  ) %>%
  # initialize the plot
  ggplot(
    # defined mapping
    mapping = aes(
      # which variable goes on the x-axis
      x = total_volume_sold,
      # which variable goes on the y-axis
      y = average_price,
      # which groups of variables to distinguish
      group = type,
      # color and fill to change by grouping variable
      fill = type,
      linetype = type,
      color = type
    )
  ) +
  # declare that we want a scatter plot
  geom_point(
    # set low opacity for each point
    alpha = 0.1
  ) +
  # add a linear model fit (for each group)
  geom_smooth(
    color = "black",
    method = "lm"
  ) +
  # change the default (normal) of x-axis to log-scale
  scale_x_log10() +
  # add dollar signs to y-axis labels
  scale_y_continuous(labels = scales::dollar) +
  # change axis labels and plot title & subtitle
  labs(
    x = 'Total volume sold (on a log scale)',
    y = 'Average price',
    title = "Avocado prices against amount sold",
    subtitle = "With linear regression lines"
  )
```



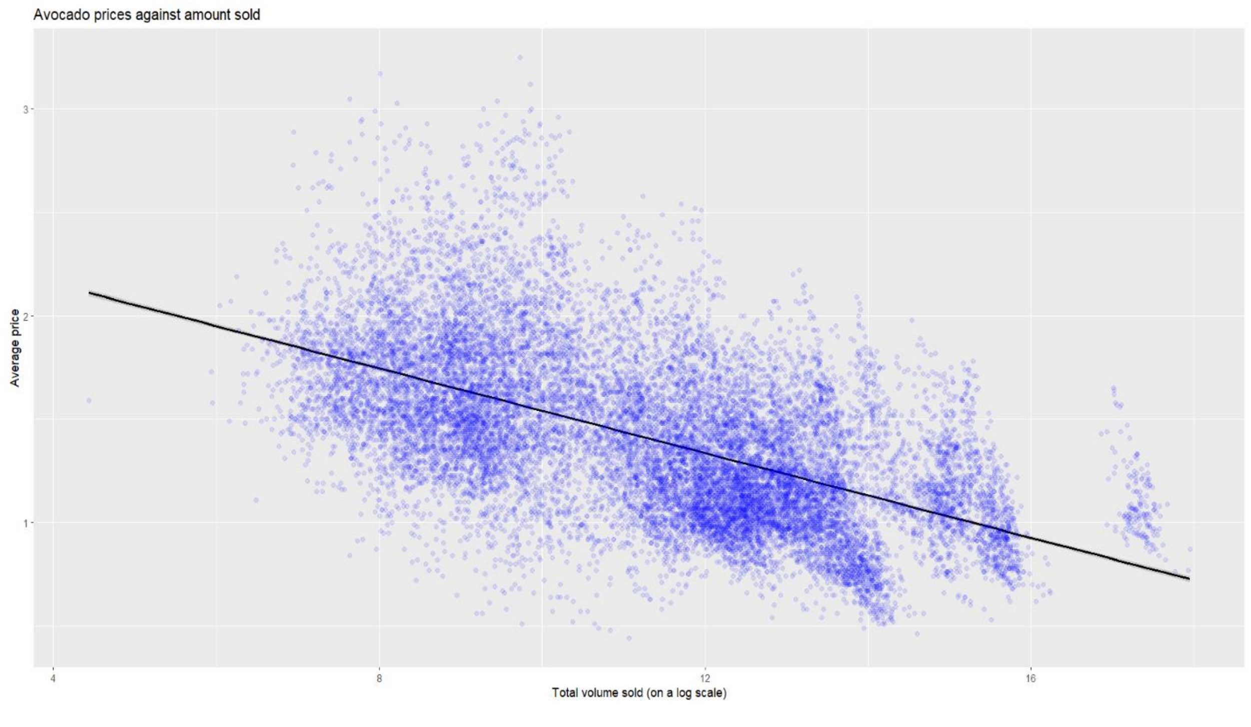
Exercise 1: [25pts] Let’s program using the avocado price data to create the following image:



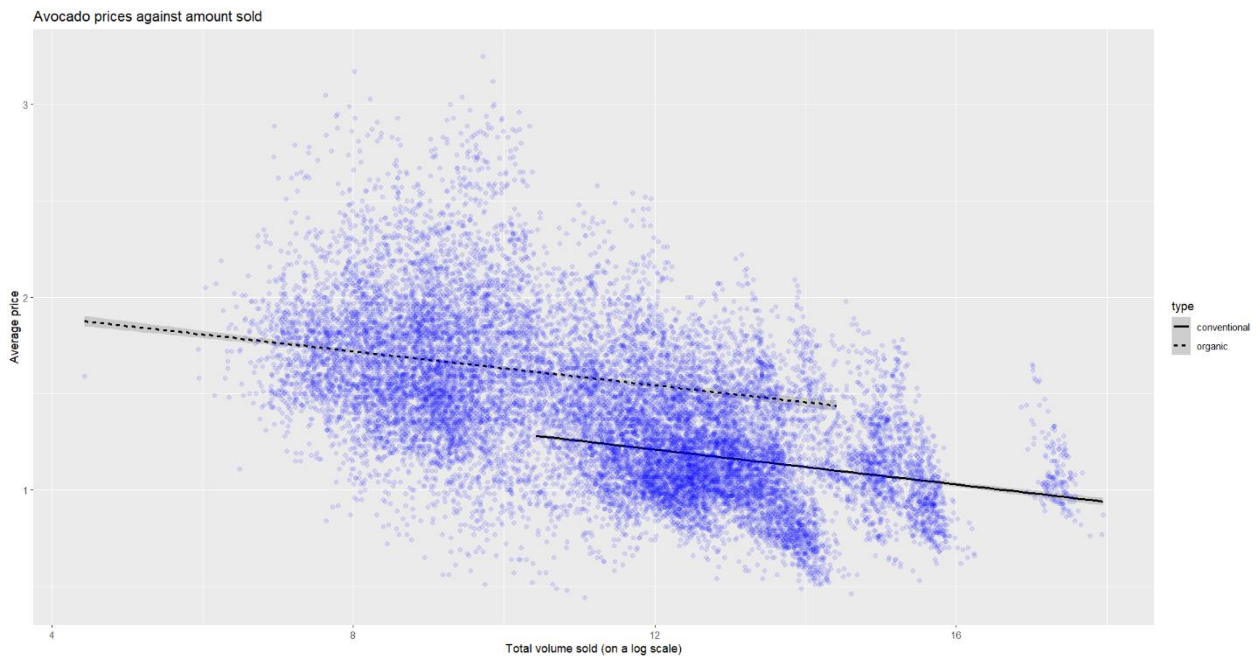
<u>Exercise 2</u>: [25pts] Let's program using the avocado price data to create the following image:



<u>Exercise 3</u>: [25pts] Let's program using the avocado price data to create the following image:



Exercise 4: [25pts] Let's program using the avocado price data to create the following image:



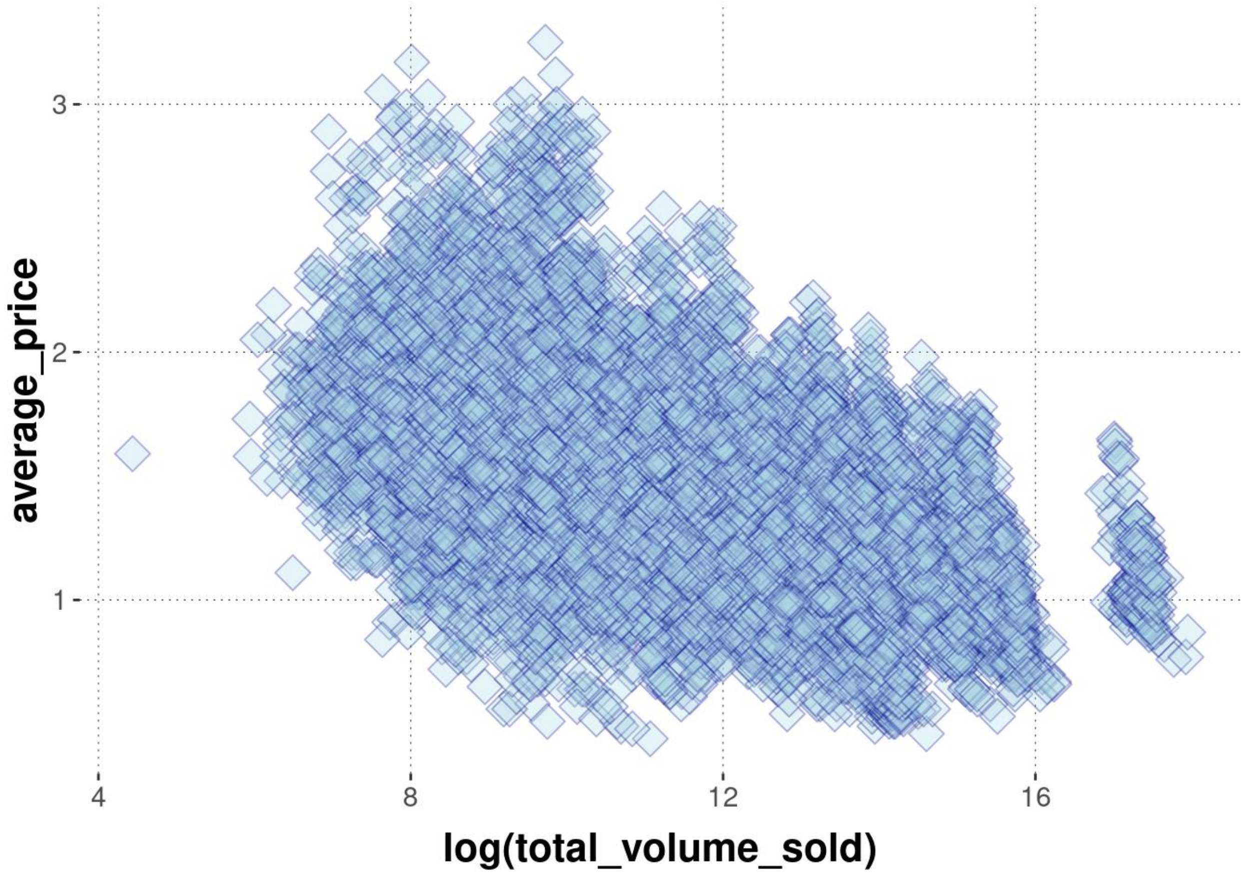
2. A rendezvous with popular geoms

In the following, we will cover some of the more basic `geom_` functions relevant for our present purposes. It might be useful to read this section top-to-bottom at least once, not to think of it as a mere reference list.

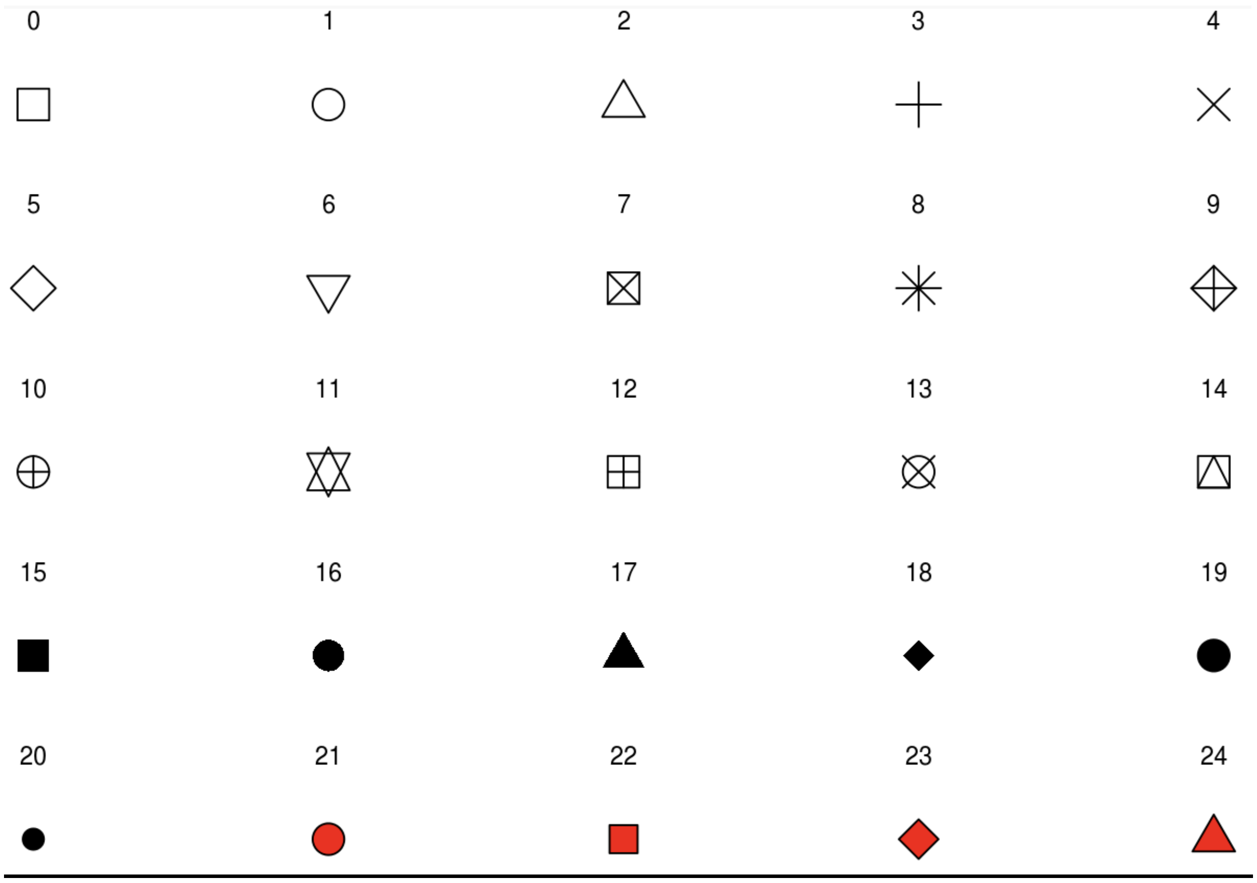
Scatter plots with `geom_point`

Scatter plots visualize pairs of associated observations as points in space. We have seen this for the avocado prize data above. Let's look at some of the further arguments we can use to tweak the presentation by `geom_point`. The following example changes the shape of the objects displayed to tilted rectangles (sometimes called diamonds, e.g., in LaTeX `\diamond`) away from the default circles, the color of the shapes, their size and opacity.

```
avocado_data %>%
  ggplot(aes(x = log(total_volume_sold), y = average_price)) +
  geom_point(
    # shape to display is number 23 (tilted rectangle, see below)
    shape = 23,
    # color of the surrounding line of the shape (for shapes 21-24)
    color = "darkblue",
    # color of the interior of each shape
    fill = "lightblue",
    # size of each shape (default is 1)
    size = 5,
    # level of opacity for each shape
    alpha = 0.3
  )
```



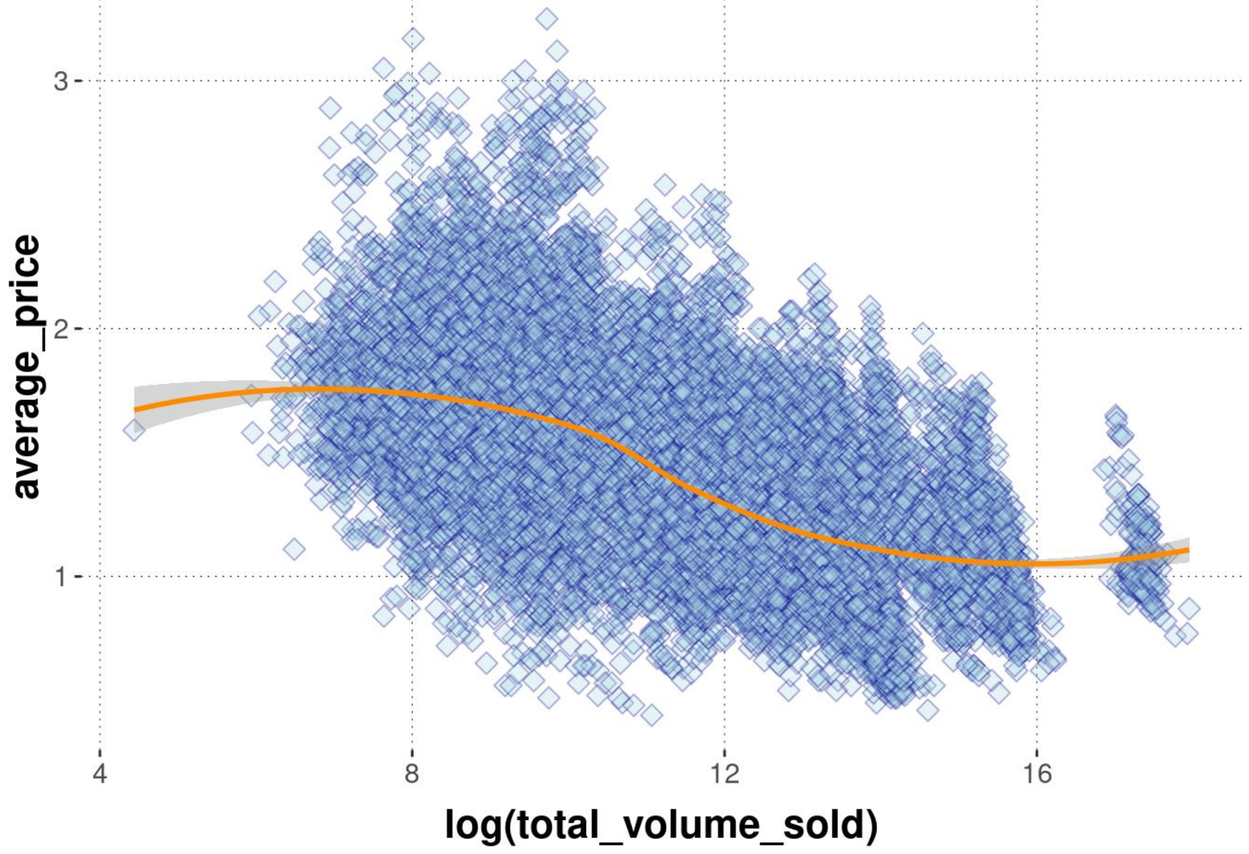
How do you know which shape is which number? - By looking at the picture in Figure below, for instance.



Smooth

The `geom_smooth` function operates on two-dimensional metric data and outputs a smoothed line, using different kinds of fitting functions. It is possible to show an indicator of certainty for the fit. We will deal with model fits in later parts of the book. For illustratio n, just enjoy a few examples here:

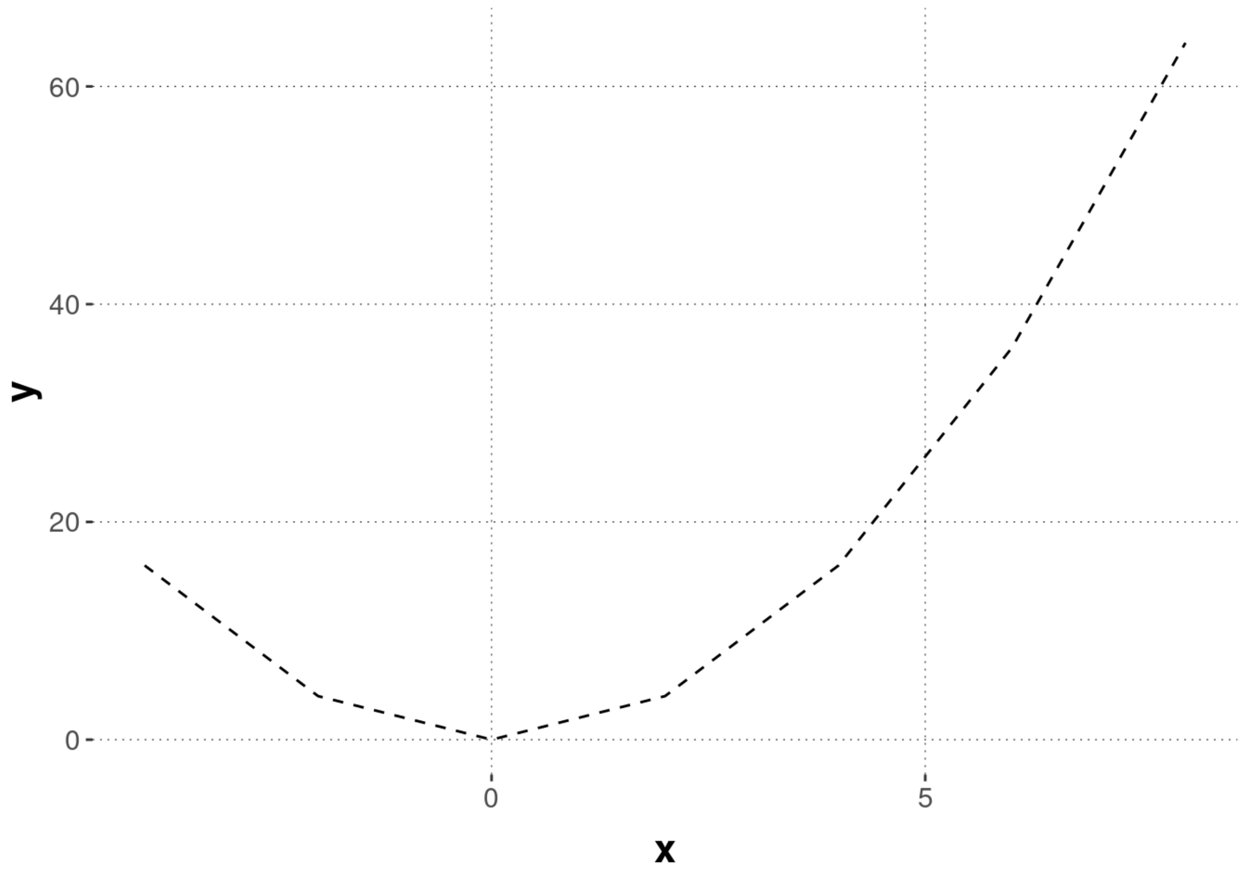
```
avocado_data %>%
  ggplot(aes(x = log(total_volume_sold), y = average_price)) +
  geom_point(
    shape = 23,
    color = "darkblue",
    fill = "lightblue",
    size = 3,
    alpha = 0.3
  ) +
  geom_smooth(
    # fitting a smoothed curve to the data
    method = "loess",
    # display standard error around smoothing curve
    se = T,
    color = "darkorange"
    )
```



Line

Use `geom_line` to display a line for your data if that data has associated (ordered) metric values. You can use argument `linetype` to specify the kind of line to draw.

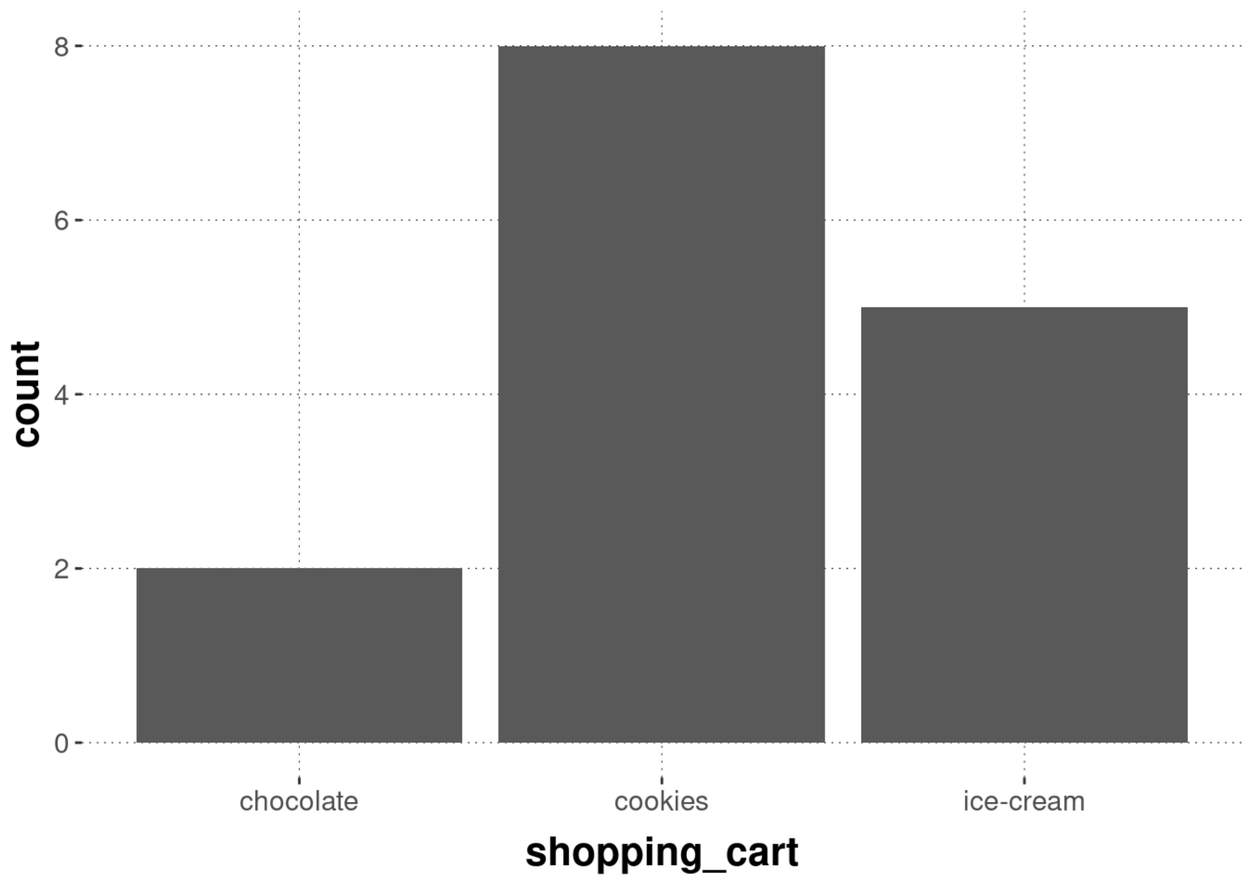
```
tibble(
  x = seq(-4, 8, by = 2),
  y = x^2
) %>%
  ggplot(aes(x, y)) +
  geom_line(
    linetype = "dashed"
  )
```



Bar plot

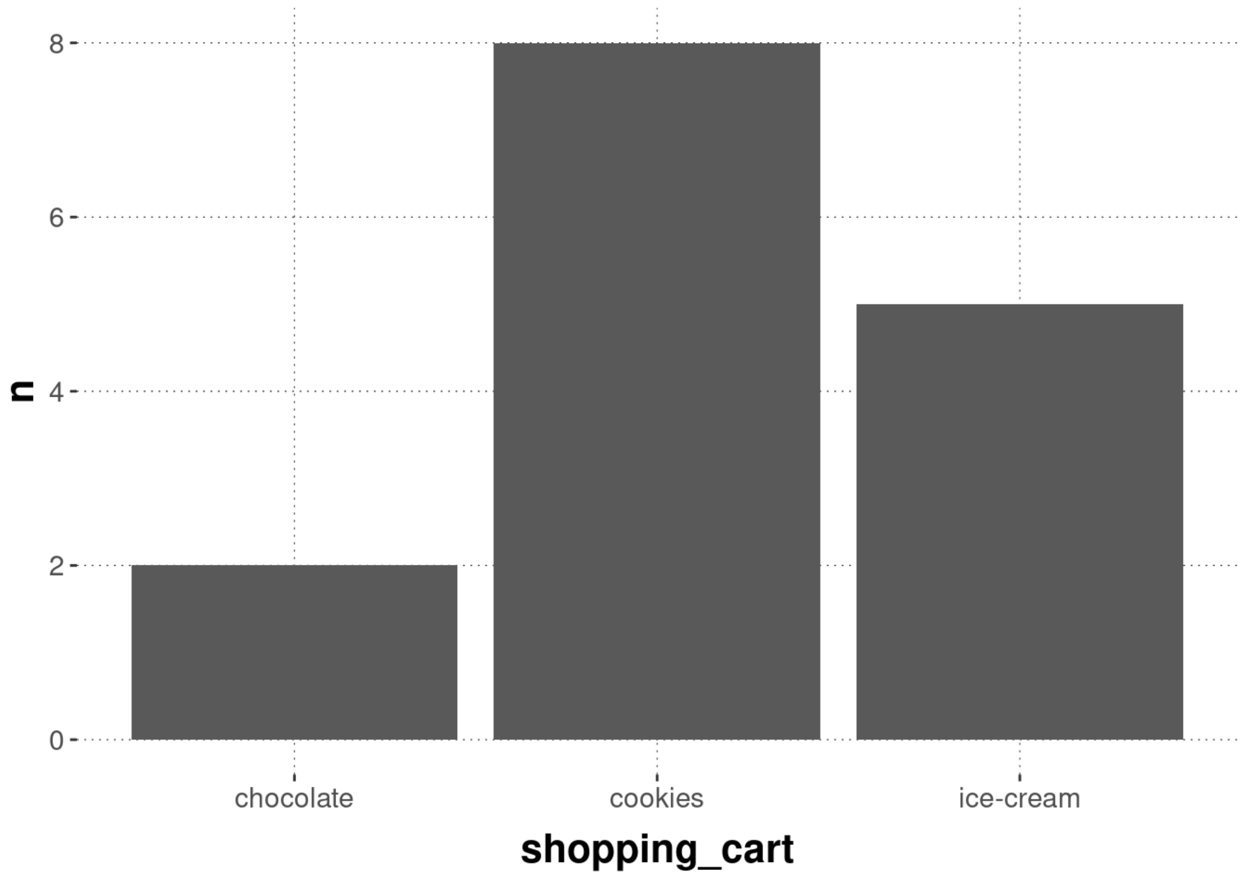
A bar plot, plotted with `geom_bar` or `geom_col`, displays a single number for each of several groups for visual comparison by length. The difference between these two functions is that `geom_bar` relies on implicit counting, while `geom_col` expects the numbers that translate into the length of the bars to be supplied for it. This book favors the use of `geom_col` by first wrangling the data to show the numbers to be visualized, since often this is the cleaner approach and the numbers are useful to have access to independently (e.g., for referring to in the text).
Here's an example of how `bar_plot` works (implicitly counting numbers of occurrences):

```
tibble(
  shopping_cart = c(
    rep("chocolate", 2),
    rep("ice-cream", 5),
    rep("cookies", 8)
  )
) %>%
  ggplot(aes(x = shopping_cart)) +
  geom_bar()
```



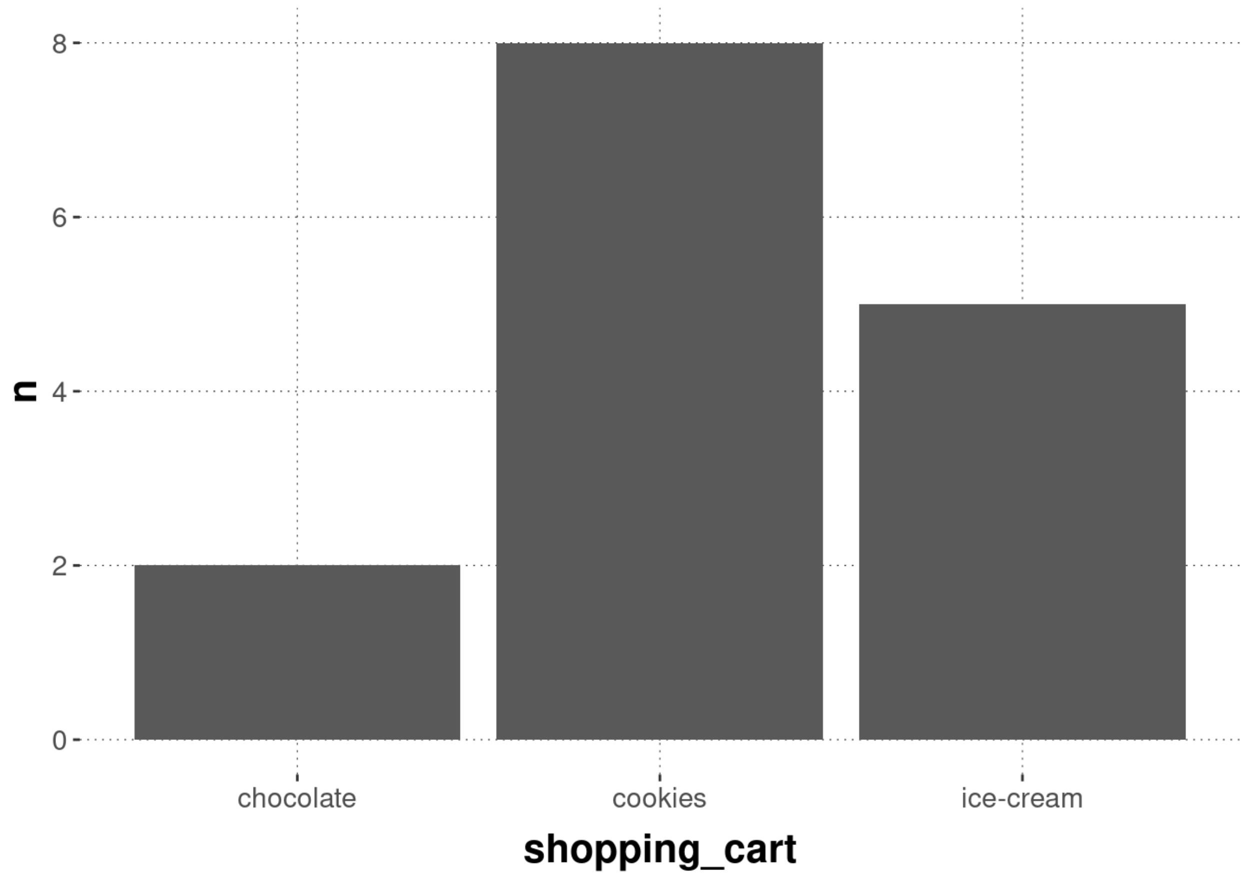
To display this data with `geom_col` we need to count occurrences first ourselves:

```
tibble(
  shopping_cart = c(
    rep("chocolate", 2),
    rep("ice-cream", 5),
    rep("cookies", 8)
  )
) %>%
  dplyr::count(shopping_cart) %>%
    ggplot(aes (x = shopping_cart, y = n)) +
    geom_col()
```



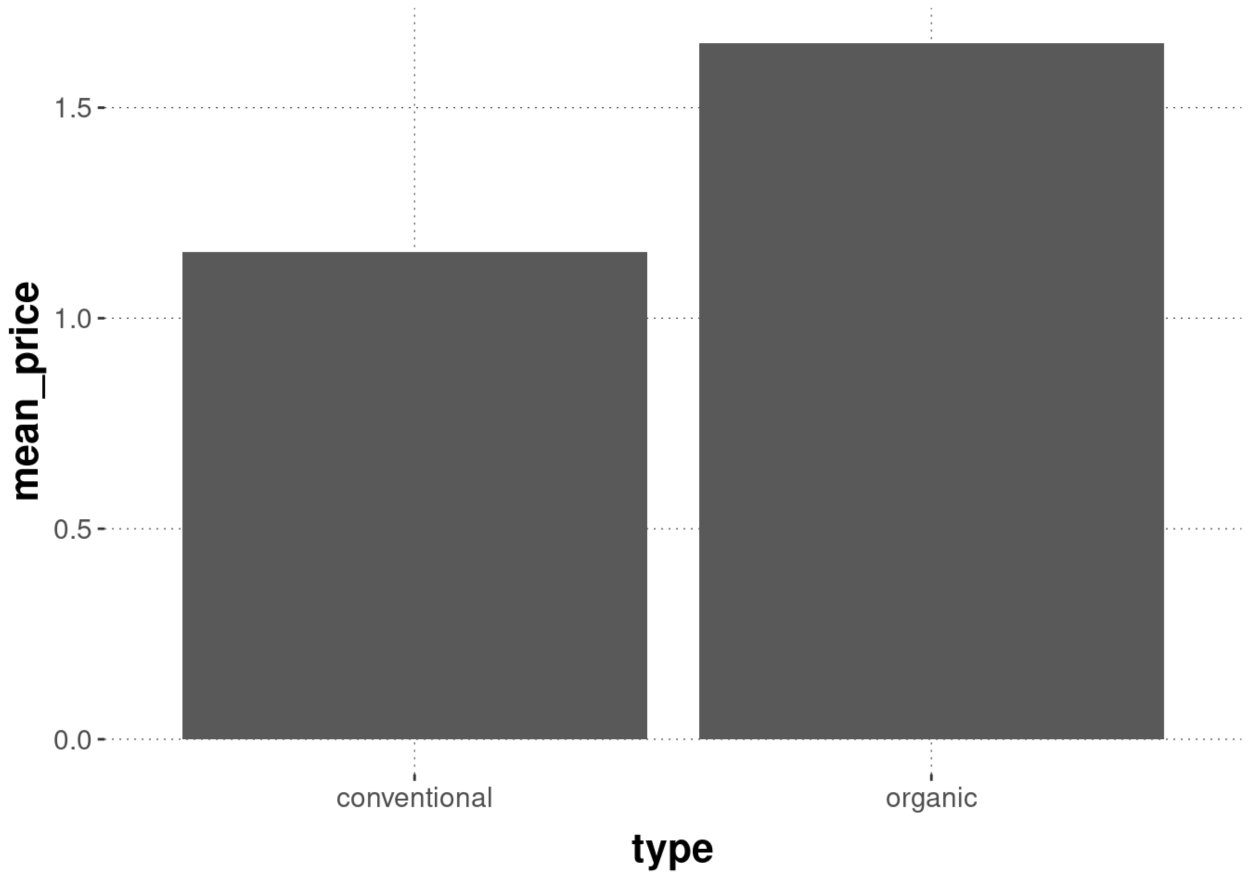
To be clear, `geom_col` is essentially `geom_bar` when we overwrite the default statistical transformation of counting to “identity”:

```
tibble(
  shopping_cart = c(
    rep("chocolate", 2),
    rep("ice-cream", 5),
    rep("cookies", 8)
  )
) %>%
  dplyr::count(shopping_cart) %>%
    ggplot(aes (x = shopping_cart, y = n)) +
    geom_bar(stat = "identity")
```



Bar plots are a frequent sight in psychology papers. They are also controversial. They often fare badly with respect to the data-ink ratio. Especially, when what is plotted are means of grouped variables. For example, the following plot is rather uninformative (even if the research question is a comparison of means):

```r
avocado_data %>%
  group_by(type) %>%
  summarise(
    mean_price = mean(average_price)
  ) %>%
  ggplot(aes(x = type, y = mean_price)) +
  geom_col()
```

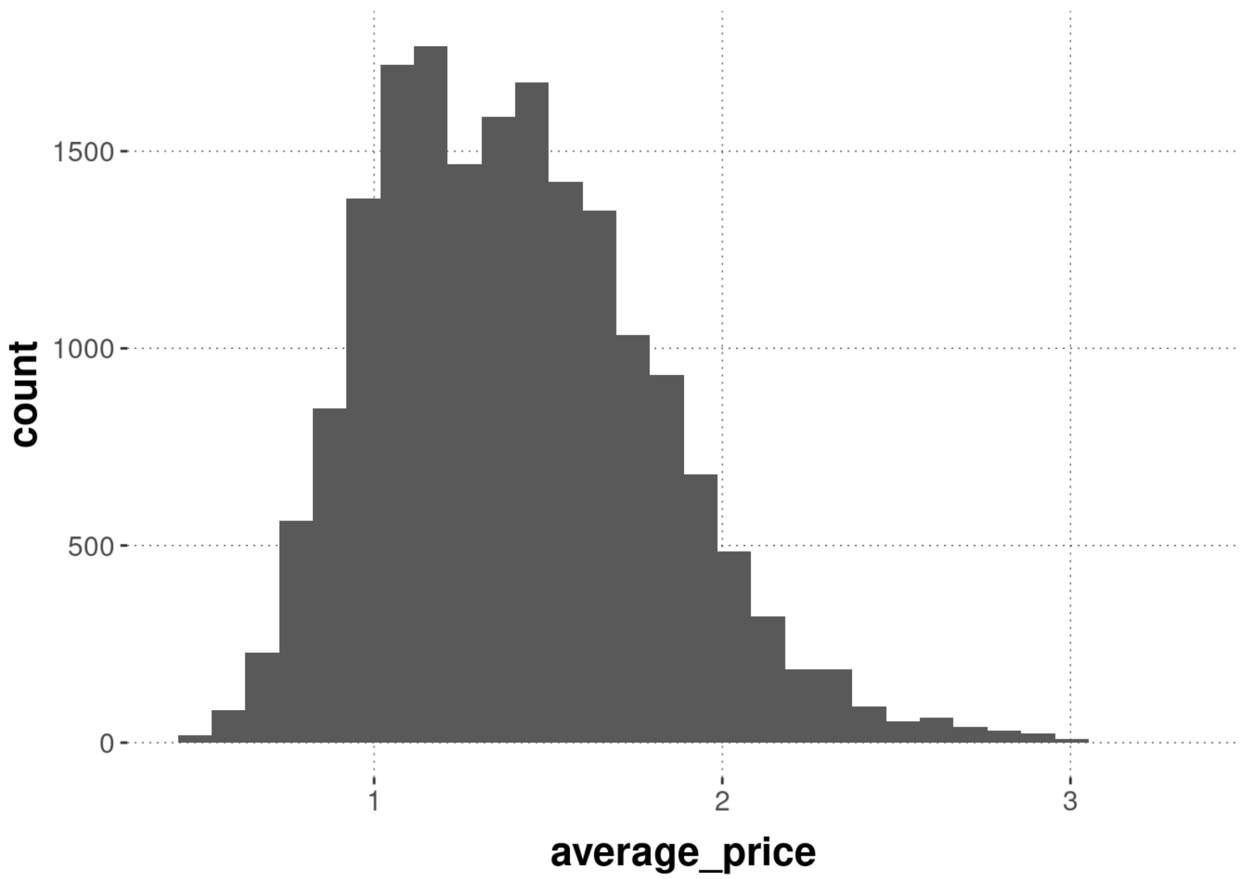


It makes sense to use the available space for a more informative report about the distribution of data points around the means, e.g., by using `geom_violin` or `geom_histogram` etc.

Plotting distributions: histograms, boxplots, densities and violins

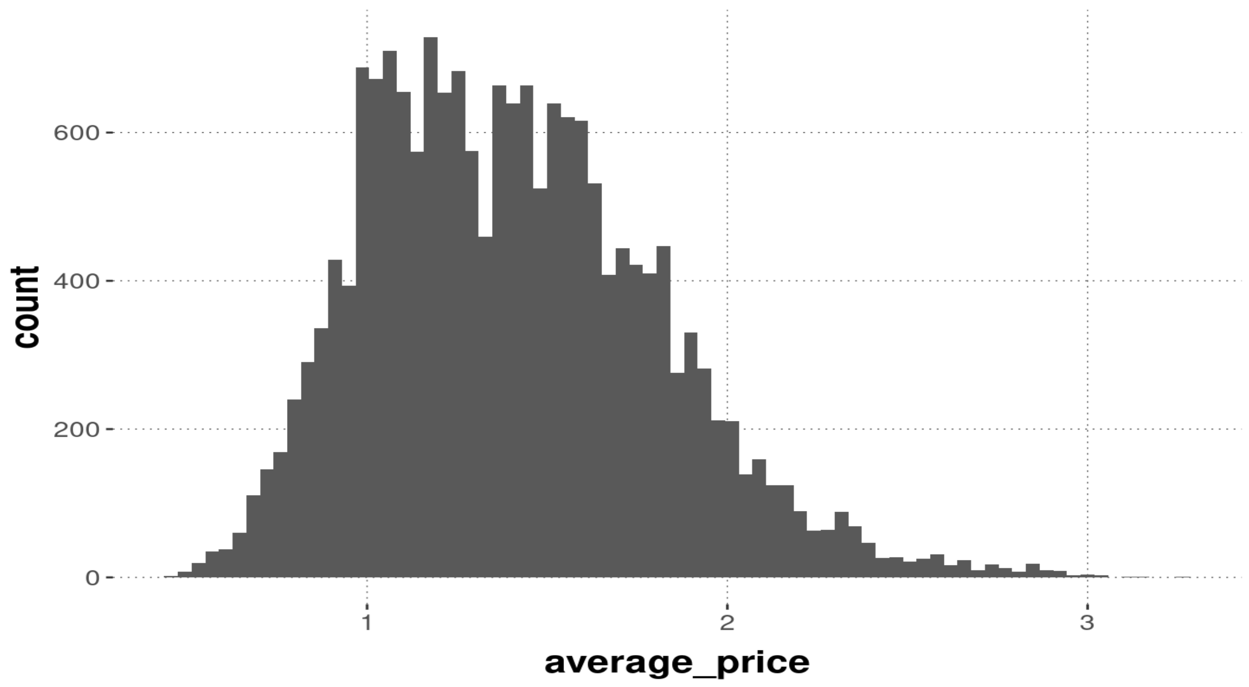
There are different ways for plotting the distribution of observations in a one-dimensional vector, each with its own advantages and disadvantages: the histogram, a box plot, a density plot, and a violin plot. Let's have a look at each, based on the `average_price` of different types of avocados.
The histogram displays the number of occurrences of observations inside of prespecified bins. By default, the function `geom_histogram` uses 30 equally spaced bins to display counts of your observations.

```
avocado_data %>%
  ggplot(aes(x = average_price)) +
  geom_histogram()
```



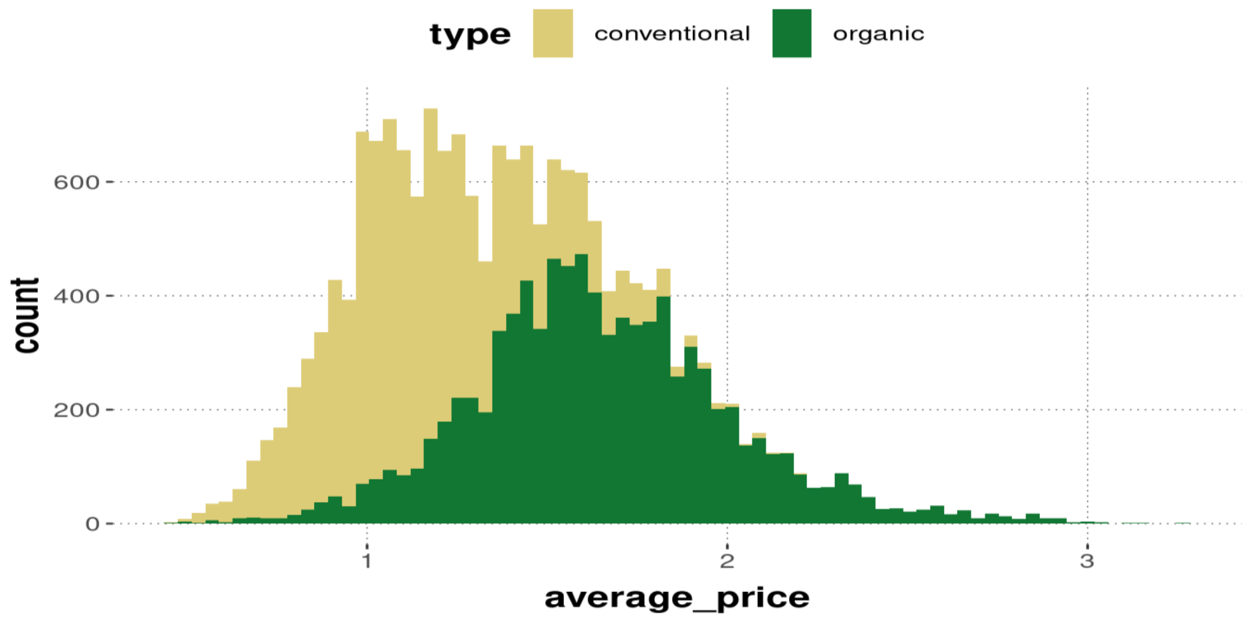
If we specify more bins, we get a more fine-grained picture. (But notice that such a high number of bins works for the present data set, which has many observations, but it would not necessarily for a small data set.)

```
avocado_data %>%
  ggplot(aes(x = average_price)) +
  geom_histogram(bins = 75)
```



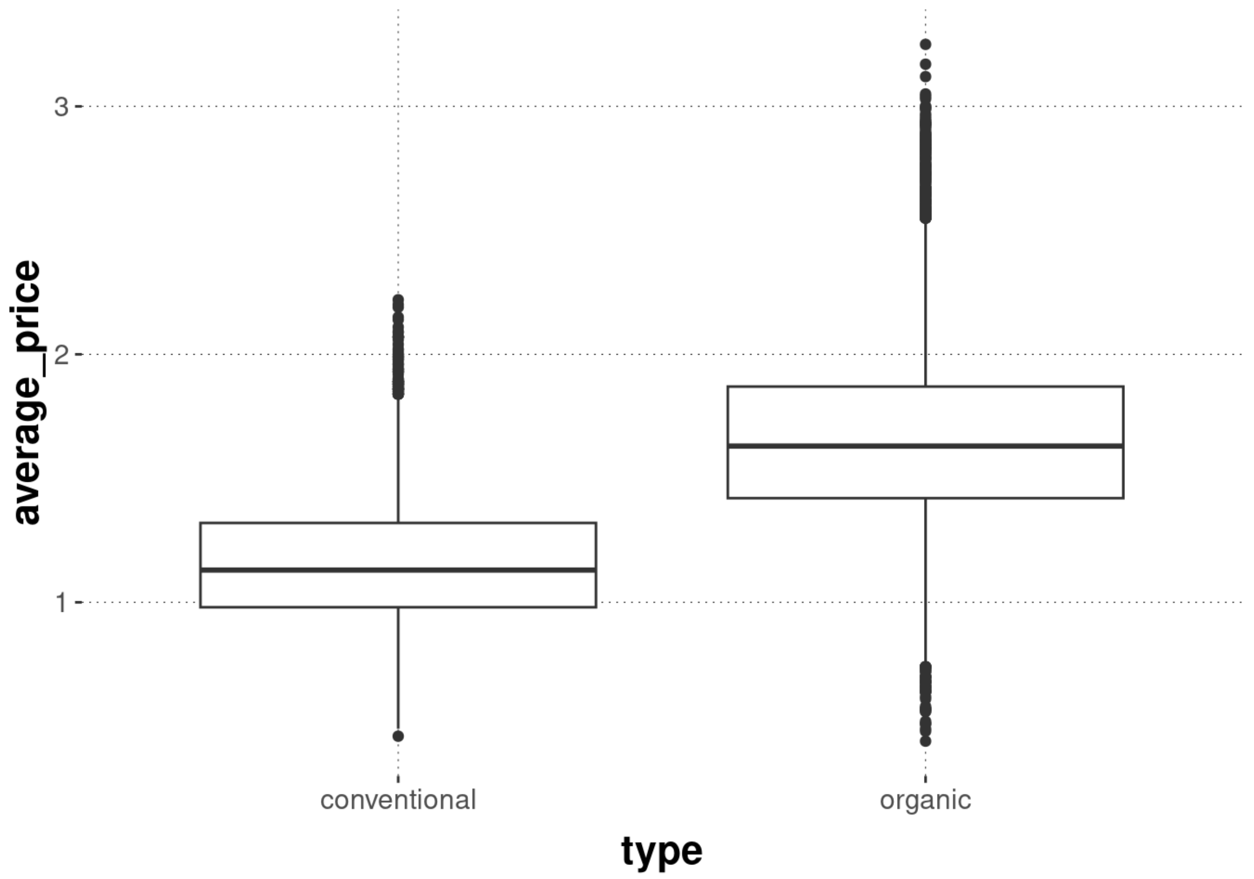
We can also layer histograms but this is usually a bad idea (even if we tinker with opacity) because a higher layer might block important information from a lower layer:

```
avocado_data %>%
  ggplot(aes(x = average_price, fill = type)) +
  geom_histogram(bins = 75)
```



An alternative display of distributional metric information is a **box plot**. Box plots are classics, also called *box-and-whiskers plots*, and they basically visually report key summary statistics of your metric data. These do work much better than histograms for direct comparison:

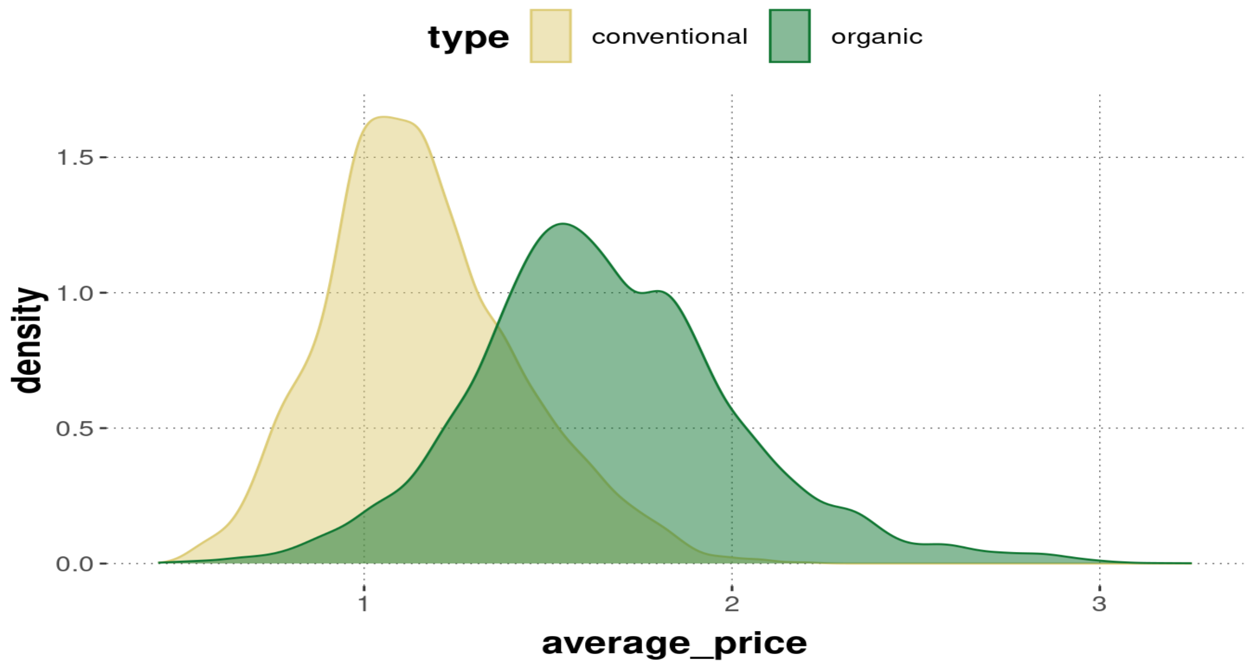
```
avocado_data %>%
  ggplot(aes(x = type, y = average_price)) +
  geom_boxplot()
```



What we see here is the median for each group (thick black line) and the 25% and 75% quantiles (boxes). The straight lines show the range from the 25% or 75% quantiles to the values given by median + 1.58 * IQR / sqrt(n), where the IQR is the “interquartile range”, i.e., the range between the 25% and 75% quantiles (boxes).

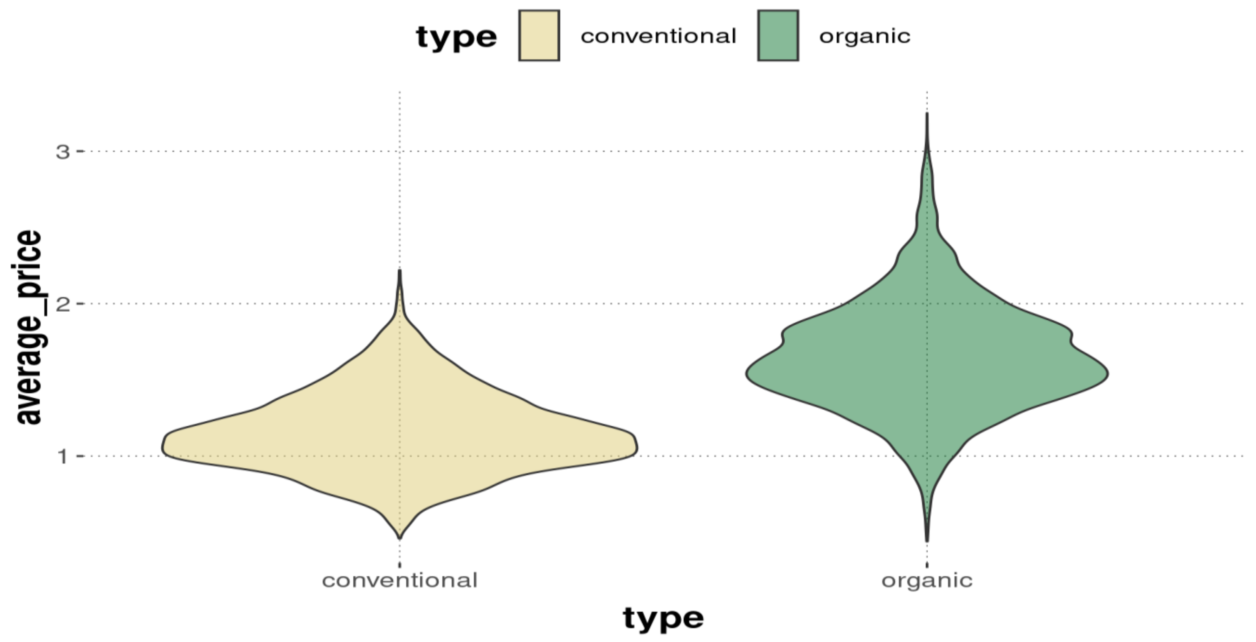
To get a better picture of the shape of the distribution, `geom_density` uses a kernel estimate to show a smoothed line, roughly indicating ranges of higher density of observations with higher numbers. Using opacity, `geom_density` is useful also for the close comparison of distributions across different groups:

```
avocado_data %>%
  ggplot(aes(x = average_price, color = type, fill = type)) +
  geom_density(alpha = 0.5)
```

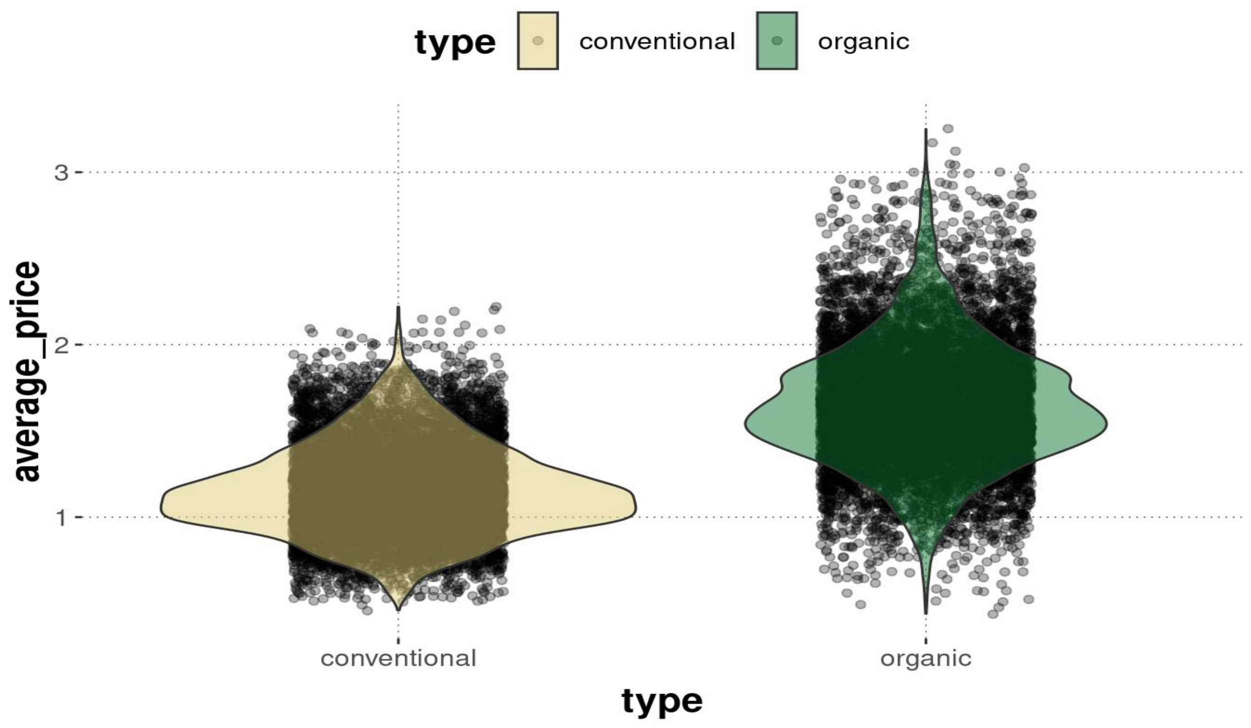


For many groups to compare, density plots can become cluttered. **Violin plots** are like mirrored density plots and are better for comparison of multiple groups:

```
avocado_data %>%
  ggplot(aes(x = type, y = average_price, fill = type)) +
  geom_violin(alpha = 0.5)
```



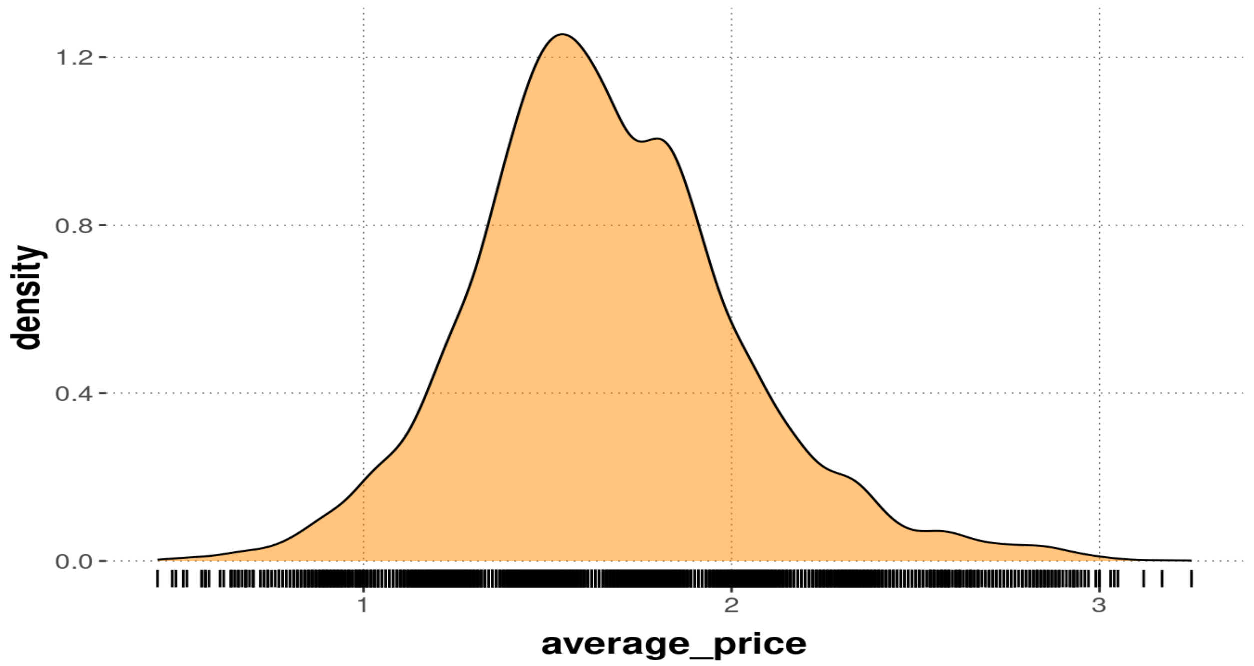
A frequently seen method of visualization is to layer a jittered distribution of points under a violin plot, like so:



Rugs

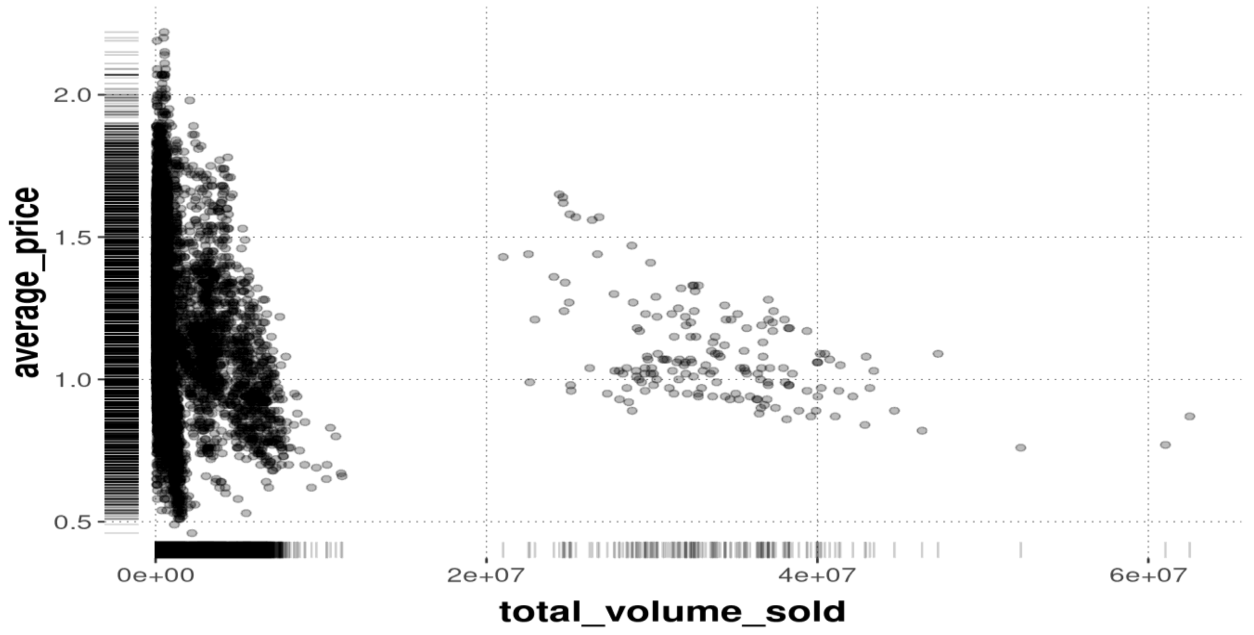
Since plotting distributions, especially with high-level abstract smoothing as in `geom_density` and `geom_violin` fails to give information about the actual quantity of the data points, rug plots are useful additions to such plots. `geom_rug` add marks along the axes where different points lie.
Here is an example of `geom_rug` combined with `geom_density`:

```
avocado_data %>%
  filter(type == "organic") %>%
  ggplot(aes(x = average_price)) +
  geom_density(fill = "darkorange", alpha = 0.5) +
  geom_rug()
```



Here are rugs on a two-dimensional scatter plot:

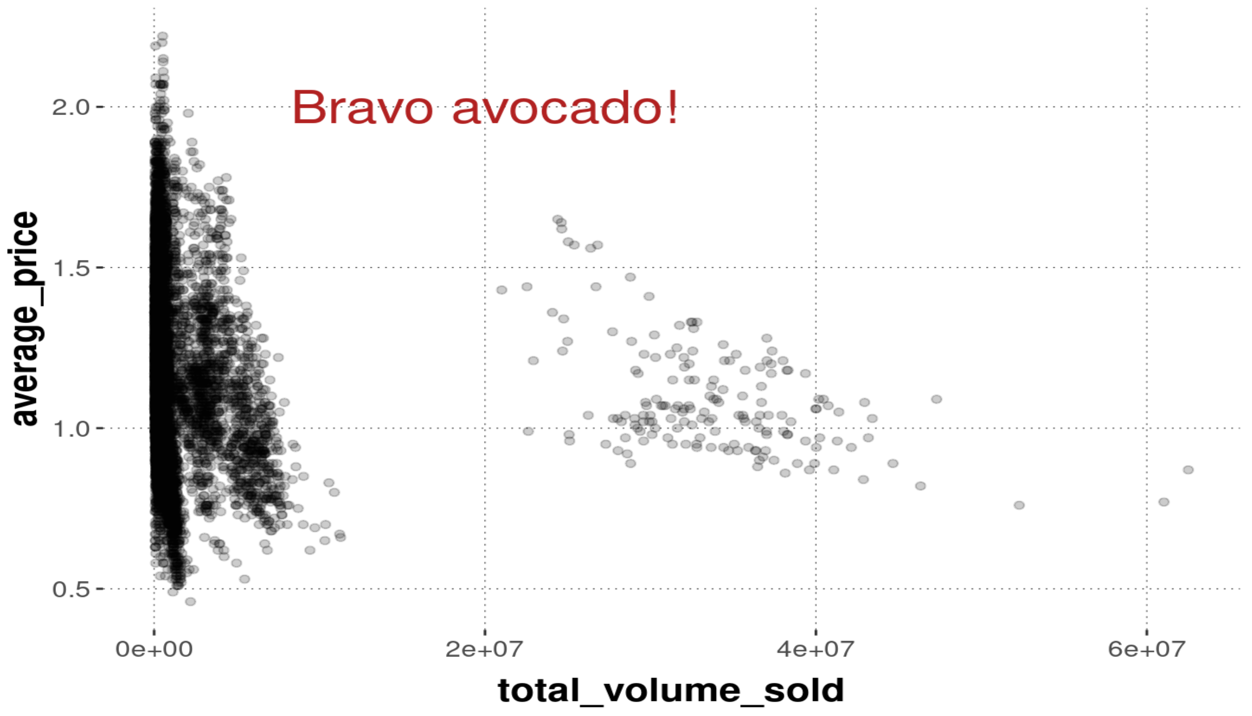
```
avocado_data %>%
  filter(type == "conventional") %>%
  ggplot(aes(x = total_volume_sold, y = average_price)) +
  geom_point(alpha = 0.3) +
  geom_rug(alpha = 0.2)
```



Annotation

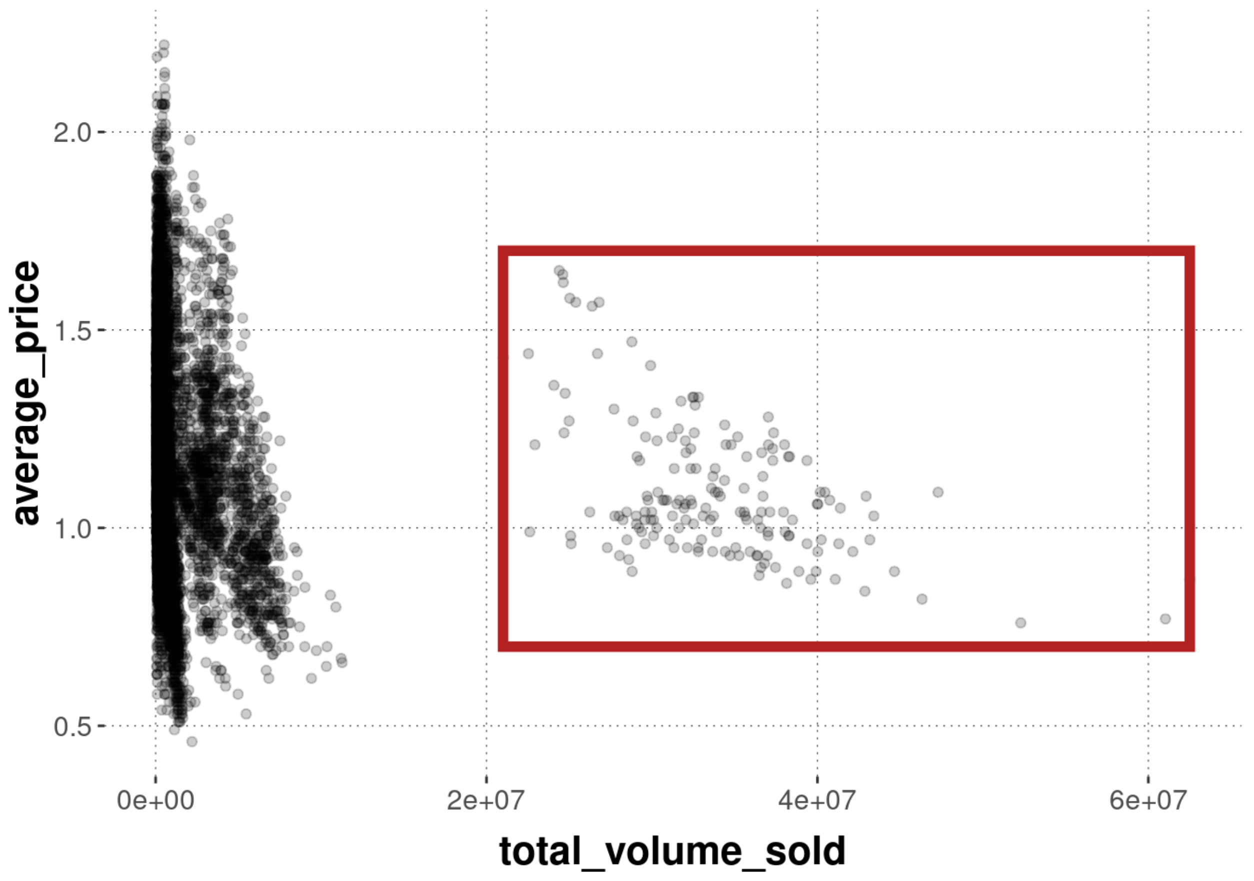
It can be useful to add further elements to a plot. We might want to add text, or specific geometrical shapes to highlight aspects of data. The most general function for doing this is `annotate`. The function `annotate` takes as a first argument a `geom` argument, e.g., `text` or `rectangle`. It is therefore not a wrapper function in the `geom_` family of functions, but the underlying function around which convenience functions like `geom_text` or `geom_rectangle` is wrapped. The further arguments that `annotate` expects depend on the geom it is supposed to realize.
Suppose we want to add textual information at a particular coordinate. We can do this with `annotate` as follows:

```
avocado_data %>%
  filter(type == "conventional") %>%
  ggplot(aes(x = total_volume_sold, y = average_price)) +
  geom_point(alpha = 0.2) +
  annotate(
    geom = "text",
    # x and y coordinates for the text
    x = 2e7,
    y = 2,
    # text to be displayed
    label = "Bravo avocado!",
    color = "firebrick",
    size = 8
  )
```



We can also single out some data points, like so:

```r
avocado_data %>%
  filter(type == "conventional") %>%
  ggplot(aes(x = total_volume_sold, y = average_price)) +
  geom_point(alpha = 0.2) +
  annotate(
    geom = "rect",
    # coordinates for the rectangle
    xmin = 2.1e7,
    xmax = max(avocado_data$total_volume_sold) + 100,
    ymin = 0.7,
    ymax = 1.7,
    color = "firebrick",
    alpha = 0,
    size = 2
  )
```



<u>What to submit</u>:

Your submission should include the following:

1. Lab report answers all exercises above and source code.
2. Please create a folder called "yourname_studentID_lab6" that includes all the required files and generate a zip file called "yourname_studentID_lab6.zip".
3. Please submit your work (.zip) to Blackboard.