Visualization for communication

Contents

Lecture learning goals

By the end of the lecture you will be able to:

1. Follow guidelines for best practices in visualization design.
2. Avoid overplotting via 2D distribution plots.
3. Adjust axes extents and formatting.
4. Modify titles of several figure elements.
5. Visualize trends using regression and loess lines.

Required activities

Before class:

Nothing

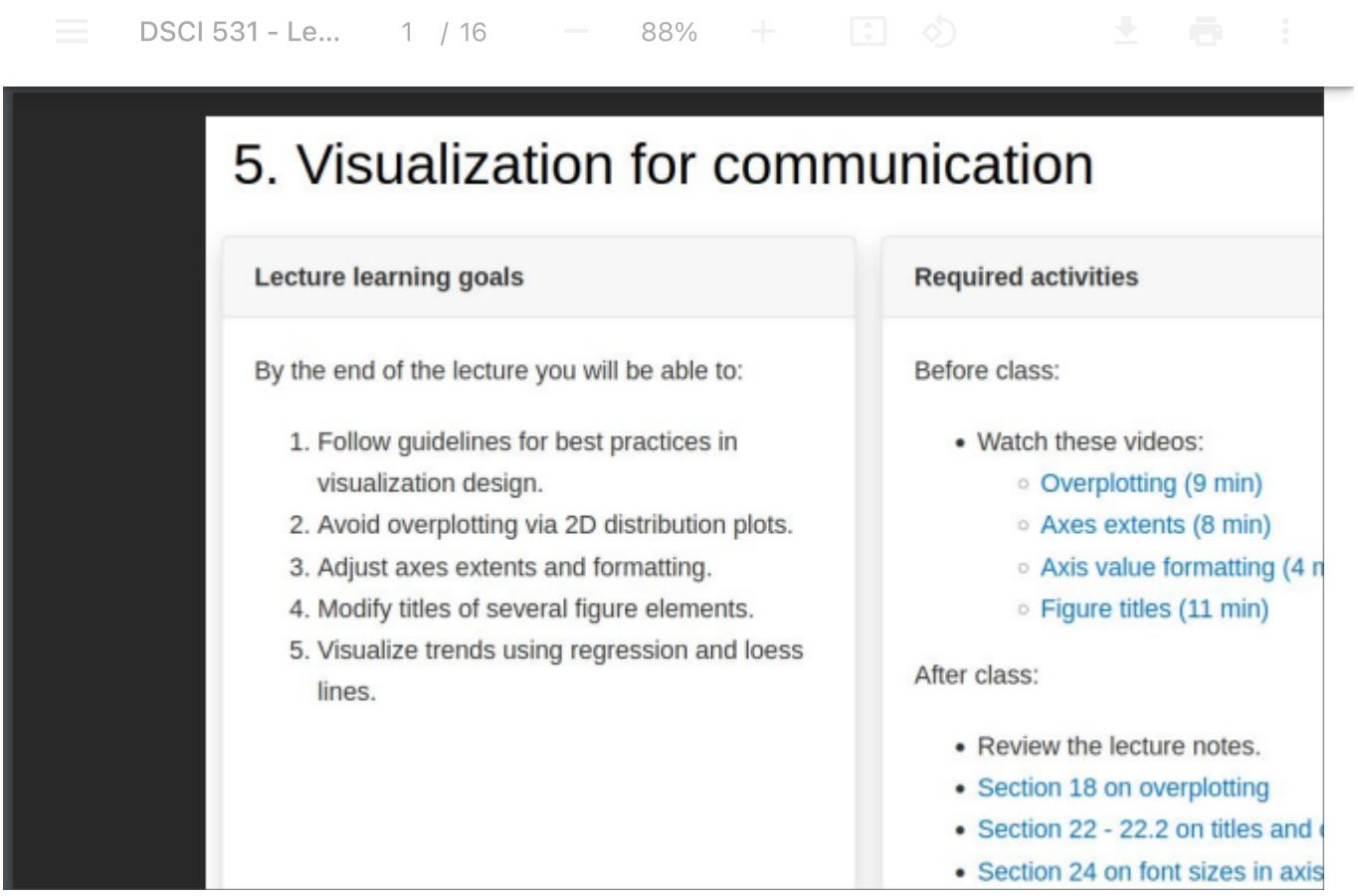
After class:

- Review the lecture notes.
  - This includes wathcing these videos of me walking through the lecture notes and describing them in more detail:
    - Overplotting (9 min)
    - Axes extents (8 min)
    - Axis value formatting (4 min)
    - Figure titles (11 min)

Skip to main content

- Section 22 - 22.2 on titles and captions
- Section 24 on font sizes in axis labels
- Section 14 - 14.2 on visualizing trends.

Lecture slides




```
import altair as alt

# Simplify working with large datasets in Altair
alt.data_transformers.enable('vegafusion')

# Load the R cell magic
%load_ext rpy2.ipython
```

The schematics of many of the guidelines shown today were from the Cato institutes

Skip to main content

this spreadsheet. The data visualization society has more interesting resources on style guidelines.

Another good way to learn is from other people's mistakes. Here the economist criticizes their own plots, not all of them are related to concepts we teach in class, but still worthwhile consideration to keep in mind when visualizing your data.

It is important to remember that many of these are guidelines and there are times when you can break such guidelines, e.g. using different colors than the good defaults if they already have a pre-association such as for fruits, or political parties. This article discusses a few more cases where breaking a guideline worked well, although it is not always the case that they gained a lot (such as the 3D ice cubes), at least breaking the guideline without ruining the viz, and having something different that stands out could be important to make a visualization more memorable.

5.1. Overplotting

5.1.1. Py

```
%%R -o diamonds
# Copy diamonds df from R to Python
library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.2 -
```

```
✔ ggplot2 3.4.2     ✔ purrr   1.0.1
✔ tibble  3.2.1     ✔ dplyr   1.1.2
✔ tidyr   1.3.0     ✔ stringr 1.5.0
✔ readr   2.1.4     ✔ forcats 1.0.0
```

```
── Conflicts ─────────────────────────────────────── tidyverse_conflicts() -
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

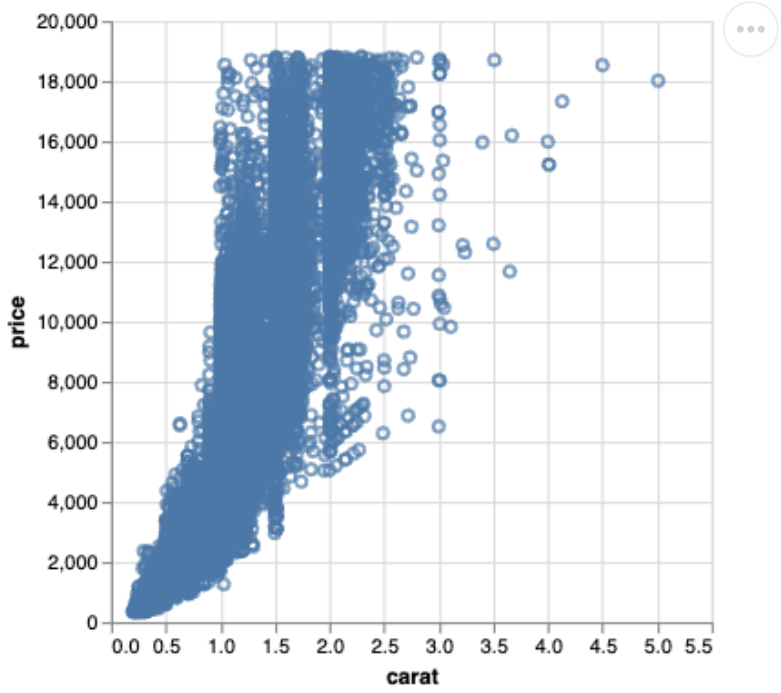
Skip to main content

Plotting all the points in this df (there are around 50,000!) takes a little bit of time and causes the plot to become saturated so that we can't see individual observations.

```
diamonds.shape
```

```
(53940, 10)
```

```
alt.Chart(diamonds).mark_point().encode(
    alt.X('carat'),
    alt.Y('price'))
```

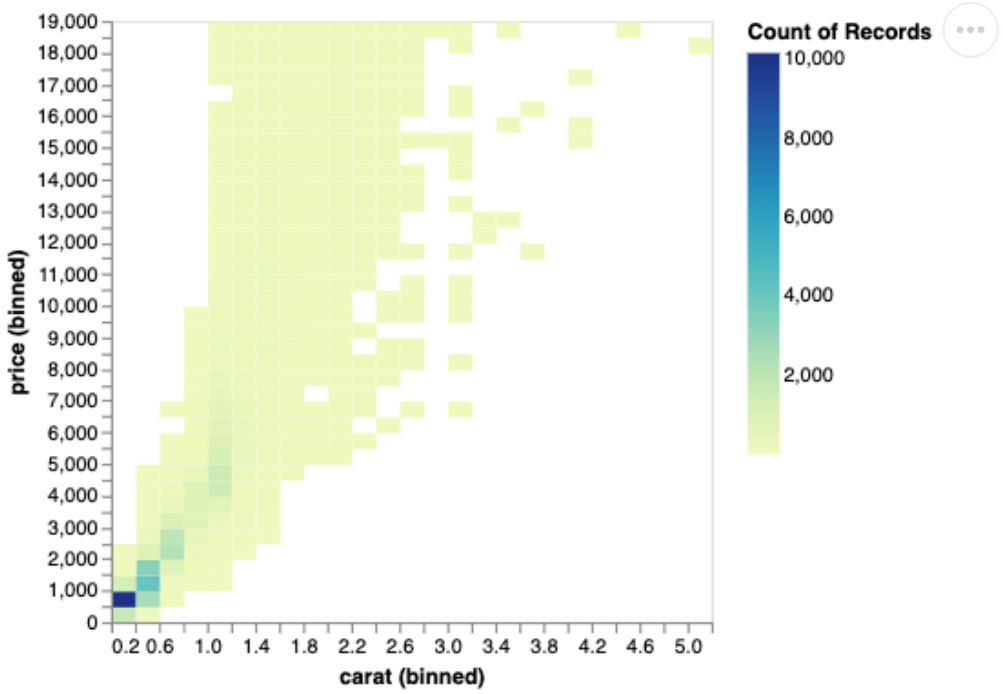


Reducing marker size and increasing opacity only helps somewhat, there are still many overplotted areas in the chart.

A better approach in this case is to create a 2D histogram, where both the x and y-axes are binned which creates a binned mesh/net over the chart area and the number of observations are counted in each bin. Just like a histogram, but the bins are in 2D instead of 1D. A 2D histogram is a type of heatmap, where count is mapped to color, you could also have used a mark that maps size to color, which might even be more effective but that is not as commonly seen.

Skip to main content

```
    alt.Y('price').bin(maxbins=40),
    alt.Color('count()')
)
```



Here we can clearer see that a small area is much more dense than the others, although they looked similar in the saturated plot. How can we zoom into this area?

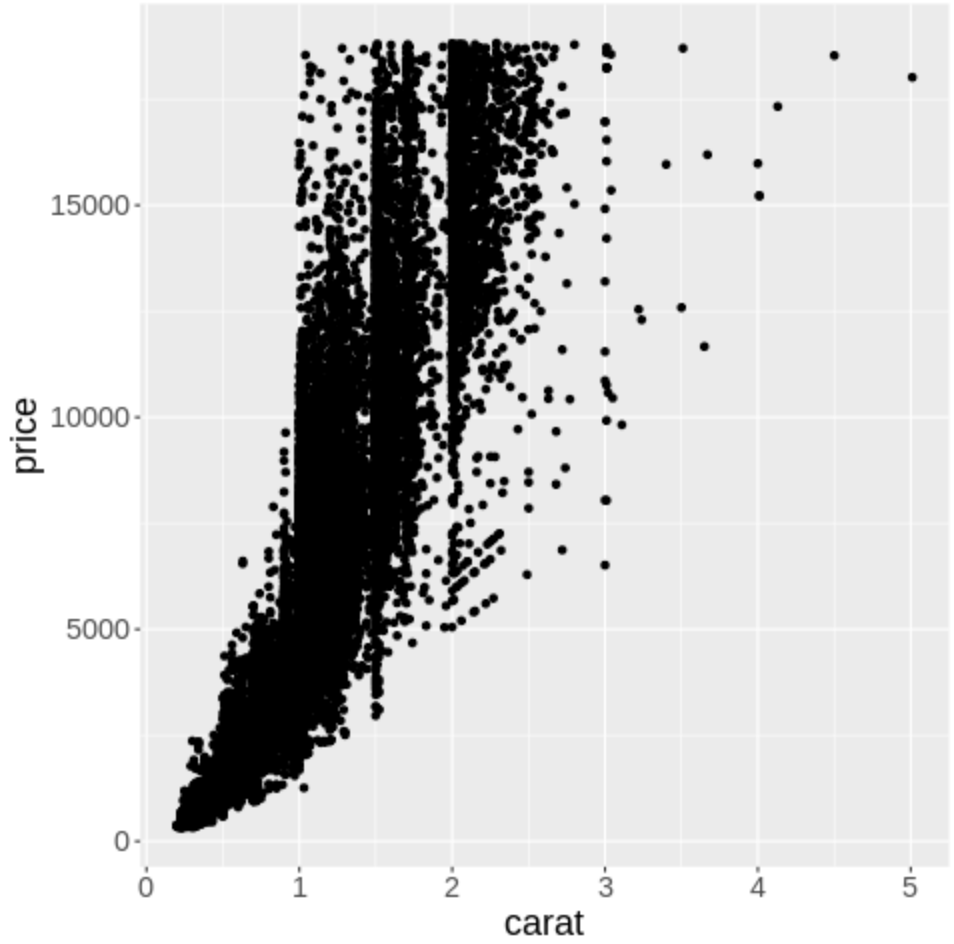
5.1.2.

In ggplot, there are more options for 2D distribution plots.

```
%%R
library(tidyverse)
theme_set(theme(text = element_text(size = 18)))

ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_point()
```

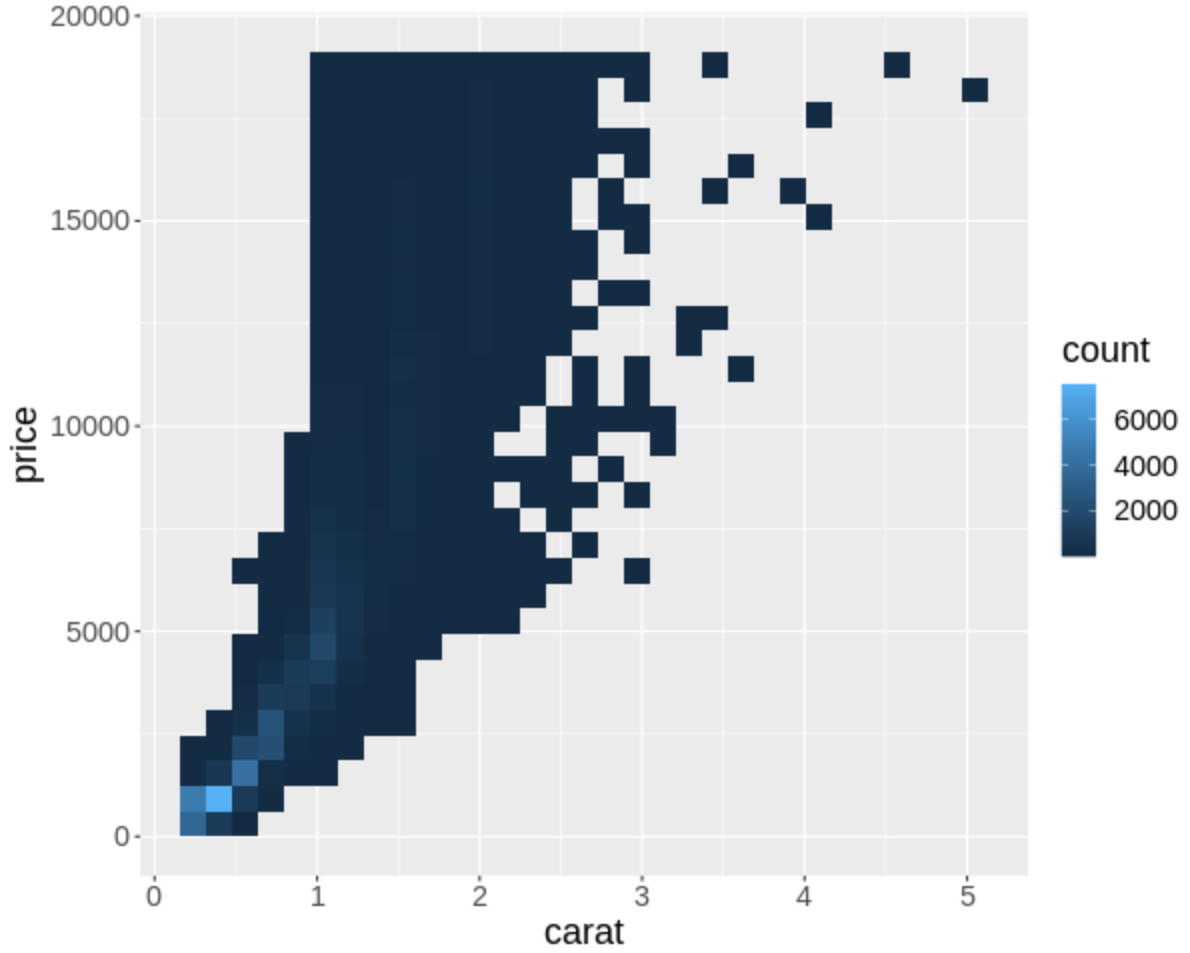
Skip to main content



As with `geom_histogram`, the binning is done by the geom, without explicitly changing the axis like in Altair.

```
%%R -w 600
library(tidyverse)
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_bin2d()
```

Skip to main content

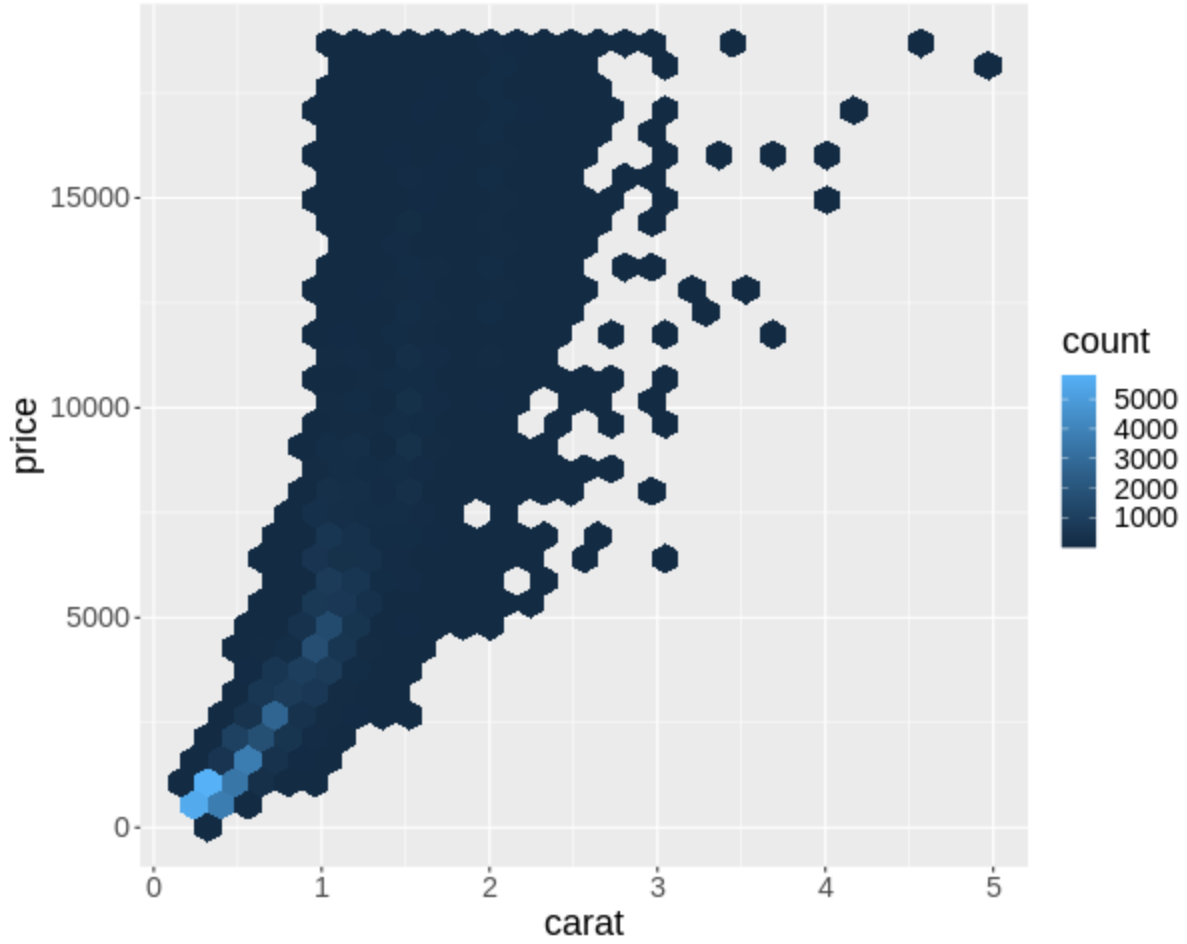


Instead of squares, hexagonal bins can be used. These have theoretically superior qualities over squares, such as a more natural notation of neighbors (1 step any direction instead of diagonal versus orthoganol neighbors), and a more circular shape ensures that data points that contribute to the count of a hexagonal bin, are not far away from the center in a corner as it could be in a square.

```
%%R -w 600
library(tidyverse)

# You need to install the package `hexbin` to run this code
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex()
```

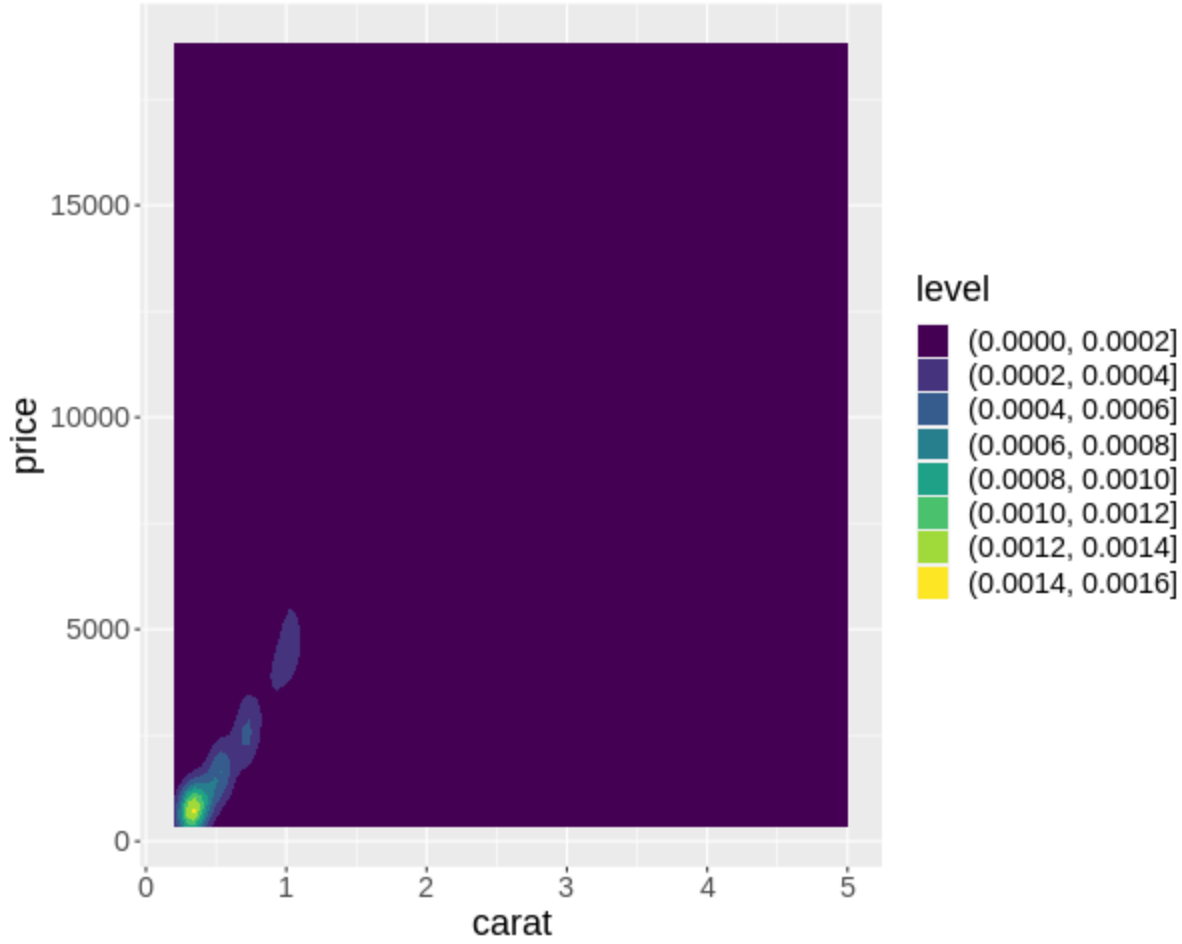
Skip to main content



We can also create 2 dimensional KDEs in ggplot. This works just like 1D KDEs, except that the kernel on each data point extends in 2 dimensions (so it looks a bit like a tent)

```
%%R -w 600
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_density_2d_filled()
```

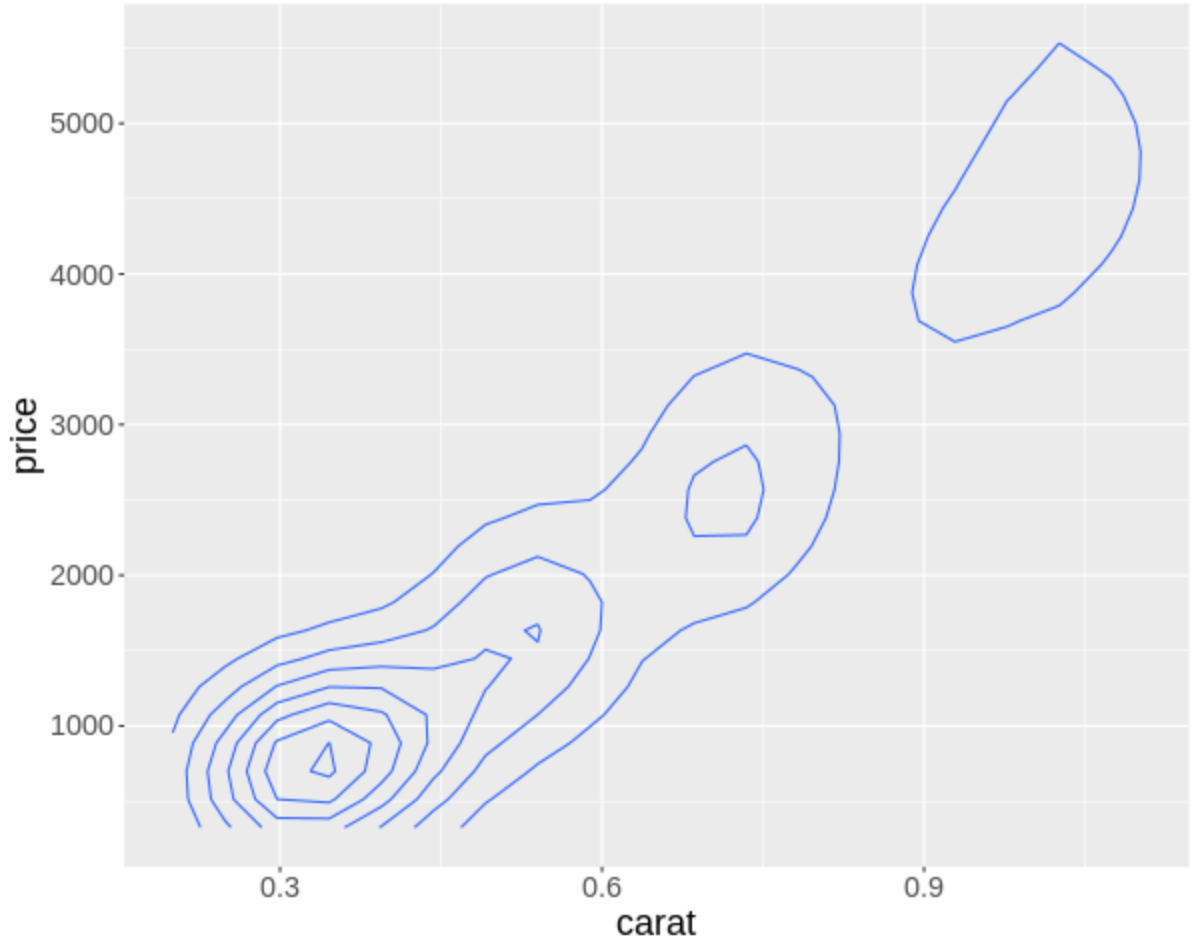
Skip to main content



In addition to indicate the density with color, we could also use ridges/contours, similar to a topographic map. This is akin to looking at a mountain range from above, so small circles indicate sharp peaks. This plot does not include a colorbar and it zooms in to the range of the contours, whether the plot above covers the full range of the data. These contour plots are often less intuitive than the density plot above, so the recommendation is to use the density plot instead.

```
%%R -w 600
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_density_2d()
```

Skip to main content



5.2. Axis label formatting

5.2.1. Py

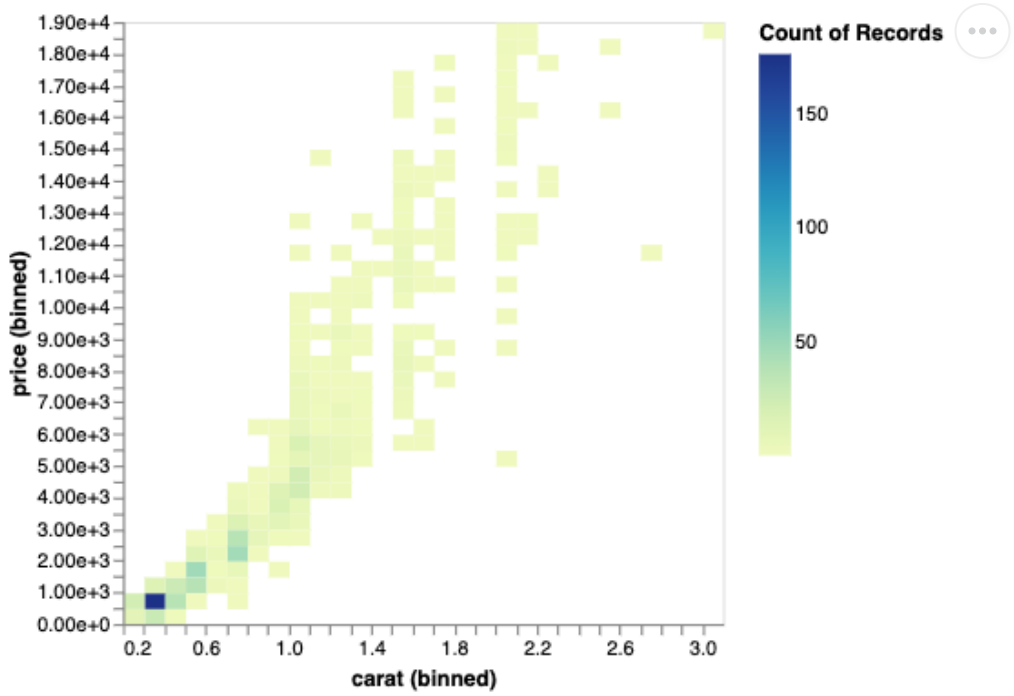
All labels formats can be found here. Notable ones include `%`, `$`, `e`, `s`.

```
# Remove the bins and take a sample of the code make the code clearer
diamonds = diamonds.sample(1000, random_state=1010)
```

Scientific notation (`10^` or `e+`) can be useful internally, but can be confusing for communicating to a more general audience.

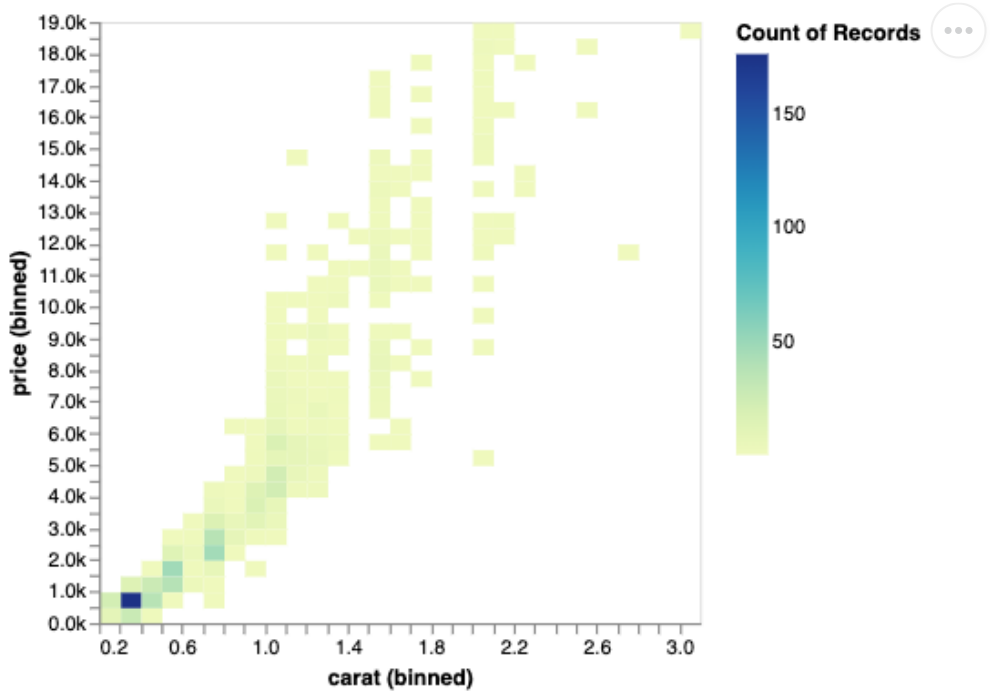
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='e'),
    alt.Color('count()')
```

Skip to main content



Standard international (SI) units are often easier to digest.

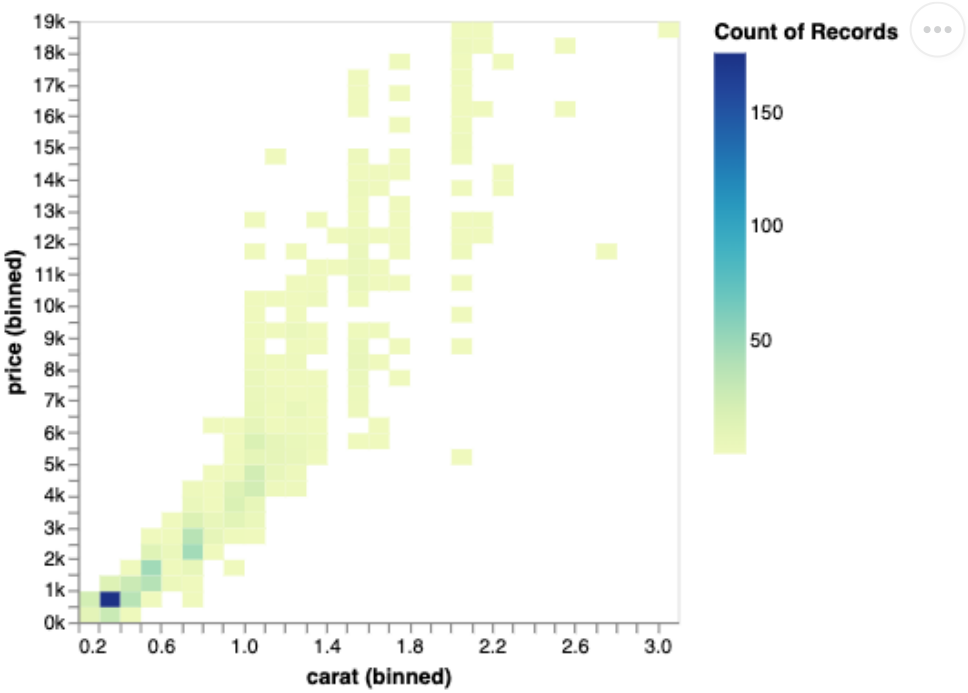
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='s'),
    alt.Color('count()')
)
```



A prefaced ~ removes trailing zeros

Skip to main content

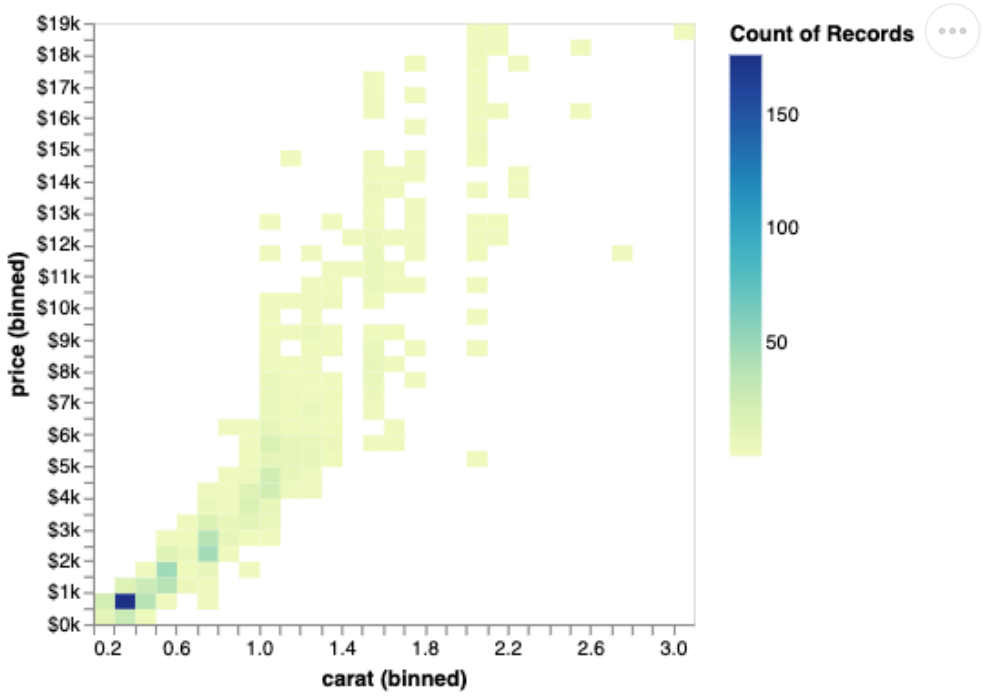
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='~s'),
    alt.Color('count()')
)
```



Formaters can also be combined.

```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='$~s'),
    alt.Color('count()')
)
```

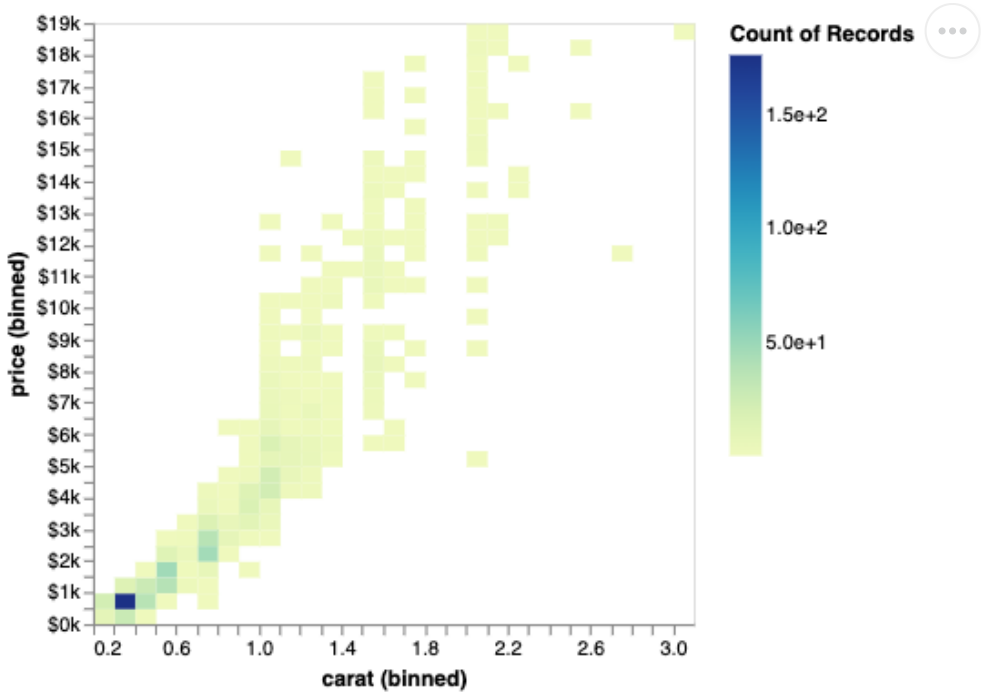
Skip to main content



The same format keys can be used for the legend. Here we use it with an to rewrite the legend values in their exponential form. This is generally something we want to avoid if we can fit the standard for of the number instead (as in the previous plot) and just used here as an example.

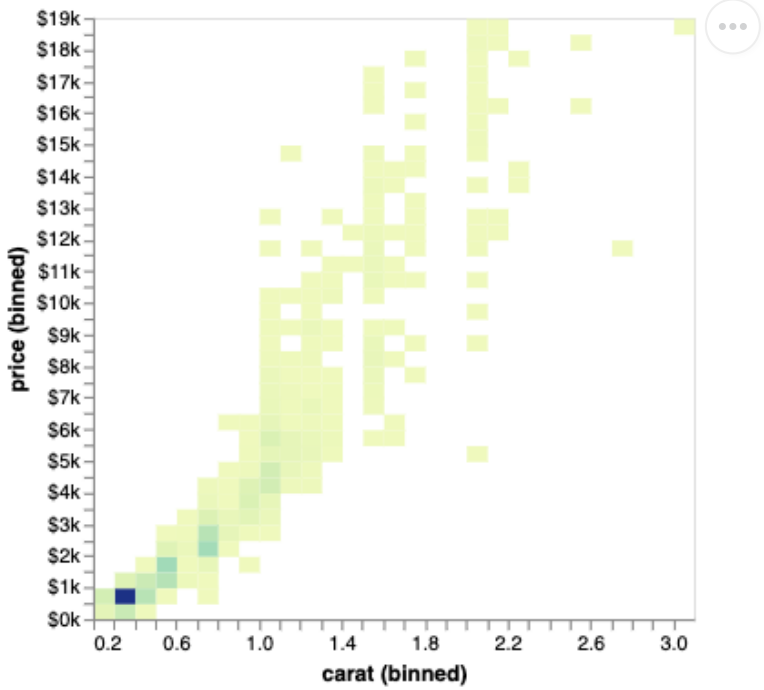
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='$~s'),
    alt.Color('count()').legend(format='e')
)
```

Skip to main content



You can remove a legend by setting it to `None`.

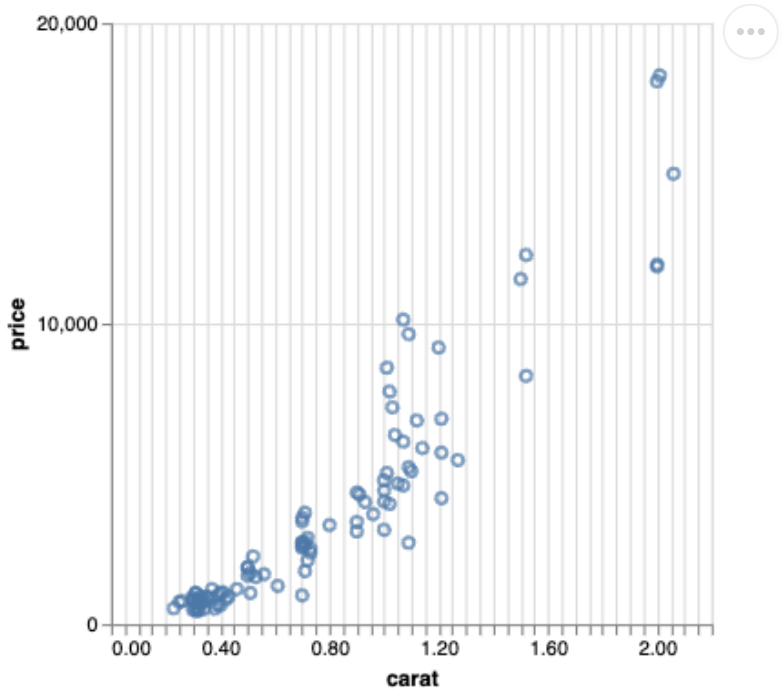
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40),
    alt.Y('price').bin(maxbins=40).axis(format='$~s'),
    alt.Color('count()').legend(None)
)
```



The number of ticks can be modified via `tickCount` but not for binned data so we are

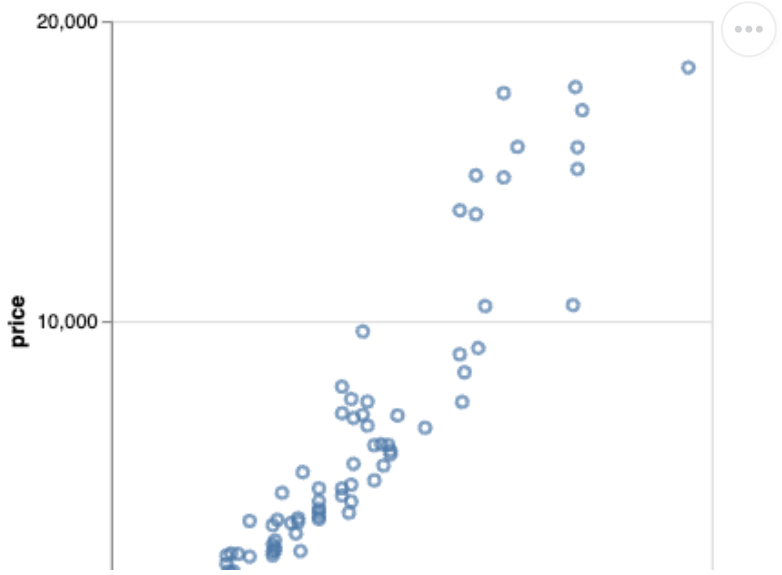
Skip to main content

```
alt.Chart(diamonds.sample(100)).mark_point().encode(
    alt.X('carat').axis(tickCount=40),
    alt.Y('price').axis(tickCount=2)
)
```



You can also remove an axis altogether by setting it to `None` as for the legend.

```
alt.Chart(diamonds.sample(100)).mark_point().encode(
    alt.X('carat').axis(None),
    alt.Y('price').axis(tickCount=2)
)
```



Skip to main content

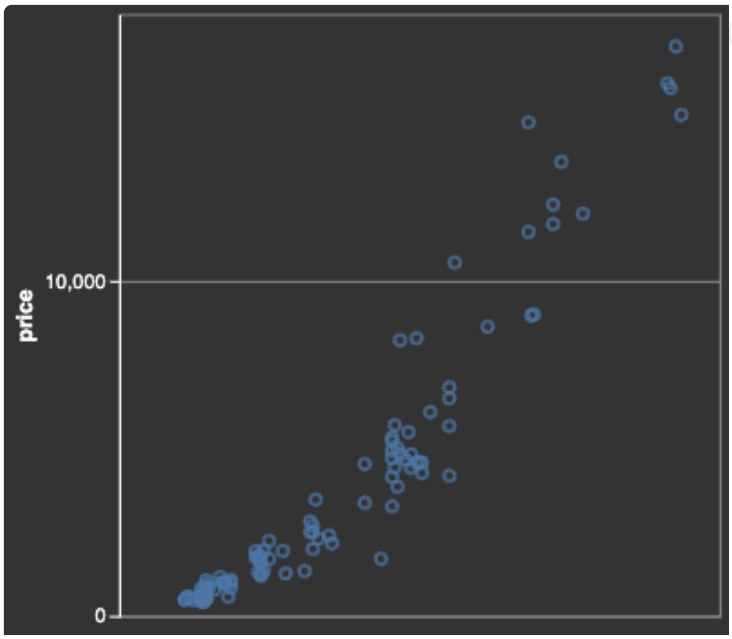
To style the chart more, we could apply a different theme. Themes are set globally, not just for one chart.

```
# Show all themes
alt.themes
```

```
ThemeRegistry(active='default', registered=['dark', 'default', 'excel', 'fivet
```

```
# This might not show properly in the course page but will work in your notebo
alt.themes.enable('dark')

alt.Chart(diamonds.sample(100)).mark_point().encode(
    alt.X('carat').axis(None),
    alt.Y('price').axis(tickCount=2)
)
```



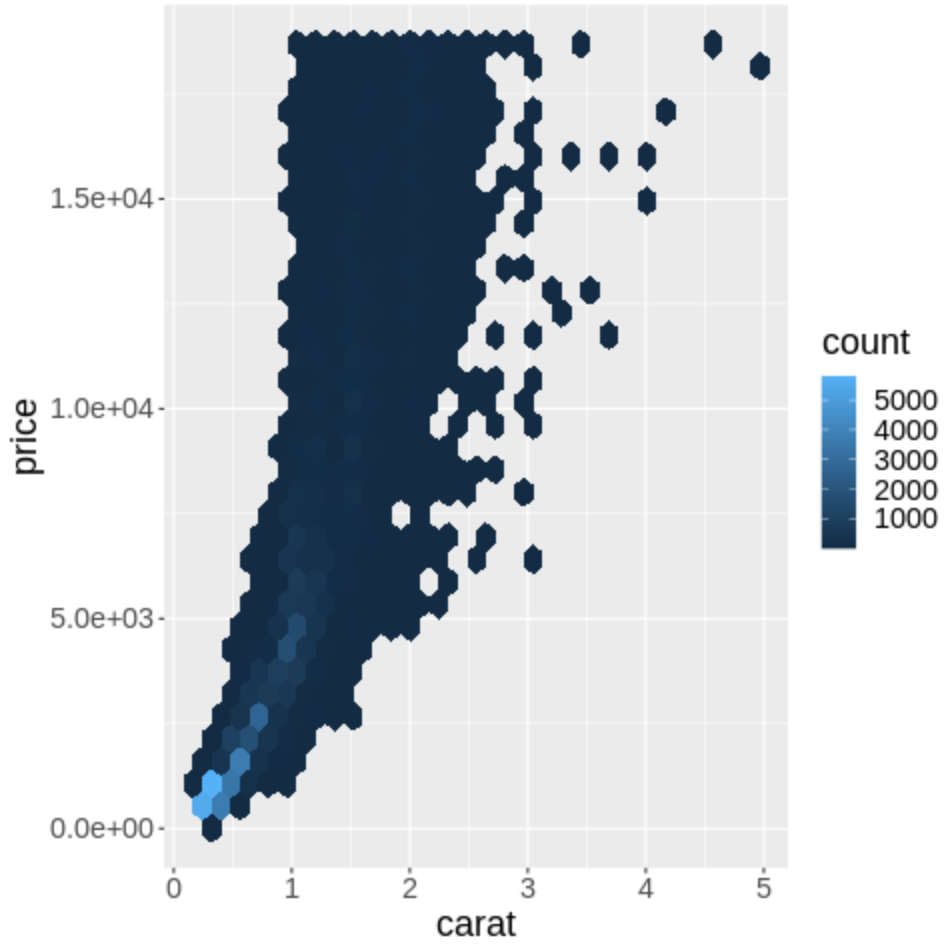
5.2.2. R

The `scales` package helps with the formatting in ggplot.

```
%%R
ggplot(diamonds) +
```

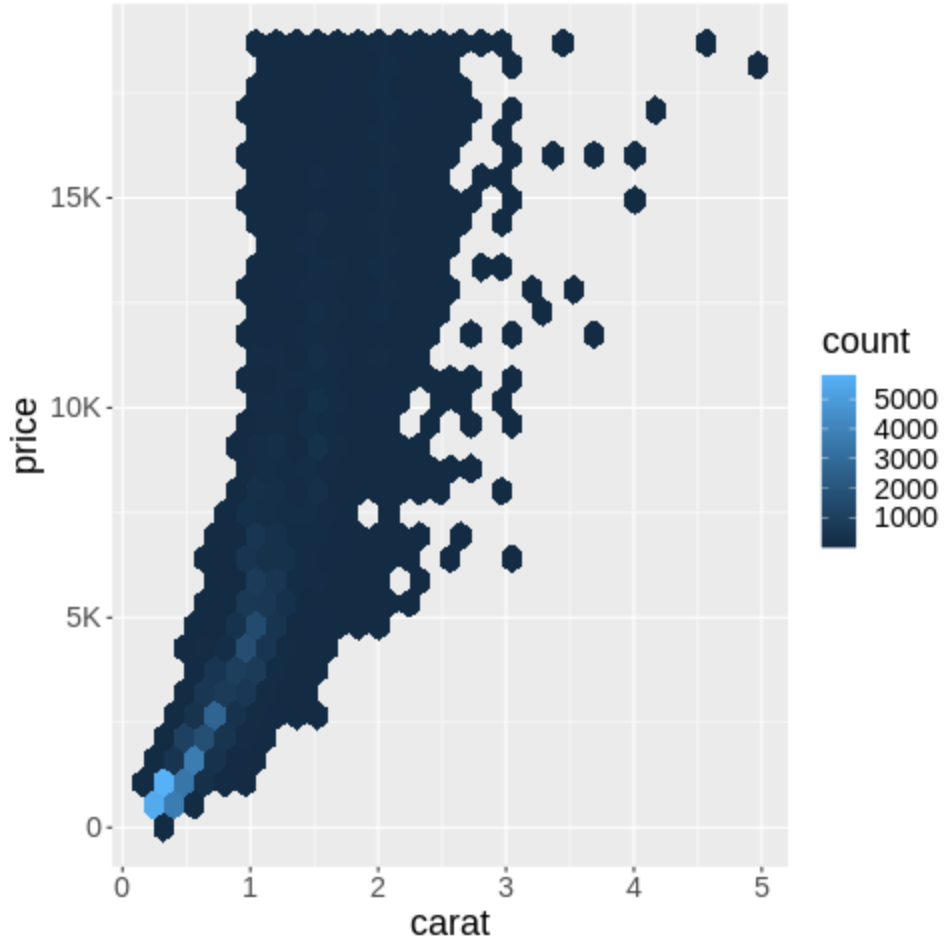
Skip to main content

```
    geom_hex() +
    scale_y_continuous(labels = scales::label_scientific())
```



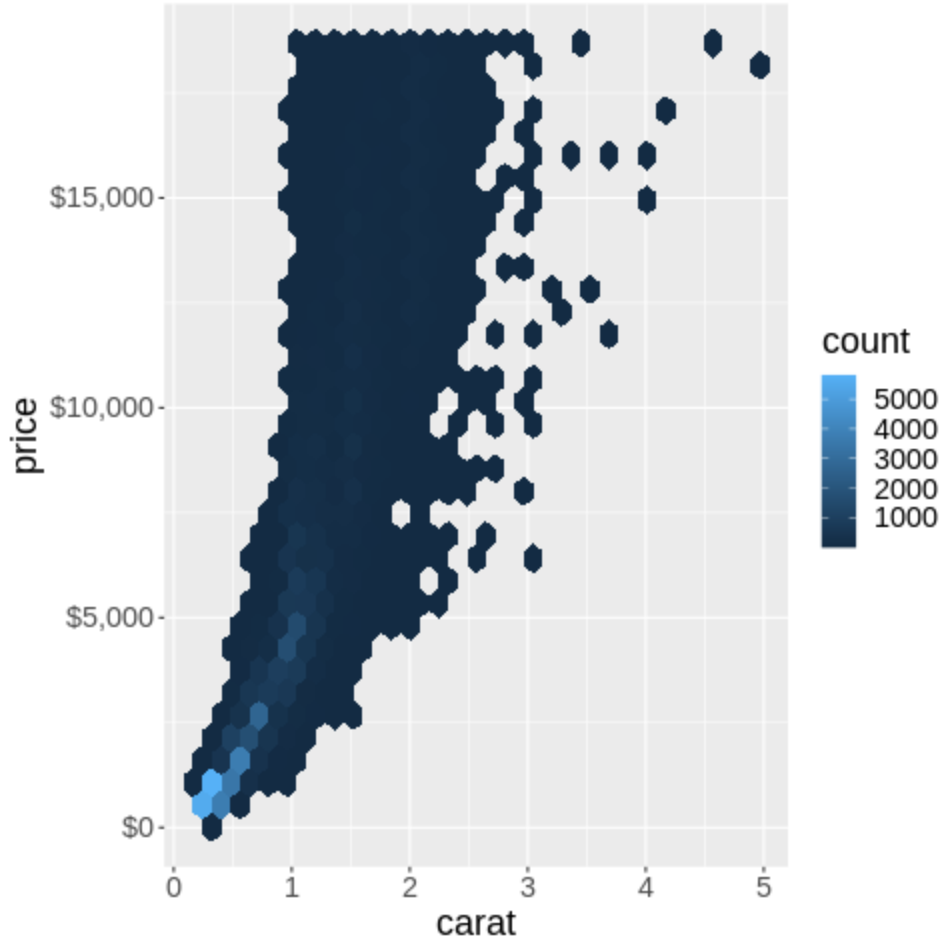

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_y_continuous(labels = scales::label_number_si())
```

Skip to main content




```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_y_continuous(labels = scales::label_dollar())
```

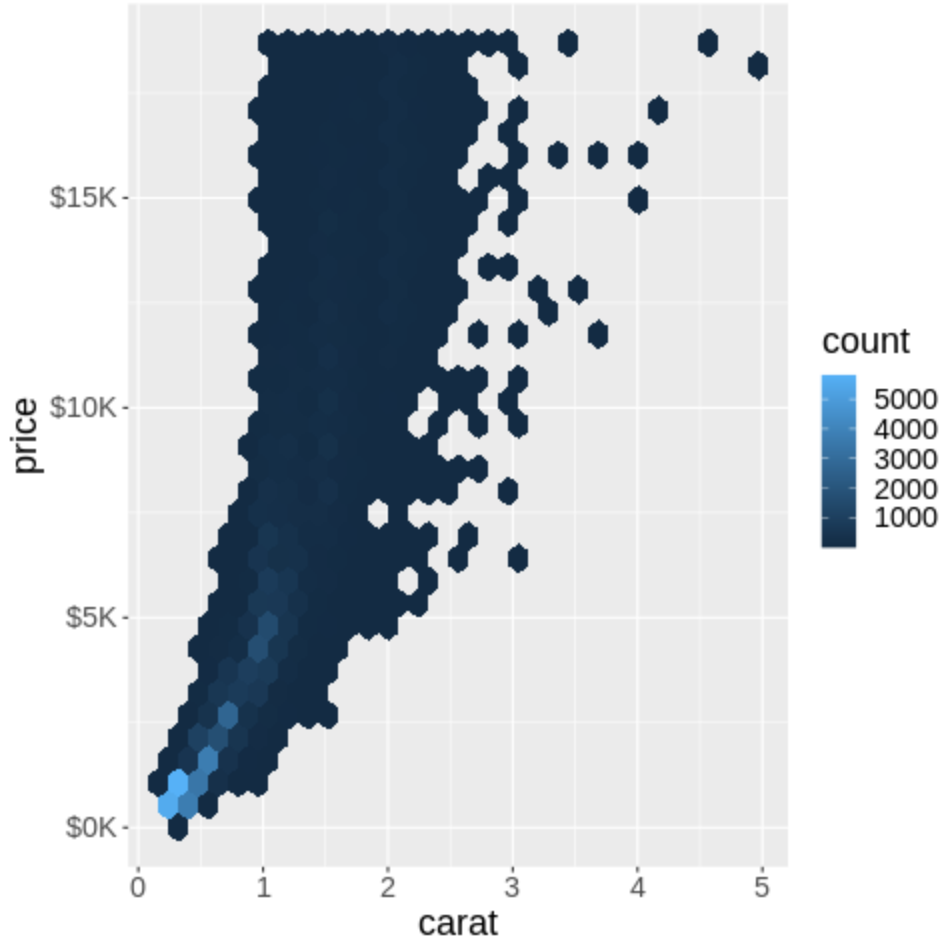
Skip to main content



You can also parameterize the functions in the `scales` package, e.g. to combine si units and dollars we could do the following.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_y_continuous(labels = scales::label_dollar(scale = .001, suffix = "K
```

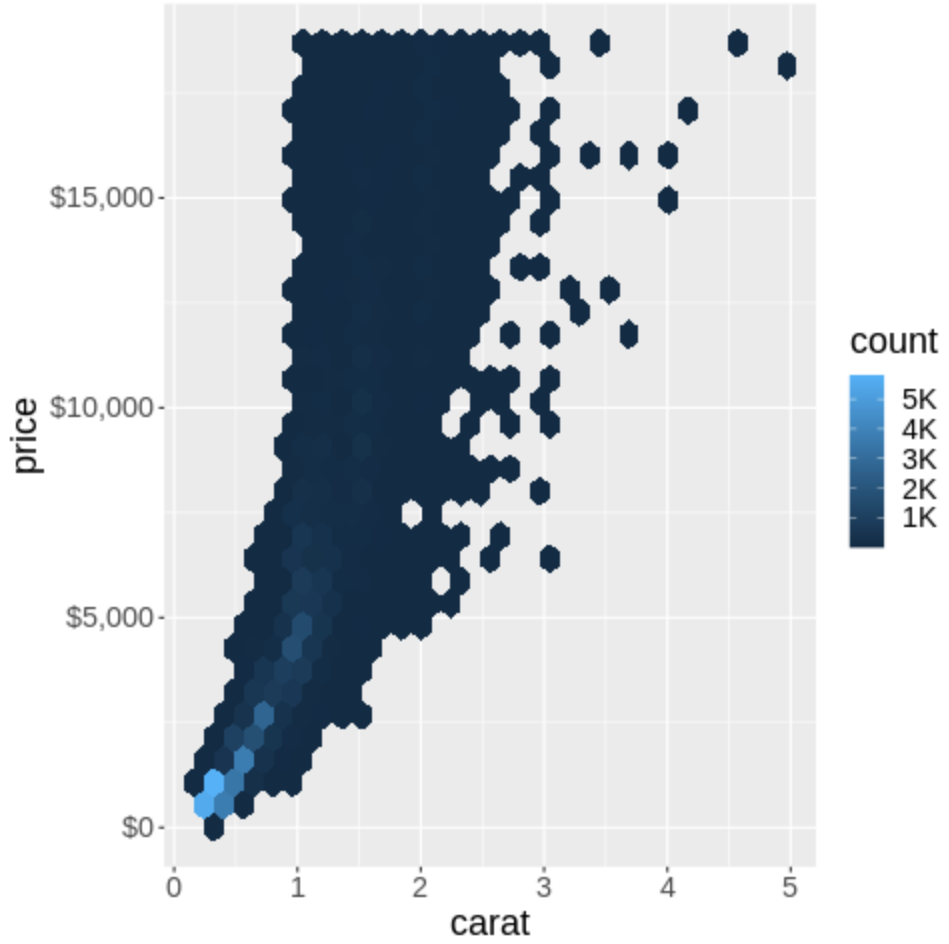
Skip to main content



The legend can be formatted via the same syntax.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_y_continuous(labels = scales::label_dollar()) +
    scale_fill_continuous(labels = scales::label_number_si())
```

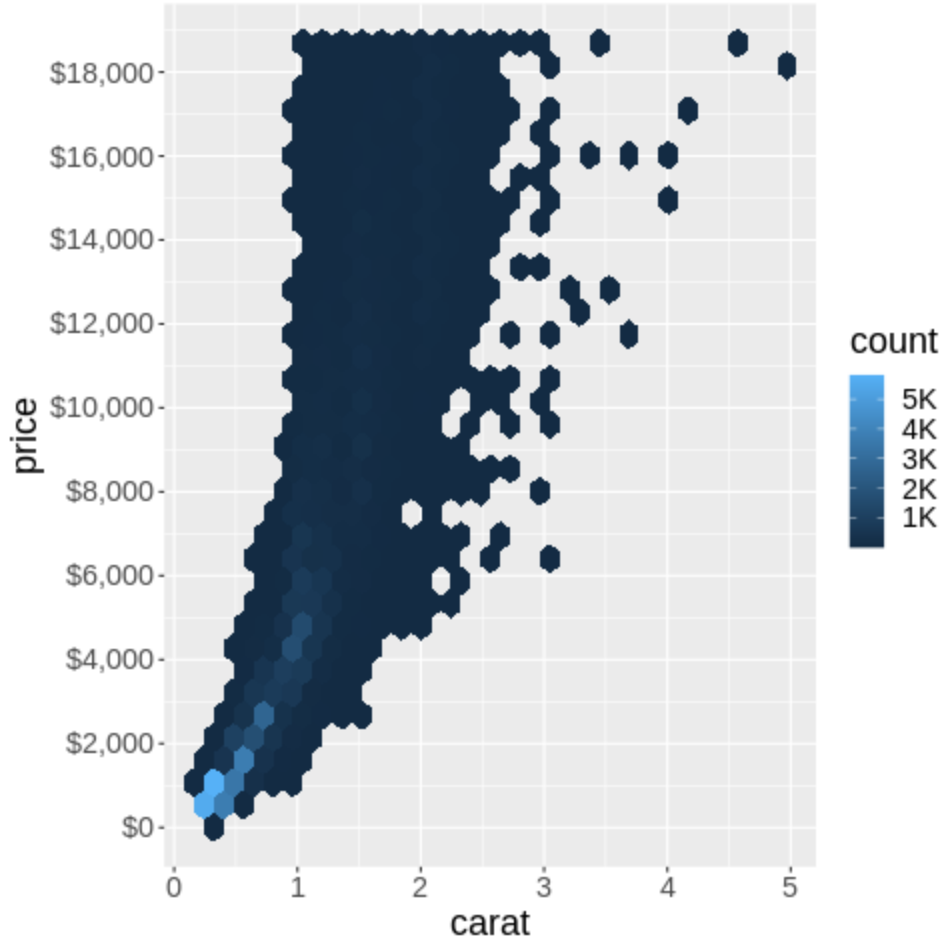
Skip to main content



The scales package also helps us setting the number of ticks (breaks) on an axis.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_y_continuous(
        labels = scales::label_dollar(),
        breaks = scales::pretty_breaks(n = 10)) +
    scale_fill_continuous(labels = scales::label_number_si())
```

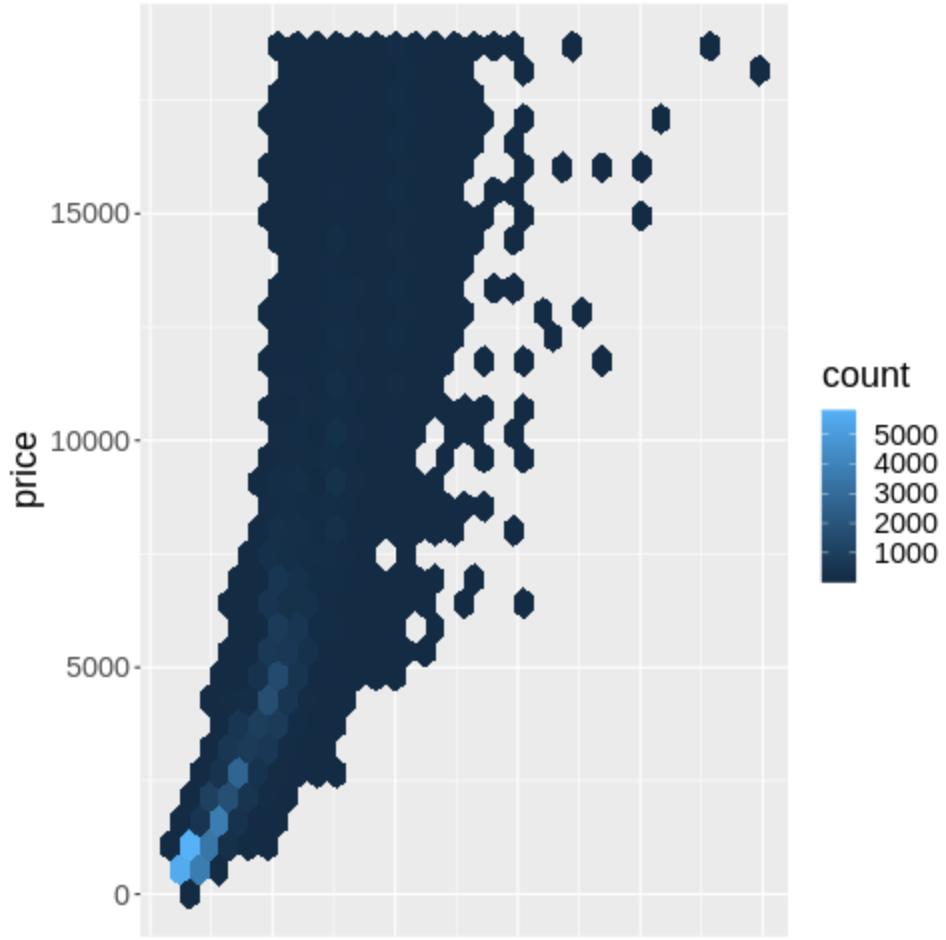
Skip to main content



You can remove an axis.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    theme(axis.title.x=element_blank(),  # Remove the title
          axis.text.x=element_blank(),  # Remove the tick text/label
          axis.ticks.x=element_blank()) # Remove the tick line/mark
```

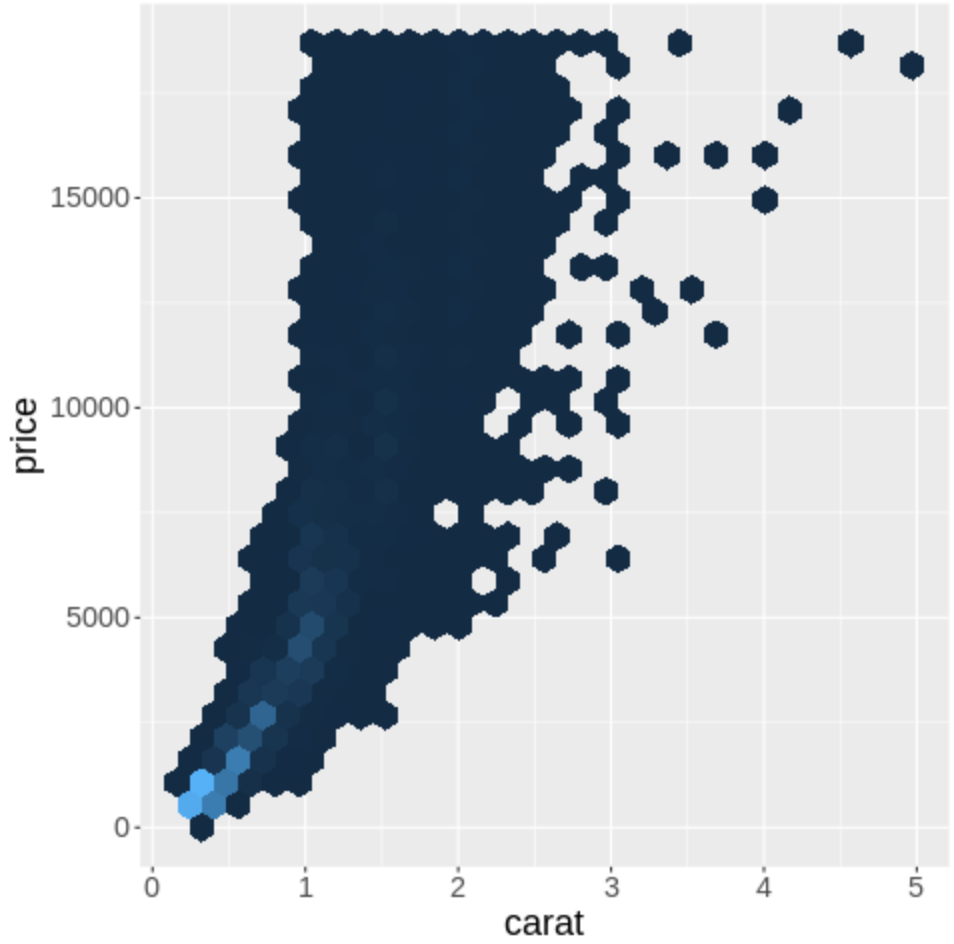
Skip to main content



Or remove a legend (referred to as a “guide” in ggplot).

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    guides(fill = "none")
```

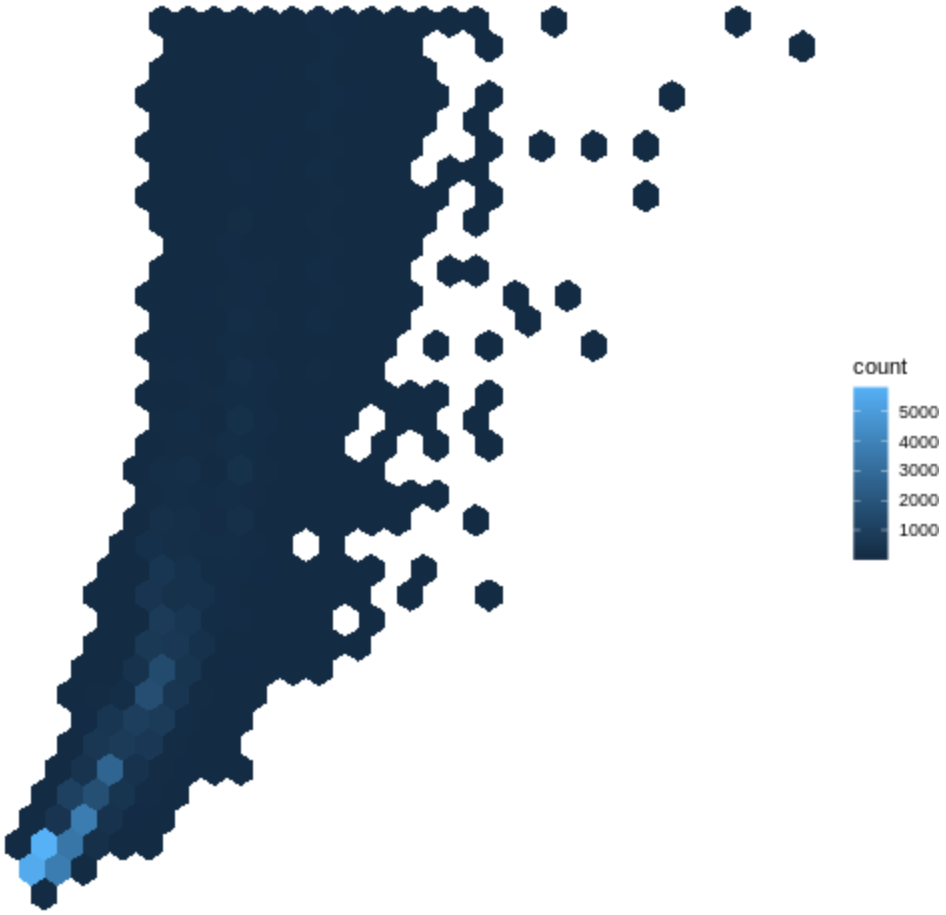
Skip to main content



Or set a theme that hides all axis objects.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    theme_void()
```

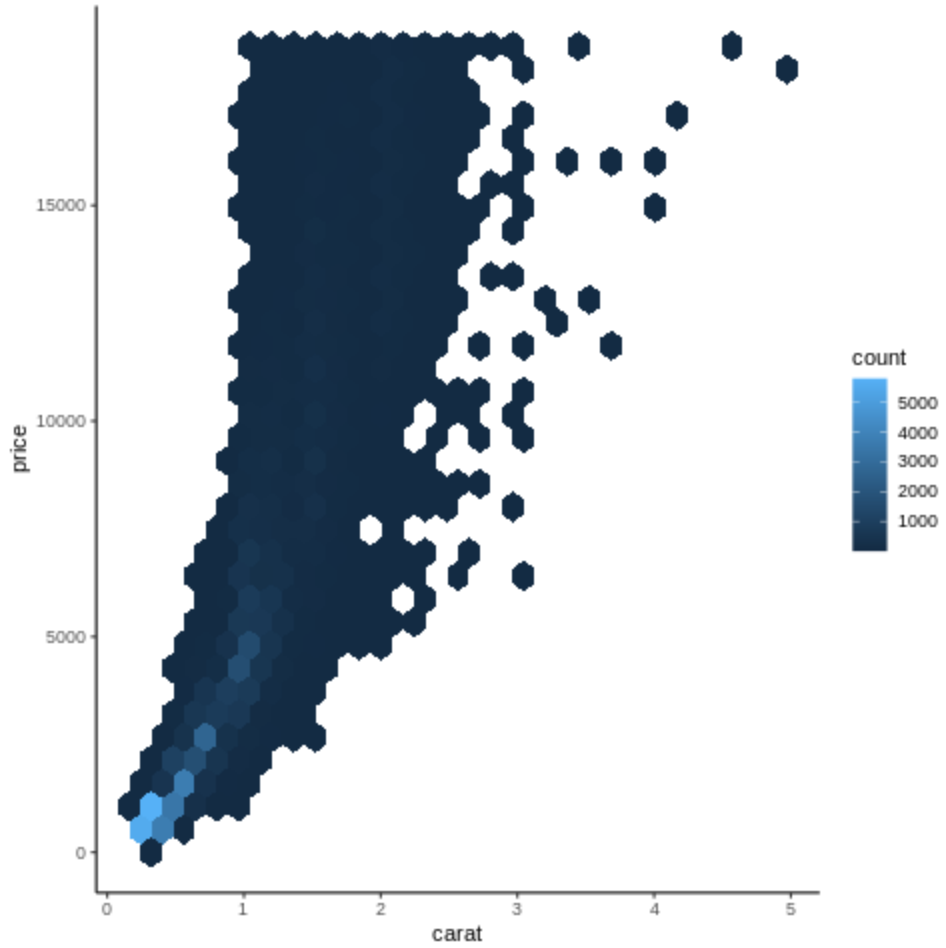
Skip to main content



The classic theme is nice. There are many more sophisticated theme in the ggthemes.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    theme_classic()
```

Skip to main content



5.3. Figure, axis, and legend titles

5.3.1. Py

When doing EDA, axis titles etc don't matter that much, since you are the primary person interpreting them. In communication however, your plots often need to be interpretable on their own without explanation. Setting descriptive titles is a big part of this, please see the required readings for more info.

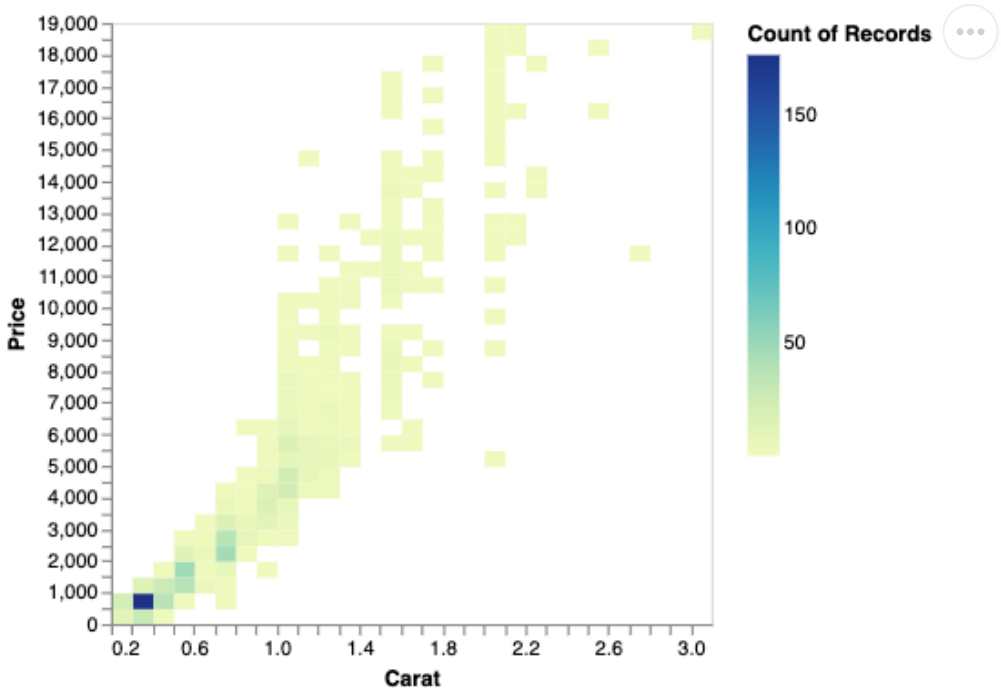
Axis titles should be capitalized and contain spaces, no variable names with underscores.

```
# Set back to defaut theme
alt.themes.enable('default')

alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40).title('Carat'),
    alt.Y('price').bin(maxbins=40).title('Price'),
```

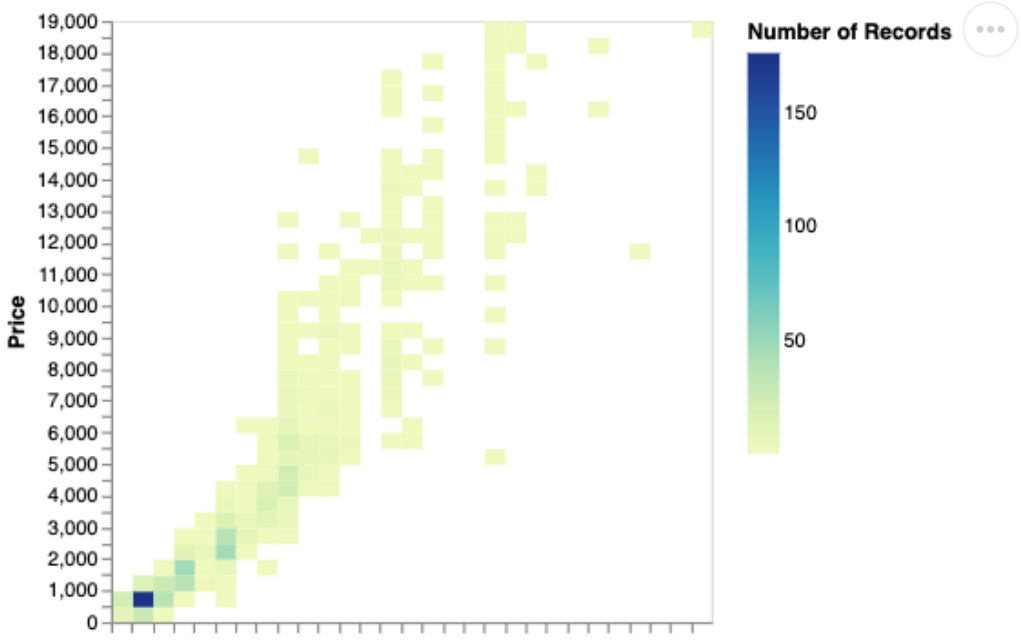
Skip to main content

```
    color='count()'
)
```



The legend title is controlled inside the encoding channel that is displays.

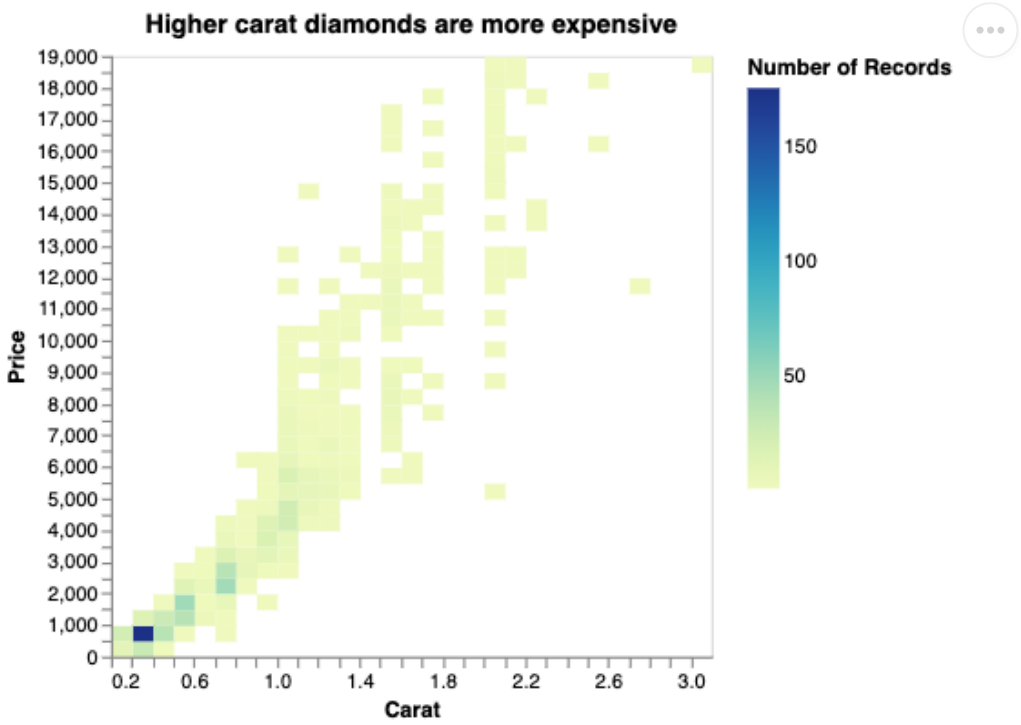
```
alt.Chart(diamonds).mark_rect().encode(
    alt.X('carat').bin(maxbins=40).title('Carat'),
    alt.Y('price').bin(maxbins=40).title('Price'),
    alt.Color('count()').title('Number of Records')
)
```



Skip to main content

It is important that the overall figure title contains the take home message of the chart (or maybe a question that evokes reader's interest), rather than just a description of the axes. For the chart below, that means that we could write something like "Higher carat diamonds are the most expensive" or "Which are the most expensive diamonds?", but not "Diamond carat versus price".

```
alt.Chart(diamonds, title='Higher carat diamonds are more expensive').mark_rec
    alt.X('carat').bin(maxbins=40).title('Carat'),
    alt.Y('price').bin(maxbins=40).title('Price'),
    alt.Color('count()').title('Number of Records')
)
```

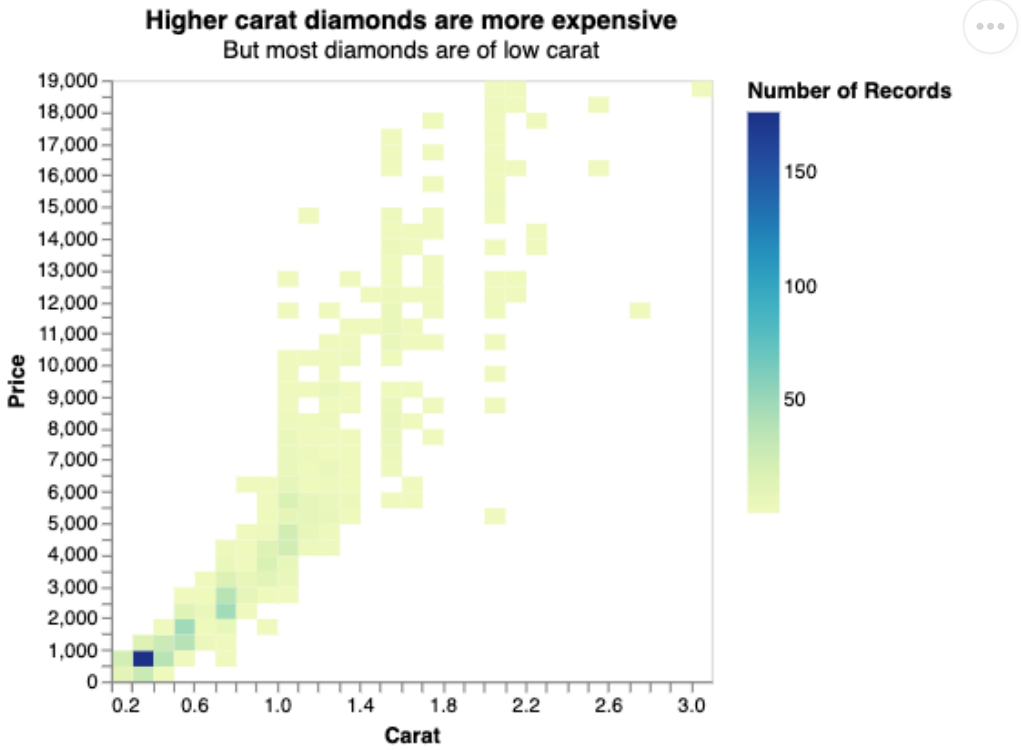


A suptitle is a property of a title. You can use the subtitle to add additional info, such as elaborating more on the takehome message, or adding some important info that is relevant for interpreting the figure correctly. Depending on where you are publishing your figure, you can be a bit looser with the fact that this is called "subtitle" and include have a longer figure caption here as well if you are not planning to put one under the figure in the text (e.g. for an article).

If you have a question in the title to peak interest, the subtitle area could be used for the takehome message that answers the question. Having just the question on its own means that you are not guiding the reader towards the answer and they might misread the chart.

Skip to main content

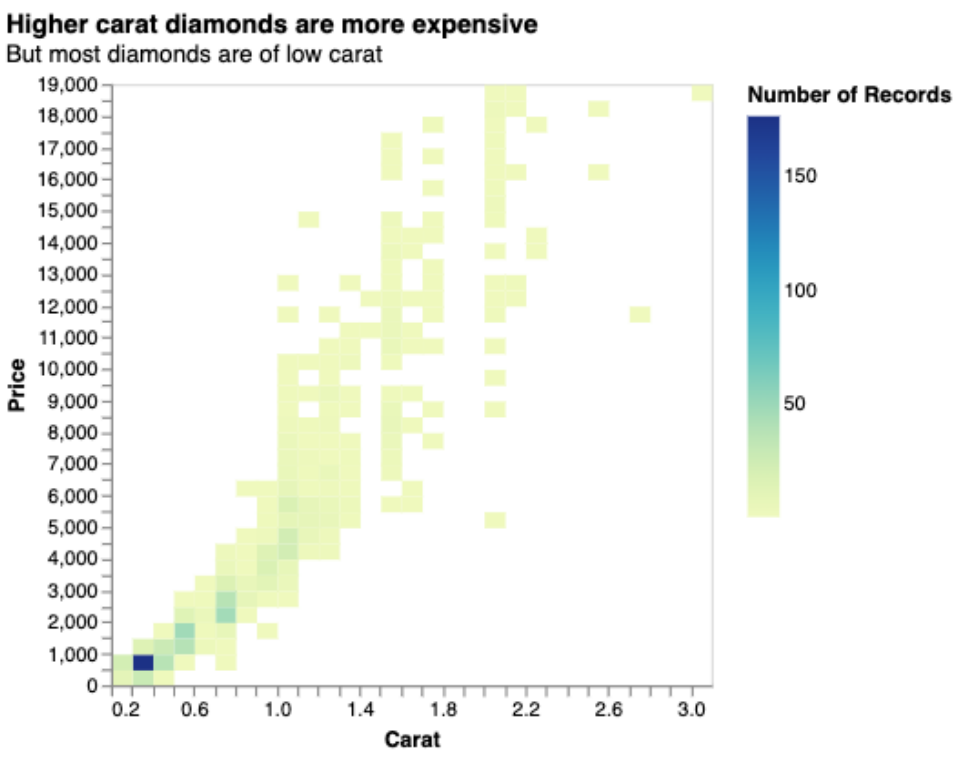
```
alt.Chart(
    diamonds,
    title=alt.Title(
        text='Higher carat diamonds are more expensive',
        subtitle='But most diamonds are of low carat'
    )
).mark_rect().encode(
    alt.X('carat').bin(maxbins=40).title('Carat'),
    alt.Y('price').bin(maxbins=40).title('Price'),
    alt.Color('count()').title('Number of Records')
)
```



You can change the title alignment with the `anchor` parameter.

```
alt.Chart(
    diamonds,
    title=alt.Title(
        text='Higher carat diamonds are more expensive',
        subtitle='But most diamonds are of low carat',
        anchor='start'
    )
).mark_rect().encode(
    alt.X('carat').bin(maxbins=40).title('Carat'),
    alt.Y('price').bin(maxbins=40).title('Price'),
    alt.Color('count()').title('Number of Records')
)
```

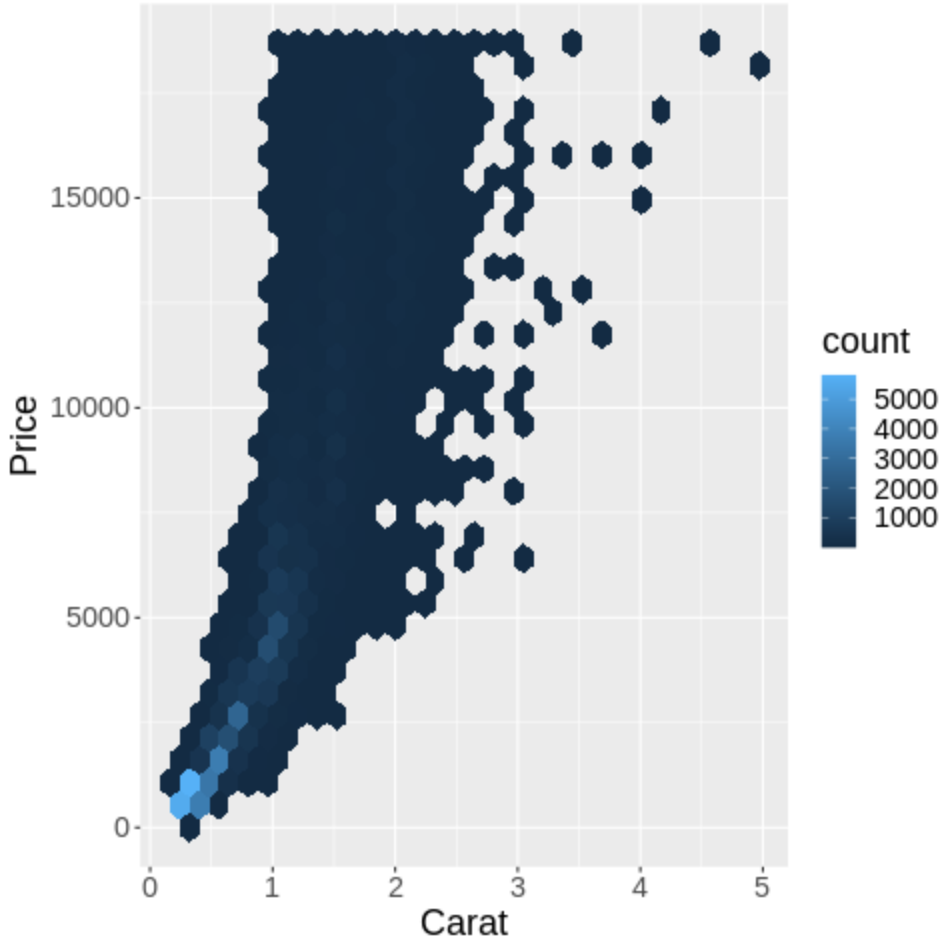
Skip to main content



5.3.2. R

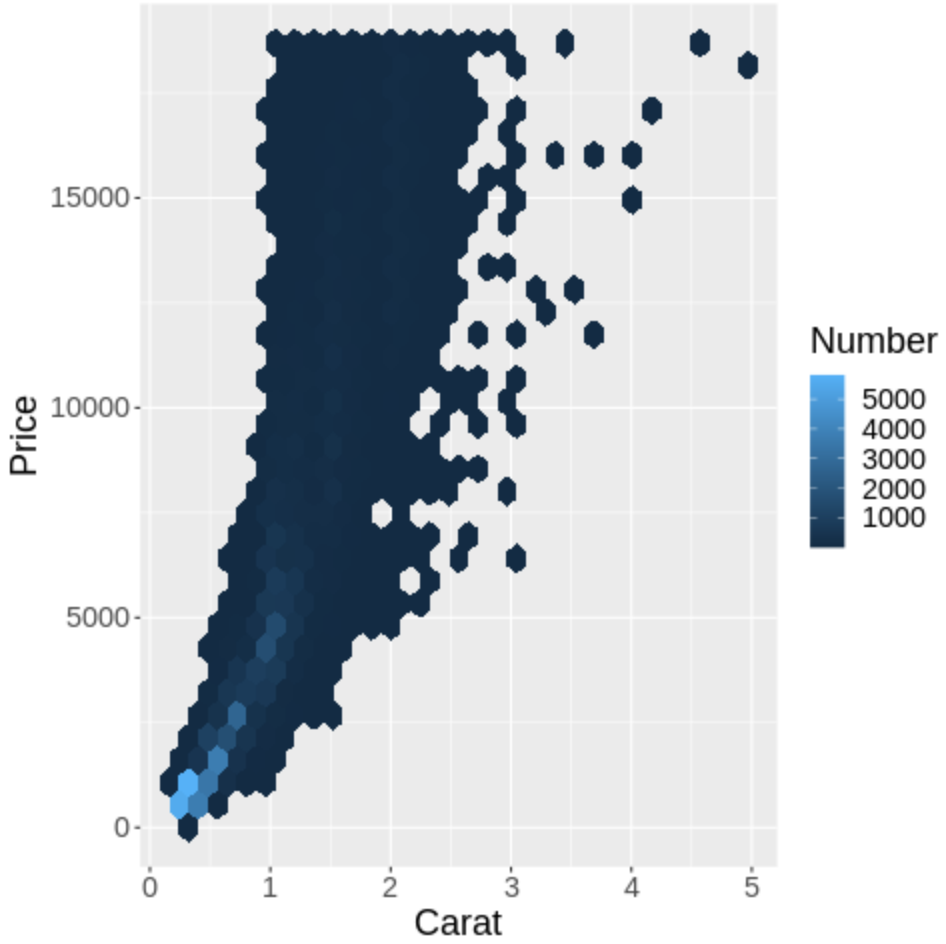
```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    labs(x = 'Carat', y = 'Price')
```

Skip to main content



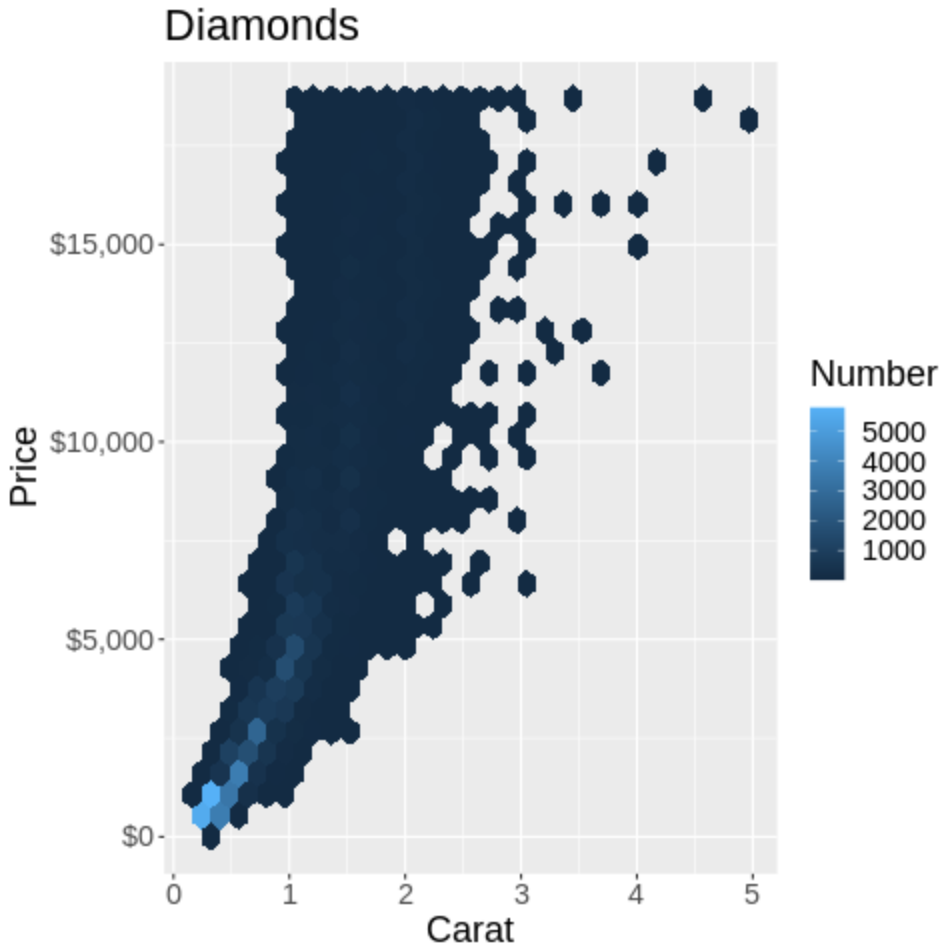

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    labs(x = 'Carat', y = 'Price', fill = 'Number')
```

Skip to main content



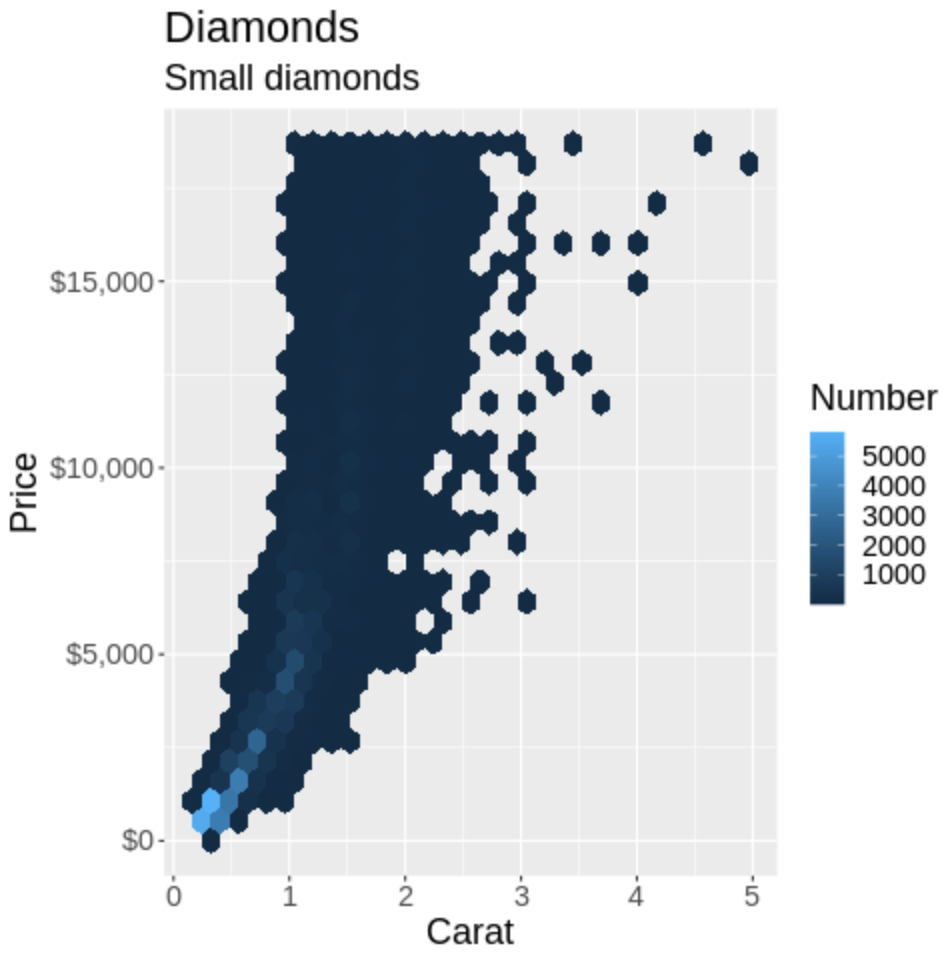

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    labs(x = 'Carat', y = 'Price', fill = 'Number', title = 'Diamonds') +
    scale_y_continuous(labels = scales::label_dollar())
```

Skip to main content




```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    labs(x = 'Carat', y = 'Price', fill = 'Number', title = 'Diamonds', subtit
    scale_y_continuous(labels = scales::label_dollar())
```

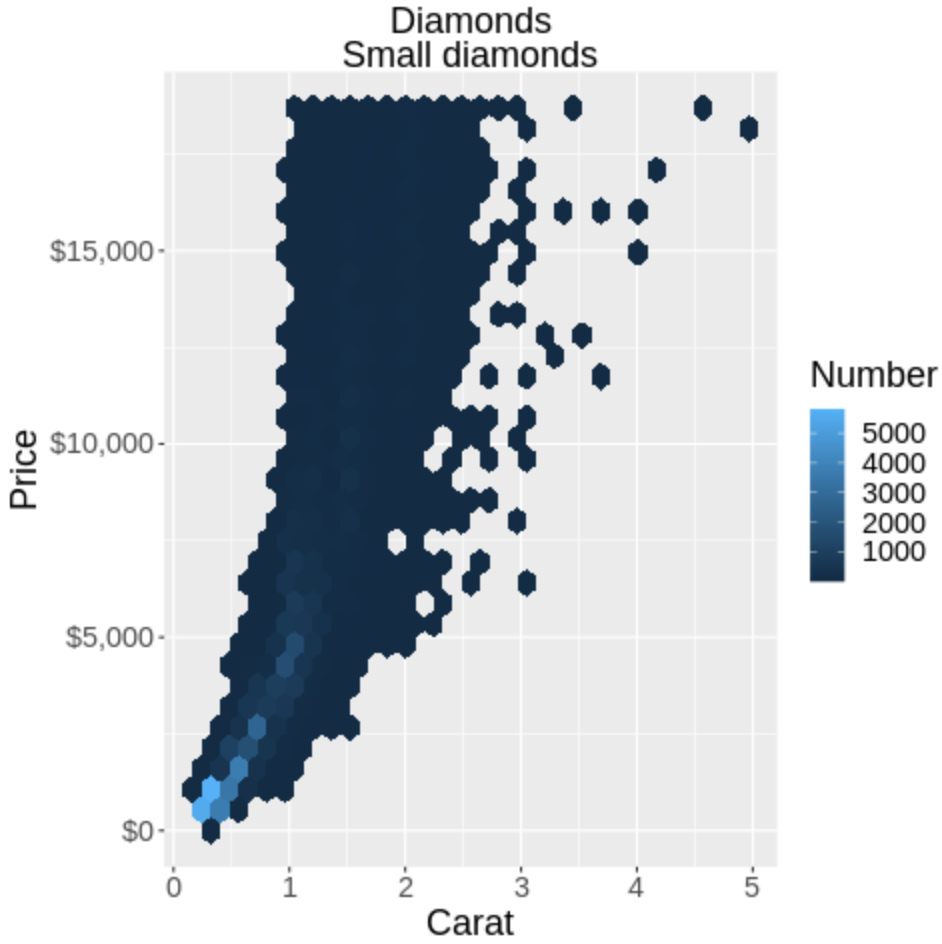
Skip to main content



You can change the alignment of the title in the theme.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    labs(x = 'Carat', y = 'Price', fill = 'Number', title = 'Diamonds', subtit
    scale_y_continuous(labels = scales::label_dollar()) +
    theme(plot.title = element_text(hjust = 0.5),
          plot.subtitle = element_text(hjust = 0.5))
```

Skip to main content



5.4. Axis ranges

5.4.1. Py

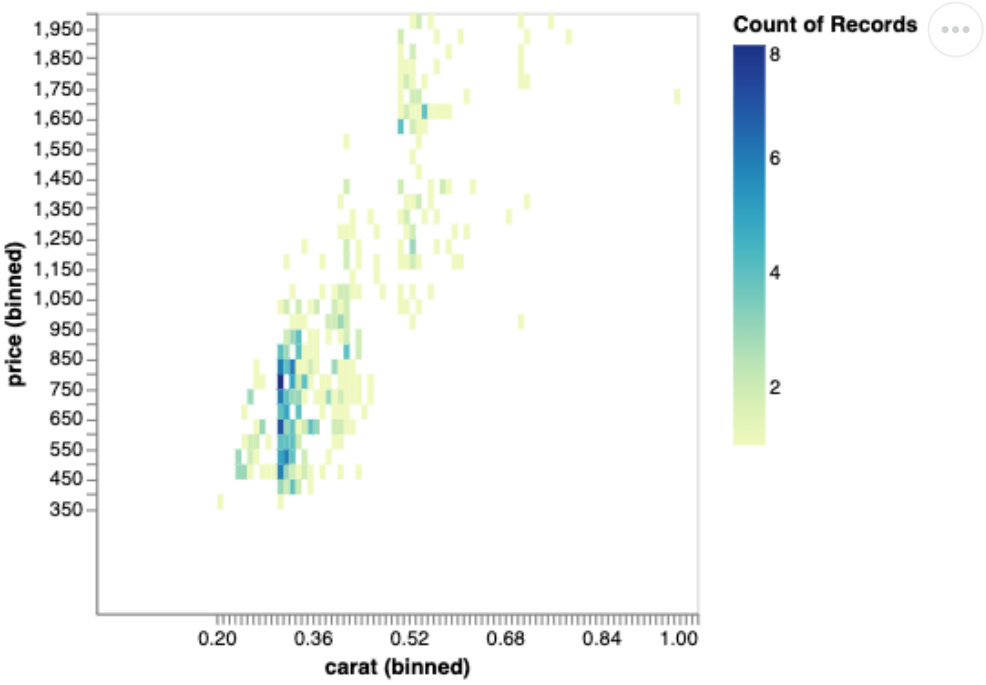
In many cases the most convenient way might be to filter the data before sending it to the chart. This was you are using the efficient pandas methods to do the heavy lifting and avoiding slowdown from plotting many points and then zoom.

The axis range is set with the `domain` parameter to `alt.Scale`. To set an axis range to less than the extent of the data, we also need to include `clip=True` in the mark, otherwise it will be plotted outside the figures. We also need to increase the number of bins to have higher resolution in this zoomed in part. Sometimes the range is padded with a bit of extra space automatically, if this is undesired `nice=False` can be set inside `alt.Scale`.

All these steps should reinforce that it is usually better to filter the data and let Altair handle the plotting.

Skip to main content

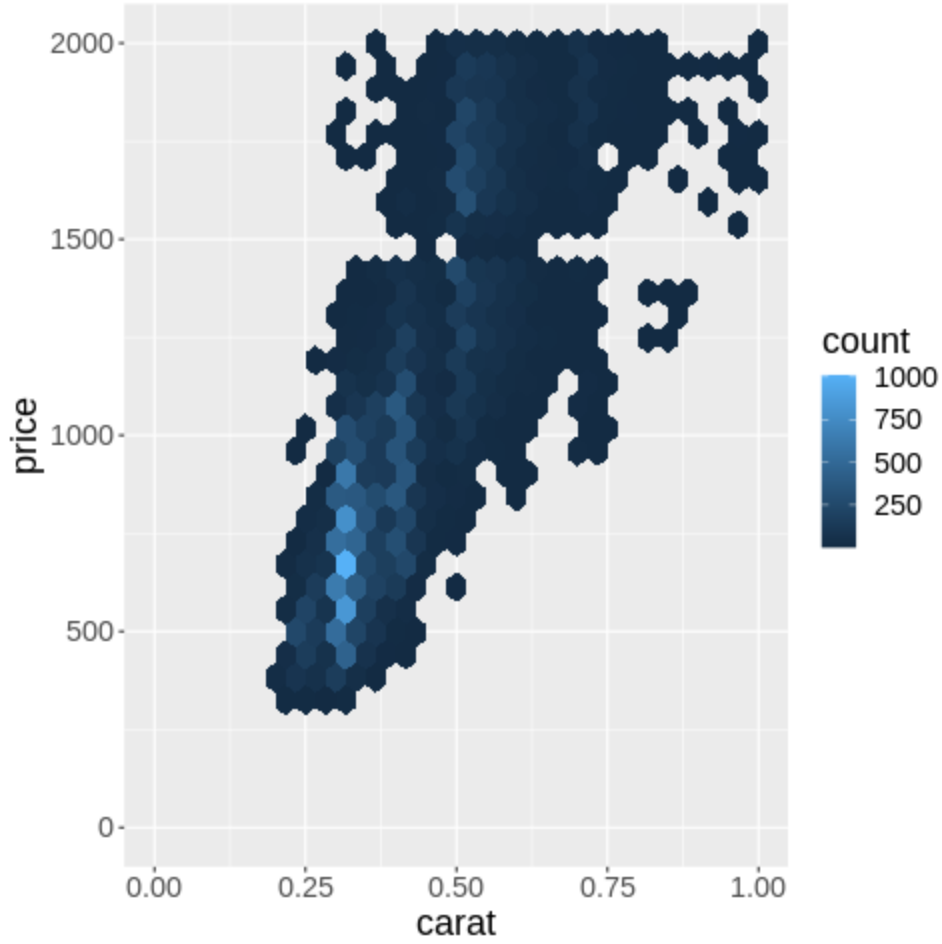
```
alt.Chart(diamonds).mark_rect(clip=True).encode(
    alt.X('carat').bin(maxbins=400).scale(domain=(0, 1)),
    alt.Y('price').bin(maxbins=400).scale(domain=(0, 2000)),
    alt.Color('count()')
)
```



5.4.2. R

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_x_continuous(limits = c(0, 1)) +
    scale_y_continuous(limits = c(0, 2000))
```

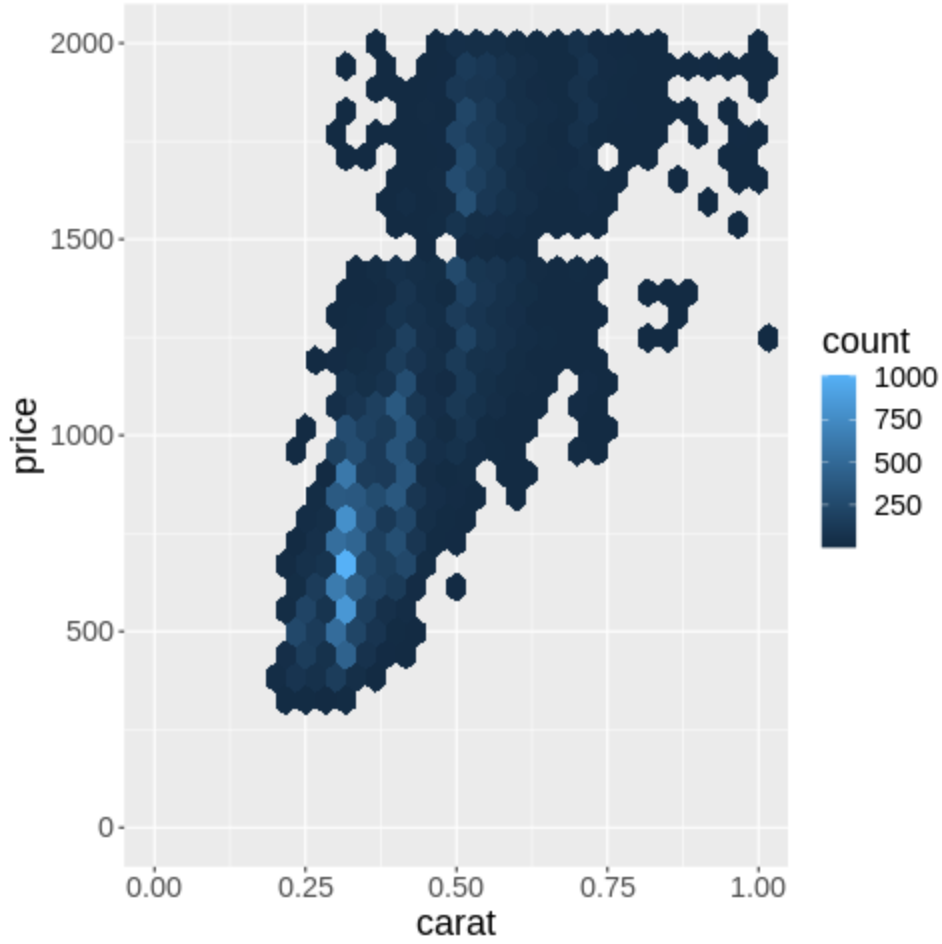
Skip to main content



By deafault ggplot removes observations outside the visible domain, so any big marks that are both inside and outside, such as bars for example, will be cut out. This is good because it makes it hard for people to make poor visualization choices, such as zooming in on bar charts instead of showing the entire domain starting from zero. In situation where you do need to include such partial graphics, you can set the out of bounds (`oob`) parameter to `scales::oob_keep` as described in https://scales.r-lib.org/reference/oob.html.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_x_continuous(limits = c(0, 1), oob = scales::oob_keep) +
    scale_y_continuous(limits = c(0, 2000))
```

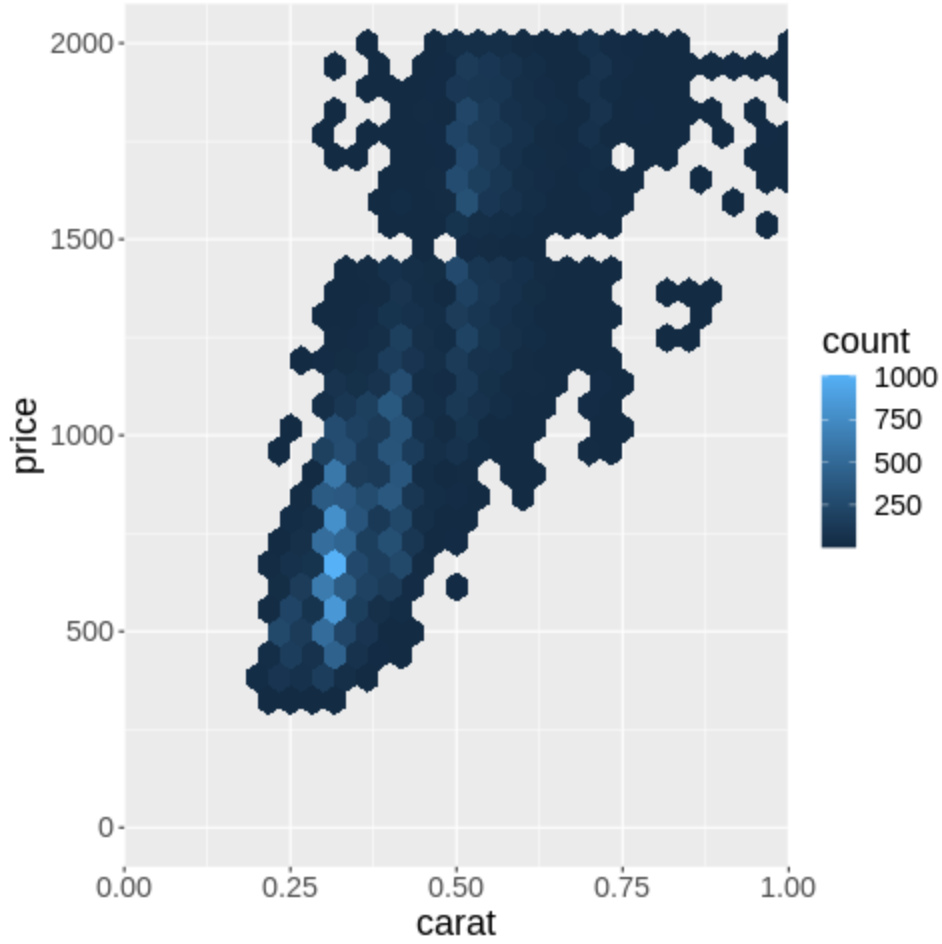
Skip to main content



You can see that there is a bit of empty space or padding on each size of the x-axis to the left of 0 and to the right of 1. If we want to get rid of this, we can set `expand = expansion(mult = c(0, 0))` in the scale we're using. The vector contains the min and max padding and changes as a multiplication of the current axis range. So if we wanted some space at the right side, we could use `mult = c(0, 0.05))` or similar instead. More details here.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_x_continuous(limits = c(0, 1), expand = expansion(mult = c(0, 0))) +
    scale_y_continuous(limits = c(0, 2000))
```

Skip to main content



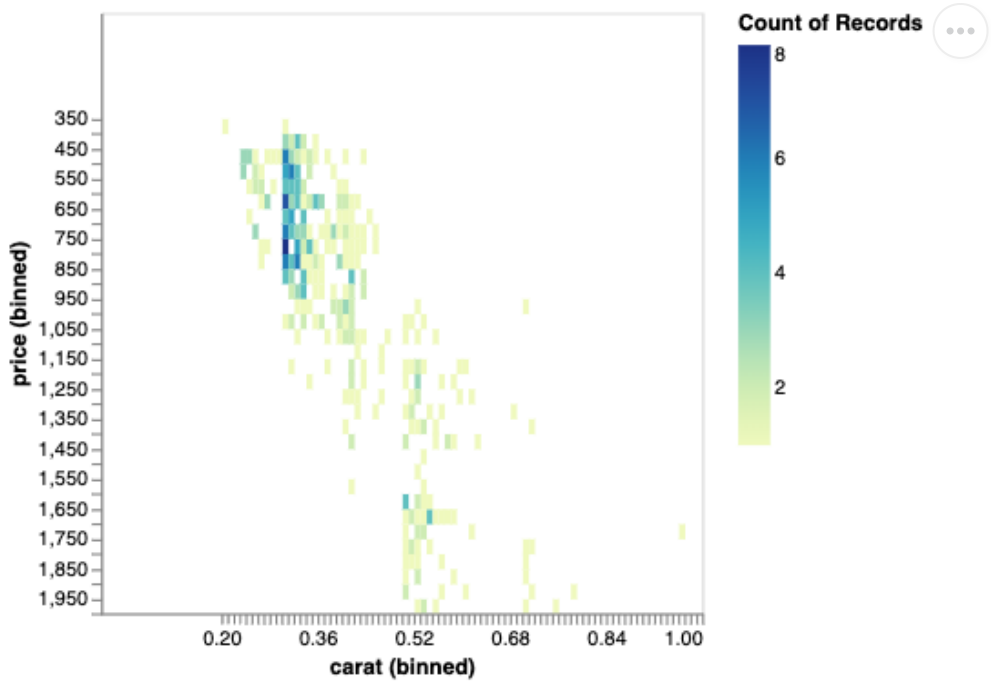
5.5. Reversing an axis

5.5.1. Py

It is possible to reverse an axis, but this is not that commonly used, so it is mostly added here as a reference.

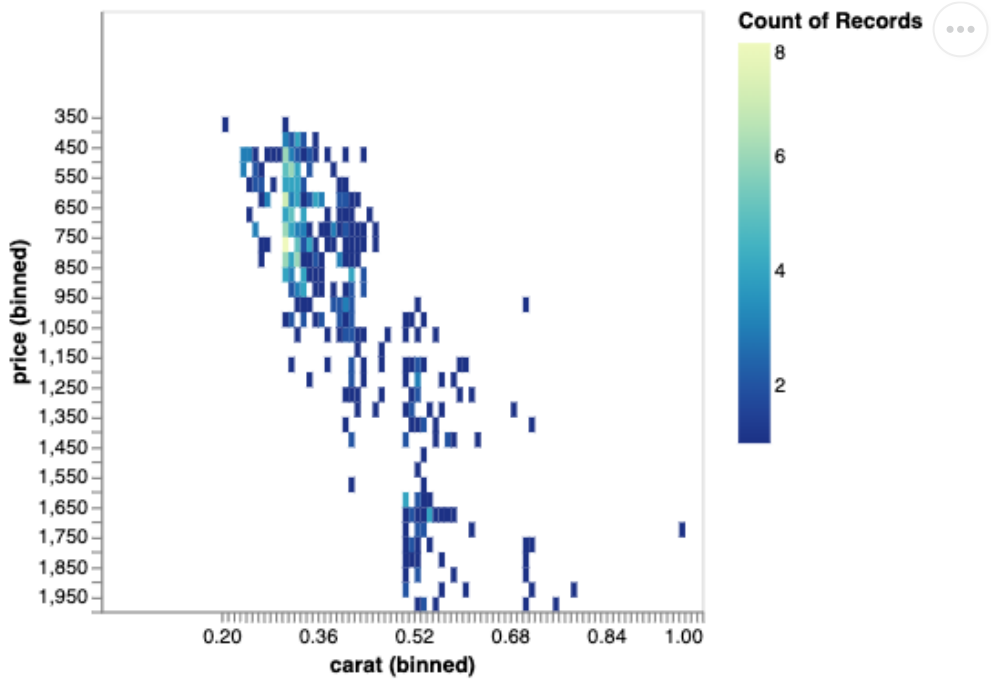
```
alt.Chart(diamonds).mark_rect(clip=True).encode(
    alt.X('carat').bin(maxbins=400).scale(domain=(0, 1)),
    alt.Y('price').bin(maxbins=400).scale(domain=(0, 2000), reverse=True),
    alt.Color('count()')
)
```

Skip to main content



This is not usually that useful for an xy-axis, but remember that color, size, etc are all scales in Altair, so they can be reversed with the same syntax! This is quite convenient and we will see more of it in following lectures.

```
alt.Chart(diamonds).mark_rect(clip=True).encode(
    alt.X('carat').bin(maxbins=400).scale(domain=(0, 1)),
    alt.Y('price').bin(maxbins=400).scale(domain=(0, 2000), reverse=True),
    alt.Color('count()').scale(reverse=True)
)
```

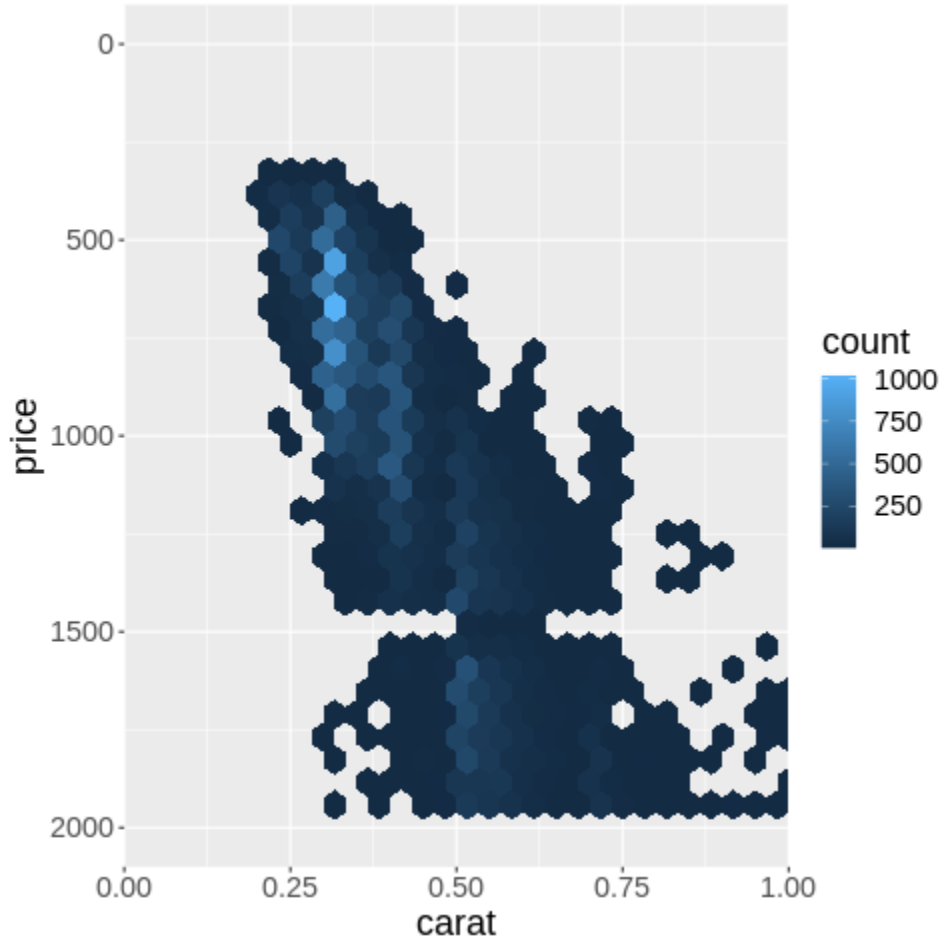


Skip to main content

5.5.2. R

To reverse the axis, we can set `trans = 'reverse'`. Other transforms include `log10` which there is also a shortcut for `scale_x_log10`. We also need to set the limits to go the opposite direction.

```
%%R
ggplot(diamonds) +
    aes(x = carat,
        y = price) +
    geom_hex() +
    scale_x_continuous(limits = c(0, 1), expand = expansion(mult = c(0, 0))) +
    scale_y_continuous(limits = c(2000, 0), trans = 'reverse')
```

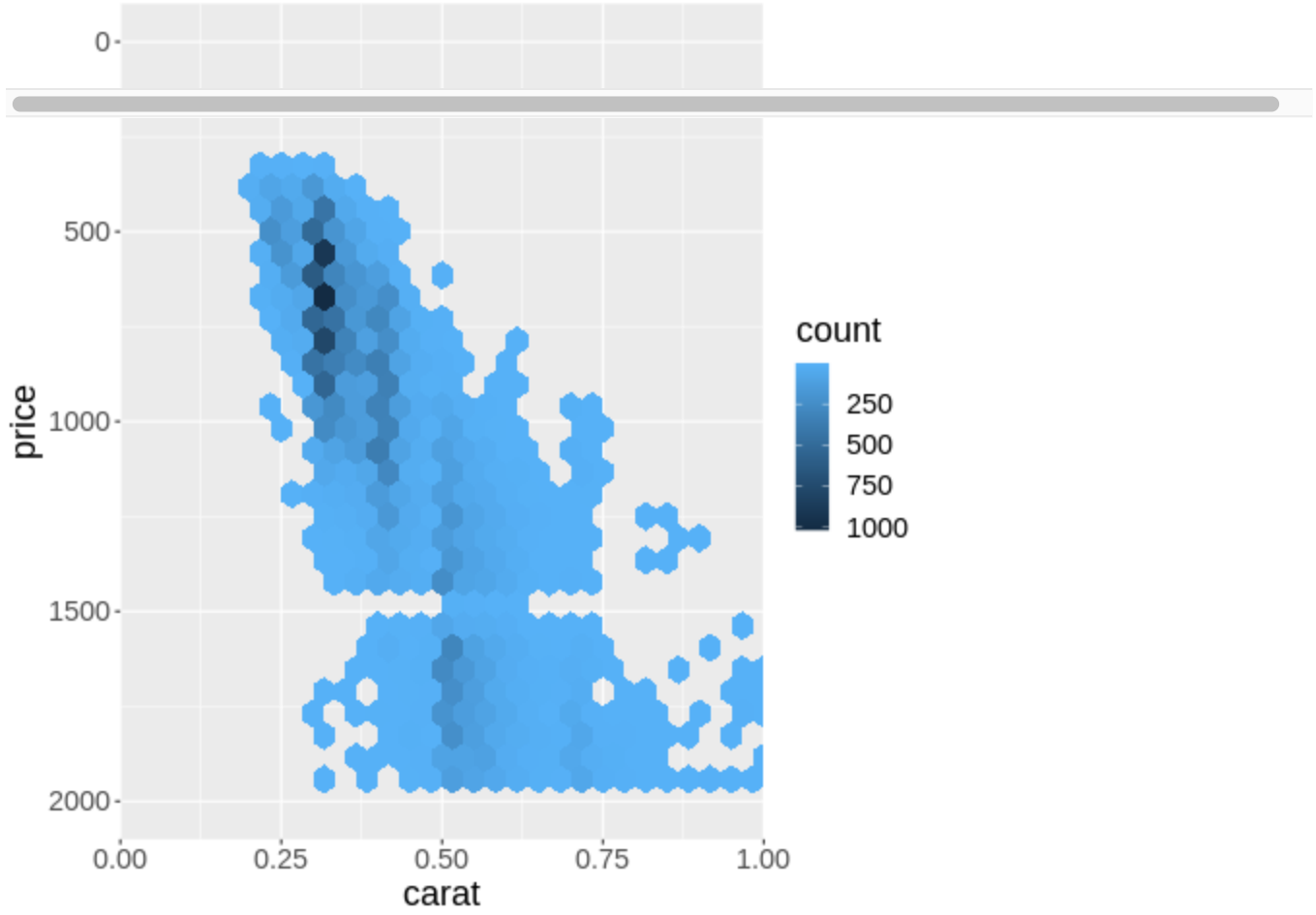


Just as in Altair, color scales can be controlled the same way as axis scales (the color for the hexagons is set via `fill` rather than `color`).

```
%%R
```

Skip to main content

```
        y = price) +
    geom_hex() +
    scale_x_continuous(limits = c(0, 1), expand = expansion(mult = c(0, 0))) +
    scale_y_continuous(limits = c(2000, 0), trans = 'reverse') +
    scale_fill_continuous(trans = 'reverse')
```



5.6. Trendlines

5.6.1. Py

Trendlines (also sometimes called “lines of best fit”, or “fitted lines”) are good to highlight general trends in the data that can be hard to elucidate by looking at the raw data points. This can happen if there are many data points or many groups inside the data.

```
from vega_datasets import data
```

Skip to main content

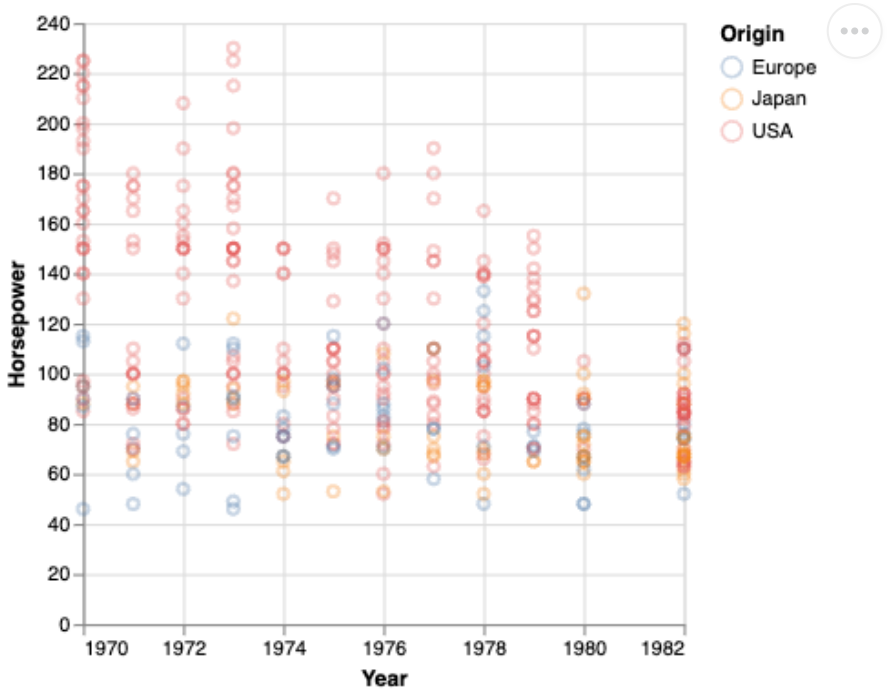
```
cars = data.cars()
cars
```

	Name	Miles_per_Gallon	Cylinders	Displacement	Horsepower	Weight_i
0	chevrolet chevelle malibu	18.0	8	307.0	130.0	
1	buick skylark 320	15.0	8	350.0	165.0	
2	plymouth satellite	18.0	8	318.0	150.0	
3	amc rebel sst	16.0	8	304.0	150.0	
4	ford torino	17.0	8	302.0	140.0	
...	...	...	...	...	...	
401	ford mustang gl	27.0	4	140.0	86.0	
402	vw pickup	44.0	4	97.0	52.0	
403	dodge rampage	32.0	4	135.0	84.0	
404	ford ranger	28.0	4	120.0	79.0	
405	chevy s-10	31.0	4	119.0	82.0	

406 rows × 9 columns

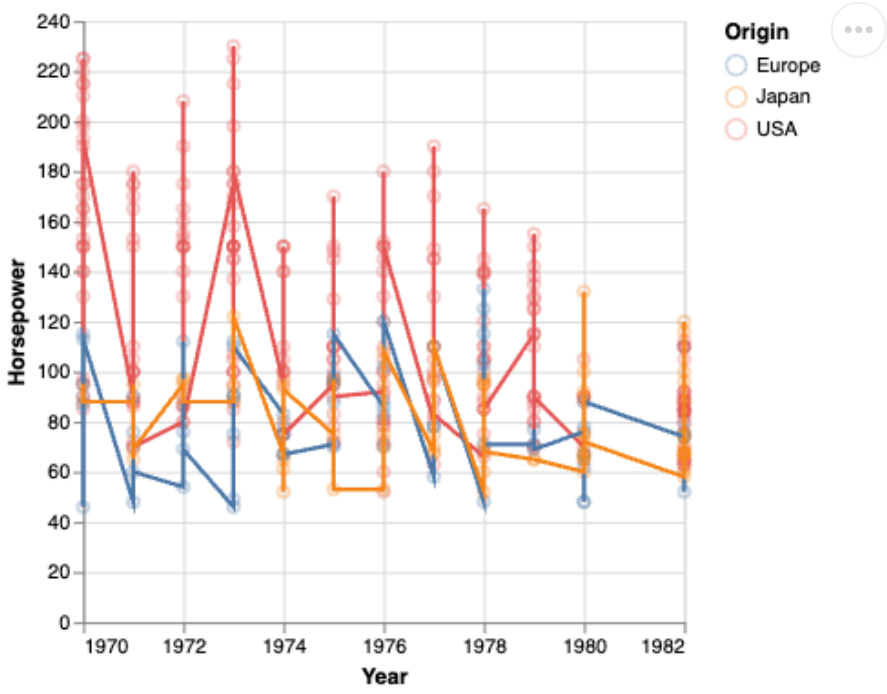
```
points = alt.Chart(cars).mark_point(opacity=0.3).encode(
    alt.X('Year'),
    alt.Y('Horsepower'),
    alt.Color('Origin')
)
points
```

Skip to main content



A not so effective way to visualize the trend in this data is to connect all data points with a line.

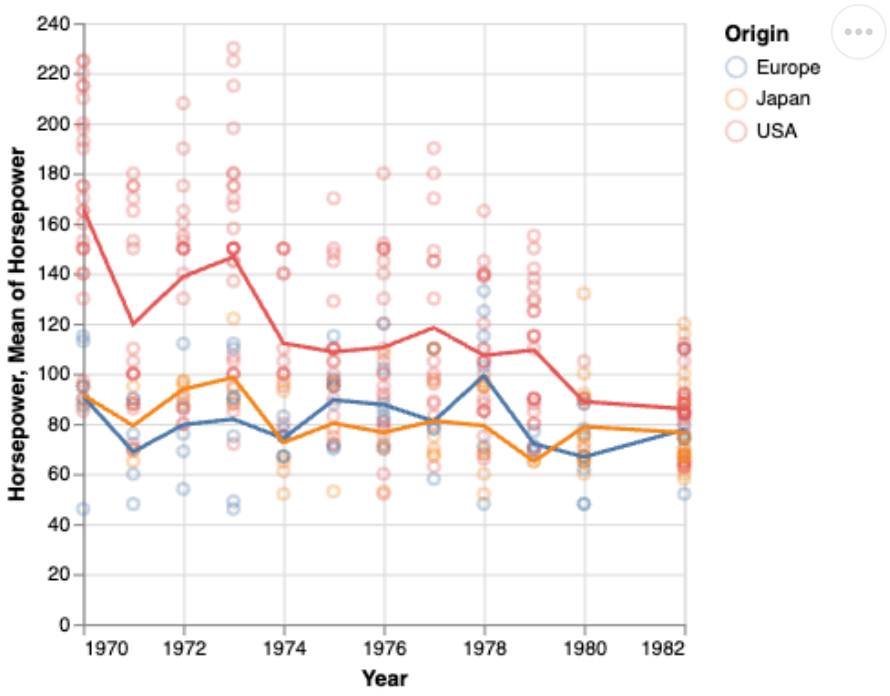
```
points + points.mark_line()
```



A simple way is to use the mean y-value at each x. This works OK in hour case because each year has several values, but in many cases with a continuous x-axis, you would need to bin it

Skip to main content

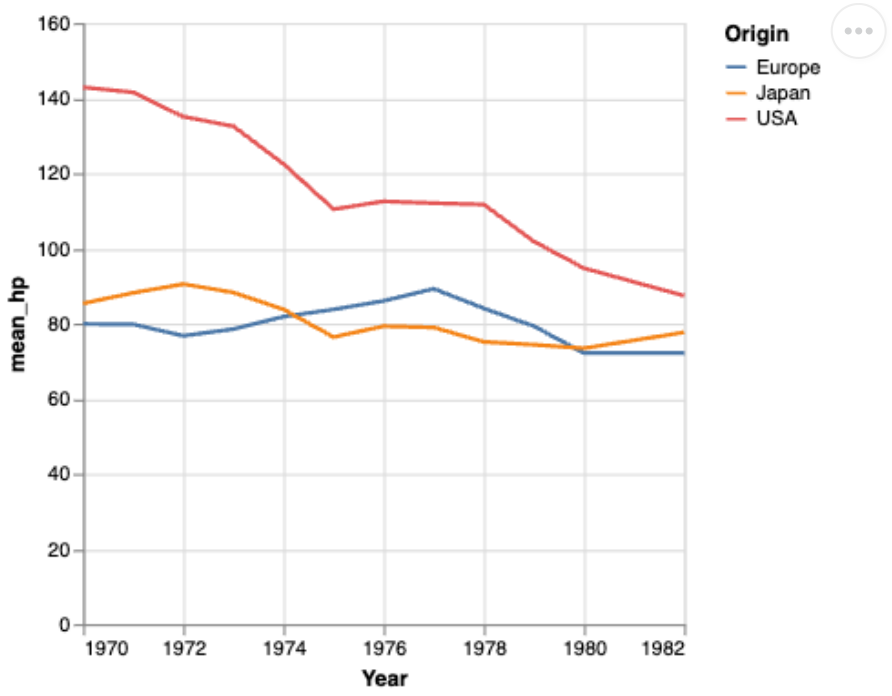
```
points + points.mark_line().encode(y='mean(Horsepower)')
```



An alternative to binning continuous data is to use a moving/rolling average, that takes the mean of the last n observations. In our example here, a moving average becomes a bit complicated because there are so many y-values for the exact same x-value, so we would need to calculate the average for each year first, and then move/roll over that, which can be done using the `window_transform` method in Altair. A more common and simpler example can be viewed in the Altair docs.

```
# You don't need to know `transform_window` for the quiz
mean_per_year = cars.groupby(['Origin', 'Year'])['Horsepower'].mean().reset_in
alt.Chart(mean_per_year).transform_window(
    mean_hp='mean(Horsepower)',
    frame=[-1, 1],
    groupby=['Origin']).mark_line().encode(
    x='Year',
    y='mean_hp:Q',
    color='Origin'
)
```

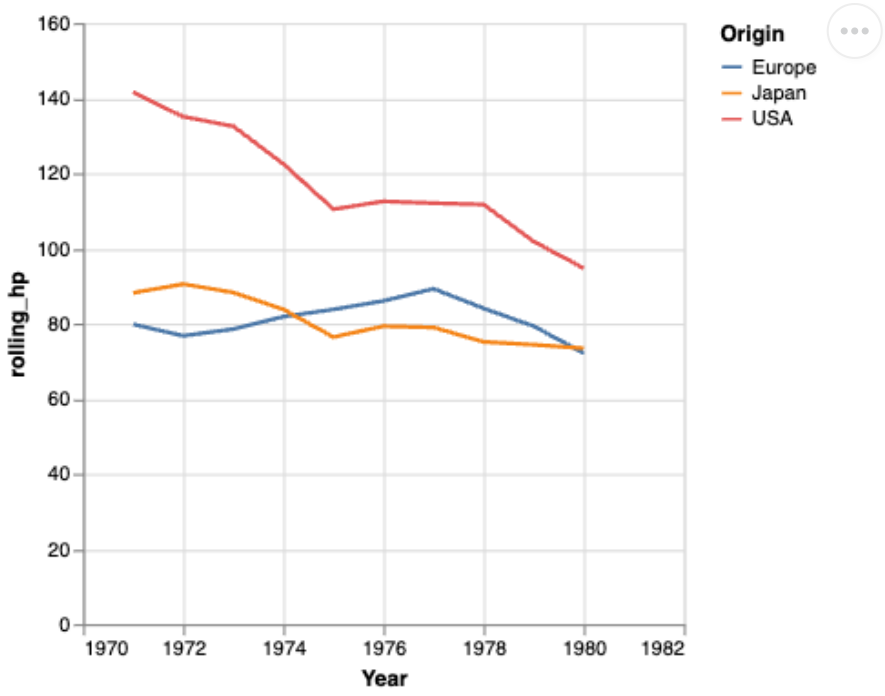
Skip to main content



We can also use the `rolling` method in pandas for this calculation, but it handles the edges a bit differently.

```
# You don't need to know `.rolling` for the quiz, just in general when a rolli
mean_per_year = cars.groupby(['Origin', 'Year'])['Horsepower'].mean().reset_in
mean_per_year['rolling_hp'] = mean_per_year.groupby('Origin')['Horsepower'].ro
alt.Chart(mean_per_year).mark_line().encode(
    x='Year',
    y='rolling_hp',
    color='Origin'
)
```

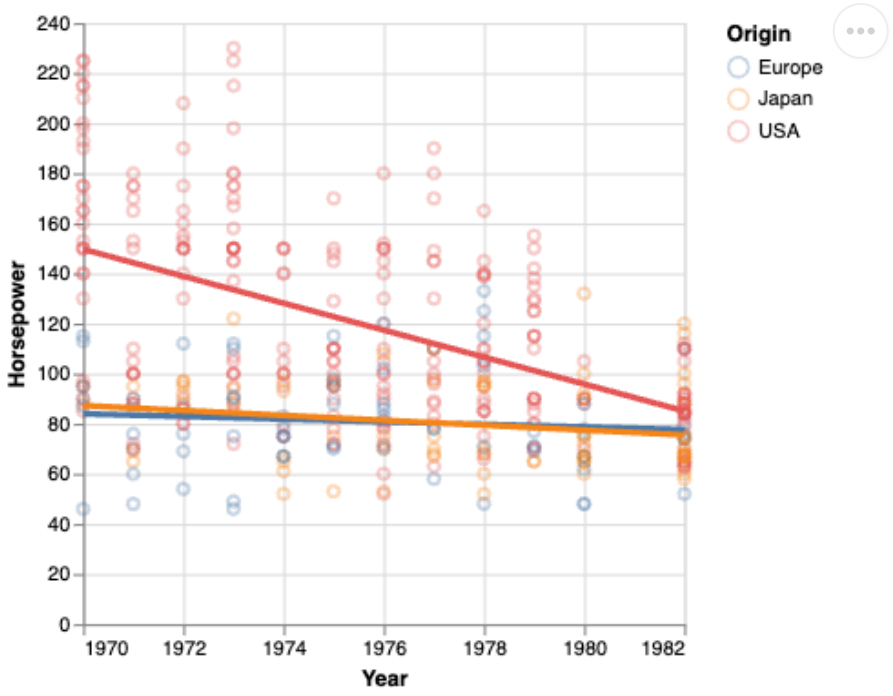
Skip to main content



Another way of showing a trend in the data is via regression. You will learn more about this later in the program, but in brief if you are using linear regression you are fitting a straight line through the data, by choosing the line that minimizes a certain cost function (or penalty), commonly the sum squared deviations of the data to the line (least squares).

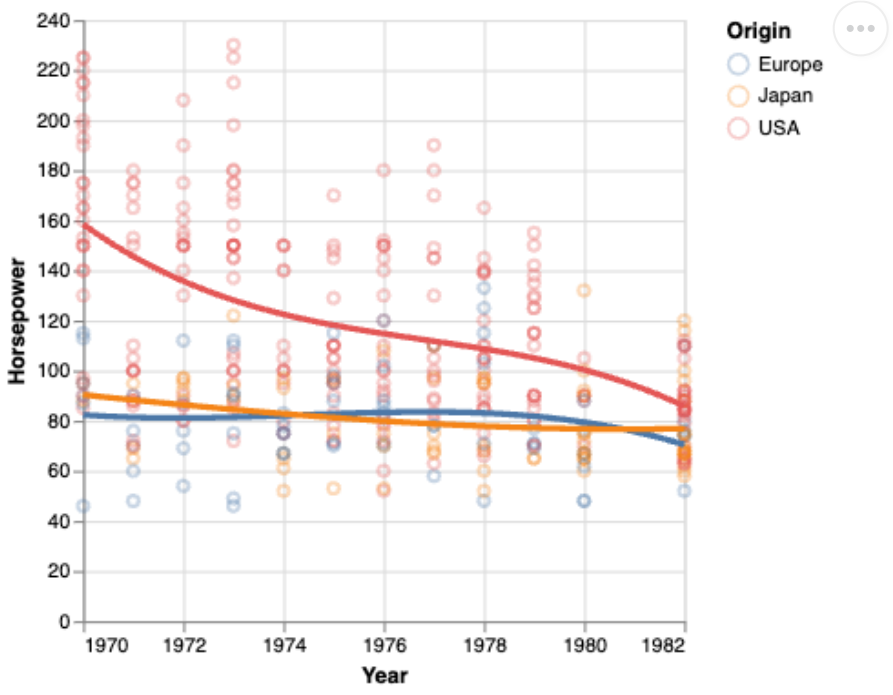
```
points +  points.mark_line(size=3).transform_regression(
    'Year',
    'Horsepower',
    groupby=['Origin']
)
```

Skip to main content



You are not limited to fitting straight lines, but can try fits that are quadratic, polynomial, etc.

```
points +  points.mark_line(size=3).transform_regression(
    'Year',
    'Horsepower',
    groupby=['Origin'],
    method='poly'
)
```



Skip to main content

Sometimes it is difficult to find a single regression equation that describes the entire data set. Instead, you can fit multiple equations (usually linear and quadratic) to smaller subsets of the data, and add them together to get the final line. This is called `loess` for locally estimated scatterplot smoothing (also called `lowess`, where the `w` stands for "weighted") and is conceptually similar to a moving/rolling average, which also only uses part of the data to create the trend line (the red line is the loess line in this plot, which is gradually built up from different subsets of the data):

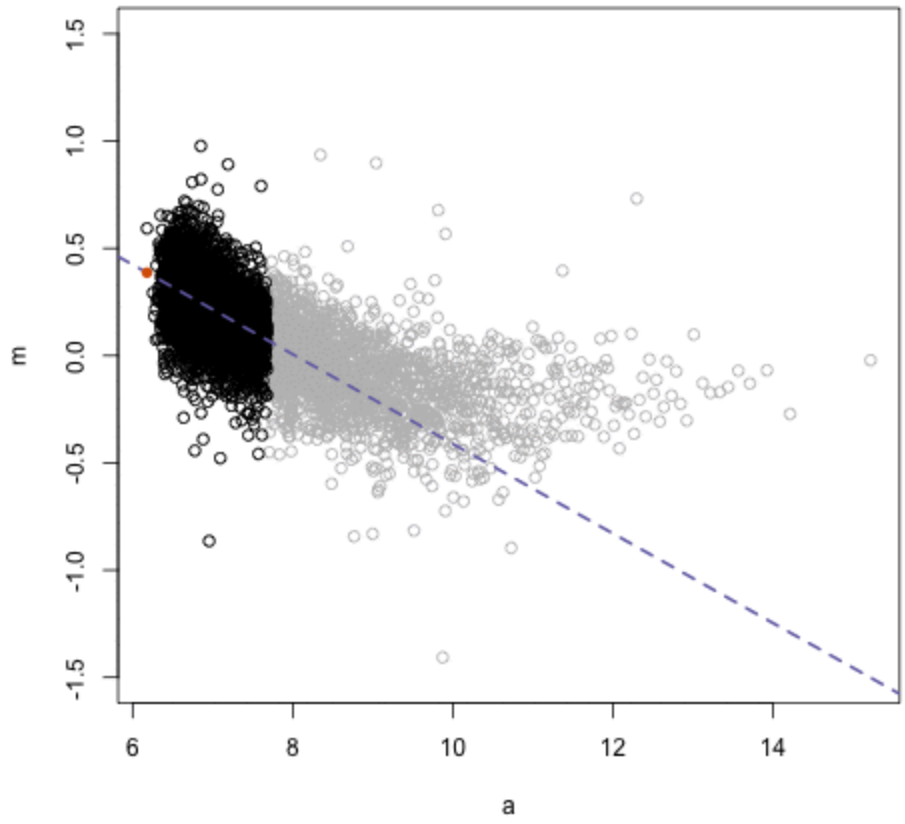
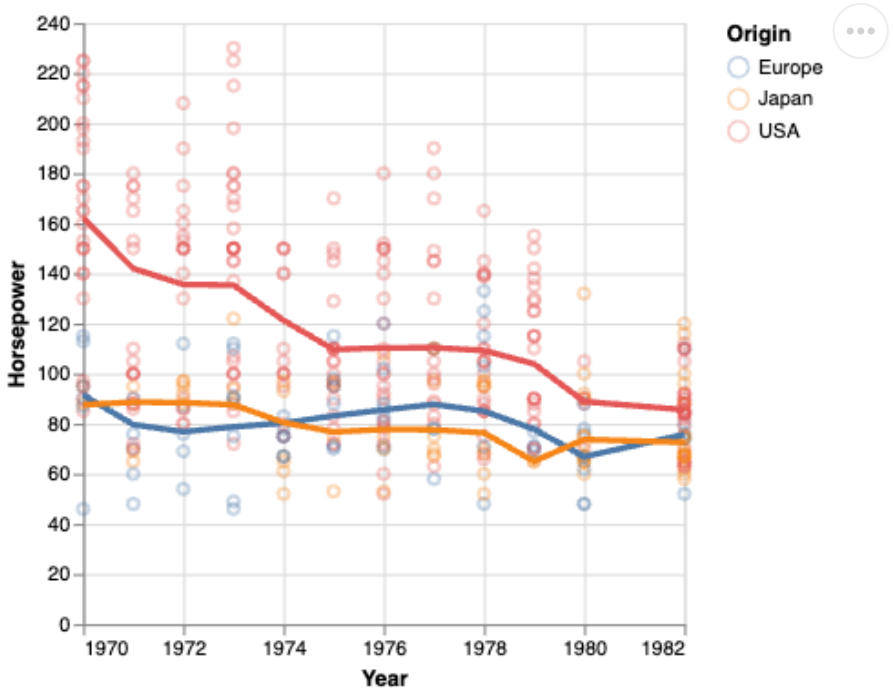


Image source: https://simplystatistics.org/posts/2017-08-08-code-for-my-educational-gifs/

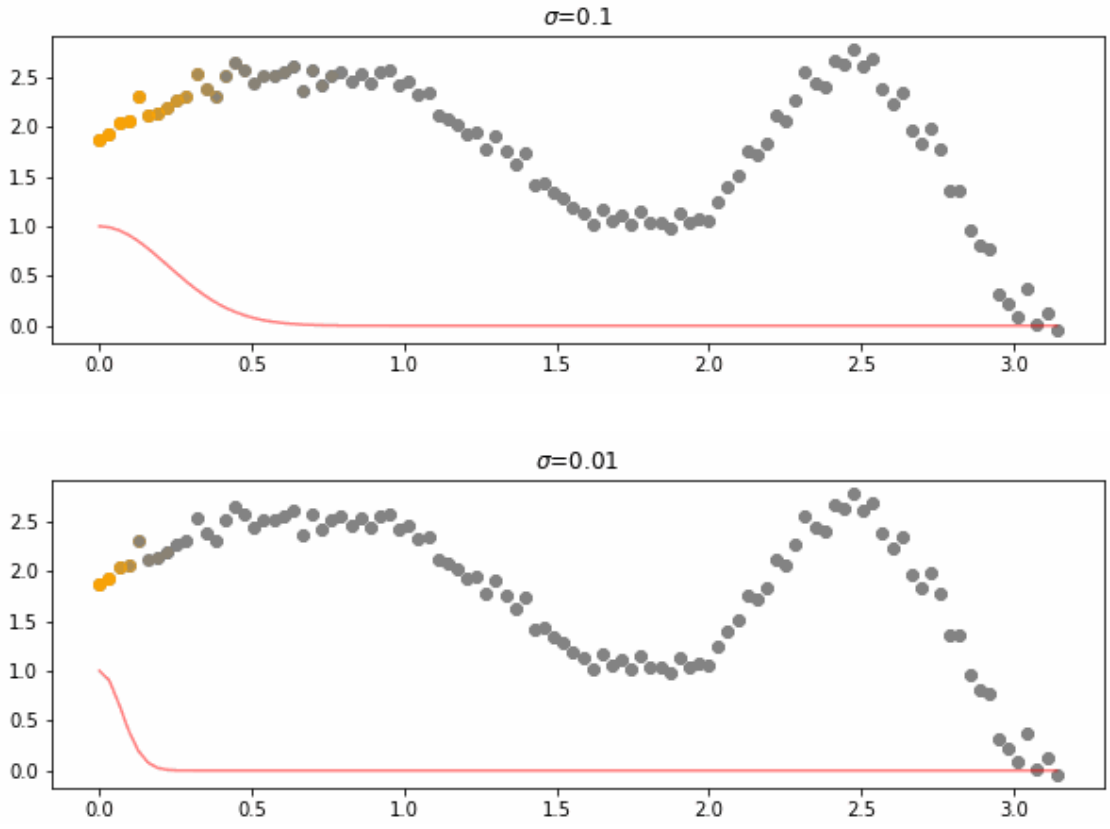
This is how we can create a loess line in Altair:

```
points +  points.mark_line(size=3).transform_loess(
    'Year',
    'Horsepower',
    groupby=['Origin']
)
```

Skip to main content



The bandwidth parameter controls how much the loess fit should be influenced by local variation in the data, similar to the effect of the bandwidth parameter for a KDE (the bandwidth is denoted by σ in this animation, and you can see that when the bandwidth is larger there is a larger subset of the data used for each regression fit which means that the loess line is smoother / more general):

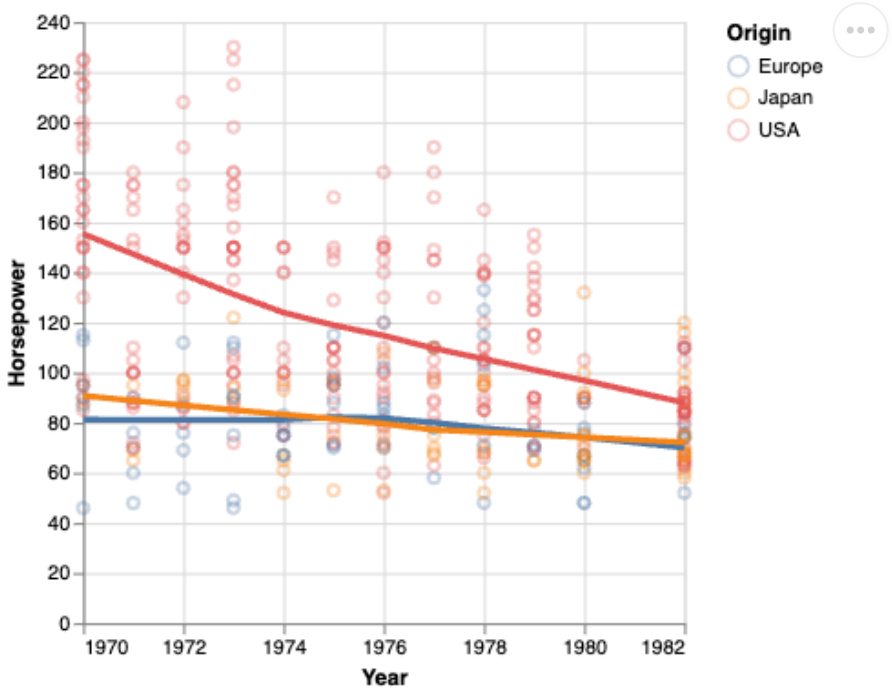


Skip to main content

Image source: https://scikit-lego.netlify.app/linear-models.html

In Altair a bandwidth of 1 corresponds to using all the data and will be similar to a linear regression (it won't be exactly the same because of some robustness optimization done behind the scenes).

```
# The default is 0.3 and 1 is often close to a linear fit.
points +  points.mark_line(size=3).transform_loess(
    'Year',
    'Horsepower',
    groupby=['Origin'],
    bandwidth=0.8
)
```

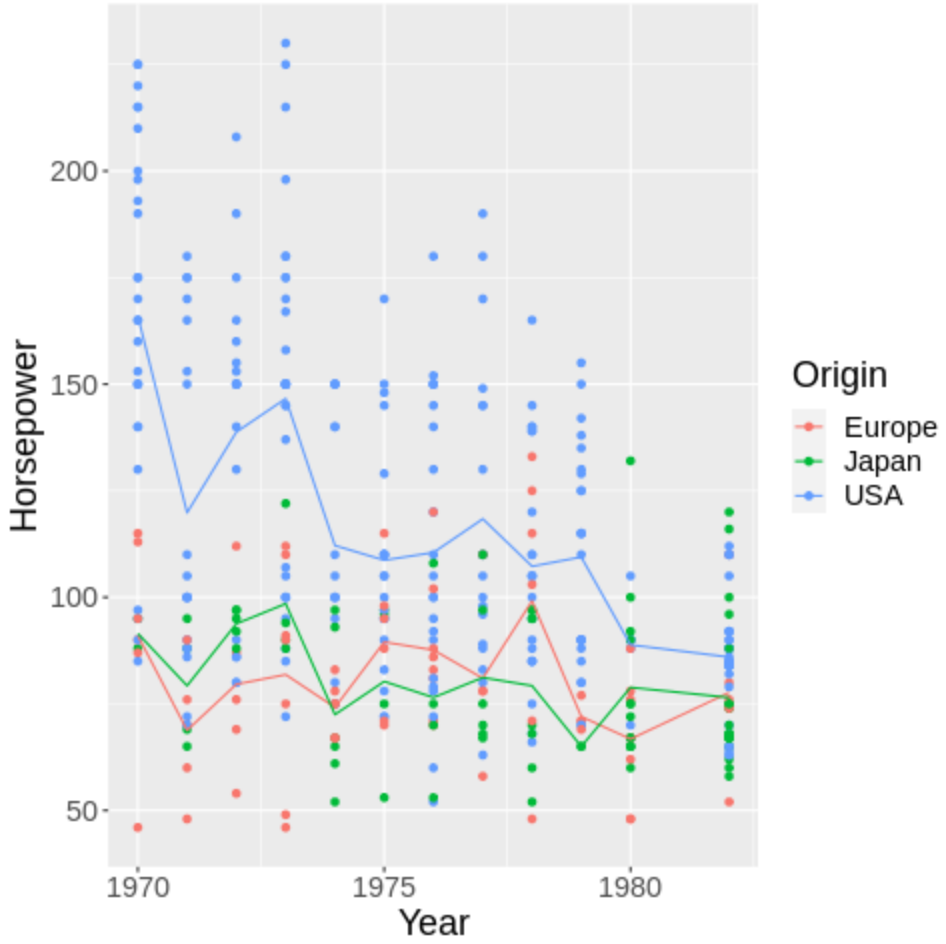


5.6.2. R

Using the mean as a trendline.

```
%%R -i cars
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin) +
    geom_point() +
    geom_line(stat = 'summary', fun = 'mean')
```

Skip to main content

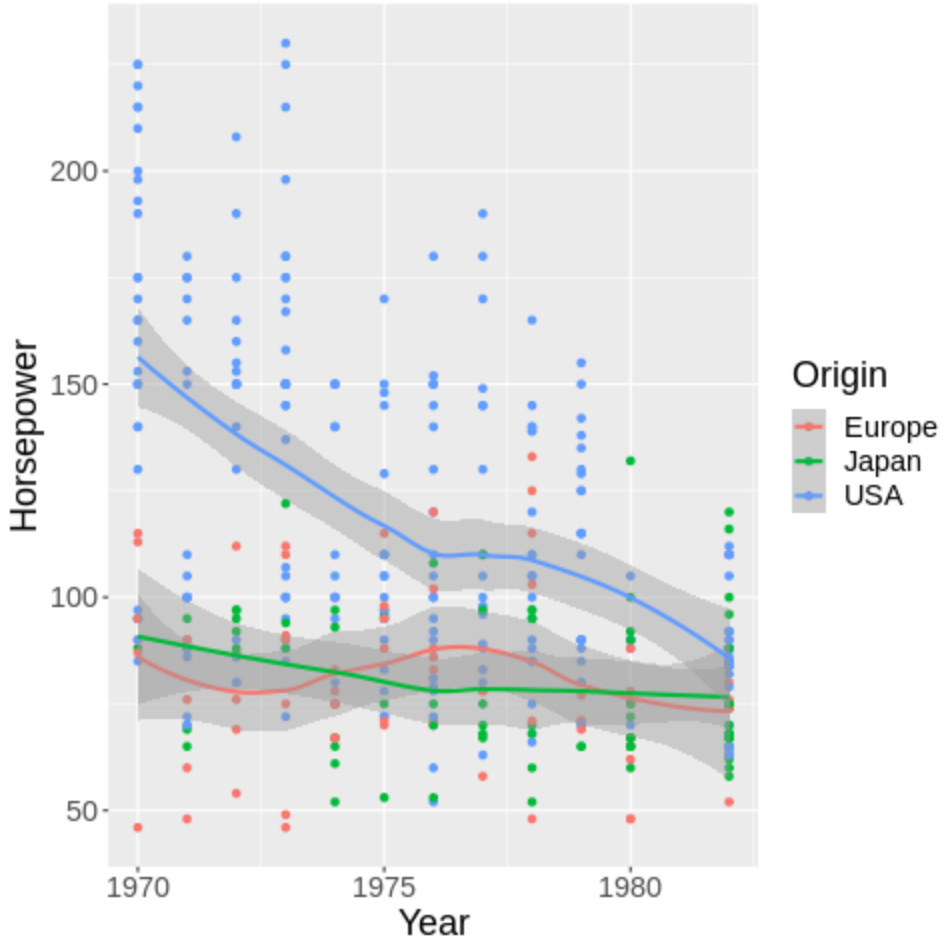


`geom_smooth` creates a loess trendline by default. The shaded gray area is the 95% confidence interval of the fitted line.

```
%%R -i cars
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin) +
    geom_point() +
    geom_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

Skip to main content

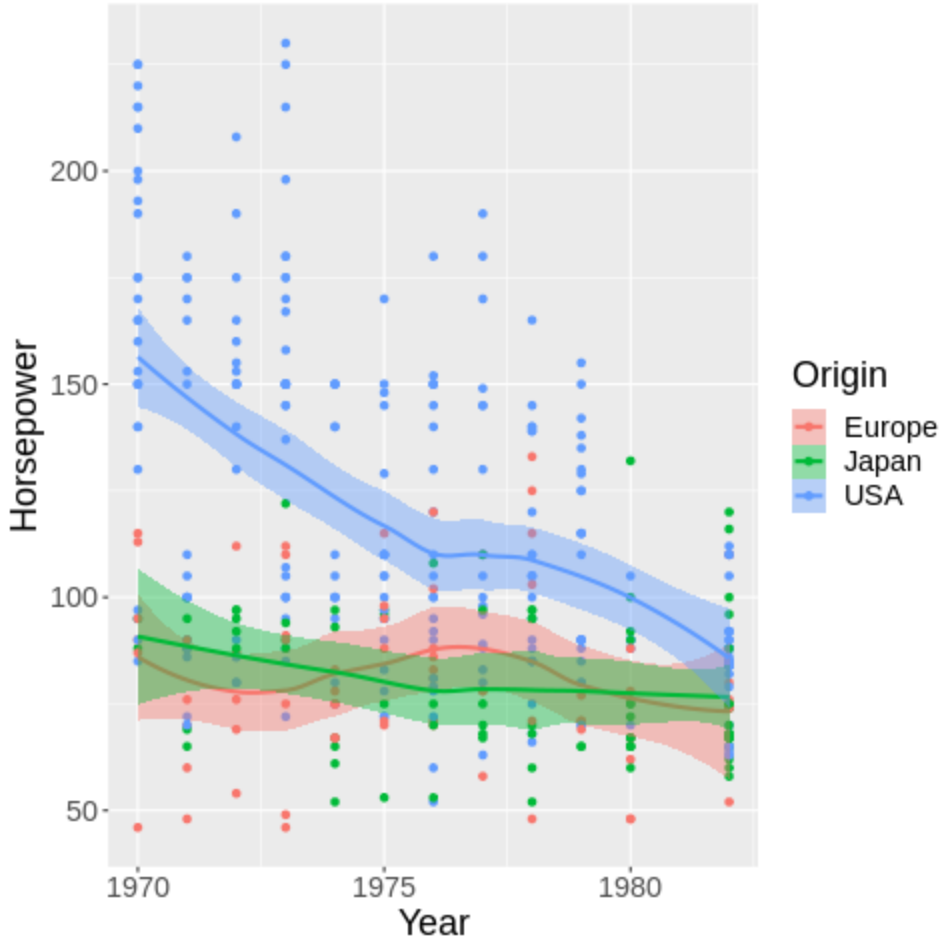


We can color the confidence interval the same as the lines.

```
%%R -i cars
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin,
        fill = Origin) +
    geom_point() +
    geom_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

Skip to main content

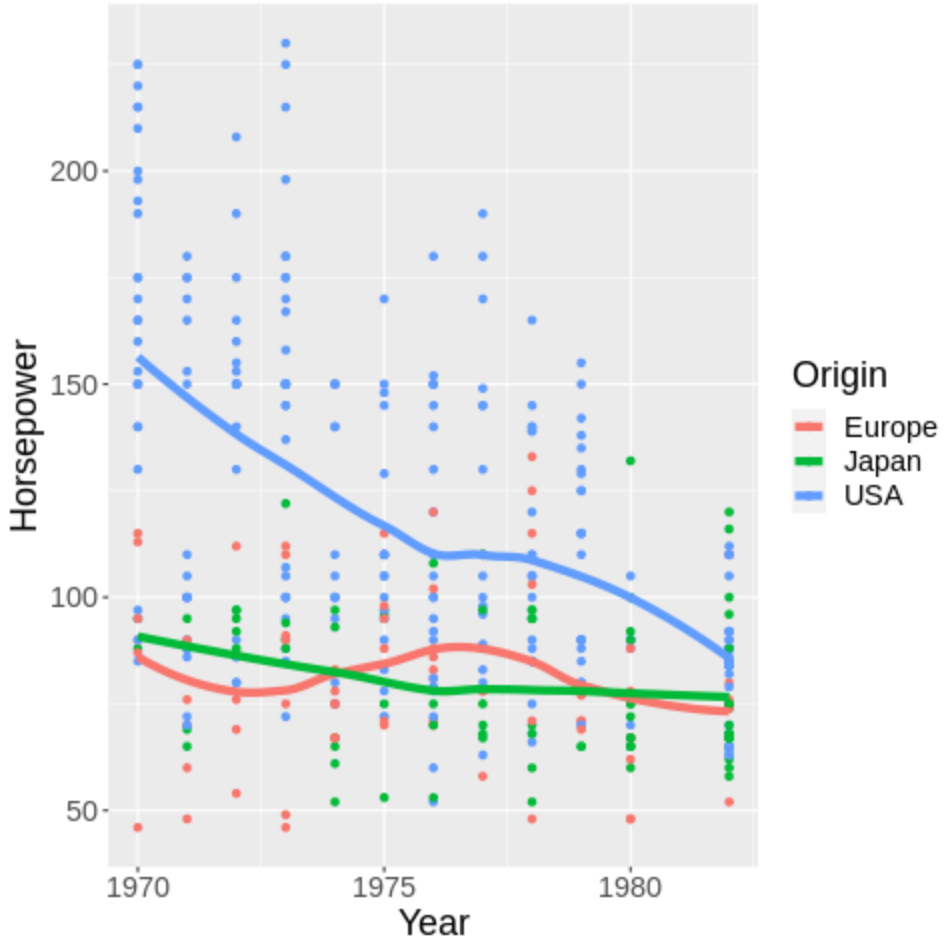


And also remove it.

```
%%R -i cars
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin,
        fill = Origin) +
    geom_point() +
    geom_smooth(se = FALSE, size = 2)
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

Skip to main content

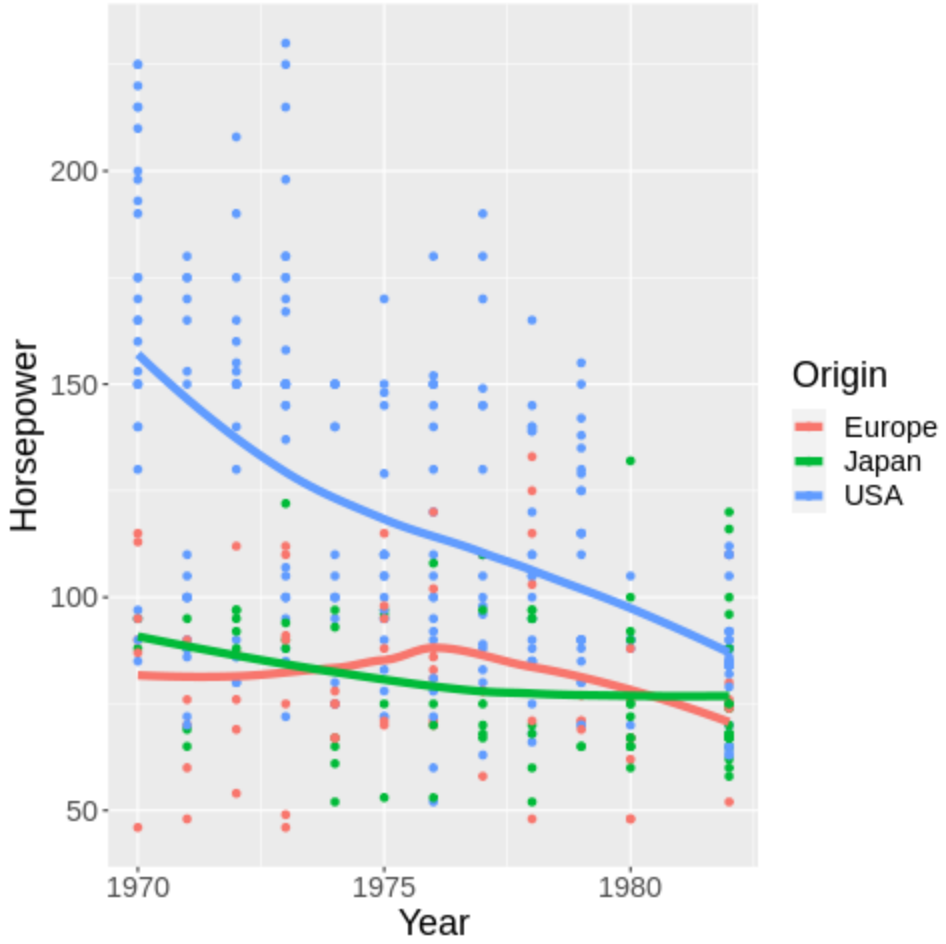


Similar to the `bandwidth` in Altair, you can set the `span` in `geom_smooth` to alter how sensitive the loess fit is to local variation.

```
%%R
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin,
        fill = Origin) +
    geom_point() +
    geom_smooth(se = FALSE, size = 2, span = 1)
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

Skip to main content

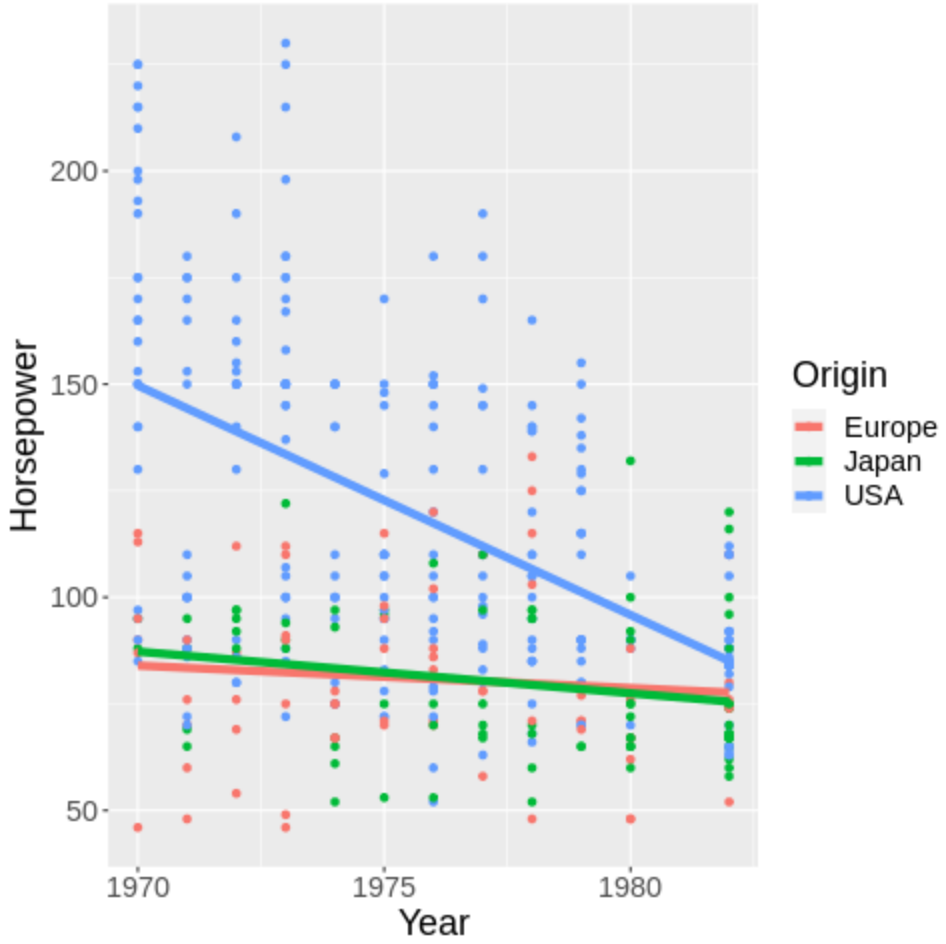


If you want a linear regression instead of loess you can set the method to `lm` (linear model).

```
%%R
ggplot(cars) +
    aes(x = Year,
        y = Horsepower,
        color = Origin,
        fill = Origin) +
    geom_point() +
    geom_smooth(se = FALSE, size = 2, method = 'lm')
```

```
`geom_smooth()` using formula = 'y ~ x'
```

Skip to main content



5.7. When to choose which trendline?

- If it is important that the line has values that are easy to interpret, choose a rolling mean (or maybe a mean if it is not too noisy). These are also the most straightforward trendlines when communicating data to a general audience.
- If you think a simple line equation (e.g. linear) describes your data well, this can be advantageous since you would know that your data follows a set pattern, and it is easy to predict how the data behaves outside the values you have collected (of course still with more uncertainty the further away from your data you predict).
- If you are mainly interested in highlighting a trend in the current data, and the two situations described above are not of great importance for your figure, then a loess line could be suitable. It has the advantage that it describes trends in data very “naturally”, meaning that it highlights patterns we would tend to highlight ourselves in qualitative assessment. It also less strict in its statistical assumption compared to e.g. a linear regression, so you don’t have to worry about finding the correct equation for the line, and assessing whether your data truly follows that equation globally. A downside is that it is

Skip to main content

Previous
4. Exploratory data analysis (EDA)

Next
6. Color theory and application