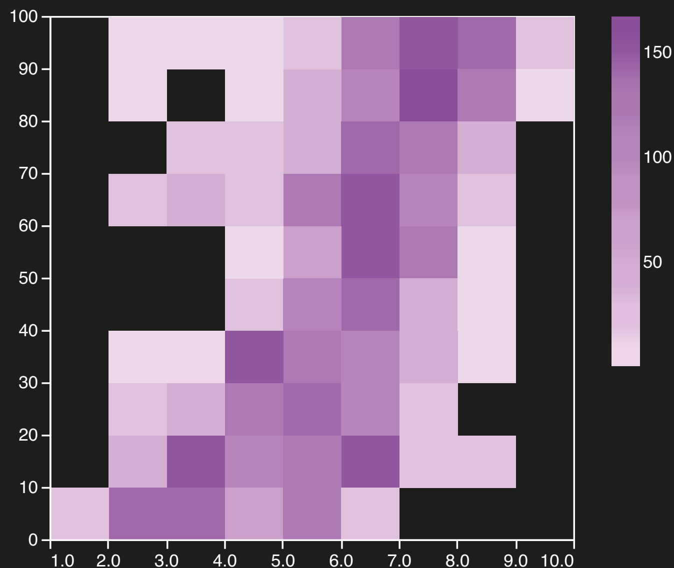
Data Visualization with Vega-Altair 5



Pere-Pau Vázquez

UPCGRAU 84

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Data Visualization with Vega-Altair 5

Pere-Pau Vázquez

UPCGRAU 84

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Primera edició octubre de 2024

© Pere-Pau Vázquez, 2024
© Iniciativa Digital Politècnica, 2024
Oficina de Publicacions Acadèmiques Digitals de la UPC
Edificio K2M, Planta S1, Despacho S103-S104
Jordi Girona 1-3, 08034 Barcelona
Tel.: 934 015 885
www.upc.edu/idp
E-mail: info.idp@upc.edu

Producció: Service Point
Pau Casals, 161-163
08820 El Prat de Llobregat (Barcelona)

ISBN:978-84-10008-89-2
ISBN digital: 978-84-10008-90-8
DL: B 6215-2024
DOI: 10.5821/ebook-9788410008908

Qualsevol forma de reproducció, distribució, comunicació pública o transformació d'aquesta obra només es pot fer amb l'autorització dels seus titulars, excepte l'excepció prevista a la llei.

Preface

In today's world, data are ubiquitous, and the ability to transform raw numbers into meaningful insights is more crucial than ever. Data visualization stands at the intersection of data science and storytelling, enabling us to detect patterns, trends, and outliers that may otherwise remain concealed within spreadsheets and databases. This book explores the power of data visualization through the lens of Altair, a declarative statistical visualization library in Python.

Altair offers a unique approach to creating visualizations. Its declarative nature allows users to specify what they want to see, rather than how to render it. This simplicity and elegance make Altair an excellent tool for both beginners and experienced data scientists. By focusing on the "what" rather than the "how," Altair enables users to generate complex visualizations with minimal code, allowing them to concentrate on the insights their data reveals.

This book introduces the fundamentals of data visualization, from basic charts to advanced interactive visualizations. The goal is not to cover every aspect of the library, but to provide a comprehensive guide to creating visualizations using the most common chart types in data visualization. Each chapter includes hands-on examples. By the end of this book, you will be able to create sophisticated visualizations with cross-interactions that can be embedded in web pages or function as standalone applications.

The charts presented in this book are created using minimal coding, sufficient to demonstrate the discussed features. Consequently, some charts may require fine-tuning, such as making the axis labels or legends more comprehensible or adjusting their capitalization. While the examples are kept simple, the text also provides guidance on configuring these variables.

Contents

Preface ... 5

Contents ... 7

1. Introduction ... 9
- 1.1. Altair versions ... 9
- 1.2. Declarative language ... 10
- 1.3. Alternatives ... 10

2. Altair library ... 13
- 2.1. Basic chart design ... 13
- 2.2. Visualization pipeline ... 14

3. Data specification ... 15
- 3.1. Basic data specification ... 15
- 3.2. Wide form vs. long form ... 16
- 3.3. Data types ... 18

4. Marks ... 21
- 4.1. Basic marks ... 21
- 4.2. Composite marks ... 27

5. Channels ... 31
- 5.1. Channel encoding ... 31
- 5.2. Channel options ... 38
- 5.3. Customization options ... 48
- 5.4. Multiple charts: simple combinations ... 50

6. Charts ... 57
- 6.1. Basic chart types ... 57
- 6.2. Variations over simple charts ... 57

7. Advanced chart types ... 65

8. Data transformations ... 79
- 8.1. Basics ... 79
- 8.2. Aggregate transforms ... 80

8.3. Bin transforms ... 81
8.4. Transforming data through calculations ... 85
8.5. Time manipulations ... 87
8.6. Filter transformation ... 92
8.7. Lookup transform ... 96
8.8. Regression transform ... 100

9. Tips and tricks ... 103
9.1. Loading large datasets ... 103
9.2. Adding labels ... 103
9.3. Customizing axes ... 106
9.4. Saving charts ... 106
9.5. Plotting graphical elements ... 107
9.6. Plotting real images ... 108

10. Interaction basics ... 109
10.1. Basic interaction: Pan and zoom ... 109
10.2. Basic interaction: Filter based on parameters ... 111

11. Selection ... 115
11.1. Individual selection ... 115
11.2. Interval selection ... 117
11.3. Selecting by fields or encodings ... 123

12. Binding interactions to user input ... 127
12.1. Sliders ... 127
12.2. Dropdown menus ... 128
12.3. Other widgets ... 132
12.4. Responsive charts ... 135
12.5. Using widgets in creative ways ... 136

13. Compound charts ... 141
13.1. Repeated charts ... 142
13.2. Faceted charts ... 143

14. Advanced maps ... 149

15. Interactive visualization of very large datasets ... 153

16. Moving forward ... 157

1
Introduction

Vega-Altair, formerly known as Altair, is a library for declarative visualization in Python. As of its current version, it supports only Python 3.6 and higher, due to the discontinuation of earlier Python versions. Altair was initially released in July 2016, and its official documentation can be found at https://altair-viz.github.io/index.html.

Architecturally, Altair generates Vega code, which is then executed in a JavaScript environment. Vega, along with its lighter counterpart, Vega-Lite, is a declarative language for interactive graphics developed by researchers at the University of Washington. Information about the Vega Lite specification and its ecosystem of tools can be found at https://vega.github.io/vega-lite/.

While there are alternative visualization libraries in Python, the majority face substantial limitations such as:

- Requiring excessive programming effort
- Inadequate support for interactivity
- A steep learning curve
- Limited functionality

To address these challenges, the developers of Altair focused on creating a library that is both simple and powerful. For users needing maximum flexibility, such as the ability to define custom shapes for all marks, JavaScript's D3 library may be a better alternative.

1.1. Altair versions

Altair was officially upgraded to version 5 in 2023, with the current version being 5.4.1, released in August 2024. Some tools, such as Google Colab, may require you to manually update to the latest version, as the default installation may be outdated.

1.2. Declarative language

In programming, two primary paradigms are commonly applied: imperative and declarative.

The **imperative** paradigm requires the user to explicitly define **how** tasks are accomplished, specifying each step necessary to achieve the result.

In contrast, the **declarative** paradigm focuses on specifying **what** needs to be done, while the system handles the details of how it is realized.

Because declarative languages require less code, they often offer advantages in usability. Additionally, they make it easier to express solutions in terms of desired outcomes rather than specific processes.

1.3. Alternatives

Alternative methods exist for accomplishing the same tasks in Python. The main difference between libraries lies in the level of detail required to produce the desired output. When selecting a library, it is important to consider several factors such as the types of charts supported, the degree of configuration available, and the types of interactions offered. Popular Python alternatives to Altair include Seaborn, Bokeh, Matplotlib, and Plotly.

Beyond Python, other alternatives can be found, such as D3, a powerful JavaScript library. However, designing visualizations from scratch with D3 typically requires far more lines of code compared to Altair. On the other hand, out-of-the-box tools like Tableau or PowerBI allow users to create visualizations without programming, although these platforms often limit the level of customization and interactivity available for creating charts.

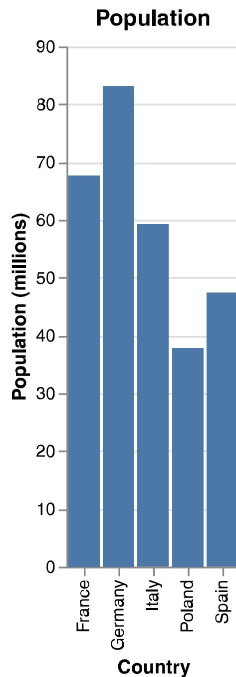
For example, generating a simple bar chart in D3 can require dozens of lines of code because it requires manual specification of even the most minimal details like chart dimensions, margins, and axis scales. In contrast, higher-level libraries such as Altair handle many of these decisions by default, enabling the creation of charts with just a few lines of code.

To illustrate, a simple bar chart in Altair requires fewer than a dozen lines:

```
# Import libraries
import altair as alt
```

```
import pandas as pd

# Data
data = pd.DataFrame(
  {'Country': ['Germany', 'France', 'Italy', 'Spain', 'Poland'],
  'Population (millions)': [83.15, 67.65, 59.23, 47.39, 37.84]})

# Creating the chart, with mark_bar to define a bar chart
alt.Chart(data).mark_bar().encode(
  alt.X('Country:N'),                # Categories in X axis
  alt.Y('Population (millions):Q'), # Length of the bars in Y
).properties(title = 'Population')       # Title of the chart
```

The previous code generates the following chart:



The Altair method Chart takes Dataframe as a parameter that contains the data, allowing users to define the marks used for visual encoding and to configure those marks accordingly.

2
Altair library

Altair is a Python library that operates with Vega-Lite, providing various data structures and methods that make converting to Vega-Lite syntax seamless for users. This means that the library's features are directly tied to the ongoing development of Vega-Lite, which is continually evolving.

When using Altair, the primary task is to load data and create one or more charts that effectively display the information as desired.

Typically, developers iterate through the design process by first creating one or more basic charts, then combining them into various views, and finally devising interaction methods.

2.1. Basic chart design

Altair charts are created using the *Chart* function, which takes the source data as a parameter and defines the type of chart and its visual properties. It includes two important methods that can be concatenated: *mark_** and *encode*. The *mark_** method has various options for defining the type of mark to be used (e.g., *mark_bar* for bar charts). The *encode* method allows us to specify the visual aspects of the marks (visual variables) in the chart and their arrangement.

To produce different types of charts, we will modify both the marks and their configurations. The *encode* function not only determines which variables are used for the X and Y axes but also lets us define the visual attributes of the marks (e.g., color, size, transparency).

While Altair supports a limited range of chart types and does not accommodate highly complex charts like Sankey diagrams or networks, it offers all the most common charts (e.g., bar charts, line charts, scatter plots, pie charts, choro-

pleths). Additionally, several advanced charts are also possible, such as violin plots, strip charts, and graduated symbol maps. Altair includes features for creating multiple views and linked interactions, which will be explained later in this tutorial.

Now that we have an understanding of what visualization in Altair looks like, we will explore it further, beginning with data specifications: how data are read in Altair and the modifications we can make to better suit our needs.

2.2. Visualization pipeline

When designing visualization applications, the typical process includes the following steps:

1. Understanding the problem
2. Gathering data
3. Cleaning data
4. Visual encoding design
5. View design
6. Interaction design
7. Evaluation

The first step involves identifying the problem to be solved. The second and third steps are not addressed by Altair; however, other tools such as OpenRefine or Data Wrangler may be helpful. Additionally, some Python libraries can assist with data cleaning.

Vega-Altair supports steps 4 to 6, albeit to varying degrees. While complete flexibility in designing marks may not be achievable, a wide range of options is available for channel and mark customization. The interaction capabilities are also somewhat limited, but they are sufficient for addressing a diverse array of issues.

3
Data specification

3.1. Basic data specification

Data are specified for each top-level chart object using a dataset encoded in one of three ways:

- A Pandas DataFrame
- A Data or related object
- A URL string pointing to a *json* or *csv* formatted text file

A Pandas Dataframe is a two-dimensional, size-mutable tabular data structure with labeled axes (rows and columns). (For more details on creating these datasets, refer to the Pandas documentation at https://pandas.pydata.org). For example, you can create a simple dataset using the following code:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'Category': ['A', 'B', 'C', 'D', 'E', 'F'],
                     'Quantity': [5, 3, 6, 7, 2, 8]})
```

Data are a class that can be used to specify data using JSON-style records. To create the same dataset using the Data class, we define it as follows:

```
import altair as alt

data = alt.Data(values=[{'x': 'A', 'y': 5},
                        {'x': 'B', 'y': 3},
                        {'x': 'C', 'y': 6},
                        {'x': 'D', 'y': 7},
                        {'x': 'E', 'y': 2}])
```

The primary difference between the two encodings is that Altair can infer the data type from a DataFrame. However, when using Data, we must explicitly specify the types of the different elements, although we can override the detected types from the DataFrame using the same specification procedure during Chart construction.

For datasets in the Vega library, we can also input data form a URL, provided the data are stored in a JSON file. The following code accomplishes this:

```
from vega_datasets import data

source = data.stocks.url
```

As in the previous case, we need to specify the data types because they cannot be extracted from the file referenced by the URL string.

We can also read data from an external URL, as shown below:

```
gapminder = pd.read_csv('https://raw.githubusercontent.com/
vega/vega/master/docs/data/gapminder-health-income.csv')
```

3.2. Wide form vs. long form

Two common conventions are observed for storing data in a DataFrame: long form and wide form.

- **Wide-form** data has one row per *independent variable*, with metadata recorded in the *row and column labels*.
- **Long-form data** has one row per *observation*, with metadata recorded as *values* within the table.

An example of wide form is provided below:

```
wide_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01',
                                   '2007-12-01'],
                          'AAPL': [189.95, 182.22, 198.08],
                          'AMZN': [89.15, 90.56, 92.64]})
print(wide_form)
```

```
         Date    AAPL   AMZN
0  2007-10-01  189.95  89.15
1  2007-11-01  182.22  90.56
2  2007-12-01  198.08  92.64
```

```
In contrast, long-form data would store
the information as follows:
long_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01',
                                   '2007-12-01', '2007-10-01',
                                   '2007-11-01', '2007-12-01'],
                          'company': ['AAPL', 'AAPL', 'AAPL',
                                      'AMZN', 'AMZN', 'AMZN'],
                          'price': [189.95, 182.22, 198.08,
                                    89.15, 90.56, 92.64]})
print(long_form)
```

```
         Date company   price
0  2007-10-01    AAPL  189.95
1  2007-11-01    AAPL  182.22
2  2007-12-01    AAPL  198.08
3  2007-10-01    AMZN   89.15
4  2007-11-01    AMZN   90.56
5  2007-12-01    AMZN   92.64
```

Altair works better with the long form version. We can specify that long-form is used in the creation call, as shown below.

```
alt.Chart(long_form).mark_line().encode(
    x = 'Date:T',
    y = 'price:Q',
    color = 'company:N'
)
```

We can convert from wide form to long form using Pandas' *melt* method or directly in Altair by using the *transform_fold* method.

```
alt.Chart(wide_form).transform_fold(
    ['AAPL', 'AMZN'],
    as_ = ['company', 'price']
).mark_line().encode(
    x = 'Date:T',
    y = 'price:Q',
    color = 'company:N'
)
```

If we want to generate charts directly from wide form, we would need to create several charts (e.g., one for each company in this example) and use layering to plot them together, as in the following example:

```
wide_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01',
                                   '2007-12-01'],
                          'AAPL': [189.95, 182.22, 198.08],
                          'AMZN': [89.15, 90.56, 92.64],
                          'GOOG': [707.00, 693.00, 691.48]})

ch1 = alt.Chart(wide_form).mark_line().encode(
    x = 'Date:T',
    y = 'AAPL:Q',
    color = alt.value('red')
)

ch2 = alt.Chart(wide_form).mark_line().encode(
    x = 'Date:T',
    y = 'AMZN:Q',
    color = alt.value('blue')
)

ch3 = alt.Chart(wide_form).mark_line().encode(
    x = 'Date:T',
    y = 'GOOG:Q',
    color = alt.value('green')
)

ch1 + ch2 + ch3
```

We will discuss compound charts later in this book.

3.3. Data types

Altair supports four basic data types (encodings):

- Quantitative
- Ordinal
- Nominal
- Temporal

These data types can be specified explicitly—when not available from the DataFrame or to override the detected type—using either verbose (e.g., “temporal”) or shorthand (e.g., “T”) in the *encode* method of the chart.

An example with explicit encoding is shown below:

```
from vega_datasets import data

cars = data.cars.url

alt.Chart(cars).mark_point().encode(
    alt.X('Acceleration', type='quantitative'),
    alt.Y('Miles_per_Gallon', type='quantitative'),
    alt.Color('Origin', type='nominal')
)
```

This is equivalent to the following encoding:

```
alt.Chart(cars).mark_point().encode(
    x = 'Acceleration:Q',
    y = 'Miles_per_Gallon:Q',
    color = 'Origin:N'
)
```

4
Marks

To create a visualization, we must transform the data into visual representations. These representations consist of two distinct types of parameters: the geometric element employed (called a **mark) and its visual attributes (known as channels or visual variables).**

For the initial examples, we will use simple charts. More advanced techniques will be presented later.

4.1. Basic marks

Altair supports a comprehensive set of basic mark types:

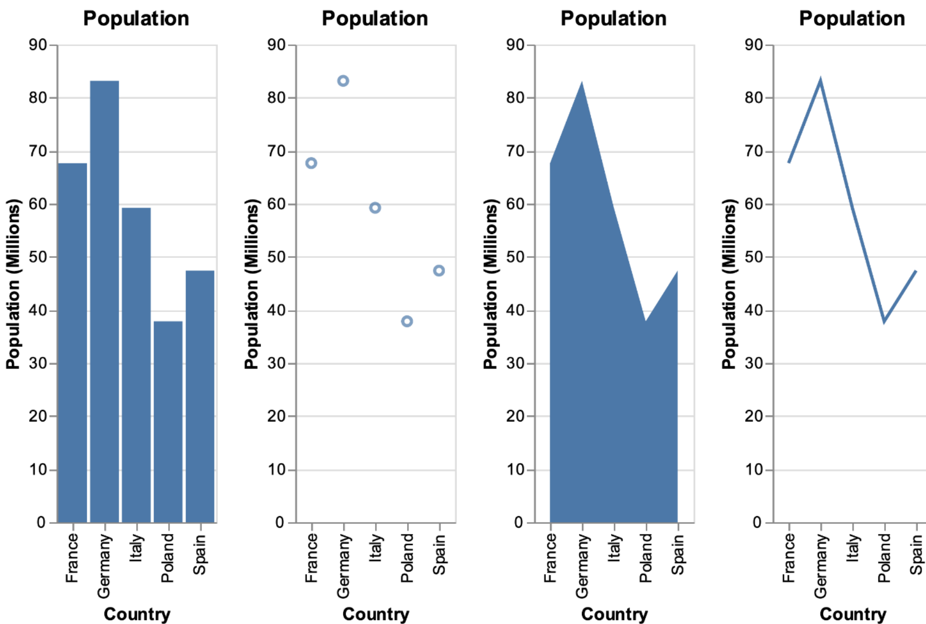
- Arc: Used to encode pie charts and donut charts
- Area: Used to plot filled area charts
- Bar: For all types of bar charts and histograms
- Circle: Used for scatter plots
- Geoshape: Used to encode spatial data
- Image: Used in scatter plots to display images instead of points
- Line: For line charts
- Point: Used for scatter plots but allows for shape customization
- Rect: A filled rectangle, usually for heatmaps
- Rule: A vertical or horizontal line spanning the axis
- Square: A scatter plot with square marks
- Text: Used to display text strings
- Tick: A vertical or horizontal tick mark
- Trail: A line with variable widths

These basic types serve to encode simple elements in charts, but composite elements are also available, as will be discussed later.

Marks are defined using the *mark_** method, where the symbol "*" is replaced with the mark name. For example, to create a **bar chart**, we use the *mark_bar()* method of the *Chart* type. The following example constructs a bar chart displaying the population of a set of countries:

```
data = pd.DataFrame(
   {'Country': ['Germany', 'France', 'Italy', 'Spain', 'Poland'],
    'Population (Millions)': [83.15, 67.65, 59.23, 47.39, 37.84]})

alt.Chart(data).mark_bar().encode(
    alt.X('Country:N'),
    alt.Y('Population (Millions):Q'),
).properties(title = 'Population')
```

By changing the mark to point (*mark_point*), area (*mark_area*), or line (*mark_line*), we will obtain these variants of the same chart: **scatter plot**, **area chart**, and **line chart**, respectively.



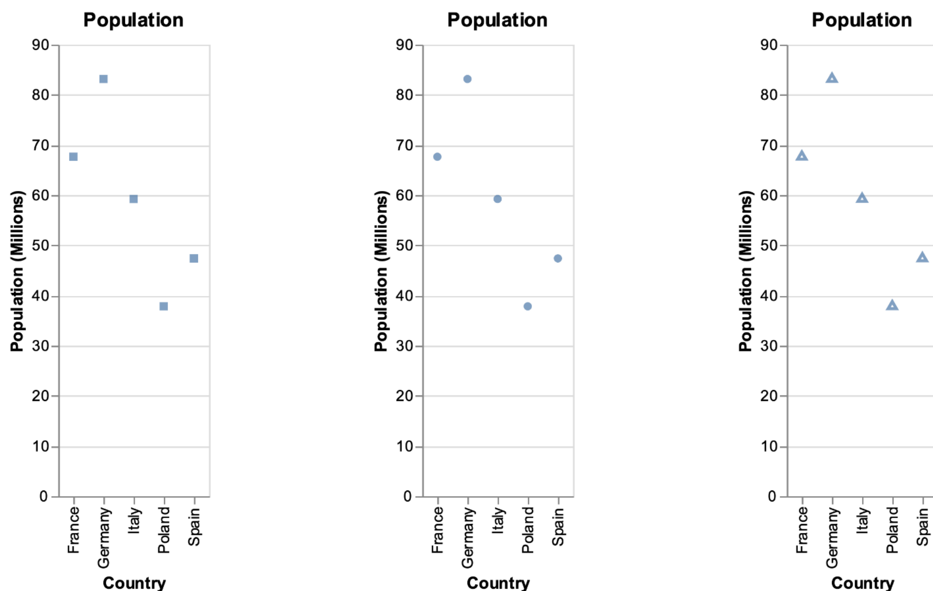
Although Altair allows us to encode data in specific ways, this does not guarantee that the resulting visualizations are sufficient. In the examples above, only the two leftmost charts are appropriate, with the first being more so than the second.

The rightmost charts may mislead the user into interpreting the data as having continuity between countries, which is incorrect. Thus, it is crucial to ensure that charts are expressive, which is a term of art meaning that they effectively communicate complex information or patterns in data in a clear and impactful way.

Additionally, the name that Altair assigns to a mark does not necessarily reflect its formal use in the chart. For instance, line charts technically use points as marks, with lines connecting them. This is intuitively clear when considering that, in a line chart, the encoded values within the data points represent positions rather than line lengths. Line lengths vary depending on the values of two consecutive data points. This principle applies to other encodings as well.

Points, squares, and circles can all be used to create similar scatter plots. In fact, when discussing shapes, we will see that various mark functions intersect with the visual channels.

The previous dataset can also be represented using squares or circles. Additionally, the shape of the *mark_point* can be defined as triangles by adding a parameter to the *mark_point* function (e.g., *mark_point*(*shape='triangle'*)). These variations are shown in the following figure for comparison.



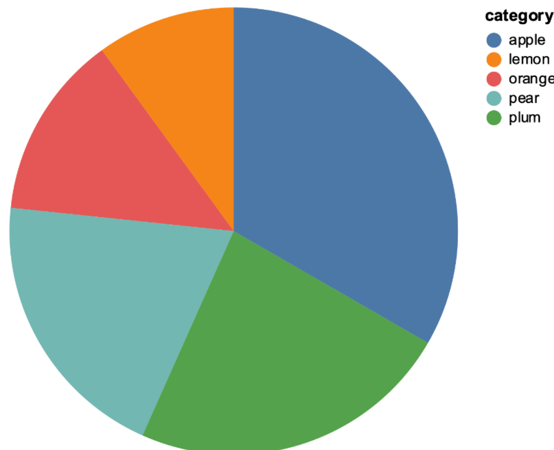
Another, widely used visualization technique is the **pie chart**. Pie charts were added to Altair relatively recently, in version 4.2. Despite their highly controversial

nature, and the fact that many visualization practitioners avoid using them, they appear in many infographics.

The most basic version that Altair supports is the regular pie chart using the *mark_angle* function. However, the default encoding may not be the most suitable. Data visualization experts recommend starting pie charts at the 12 o'clock position, with slices arranged clockwise in descending order of size. To achieve this, marks must be assigned in descending *order* using the order parameter of the *encode* function. An example is depicted next:

```
source = pd.DataFrame({"category": ['orange',
                        'pear', 'apple', 'lemon', 'plum'],
                        "value": [4, 6, 10, 3, 7]})

alt.Chart(source).mark_arc().encode(
    theta=alt.Theta("value"),
    order = alt.Order('value', sort = 'descending'),
    color="category:N"
)
```

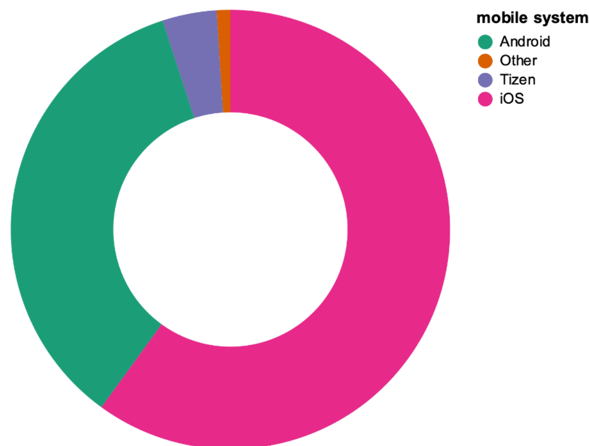
The result of this code is as follows:



However, pie charts have different variants, such as **donut charts**, which feature a hole in the center. To create a donut chart, we simply need to define the size of the hole using the *innerRadius* parameter, which can be passed to the arc mark as a configuration value, as shown in the following example:

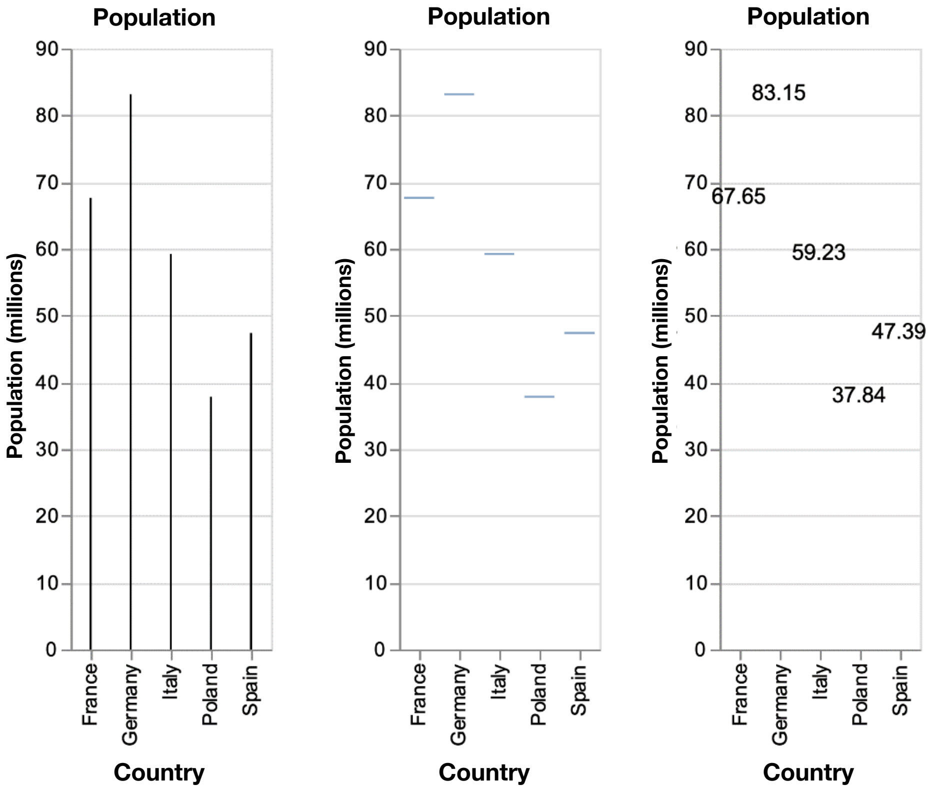
```
source = pd.DataFrame({"mobile system": ['iOS', 'Android',
                                          'Tizen', 'Other'],
                       "percentage": [60, 35, 4, 1]})

alt.Chart(source).mark_arc(innerRadius = 80).encode(
    theta= alt.Theta(field = 'percentage', type = 'quantitative'),
    order = alt.Order('percentage', sort = 'descending'),
    color= alt.Color(field="mobile system", type="nominal",
                     scale = alt.Scale(scheme = 'dark2')),
)
```

In this example, we also modified the default palette to use a different color scheme (dark2). A variety of color schemes are available, as described on Vega's website: https://vega.github.io/vega/docs/schemes/.



The arc mark also allows for more complex charts, such as the polar area chart, which will be discussed later.

Two other less familiar yet basic types of mark are rule and tick. The rule mark is typically used to create markers (e.g., reference lines for specific values) but can also be employed to create charts on its own, as shown next. The tick mark is another element often used to highlight reference values. As above, we compare these marks by illustrating how they plot the same data in a chart.



The text mark is highly useful for illustrating values, either directly within a chart or as a tooltip. In the previous example, we can use the text mark to display the population values:

```
data = pd.DataFrame(
   {'Country': ['Germany', 'France', 'Italy', 'Spain', 'Poland'],
    'Population (millions)': [83.15, 67.65, 59.23, 47.39, 37.84]})

alt.Chart(data).mark_text().encode(
    alt.X('Country:N'),
    alt.Y('Population (millions):Q'),
    text = 'Population (millions)'
).properties(title = 'Population')
```

We will deal with the other objects later in this book.

4.2. Composite marks

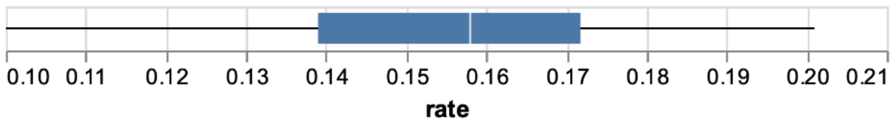
In addition to the basic marks, there are other specific types:

- Box Plot: Used to display boxplots
- Error Band: A continuous band around a line
- Error Bar: An error bar around a point

In the following example, we show the obesity rate of all the US states in a **boxplot.**

```
from vega_datasets import data

source = data.obesity()

alt.Chart(source).mark_boxplot().encode(
    alt.X("rate:Q", scale=alt.Scale(zero=False)),
)
```

The result is as follows:

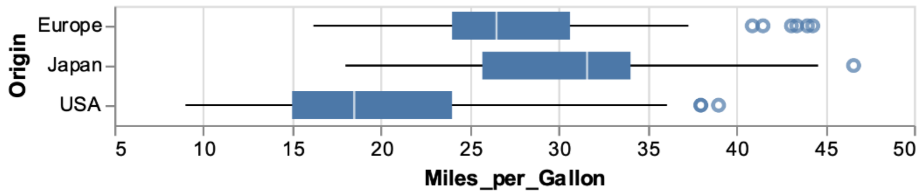


The default version of boxplots follows Tukey's approach, which has whiskers that span from the smallest to the largest data points within the range [Q1 – k * IQR, Q3 + k * IQR], where Q1 and Q3 are the first and third quartiles, respectively, and IQR is the interquartile range (Q3–Q1). The constant can be explicitly specified using the extent value; otherwise, it is defined as 1.5. In the previous example, no outliers were present outside the extent.

Data with outliers will show them as small circles:

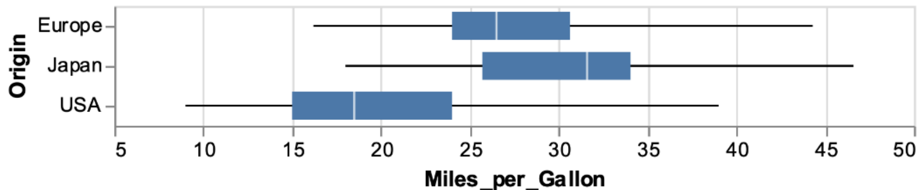
```
source = data.cars()

alt.Chart(source).mark_boxplot().encode(
    alt.X("Miles_per_Gallon:Q",scale=alt.Scale(zero=False)),
    alt.Y("Origin:N"),
)
```

As shown in this result:



If we modify the extent to cover the min-max range, the whiskers will span the entire range. The following example uses this approach:

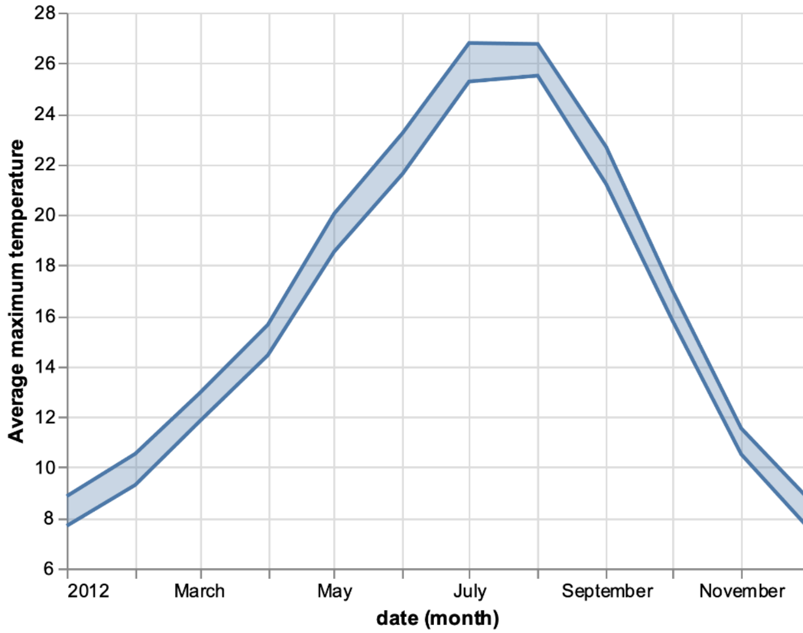
```
source = data.cars()

alt.Chart(source).mark_boxplot(extent = 'min-max').encode(
    alt.X("Miles_per_Gallon:Q", scale=alt.Scale(zero=False)),
    alt.Y("Origin:N"),
)
```



For this type of data, we can also use error bands. In the following example, the average maximum temperatures recorded per month in Seattle are displayed as a range:

```
from vega_datasets import data

source = data.seattle_weather()

alt.Chart(source).mark_errorband(extent="ci", borders=True).encode(
    alt.X("month(date)"),
    alt.Y(
        "average(temp_max):Q",
        scale=alt.Scale(zero=False),
        title="Average maximum temperature",
    ),
)
```

The result of the previous code is:



5
Channels

The visual attributes of marks are referred to as channels. Altair offers various ways to configure channels, such as position, shape, or color. To adjust their default values, we can either modify their visual properties using different *markRef* values or set their positions by defining fields like the x and y coordinates.

5.1. Channel encoding

Several aspects of marks can be configured. First, the position can be set using the following channels:

- **x**: X coordinates of the marks or the width of horizontal bars (and area marks).
- **y**: Y coordinates of the marks or the height of vertical bars (and area marks).
- **x2**: X2 coordinates for ranged shapes (area, bar, rect, and rule).
- **y2**: Y2 coordinates for ranged shapes (area, bar, rect, and rule).
- **longitude:** GPS longitude for geographical charts.
- **latitude:** GPS latitude for geographical charts.
- **longitude2:** Second longitude value for ranges in geographical charts.
- **latitude2:** Second latitude value for ranges in geographical charts.
- **xError:** X-axis error value.
- **yError:** Y-axis error value.
- **xError2:** Second x-axis error value.
- **yError2:** Second y-axis error value.
- **xOffset:** Offset for the x position.
- **yOffset:** Offset for the y position.
- **theta:** Start angle for arcs.
- **theta2:** End angle for arcs (given in radians), used for incomplete pie charts (e.g., Pacman-like charts).

To adjust the appearance of channels, the following properties can be modified:

- **angle**: Defines the angle of the mark, as in pie charts.
- **color**: Default color of the mark.
- **fill**: Color that fills the mark (overrides color).
- **fillOpacity**: Float indicating the opacity, ranging from 0 to 1.
- **opacity**: Opacity of the mark.
- **radius**: Radius of the mark (for radial charts).
- **shape**: Specifies the shape of point marks, including:
- *circle, square, cross, diamond, triangle (up, down, right, left), arrow, or wedge.*
- **size**: Size of the shape. For point, circle, and square marks, this refers to the pixel area.
- **stroke**: The stroke of the mark.
- **strokeDash**: Style of the stroke, typically used in line charts.
- **strokeOpacity**: Opacity of the line.
- **strokeWidth**: Width of the line.

Other specific encodings apply for text and tooltips:

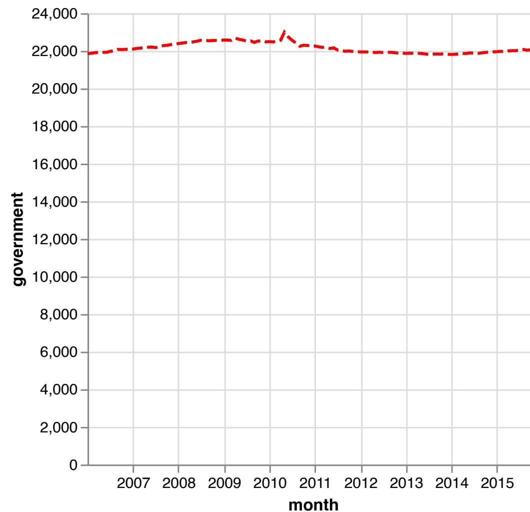
- **text**: The text to display for the mark.
- **tooltip**: The value displayed in the tooltip. This can include multiple fields.

Other useful properties for configuring the chart include “detail,” which can group elements, and “order,” which defines the sorting of elements within a channel.

In the following example, we modify the plot using both color and strokeDash:

```
from vega_datasets import data
df = data.us_employment.url
alt.Chart(df).mark_line(strokeDash=[7,3],color = “red”).encode(
    x=’month:T’,
    y=’government:Q’
)
```

The *stroke* property takes precedence over the *color* property. In the following chart, both properties are utilized, and the stroke width is also adjusted.

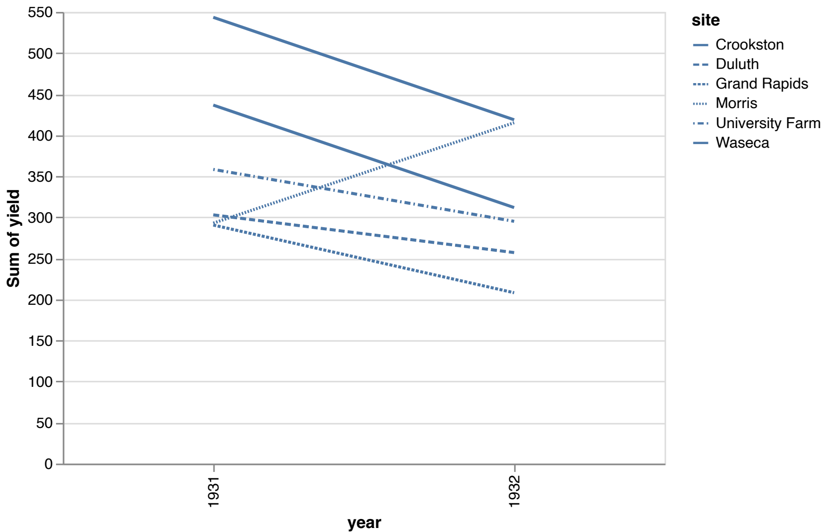



```
df = data.us_employment.url

alt.Chart(df).mark_line(strokeDash=[3,7], color = "red",
    stroke = 'purple', strokeWidth = 5,
    strokeOpacity = 0.5, size = 10,
).encode(
    x='month:T',
    y='government:Q'
)
```

We can also use this parameter to apply different styles to the various sources of barley in the chart, as shown next.

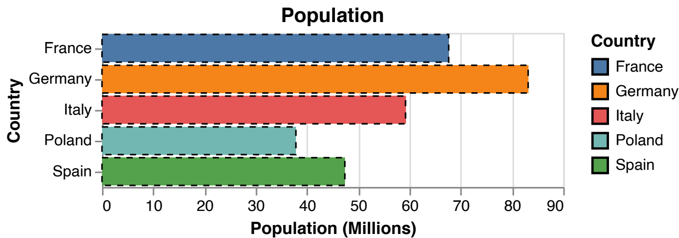
```
from vega_datasets import data

# Dimensions: year (2 values), site, yield, variety
source = data.barley()

alt.Chart(source).mark_line().encode(
    alt.X('year:O'),
    alt.Y('sum(yield):Q'),
    strokeDash='site:N',
).properties(width = 300)
```

The result is as follows:



The previous example resembles a slope chart, which consists of a line chart with only two samples along the x-axis.

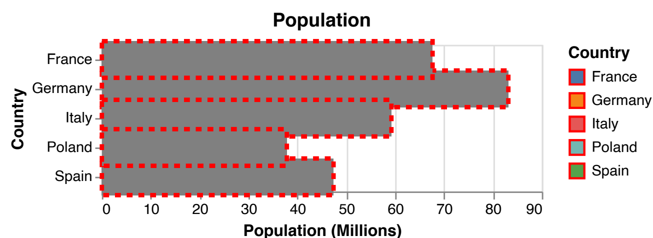
The stroke color can also be applied to other marks, such as bars, as shown in the following example, where we also adjusted the dimensions to create a horizontal bar chart:




```
data = pd.DataFrame(
    {'Country': ['Germany', 'France', 'Italy', 'Spain', 'Poland'],
     'Population (Millions)': [83.15, 67.65, 59.23, 47.39, 37.84]})

alt.Chart(data).mark_bar(stroke = 'black',
strokeDash = [4,4]).encode(
    alt.X('Population (Millions):Q'),
    alt.Y('Country:N'),
    alt.Color('Country:N')
).properties(title = 'Population')
```

Note that, if we are not careful, we may also create plots that are difficult to read, such as those using colors that lack sufficient contrast or where marks overlap unnecessarily, which can compromise clarity:

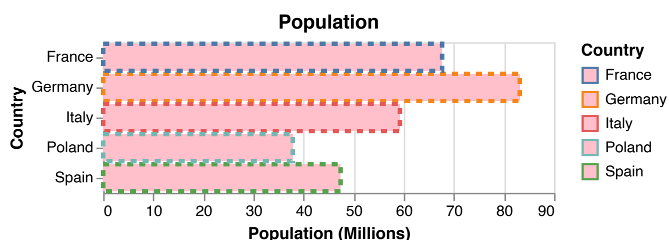
```
alt.Chart(data).mark_bar(color = 'pink', fill = 'gray',
                         stroke = 'red', strokeDash = [4,4],
                         strokeWidth = 3, height = 25).encode(
    alt.X('Population (Millions):Q'),
    alt.Y('Country:N'),
    alt.Color('Country:N')
).properties(title = 'Population')
```



Additionally, some global parameters, such as the gray fill, may overlap and obscure local parameters, such as the colors assigned to countries. Although these colors still appear in the legend, they are not visible in the plot.

We can also use the data to determine the colors of the strokes:

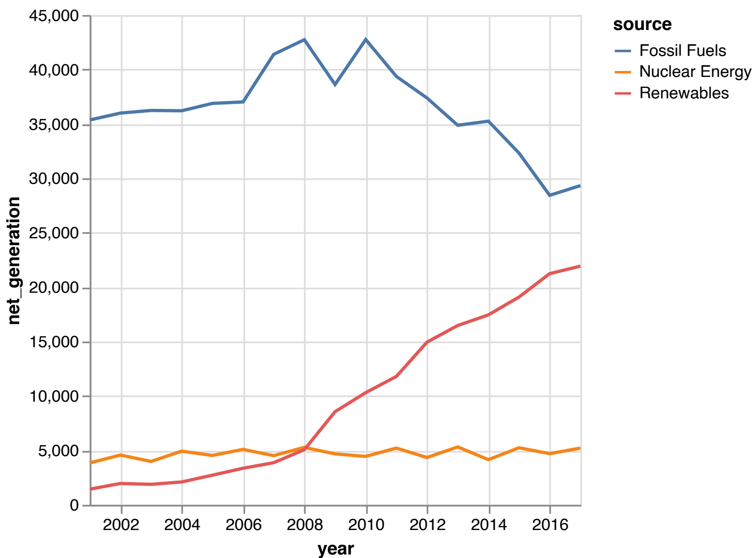
```
alt.Chart(data).mark_bar(color = 'pink', strokeWidth = 3,
                         stroke = 'red',
strokeDash = [4,4]).encode(
    alt.X('Population (Millions):Q'),
    alt.Y('Country:N'),
    stroke = alt.Color('Country:N'),
).properties(title = 'Population')
```



Grouping data is essential in many visualizations. A channel named *detail* allows us to group elements without mapping the fields to any visual properties. For example, to plot the energy sources for Iowa's electricity, we can create the following **line chart**:

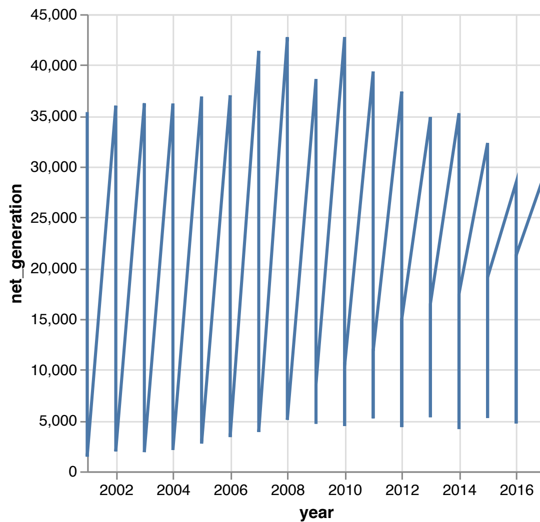
```
df = data.iowa_electricity()

alt.Chart(df).mark_line().encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q'),
    color = 'source:N'
)
```

This code renders the sources of electricity in different colors:



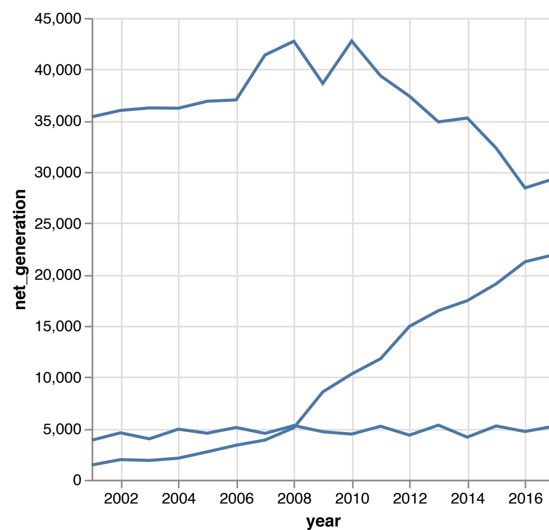
The various sources are separated and plotted in distinct colors, thanks to the *"color = 'source'"* channel. However, we may want to plot all lines in the same color. Removing the color encoding based on the data would generate the following chart:

In this chart, no distinction is made among the sources, so the system interprets all the data as a single source, which is misleading. We can separate the energy sources and obtain line charts for each source by using the *detail* field:




```
df = data.iowa_electricity()

alt.Chart(df).mark_line().encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q'),
    detail = 'source'
)
```

This will generate three lines in the same color:



To further distinguish the lines, we need a visual cue, such as a label, which can be added as an overlaid view. However, the "detail" parameter can be useful on its own, as other elements may help differentiate the individual groups.

5.2. Channel options

Channels can be configured with additional options to perform operations on the data, such as aggregation, sorting, or binning. The available operations depend on the data type.

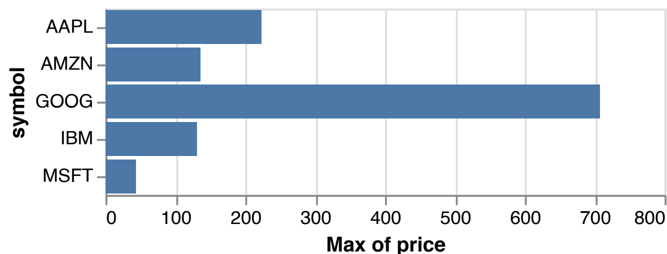
Options for x and y encodings:

- **aggregate**: Applies an aggregation function (e.g., mean, sum, median) to the field.
- **axis**: Modifies axis properties.
- **bandPosition**: Sets the relative position within the band of a stacked, binned, time unit, or band scale. For example, marks will be positioned at the beginning of a band if set to 0, and in the middle if set to 0.5.
- **bin**: Flags a quantitative field for binning.
- **field**: A required field (unless using the *count* aggregate function) that defines the name of the field from which data values are drawn. It can also be an object defining iterated values from the repeat operator. Dots and brackets can access nested objects, and if a field name contains dots or brackets that are not for nesting, they must be escaped with "\\".
- **impute**: Defines the properties of the impute operation.
- **scale**: Scales properties proportional to the data; when disabled, the data are directly encoded.
- **sort**: Defines the sort order of the encoded field.
- **stack**: Used to stack values of x or y when they encode values of continuous domains.
- **timeUnit**: Specifies the time unit (e.g., year, yearmonth, month, hours) for a temporal field.
- **title**: Sets a title for the field.
- **type**: Specifies the data type of measurement (e.g., "quantitative," "temporal," "ordinal," "nominal," or "geojson") for the encoded field or constant value (datum).

A very common operation in x and y encoding is data aggregation. This is achieved through a set of functions that perform calculations such as maximum, minimum, average, and counts. For instance, to calculate the maximum price of each company (represented as 'symbol' in the *stocks* dataset), Altair can com-

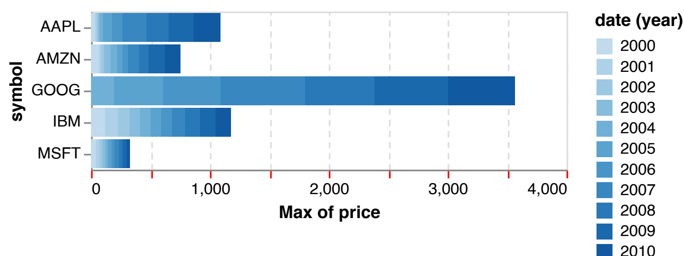
pute the maximum directly from the field itself (positioning values along the x-axis for space optimization):

```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q'),
    alt.Y('symbol:N'),
)
```



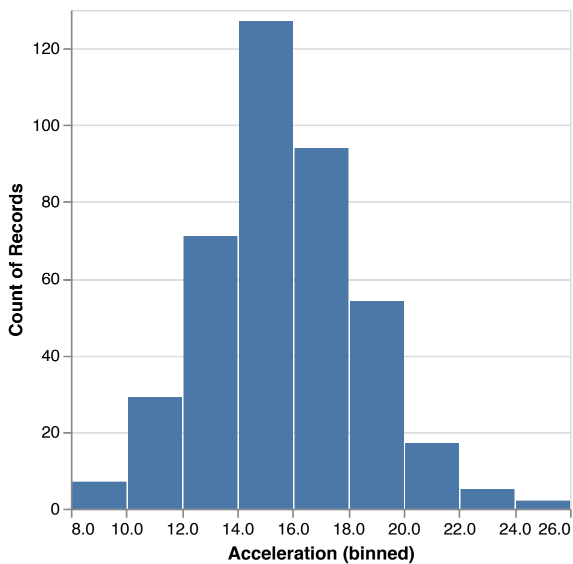
To check the maximum stock price per year, we can separate the data by year. This will generate a stacked bar chart, where each bar represents the maximum stock price for each element within that year:

```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q', axis = alt.
Axis(gridDash = [4,3], tickColor = 'red')),
    alt.Y('symbol:N'),
    alt.Color('year(date):O')
)
```



To create **histograms**, we can use the *bin* option. In the following example, cars are grouped by their acceleration:

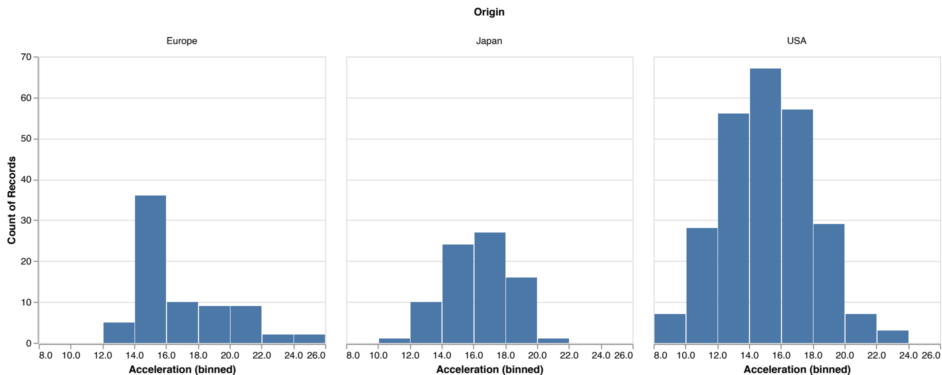
```
df = data.cars.url

alt.Chart(df).mark_bar().encode(
    alt.X('Acceleration:Q', bin=True),
    alt.Y('count():Q'),
)
```



We can further separate the cars by origin and add a column for each by including the *column* option in the plot:

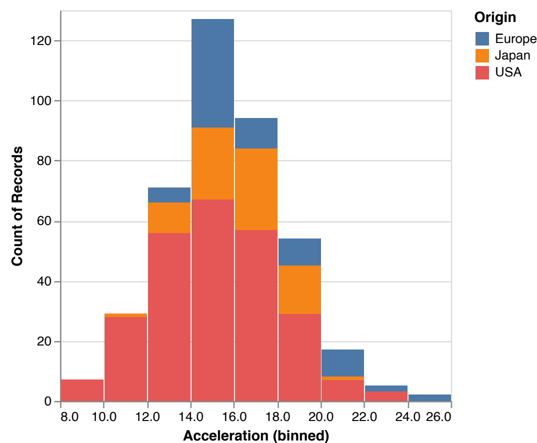
```
df = data.cars.url

alt.Chart(df).mark_bar().encode(
    alt.X('Acceleration:Q', bin=True),
    alt.Y('count():Q'),
    column = 'Origin:N'
)
```

This will generate three bar charts:



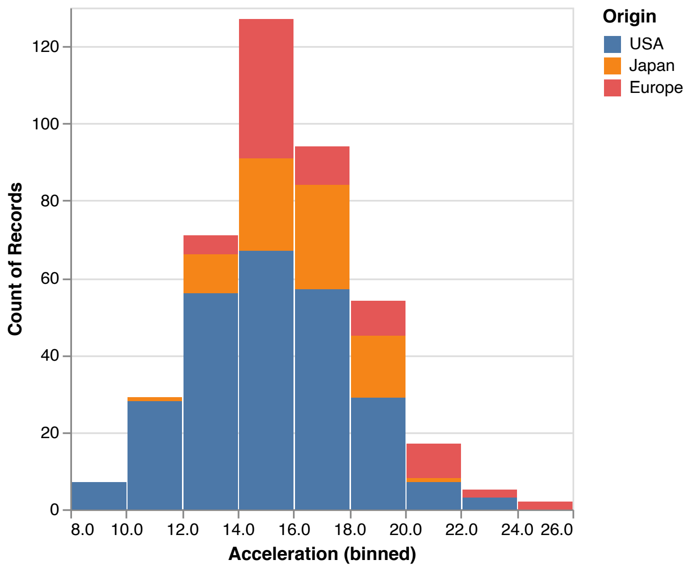
A common variation of bar charts is the **stacked bar chart**, where bars are stacked to show the composition of each category:

```
df = data.cars.url

alt.Chart(df).mark_bar().encode(
    x=alt.X('Acceleration:Q', bin=True),
    y='count():Q',
    color = 'Origin:N'
)
```



Another common channel option is sorting. To *sort* the segments of the bars, we can use the *sort* option to specify the desired sorting method. For example, to sort the stacks by the values of the origin field in ascending order, we would set the sort option for the color encoding:

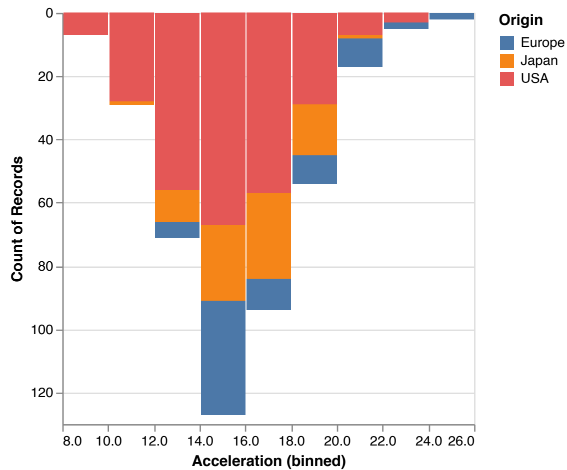
```
df = data.cars.url

alt.Chart(df).mark_bar().encode(
    alt.X('Acceleration:Q', bin=True),
    alt.Y('count():Q'),
    alt.Color('Origin:N', sort = 'descending')
)
```



We can also sort the y-axis data using the sort option:

```
df = data.cars.url

alt.Chart(df).mark_bar().encode(
    alt.X('Acceleration:Q', bin=True),
    alt.Y('count():Q', sort = 'descending'),
    alt.Color('Origin:N')
)
```

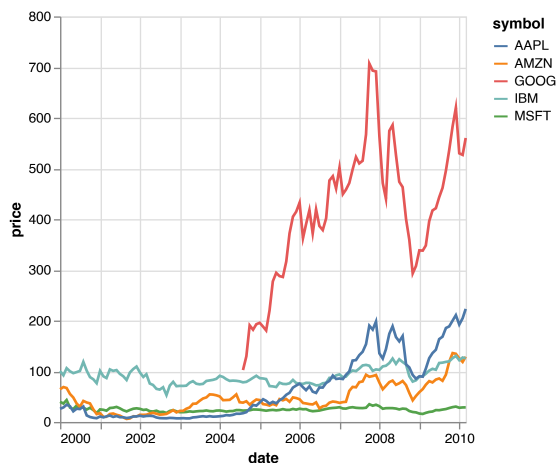
This will set the top of the chart as the origin of the y-axis:



However, we should generally avoid using non-standard axes.

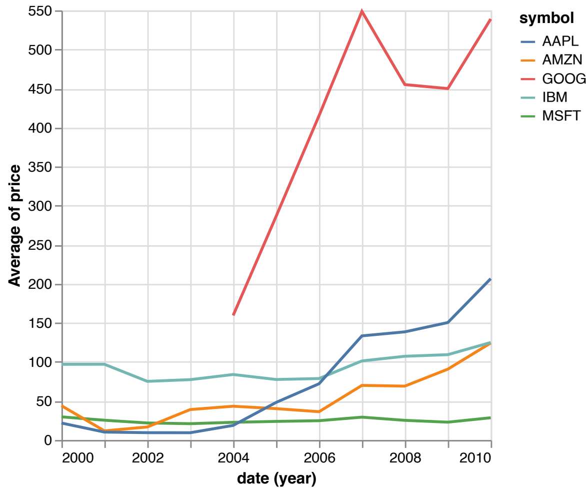
We can also bin time by using the *timeUnit* option, which allows us to group data into different time intervals. For instance, the *stocks* dataset can be visualized with fine-grained data:

```
df = data.stocks.url

alt.Chart(df).mark_line().encode(
    alt.X('date:T'),
    alt.Y('price:Q'),
    alt.Color('symbol:N')
)
```



Alternatively, it can be visualized by averaging the values yearly:

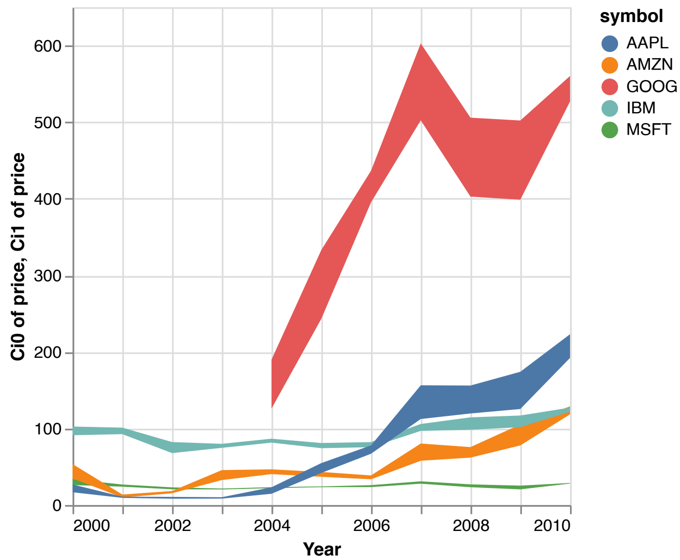
```
df = data.stocks.url

alt.Chart(df).mark_line().encode(
    alt.X('date:T', timeUnit = 'year'),
    alt.Y('average(price):Q'),
    alt.Color('symbol:N')
)
```



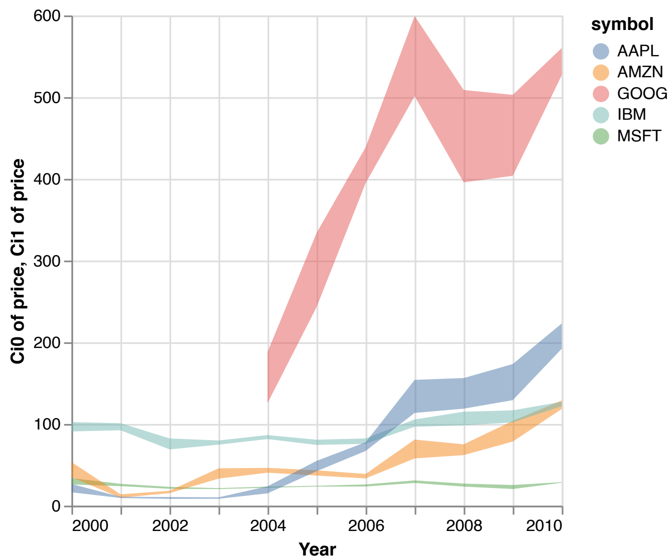
In this case, we do not know how much the price has changed along the line. We could add a confidence interval to the function using *ci0* and *ci1* to encode the initial and final values of the intervals on the y-axis with the *y* and *y2* options:

```
df = data.stocks.url

alt.Chart(df).mark_area().encode(
    alt.X('date:T', timeUnit = 'year').title('Year'),
    alt.Y('ci0(price):Q'),
    alt.Y2('ci1(price):Q'),
    alt.Color('symbol:N')
)
```

For this purpose, we use the area mark, as we want to create an area region. If the *mark_line* function is selected, we would see lines extending from the maximum to the minimum stock prices each year, which is not our intention. Additionally, overlapping values would make it difficult to determine the exact start and end points of the different values in some regions.



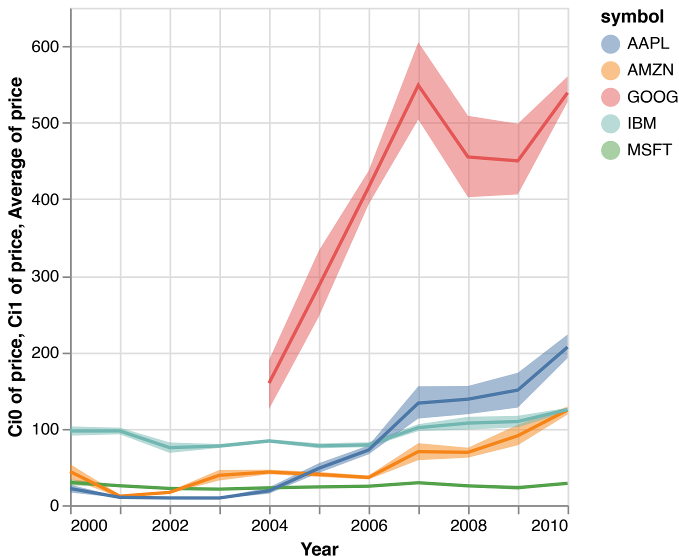
This chart can be improved by adjusting the opacity in the *mark_area* function (opacity = 0.5), making the result easier to interpret:



We can also combine both charts by overlapping them as we saw previously, resulting in:

```
df = data.stocks.url

chArea = alt.Chart(df).mark_area(opacity = 0.5).encode(
    alt.X('date:T', timeUnit = 'year').title('Year'),
    alt.Y('ci0(price):Q'),
    alt.Y2('ci1(price):Q'),
    alt.Color('symbol:N')
)

chLine = alt.Chart(df).mark_line().encode(
    alt.X('date:T', timeUnit = 'year').title('Year'),
    alt.Y('average(price):Q'),
    alt.Color('symbol:N')
)

chArea + chLine
```

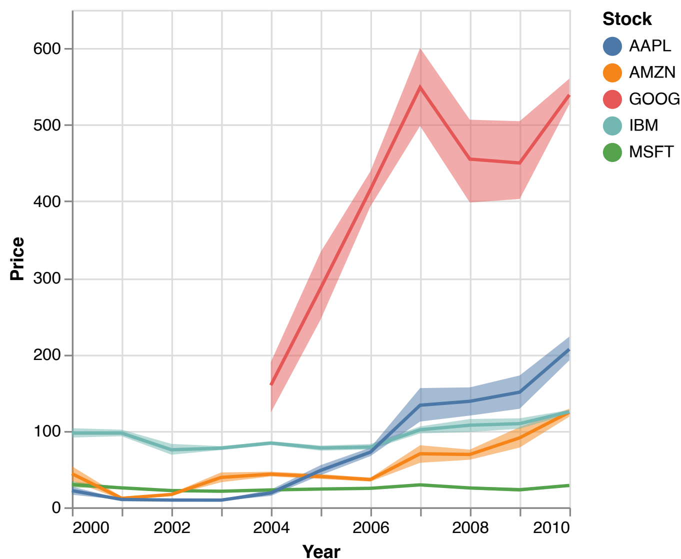
This code will generate the following chart:



We can enhance the result by ensuring that the axes are properly labeled and that the symbols in the legend are rendered opaque. The following changes will accomplish this:

```
df = data.stocks.url

chArea = alt.Chart(df).mark_area(opacity = 0.5).encode(
    alt.X('date:T', timeUnit = 'year').title('Year'),
    alt.Y('ci0(price):Q').title(''),
    alt.Y2('ci1(price):Q').title(''),
    alt.Color('symbol:N',
              legend = alt.Legend(title = 'Stock',
                                  symbolOpacity = 1.0))
)

chLine = alt.Chart(df).mark_line().encode(
    alt.X('date:T', timeUnit = 'year').title('Year'),
    alt.Y('average(price):Q').title('Price'),
    alt.Color('symbol:N')
)

chArea + chLine
```

With the following result:



Other channels like color, fill, stroke, and shape also accept similar sets of options.

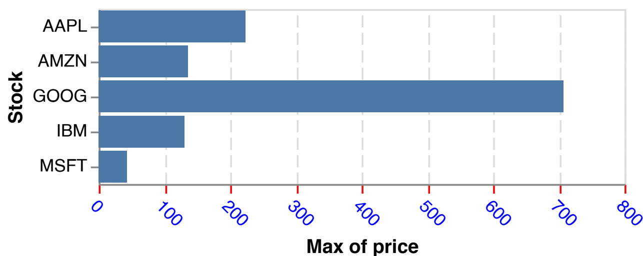
5.3. Customization options

Additional configurations can be applied to customize the visualization. In the previous examples, we have used functional naming for the encodings (e.g., *alt.X* for the x-axis and *alt.Y* for the y-axis) rather than the older convention (*x* = and *y* =) because these functions accept several parameters (separated by commas) that allow further customization.

To modify axis properties, such as adding a dashed style to the grid, we can configure the axis parameter of the *alt.X()* function.

In the following example, we add dashes, color the ticks, rotate the labels to a 45-degree angle, and change their color to blue:

```
df = data.stocks.url
alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q',
          axis = alt.Axis(gridDash = [7,3], tickColor = 'red',
          labelColor = 'blue', labelAngle = 45)),
    alt.Y('symbol:N').title('Stock'),
)
```



Note that the same syntax is used for both the dashed grid lines and the strokes.

Numerous customizations can be made using these parameters. In most editors, options will appear in a floating window as you begin typing.

However, with great power comes great responsibility. Just because we can make many modifications to the original layout does not mean that they will enhance user experience or simplify data interpretation. It is essential to avoid

adding unnecessary embellishments that may hinder the user's ability to accurately interpret the data being presented.

In addition to customizing individual elements, other options can be used to affect the overall layout, such as the title and chart size. The size of the chart can be adjusted using *width* and *height* properties, as demonstrated below.

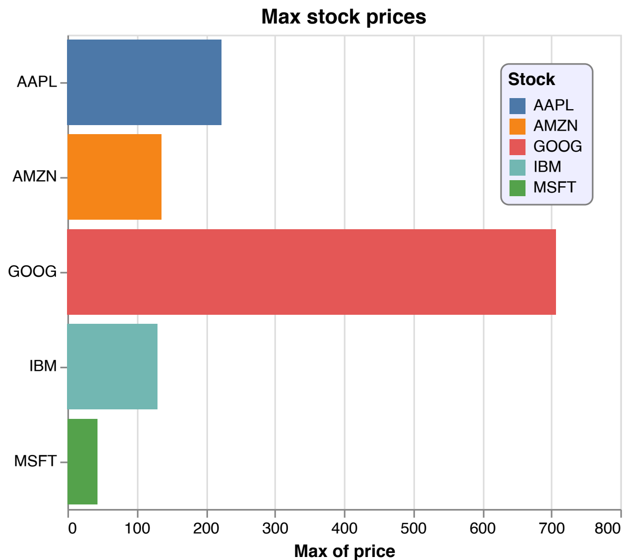
```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q'),
    alt.Y('symbol:N').title(''),
    alt.Color('symbol:N').title('Stock')
).properties(title = 'Max stock prices',
width = 350, height = 300)
```

We can also modify the title with the *title* property, which accepts a string.

Finally, other useful elements to configure include the legend. Parameters such as the title, stroke color, fill color, padding, and orientation can be adjusted using the *configure_legend* function, as shown in the following example:

```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q'),
    alt.Y('symbol:N').title(''),
    alt.Color('symbol:N').title('Stock')
).properties(title = 'Max stock prices',
             width = 350,
             height = 300).configure_legend(
                 strokeColor = 'gray',
                 fillColor = '#EEEEFF',
                 padding = 5,
                 cornerRadius = 5,
                 orient = 'top-right'
             )
```

This configuration results in the following chart:



Note that the legend uses the title provided for the y-axis, but the axis itself was disabled using the option *title* = ‘’.

Another option for setting the *width* and *height* values involves making them responsive to the size of the HTML page or container. To do this, simply set the *width* value to *container*. This adjusts the chart size based on the available space in the HTML page. The advantage of this feature is that the size of the chart will automatically adapt to window resizing. However, note that this function may not work well in Google Colab.

5.4. Multiple charts: simple combinations

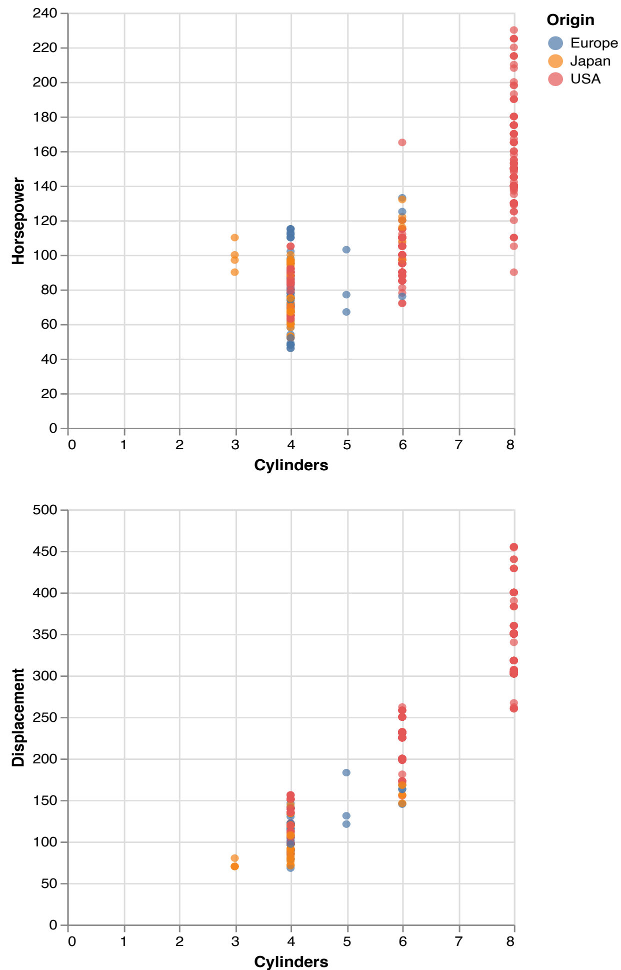
When addressing multiple questions using plots, we often need to create multiple charts. Previously, we used the ‘+’ operator to overlay two charts. Altair offers several ways to combine charts, such as using layers. In this section, we provide additional examples of simple chart combinations.

To arrange multiple charts vertically, we use the ‘&’ operator. This positions the charts one above the other. We can achieve this by assigning names to each chart or by using parentheses to group the plotting method:

```
cars = data.cars.url
```

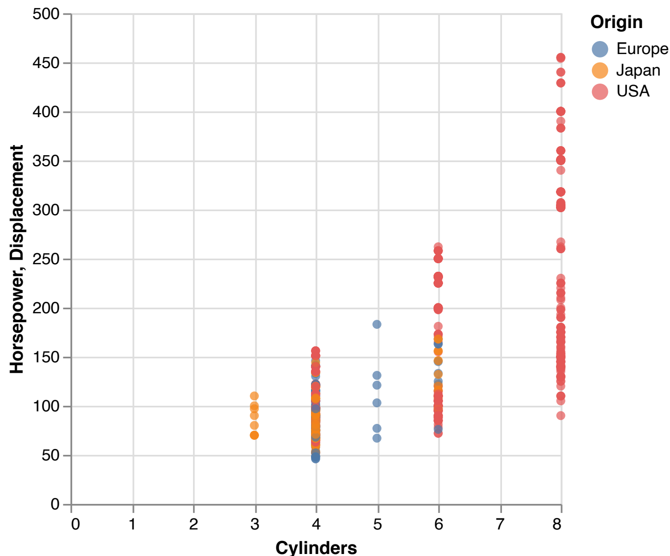
```
ch1 = alt.Chart(cars).mark_circle().encode(
    x = 'Cylinders:Q',
    y = 'Horsepower:Q',
    color = 'Origin:N'
)
ch2 = alt.Chart(cars).mark_circle().encode(
    x = 'Cylinders:Q',
    y = 'Displacement:Q',
    color = 'Origin:N'
)
ch1 & ch2
```

This will generate:



Note that the legend is shared.

However, we can also overlay the charts by replacing the '&' symbol with a '+'. The result will be:



In this case, ambiguity arises because we are using color encodings for both plots. If the plots represent different data, marks must differ to avoid confusion.

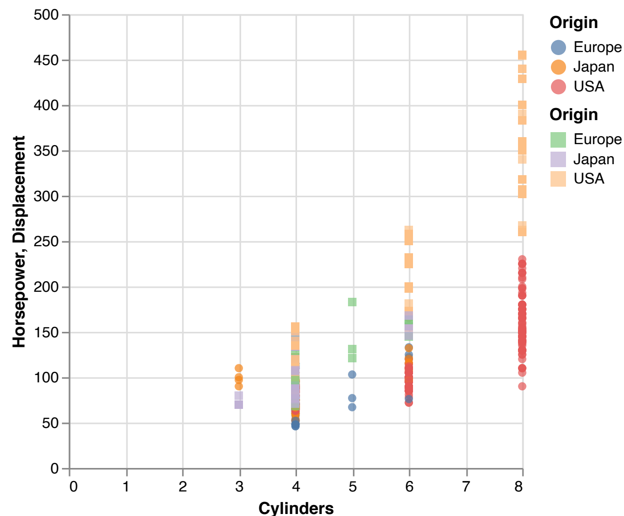
To enhance visual separation, you can change the marks of one of the charts. Additionally, the color schemes can also differ between the charts. However, since the color property is shared by default among all overlapping charts, you must explicitly instruct Altair to treat the colors independently. This can be accomplished by using a combination of the *scale* property to set the palette and the *resolve_scale* method to ensure that the colors are treated as independent.

In the following example, one chart uses squares as shapes, while the other uses circles. We will then set the color scheme of one chart to the *accent* color scheme (refer to the Vega documentation for available color schemes: https://vega.github.io/vega/docs/schemes/). Finally, we will ensure that both schemes are treated independently:

```
cars = data.cars.url

ch1 = alt.Chart(cars).mark_circle().encode(
```

```
    alt.X('Cylinders:Q'),
    alt.Y('Horsepower:Q'),
    alt.Color('Origin:N')
)
ch2 = alt.Chart(cars).mark_square().encode(
    alt.X('Cylinders:Q'),
    alt.Y('Displacement:Q'),
    alt.Color('Origin:N', scale = alt.Scale(scheme = 'accent'))
)

(ch1 + ch2).resolve_scale(color = 'independent')
```

The result is:



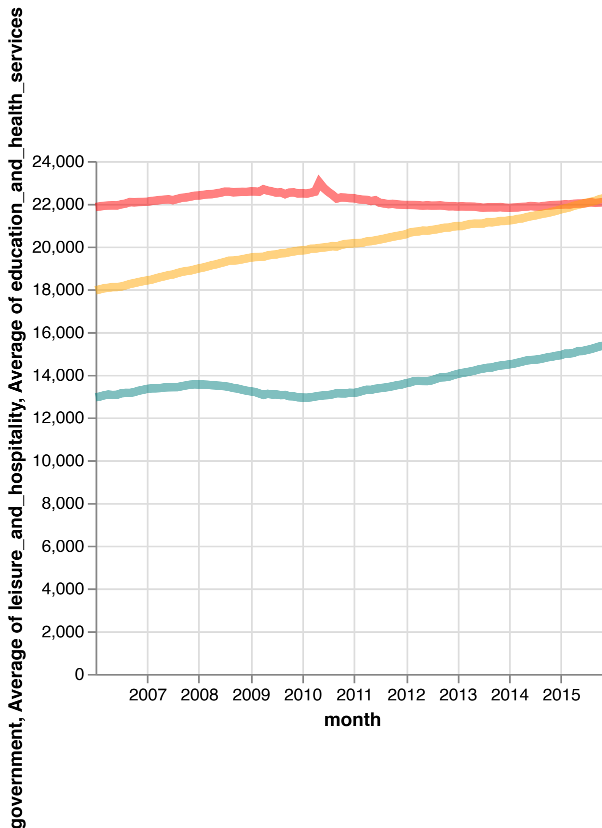
Layers can be used to visualize wide-form data, such as the *stocks* dataset:

```
df = data.us_employment.url

base = alt.Chart(df).mark_line(
    color = "red", strokeWidth = 5, strokeOpacity = 0.5
).encode(
    alt.X('month:T'),
    alt.Y('government:Q')
)
```

```
alt.layer(base,
          base.encode(
              alt.Y('average(leisure_and_hospitality):Q'),
                    color = alt.value('teal')),
          base.encode(
              alt.Y('average(education_and_health_services):Q'),
                    color = alt.value('orange'))
          )
```

Layers are equivalent to the '+' operator in charts but allow you to add more than two elements. In this case, because the data are wide-form, we do not have a row for each entry. Therefore, to visualize all the columns, we would need to specify them individually.

The result of the previous code is:

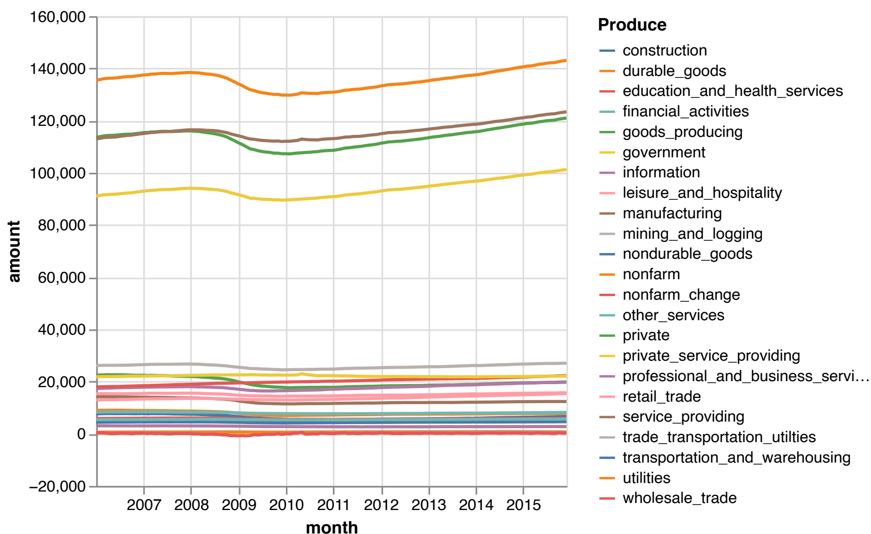


This approach is far less convenient than when the data are in long-form. Additionally, note how the names of the axes now concatenate all the marks from the different layers, which can be inconvenient. We can set appropriate names for the axes within the *encode* function for both x and y, and we can also remove all names if necessary.

To render the entire column set, we can melt the data using a function from Pandas and then render it as if it were in long-form:

```
df = data.us_employment()

df2 = df.melt('month', var_name = 'Produce',
              value_name = 'amount')

alt.Chart(df2).mark_line().encode(
    alt.X('month:T'),
    alt.Y('amount:Q'),
    alt.Color('Produce:N')
)
```

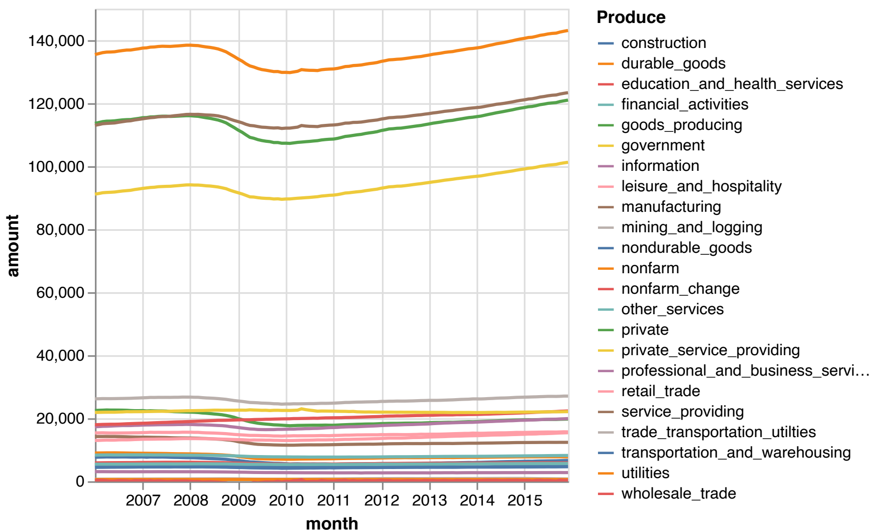
This results in the following chart:



The previous chart plots negative values, causing the y-axis to extend down to −20,000. We can clip this by defining the axis domain using the *scale* property of the y parameter:

```
df = data.us_employment()

df2 = df.melt('month', var_name = 'Produce',
              value_name = 'amount')

alt.Chart(df2).mark_line(clip = True).encode(
    alt.X('month:T'),
    alt.Y('amount:Q', scale = alt.Scale(domain = (0, 150000))),
    alt.Color('Produce:N')
)
```

Adding the extra option of *clip* to *True* will clip the negative values (which would be plotted even though the axis starts at 0). The result is shown in the following figure.



Note that the chart here is not only quite cluttered but also has other issues, such as an insufficient number of colors in the categorical palette for the categories being plotted. This may lead to incorrect interpretations and should be addressed, for example by providing a categorical palette (which can be set using Altair) with enough distinct colors.

6
Charts

We have already seen how the selection of a mark determines the type of chart. For example, mark_point generates scatter plots, while mark_bar produces bar charts. In this section, we will review these basic chart types and introduce more complex ones.

6.1. Basic chart types

The simplest chart types that have been presented in previous sections are:

- **Scatter plot:** Created by encoding the mark as a point using *mark_point*, with the x and y fields representing its coordinates.

- **Bar chart:** Uses the mark as a bar with *mark_bar*. The x field encodes the variable and the y field represents its value.

- **Line chart:** The mark is configured as a line using *mark_line*, with *x* containing the first field and the *y* coordinate the second.

- **Area chart:** Similar to the line chart, but with the mark configured as an area using *mark_area*.

6.2. Variations over simple charts

Numerous modifications can be made to basic plots. By using the various configurations in Altair, you can achieve a different look and feel for the same chart or even transform the chart type. In this section, we will explore different ways to modify simple charts.

Fine-tuning can be applied not only to the appearance of the marks but also to auxiliary elements such as axes and labels. In the following example, we change the values on the y-axis to display them as percentages:

```
source = data.jobs.url

alt.Chart(source).mark_line().encode(
    alt.X('year:O'),
    alt.Y('perc:Q', axis=alt.Axis(format='%')),
    color='sex:N'
).transform_filter(
    alt.datum.job == 'Welder'
)
```

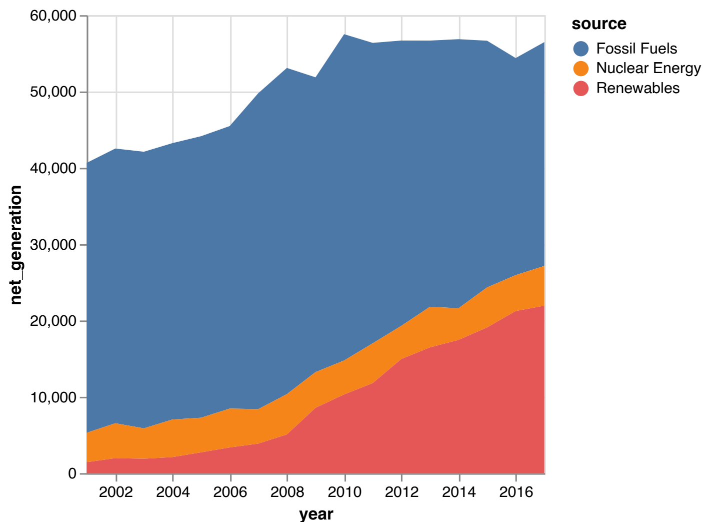
Otherwise, the y-axis would display values such as 0.018 instead of 1.8%.

We can also add points to line charts simply by including the option *point=True* in the mark properties. Additionally, we can configure the line thickness by adding the option *size* to the encoding, allowing it to vary based on a specific variable, such as the one encoded on the y-axis.

We have previously discussed **area charts**, which can be configured in two ways: stacked and unstacked. The default configuration is stacked; however, this may complicate the calculation of ratios. For example, if we display the electricity production sources for Iowa, we would get:

```
df = data.iowa_electricity()

alt.Chart(df).mark_area().encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q'),
    alt.Color('source:N')
)
```

Alternatively, if we want to compare them side by side, we should specify that they are not stacked:

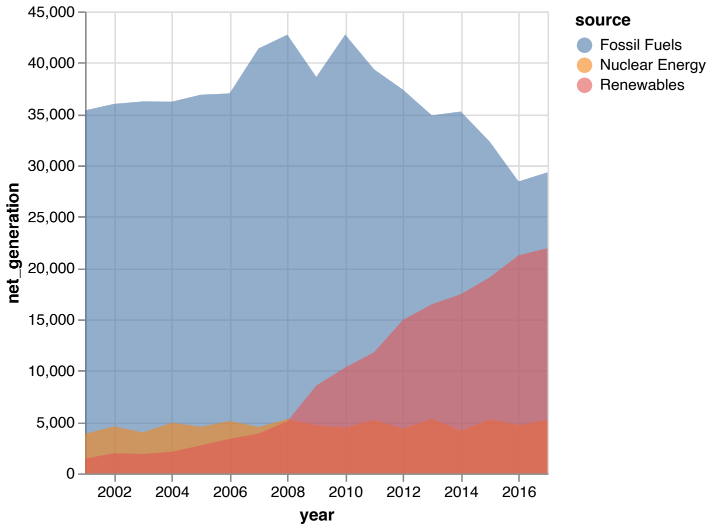



```
df = data.iowa_electricity()

alt.Chart(df).mark_area().encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', stack = False),
    alt.Color('source:N')
)
```

The current configuration, however, generates overlapping areas. To prevent some data points from completely obscuring others, we can add transparency to facilitate visual comparison:

```
df = data.iowa_electricity()

alt.Chart(df).mark_area(opacity = 0.6).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', stack = False),
    alt.Color('source:N')
)
```

Another option is to stack the data while normalizing it, which makes the relative ratios easier to appreciate. The result is called a **normalized area chart:**

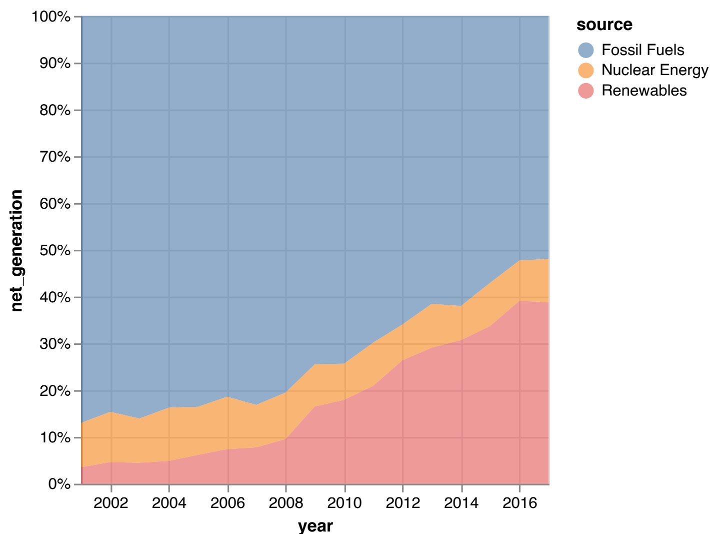



```
df = data.iowa_electricity()

alt.Chart(df).mark_area(opacity = 0.6).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', stack = 'normalize'),
    alt.Color('source:N')
)
```

Note that by maintaining a certain level of transparency, we can keep the grid lines visible, making it easier to evaluate magnitudes visually. If the plot is completely opaque, these lines become obscured.

We could also represent each category in a separate chart by creating a Trellis chart. To do this, we simply instruct Altair to assign a different row for each category:

```
df = data.iowa_electricity()

alt.Chart(df).mark_area(opacity = 0.6).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q'),
    alt.Color('source:N'),
    alt.Row('source:N')
)
```

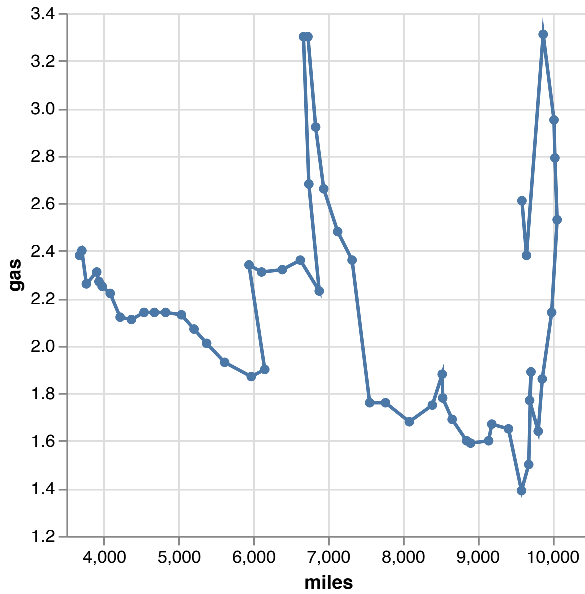


Note that although Trellis charts can be useful for visual comparison, the current configuration is not ideal. Users would need to scroll up and down to get a complete view of all the charts. We can also generate small multiples horizontally by changing *alt.Row* to *alt.Column*.

Another, less common, chart is the connected line chart with ordering, also known as a **connected scatter plot**. This chart is useful when we need a scatter plot but the data points do not appear in the same sequence as the *x* values. We can address this by using a regular line plot with ordering. In the following example, we sort the points using a variable that is not plotted, that is, the year:

```
df = data.driving()

alt.Chart(df).mark_line(point = True).encode(
    alt.X('miles:Q').scale(zero = False),
    alt.Y('gas:Q').scale(zero = False),
    alt.Order('year:O'),
)
```

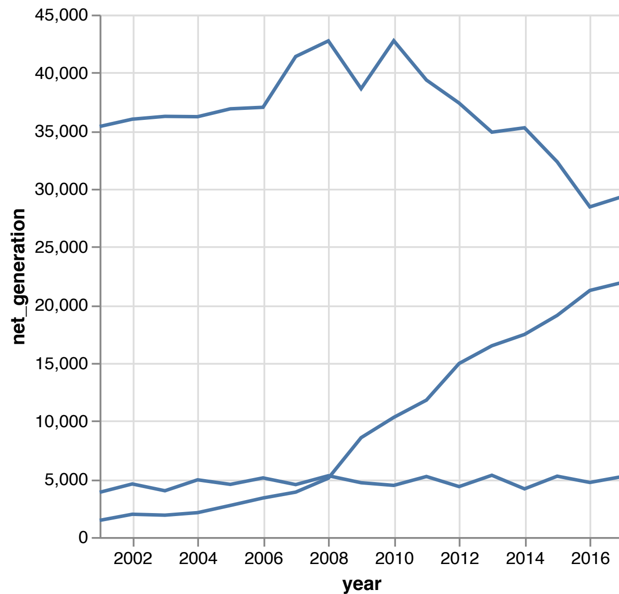
This results in:



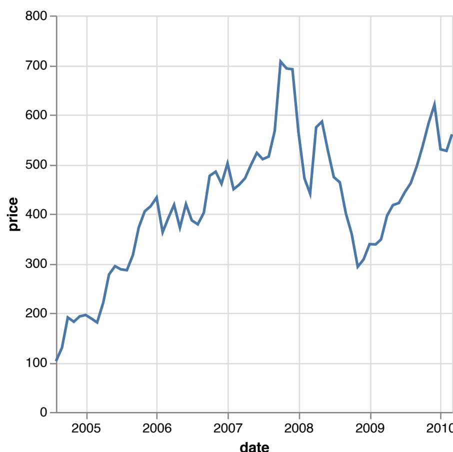
We can also modify the line chart by using the *trail* channel, allowing the trail to have varying widths based on a variable:

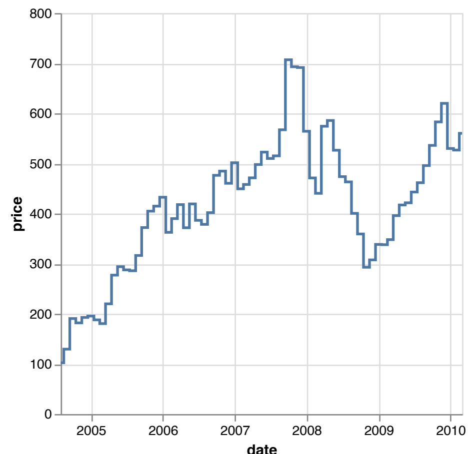
```
df = data.iowa_electricity()

alt.Chart(df).mark_trail().encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', title = 'Net generation'),
    alt.Color('source:N'),
    alt.Size('net_generation')
)
```

Note that another legend will now appear to explain the new encoding:



A different way to configure line plots is by changing the line used to connect the points through the *interpolate* option of the *mark_line* function. If we interpolate the previous stock values of Google (using the *stocks* dataset) with either *linear* or *step* methods, we obtain the following two different charts:





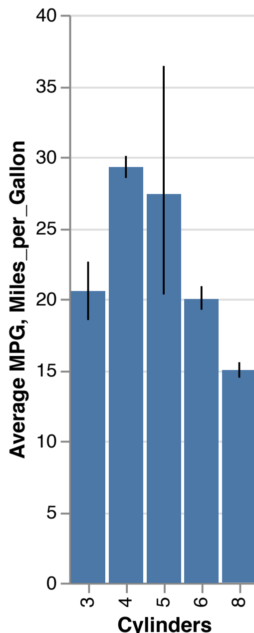
The interpolation options include *linear, linear-closed, step, step-before, step-after, basis, basis-open, basis-closed, cardinal, cardinal-open, cardinal-closed, bundle*, and *monotone*. However, some of these options do not preserve the loca-

tion of the data points, which may lead to misinterpretation of quantities; so they should be used with caution.

We can also enhance bar charts with additional information, such as error bars. This can be easily implemented by creating a layer that contains the error bars for the dataset, as shown in the following example using the *cars* data:

```
cars = data.cars.url

base = alt.Chart(cars).mark_bar().encode(
    alt.X('Cylinders:O'),
    alt.Y('mean(Miles_per_Gallon):Q', title = 'Average MPG'),
)

errorbars = alt.Chart(cars).mark_errorbar(extent = 'ci').encode(
    alt.X('Cylinders:O'),
    alt.Y('Miles_per_Gallon:Q')
)
base + errorbars
```

The result is:



7

Advanced chart types

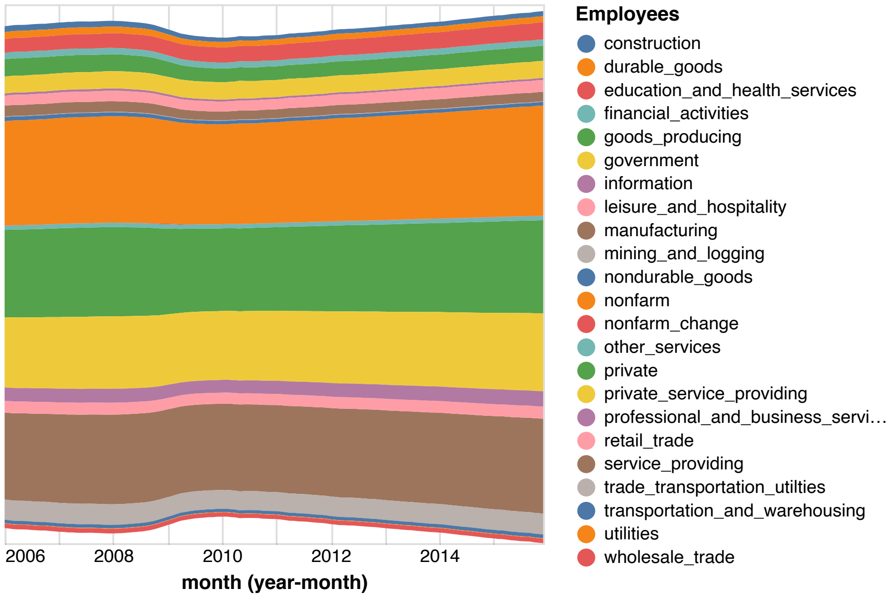
In addition to simple charts and their variations, more sophisticated visualizations help convey specific types of information.

Streamgraphs, a variant of area charts, depict large numbers of categories whose values change over time. Streamgraphs focus on highlighting variations over time, emphasizing the evolution of different elements rather than their exact quantities. The distinctive feature of streamgraphs are the stacking of areas, which are distributed both above and below the x-axis.

In Altair, streamgraphs may be created by configuring the system to stack elements at the center, as demonstrated below:

```
df = data.us_employment()

df2 = df.melt('month', var_name = 'Employees',
              value_name = 'amount')

alt.Chart(df2).mark_area().encode(
    alt.X('yearmonth(month):T',
          axis = alt.Axis(format = '%Y',
                          domain = False,
                          tickSize = 0)),
    alt.Y('sum(amount):Q', stack = 'center', axis = None),
    alt.Color('Employees:N')
)
```

This results in:

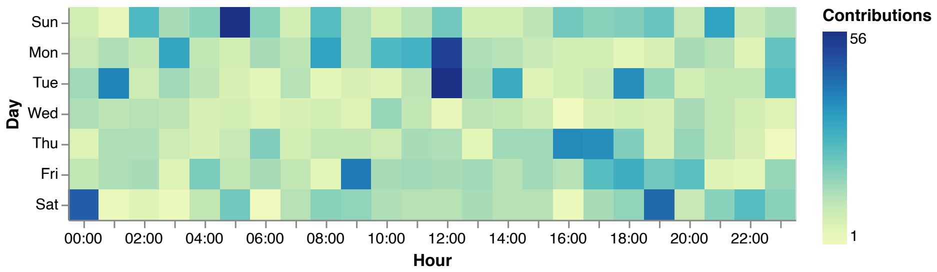


Since the focus is on changes over time, displaying the vertical axis values becomes less important and they have thus been omitted. As in previous examples, the categorical palette used here lacks enough distinct colors to encode all values, making it impossible to distinguish between several categories, which may lead to incorrect interpretation.

Another commonly used visualization is the heatmap, which represents complex data in a tabular format. In the following example, we illustrate the number of contributions made to GitHub each day of the week, broken down by hourly range. The data comes from the github dataset:

```
githubData = data.github.url

alt.Chart(githubData).mark_rect().encode(
    alt.X('hours(time):O'),
    alt.Y('day(time):O'),
    alt.Color('sum(count):Q',
              legend = alt.Legend(title='Contributions'))
)
```

And the result will be the following heatmap:

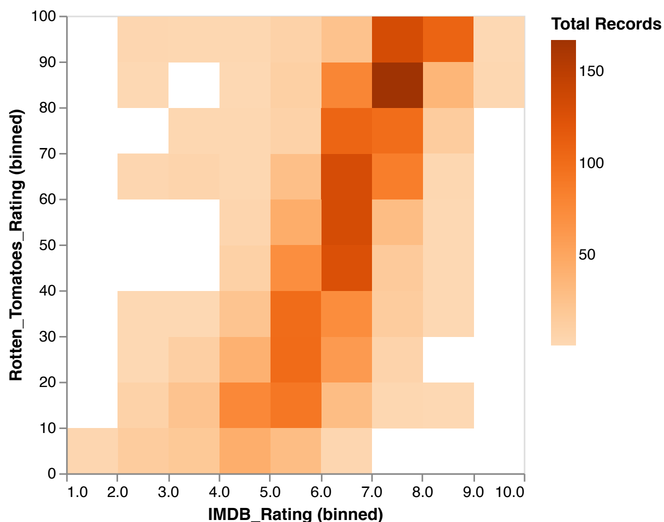


Note that we changed the legend title in the *color* field to ensure clarity. Otherwise, it would have read “Sum of count”.

The next example presents the ratings of several movies from the Rotten Tomatoes website. The data can be obtained from the *movies* dataset.

```
movies = data.movies.url

alt.Chart(movies).mark_rect().encode(
    alt.X('IMDB_Rating:Q', bin = True),
    alt.Y('Rotten_Tomatoes_Rating:Q', bin = True),
    alt.Color('count()',
              scale= alt.Scale(scheme = 'oranges'),
              legend= alt.Legend(title = 'Total Records'))
)
```

The result is as follows:

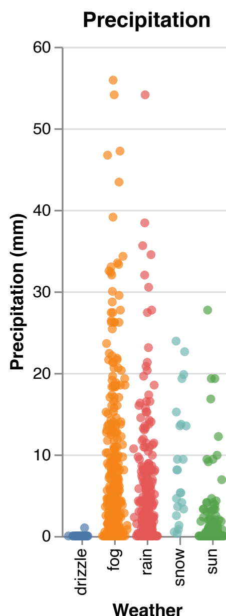


Another way to display data distributions is by plotting all points in the dataset, which is effective for a relatively large number of points. One technique for this is the **strip chart**, which plots all data points in a distribution and uses offsets to avoid overlapping. In Altair, strip charts can be created using the offset parameters, such as *xOffset*.

The following example illustrates the different precipitation amounts in Seattle for various weather conditions:

```
source = data.seattle_weather.url

alt.Chart(source, title = 'Precipitation').mark_circle().encode(
    alt.X('weather:N', title = 'Weather'),
    alt.Y('precipitation:Q'),
    alt.Color('weather:N').legend(None),
    xOffset = 'jitter:Q'
).transform_calculate(
    jitter='sqrt(-2*log(random()))*cos(2*PI*random())'
)
```

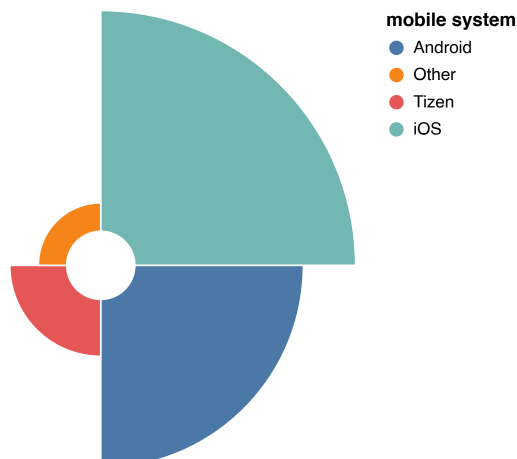
The offset is calculated using a Gaussian filter with a Box-Muller transform. The result is as follows:



Polar area charts are a specific type of circular chart in which quantities are encoded in the radius rather than the angle. They can be implemented using the theta and theta2 parameters to define the beginning and end of each arc, along with the radius parameter to determine the length. The following example illustrates this:

```
import numpy as np

source = pd.DataFrame({"mobile system":
                       ['iOS', 'Android', 'Tizen', 'Other'],
                       "percentage": [60, 35, 4, 1],
                       "sector": [0, 0.5*np.pi,
                                 1.*np.pi, 1.5*np.pi],
                       "sector2": [0.5*np.pi, np.pi,
                                  1.5*np.pi,2.*np.pi]})
alt.Chart(source).mark_arc(innerRadius = 20, stroke="#fff").encode(
    theta=alt.Theta(field = 'sector',
                    type = 'quantitative', scale = None),
    theta2 = 'sector2:Q',
    radius = alt.Radius("percentage",
                        scale = alt.Scale(type = "sqrt",
                        zero = True, rangeMin = 20)),
    color=alt.Color(field="mobile system", type="nominal"),
)
```

Note that in this case, the angle values are stored explicitly in the DataFrame because the function used is *mark_angle*, which requires angles. Alternatively, we could calculate the angles on the fly, with the following result:

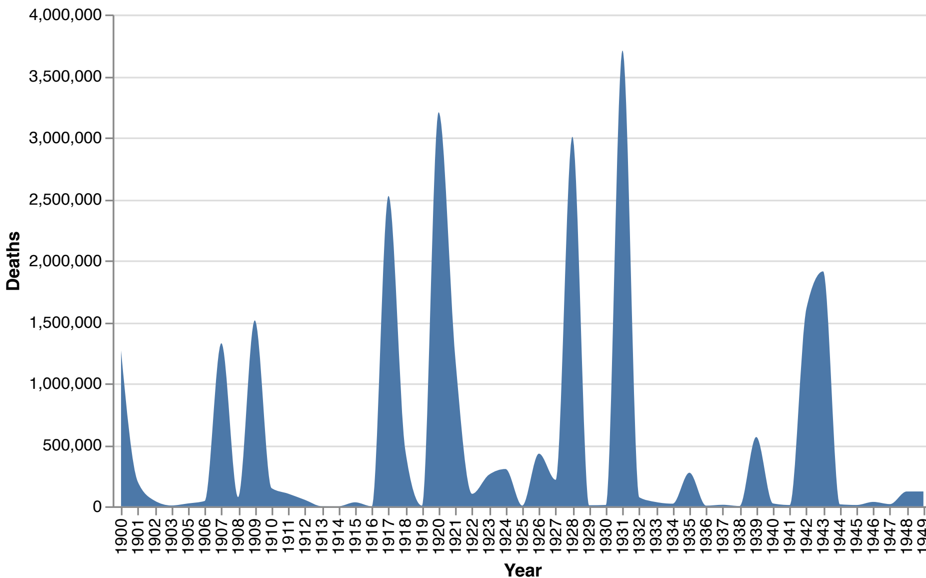


Horizon graphs are a modified version of area charts designed to minimize the vertical space required for the plot. They achieve this effect by employing a combination of clipping, overlaying, and transparency. While these can be very useful, they are often more challenging to read, requiring users to undergo a certain amount of training. Horizon graphs are also intended to represent a single category; otherwise, they become illegible.

Let's consider a plot like the following:

```
df = data.disasters()

alt.Chart(df).mark_area(interpolate = 'monotone').encode(
    alt.X('Year:O'),
    alt.Y('Deaths:Q')
).transform_filter(
    (alt.datum.Entity == 'All natural disasters') &
    (alt.datum.Year < 1950)
).properties(width = 500)
```

In this example, we plot the deaths caused by natural disasters prior to 1950, using the dataset *disasters*.



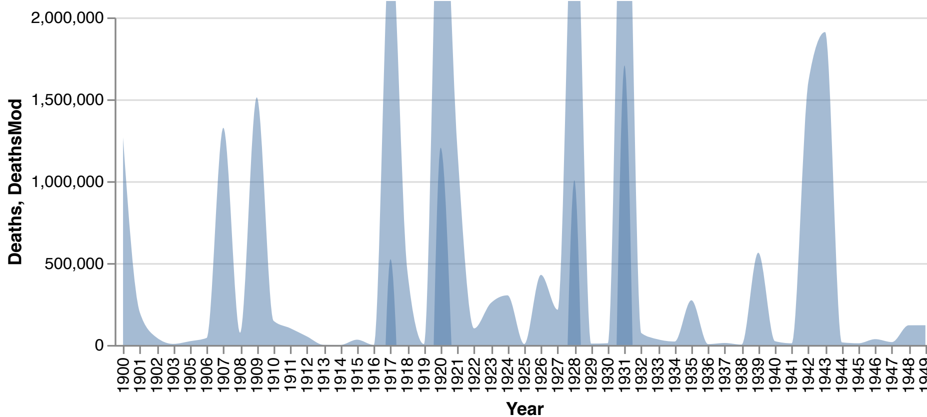
The data contains both very high and very low values, necessitating a large amount of vertical space. To conserve space, horizon graphs employ the following strategy: The y-axis is clipped at the center, with the data in the upper section overlaid onto the lower section as a second chart. This overlay uses a different color coding to indicate that the quantities are higher than those represented in the background. This process is repeated multiple times, with each iteration saving half of the vertical space, albeit at the cost of increased overlay.

In Altair, we can generate horizon graphs using the following technique: First, we plot the original chart clipped to 2 million casualties. Next, we overlay a second chart with the original data modified by subtracting the 2 million amount and applying a different color.

The following example illustrates this approach:

```
df = data.disasters()

base = alt.Chart(df).mark_area(
    clip = True, interpolate = 'monotone'
    ).encode(
    alt.X('Year:O'),
    alt.Y('Deaths:Q').scale(domain = [0, 2000000]),
    opacity = alt.value(0.5)
).transform_filter(
    (alt.datum.Entity == 'All natural disasters') &
    (alt.datum.Year < 1950)
).properties(width = 500, height = 200)

base2 = base.encode(
    alt.Y('DeathsMod:Q').scale(domain = [0, 2000000]),
).transform_calculate('DeathsMod', alt.datum.Deaths - 2000000)

base + base2
```

This results in:



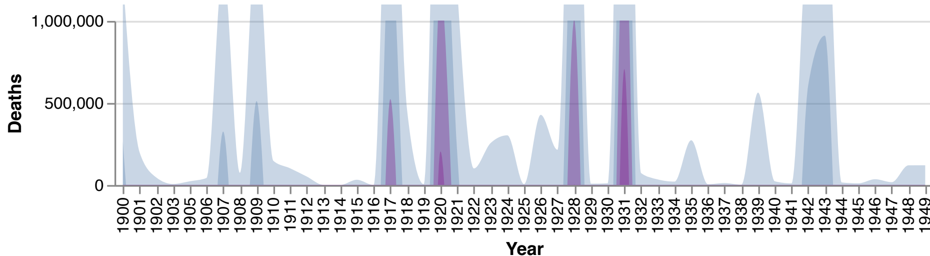
This chart uses a transformation, which is explained in the following section.

The process for creating the horizon chart can be repeated several times. In the following example, we iterated three times and changed the colors for the third and fourth iterations:

```
df = data.disasters()

base = alt.Chart(df).mark_area(
    clip = True, interpolate = 'monotone'
    ).encode(
    alt.X('Year:O'),
    alt.Y('Deaths:Q').scale(domain =
          [0, 1000000]).title('Deaths'),
    opacity = alt.value(0.3)
).transform_filter(
    (alt.datum.Entity == 'All natural disasters') &
    (alt.datum.Year < 1950)
).properties(width = 500, height = 100)

base2 = base.encode(
    alt.Y('DeathsMod:Q').scale(domain = [0, 1000000]),
).transform_calculate('DeathsMod', alt.datum.Deaths - 1000000)

base3 = base.encode(
    alt.Y('DeathsMod2:Q').scale(domain = [0, 1000000]),
    color = alt.value('purple')
).transform_calculate('DeathsMod2', alt.datum.Deaths - 2000000)
```

```
base4 = base.encode(
    alt.Y(‘DeathsMod3:Q’).scale(domain = [0, 1000000]),
    color = alt.value(‘purple’)
).transform_calculate(‘DeathsMod3’, alt.datum.Deaths - 3000000)

base + base2 + base3 + base4
```

We also reduced the vertical size of the chart to 100 units. The result is as follows:

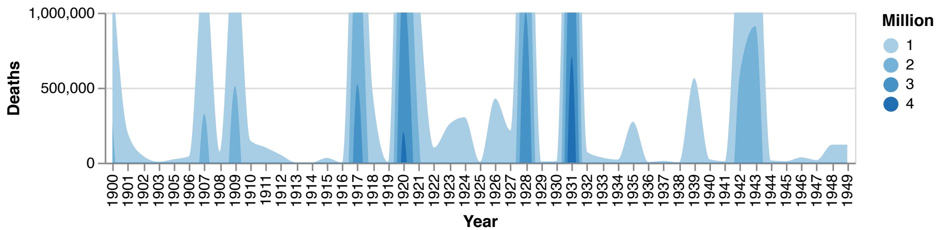


We can further customize the chart to improve its readability. In the following example, we created a new variable to automatically assign colors using a quantitative palette and renamed some calculated variables for easier interpretation:

```
df = data.disasters()

base = alt.Chart(df).mark_area(
    clip = True, interpolate = ‘monotone’
    ).encode(
    alt.X(‘Year:O’),
    alt.Y(‘Deaths:Q’).scale(domain = [0, 1000000]).title(‘Deaths’),
    color = ‘Million:O’
).transform_calculate(Million = ‘1’
).transform_filter(
    (alt.datum.Entity == ‘All natural disasters’) &
    (alt.datum.Year < 1950)
).properties(width = 500, height = 100)

base2 = base.encode(
    alt.Y(‘Deaths>1M:Q’).scale(domain = [0, 1000000]),
).transform_calculate(‘Deaths>1M’, alt.datum.
Deaths - 1000000, Million = ‘2’)
```

```
base3 = base.encode(
    alt.Y('Deaths>2M:Q').scale(domain = [0, 1000000]),
).transform_calculate('Deaths>2M', alt.datum.
Deaths - 2000000, Million = '3')

base4 = base.encode(
    alt.Y('Deaths>3M:Q').scale(domain = [0, 1000000]),
).transform_calculate('Deaths>3M', alt.datum.
Deaths - 3000000, Million = '4')

base + base2 + base3 + base4
```

The result is this improved chart:



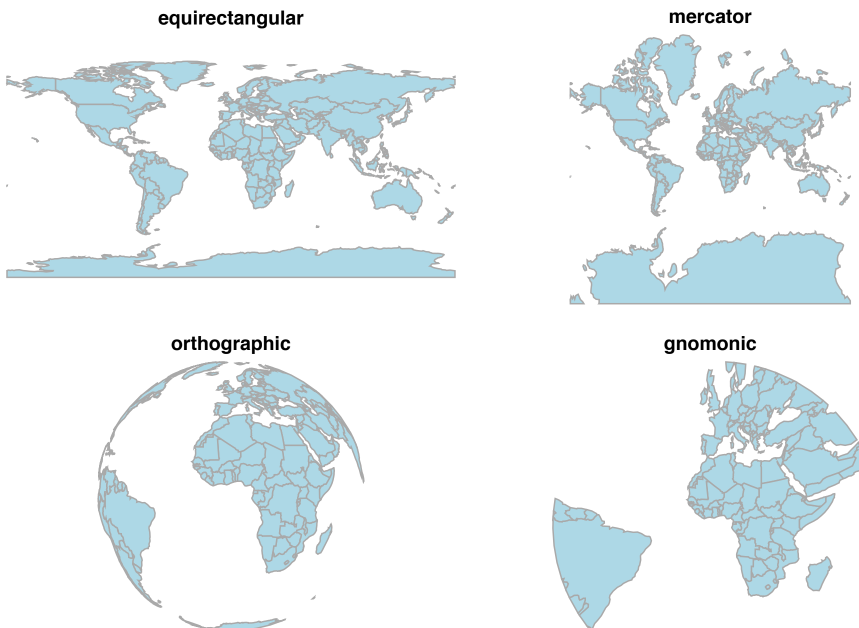
Maps are widely used to visualize various types of data linked to geographic locations, including routes, flows, and demographic or economic datasets. In Altair, maps can be generated using the 1type, though it is more complex than other marks. First, we need a dataset that encodes the geometry of the regions to be plotted, typically obtained from a *geojson* file. Then, we must link this geographic data to the data being visualized, as the geographic dataset often contains only region names (e.g., country, province, city). Combining these datasets usually requires a *lookup* operation, which will be addressed later.

The initial examples focus only data encoded directly within the same geographic file. In the following example, we demonstrate how to project a world map using different projections:

```
source = alt.topo_feature(data.world_110m.url, 'countries')

base = alt.Chart(source).mark_geoshape(
    fill = 'lightblue',
    stroke = 'darkgray',
```

```
).properties(
    width = 300,
    height = 180
)

projections = ['equirectangular', 'mer-
cator', 'orthographic', 'gnomonic']
charts = [base.project(proj).properties(title=proj)
  for proj in projections]

alt.concat(*charts, columns = 2)
```

The result is as follows:



In the following example, from the Altair website, we plot the US map and overlay a circular mark representing the number of airports per state. In this case, we have two sources of information that are not connected. The procedure is as follows: First, we draw the map using *mark_geoshape*. Next, the airport positions are encoded as longitude and latitude in another chart. Finally, we overlay both charts to achieve the desired result.

The first step is to load the US map and plot the geography:

```
map = alt.topo_feature(data.us_10m.url, feature = 'states')

usChart = alt.Chart(map).mark_geoshape(
    fill = 'lightgray',
    stroke = 'white',
).properties(
    width = 500,
    height = 300
).project('albersUsa')
```

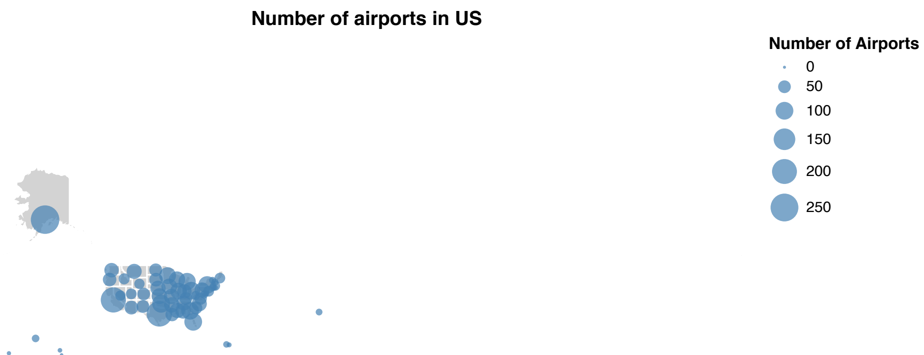
To incorporate the airports information, we gather data from the *airports* dataset and calculate their average longitude and latitude by grouping the data by state (all this information is stored in the *airports* dataset). By performing this aggregation, we can count the number of airports and calculate the average in longitude and latitude of their positions, allowing us to generate 2D points on the map to represent those quantities in size-graded circles. The result is a **graduated symbol map**.

```
airports = data.airports.url

airportsMap = alt.Chart(airports, title
= 'Number of airports in US'
).transform_aggregate(
    latitude='mean(latitude)',
    longitude='mean(longitude)',
    count='count()',
    groupby=['state']
).mark_circle().encode(
    longitude='longitude:Q',
    latitude='latitude:Q',
    size=alt.Size('count:Q').title('Number of Airports'),
    color=alt.value('steelblue'),
    tooltip=['state:N','count:Q']
)
```

The result of overlaying both maps is as follows:

Note that we are using the *Albers* projection, a conic equal-area map projection characterized by minimal distortion between standard parallels, although it does not preserve scale and shape. The projection is utilized by various institutions, including the US Census Bureau. The specific version of the Albers projection used here focuses only on the US.

It is important to note that Altair has mapped the longitude and latitude provided in the second chart to their correct positions on the map, even though Alaska appears in a different location than its actual position according to the Albers USA projection. If we used the Mercator projection, the positions would align correctly (e.g., Puerto Rico would appear where it should). However, the Mercator projection is designed to show the entire world, while our geographic data contains only the US states, leaving much of the map devoid of countries.



Finally, plotting the points first and then the map results in an incorrect configuration.

More advanced maps are presented later in the book.

8
Data transformations

8.1. Basics

It is common to transform input data when generating visualizations. The most flexible and powerful approach is to use Pandas for data transformations. Performing transformations before plotting the chart is generally more efficient unless the transformation is specifically required for exploratory tasks.

Altair can load data from *JSON* files, *CSV* files, or *URLs*. In such cases, using Pandas for data manipulation may be less suitable.

Altair also allows data transformations within the chart specification through a set of *transform_** functions. Some of the most commonly used transformations include:

- **transform_aggregate**: Creates a new data column by aggregating an existing one.
- **transform_bin**: Bins an existing column to create a new one.
- **transform_calculate**: Creates a new column using arithmetic expressions applied to an existing one.
- **transform_filter**: Selects a subset of the input data.
- **transform_fold**: Converts wide-form data into long-form data.
- **transform_lookup**: Performs a one-sided join of two datasets based on a lookup key.
- **transform_pivot:** Converts long-form to wide-form data.
- **transform_stack**: Computes values related to stacked versions of encodings.
- **transform_timeunit**: Discretizes a date by a specified time unit (e.g., day, month, year).

Note that transformations creating new columns do not add them to the original DataFrame; they are created only for visualization purposes. Therefore, the input data remain unchanged.

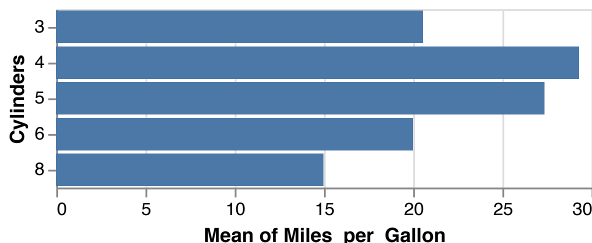
8.2. Aggregate transforms

We have previously covered some aggregate transforms, such as counting. Altair provides two primary methods for calculating aggregations: within the encoding itself (by specifying how the data should be aggregated and displayed) or using an aggregate transform.

The aggregate property of a field definition allows computing aggregate summary statistics (e.g., median, minimum, maximum) over groups of data. When at least one field in the specified encoding channels contains an aggregate function, the resulting visualization will display aggregated data. Fields without a specified aggregation function are treated as group-by fields in the aggregation process.

We demonstrated this in earlier examples. For instance, to render the cars dataset and display the average displacement for cars with different number of cylinders, we can apply an aggregation using the *mean* function on the second field:

```
cars = data.cars.url

alt.Chart(cars).mark_bar().encode(
    x = 'mean(Miles_per_Gallon):Q',
    y = 'Cylinders:O'
)
```



This code is equivalent to expressing the *x* field as:

```
alt.X('Miles_per_Gallon', aggregate =
                'mean', type = 'quantitative')
```

The *transform_** functions allow us to calculate new columns using specific operations. For example, *transform_aggregate* can be used to compute the same result by creating a new column with the aggregated values. This can be done as follows:

```
cars = data.cars.url

alt.Chart(cars).mark_bar().encode(
    x = 'avgMilesG:Q',
    y = 'Cylinders:O'
).transform_aggregate(
    avgMilesG = 'mean(Miles_per_Gallon)',
    groupby = ['Cylinders']
)
```

The *transform_aggregate* function requires three key pieces of information: the output field name (in this case, *avgMilesG*) for each aggregated field, the field to aggregate (*Miles_per_Gallon*), and the operation to perform (in this case, *mean*). In this example, we also group the data using the *groupby* parameter; otherwise, all data points would be treated as a single group.

Many aggregation operations are available, including: count, sum, mean, variance, stdev, median, q1, q3, ci0, ci1, min, and max.

8.3. Bin transforms

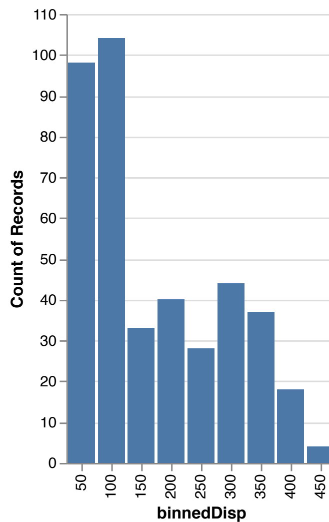
As with the previous case, we create bin transforms through the encoding and by explicitly defining them using the following code:

```
cars = data.cars.url

alt.Chart(cars).mark_bar().encode(
    alt.X('Displacement:O', bin = True),
    alt.Y('count()')
)
```

This is equivalent to:

```
cars = data.cars.url
alt.Chart(cars).mark_bar().encode(
    alt.X('binnedDisp:O'),
    alt.Y('count()')
).transform_bin(
    'binnedDisp', field = 'Displacement'
)
```

Both encodings produce the following chart:

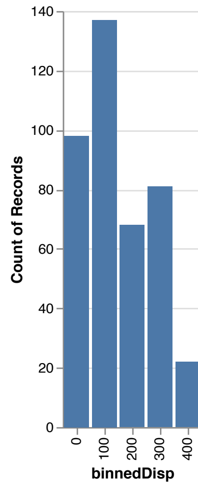


The only difference in the codes is the name of the x-axis (which can be changed in any case).

We can also limit the number of bins during the creation of the binning:

```
cars = data.cars.url

alt.Chart(cars).mark_bar().encode(
    alt.X ('binnedDisp:O'),
    alt.Y ('count()')
).transform_bin(
    'binnedDisp', field = 'Displacement',
     bin = alt.Bin(maxbins = 5)
)
```

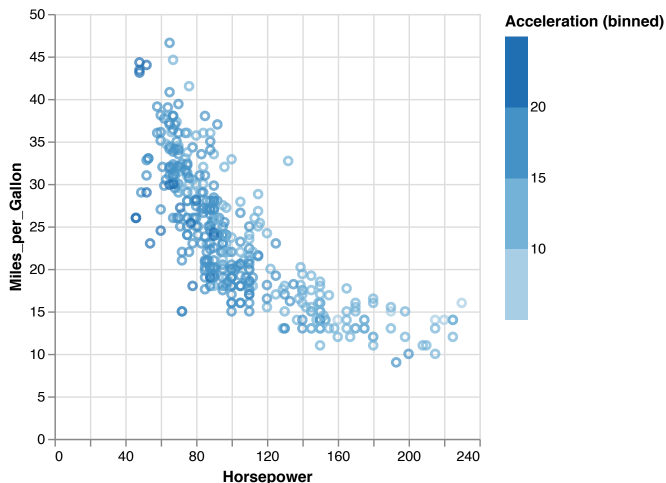
This code would render:



Additionally, we can transform the color scale using a top-level bin transform:

```
cars = data.cars.url

alt.Chart(cars).mark_point().encode(
    alt.X('Horsepower:Q'),
    alt.Y('Miles_per_Gallon:Q'),
    alt.Color('Acceleration:Q', bin = alt.Bin(maxbins = 5))
)
```

This code generates the following output:

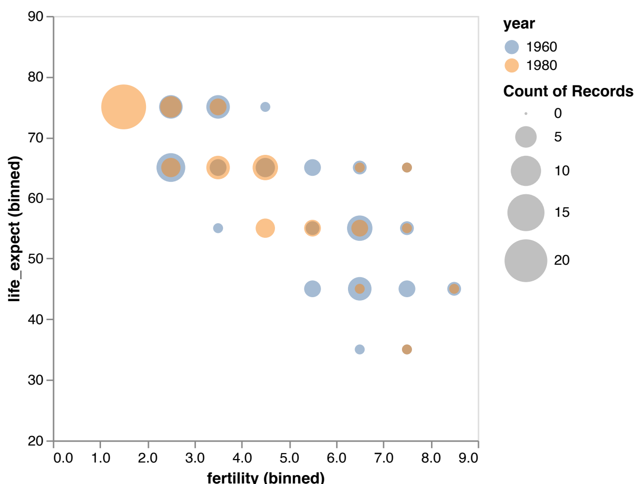


The advantage of the top-level transform is that the same named field can be used in multiple places within the chart if desired. Note the slight difference in binning behavior between encoding-based binning (which preserves the range of the bins) and transform-based binning (which collapses each bin to a single representative value).

An example of bin transforms that create a specific version of the plot is **binned scatter plots**. Binned scatter plots simplify the appearance of a scatter plot and allow for density analysis. They can illustrate the non-parametric relationship between two variables. Creating binned scatter plots in Altair is as simple as creating a scatter plot and setting the *bin* option to *True* for the variables we want to aggregate. The following example compares the fertility rate and life expectancy for all countries in the *gapminder* dataset for the years 1960 and 1980. Here, we also use selection, which is a technique that will be explained later.

```
source = data.gapminder.url

base = alt.Chart(source).mark_circle(opacity = 0.5).encode(
    alt.X('fertility:Q', bin = True),
    alt.Y('life_expect:Q', bin = True),
    alt.Size('count()'),
    alt.Color('year:N')
)
alt.layer(base.transform_filter(alt.datum.year == 1960),
          base.transform_filter(alt.datum.year == 1980))
```

The result is this plot:



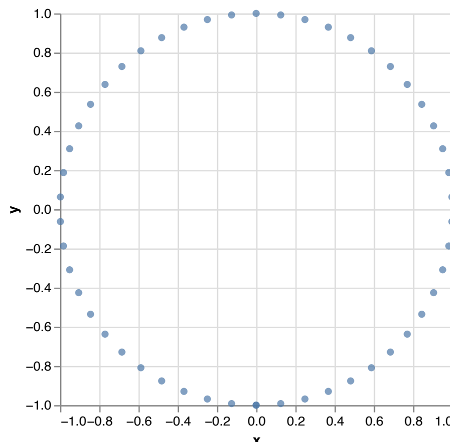
8.4. Transforming data through calculations

We can create new data from the input by using arithmetic calculations. This can be accomplished with the *transform_calculate* function, which allows us to create one or more new columns based on specified arithmetic operations. For example, if we want to draw a set of circles in a circular layout, we can calculate the circle positions using the sine and cosine functions:

```
data = pd.DataFrame({ 'angle': range(51)})
alt.Chart(data).mark_circle().encode(
    alt.X('x:Q'),
    alt.Y('y:Q'),
    order='angle:Q'
    ).transform_calculate(
        x='sin(PI*(datum.angle - 25) / 25.)',
        y='cos(PI*(datum.angle - 25) / 25.)'
    )
```

In the previous code, *x* and *y* are the names of the new fields that were added, and their values depend on the sine and cosine of the variable angle, which was created as an array ranging from 0 to 50.

Plotting the result yields:



Keep in mind that the sizes of the plot may differ from a square, which could result in a distorted appearance. In the default configuration, this is acceptable,

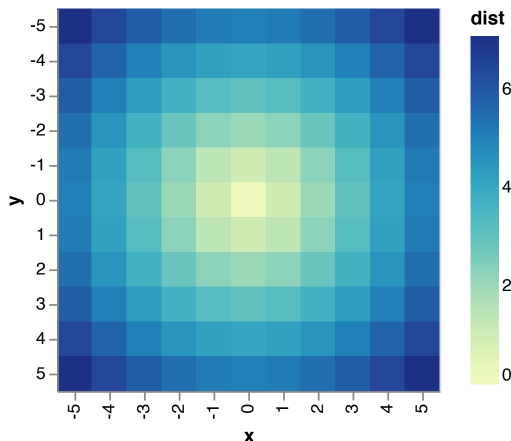
but when defining specific dimensions or if the size is determined by the container, the appearance might change.

To calculate new data, Altair uses expressions that can reference the current dataset as input. This data can be referred to by the name *datum*.

In the following example, we will create a table with elements ranging from (−5, −5) to (5, 5). These values represent the coordinates of the pixels, and we will plot a heatmap based on these data. Then we will use *transform_calculate* to compute the distances of the pixels from the center. Specifically, we will perform two tasks: First, we will calculate the distances from each pixel to the center by deriving a new column that contains the Euclidean distance to the point (0, 0). Then, we will plot a heatmap that uses these newly calculated values to encode the color:

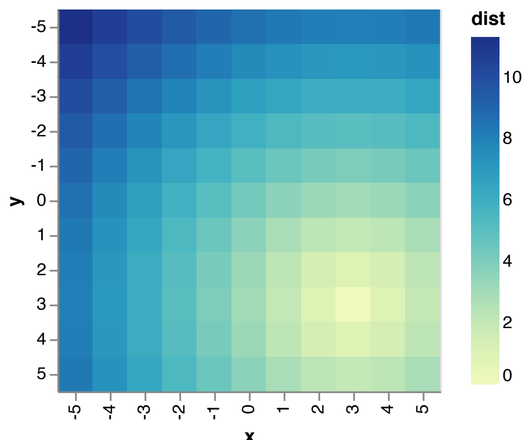
```
import numpy as np

x, y = np.meshgrid(range(-5, 6), range(-5, 6))
df = pd.DataFrame({'x': x.ravel(), 'y': y.ravel()})

alt.Chart(df).mark_rect(size=20).encode(
    alt.X('x:O'),
    alt.Y('y:O'),
   alt.Color('dist:Q')
).transform_calculate(
    dist = 'sqrt(datum.x*datum.x + datum.y*datum.y)'
)
```

This results in:



We can further work upon this function. For instance, we could calculate new columns to generate the *dist* value. Suppose we want to shift the center 3 pixels to the right and downward. We can achieve this by creating a new column that subtracts 3 from the x values and another column that subtracts 3 from the y values. These columns will be named xShift and yShift, respectively. We can then use these columns in the *sqrt* calculation (referencing them as *datum.xShift* and *datum.yShift*). The code and resulting output are shown below:

```
alt.Chart(df).mark_rect(size=20).encode(
    alt.X('x:O'),
    alt.Y('y:O'),
    alt.Color('dist:Q')
).transform_calculate(
    xShift = 'datum.x - 3',
    yShift = 'datum.y - 3',
    dist = 'sqrt(datum.xShift*datum.
xShift + datum.yShift*datum.yShift)'
)
```



8.5. Time manipulations

Various methods can be used to aggregate information based on time units. When dealing with temporal data, we can aggregate it using time units that work as functions over a temporal field. Some relevant functions are:

- *year, yearmonth, yearmonthdate*: These aggregate data, respectively, by year; by year and month; or by year, month, and day. This means that aggregation operations will occur over the data sharing these properties.

- *month, monthdate*: These aggregate data from the same month or from the same month and day.

- *date*: This function calculates aggregate functions based on the day of the month (i.e., 1–31).

- *day*: This function considers the day of the week.

Other units account for seconds or minutes. Interested readers can consult the documentation to see which values for the *timeUnit* function are available (https://altair-viz.github.io/user_guide/transform.html#timeunit-transform). Be mindful that aggregating data in ways that do not make sense can lead to misleading results. For example, aggregating temporal data from different years may not yield meaningful insights. It is your responsibility to assess whether certain aggregations are appropriate.

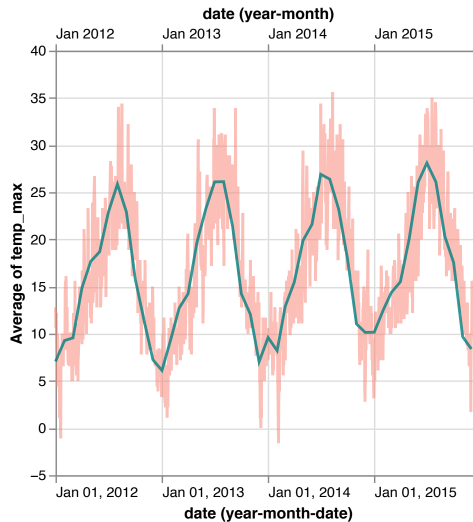
The following example uses the *seattle_weather.csv* dataset from the Vega datasets:

```
source = data.seattle_weather()

ch1 = alt.Chart(source).mark_line(
    opacity = 0.5, stroke = 'salmon').encode(
        alt.X('yearmonthdate(date):T',
              axis = alt.Axis(labelAlign='left')),
        alt.Y('average(temp_max):Q'),
    )
ch2 = alt.Chart(source).mark_line(
    opacity = 0.8, stroke = 'teal').encode(
        alt.X('yearmonth(date):T',
              axis = alt.Axis(labelAlign='left')),
        alt.Y('average(temp_max):Q'),
    )
alt.layer(ch1, ch2).resolve_scale(
    color = 'independent').resolve_axis(
        'independent'
    )
```

The plot displays two different encodings of the dataset: One averages data per month, while the other averages data per day of the month.

To ensure accurate calculations, the axes are independent, allowing you to observe the time range used for the plots.

The result is shown the following:

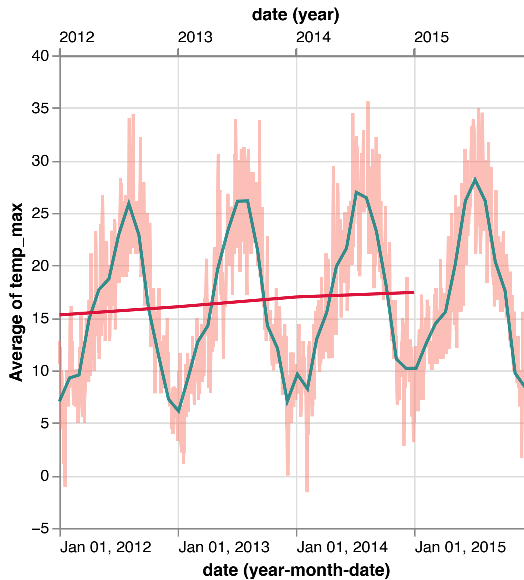


We can also demonstrate the result of aggregating by year. In this case, we plot three charts. Note that when we ask Altair to treat the three axes as independent, two of them are overlaid:

```
source = data.seattle_weather()

ch1 = alt.Chart(source).mark_line(
    opacity = 0.5, stroke = 'salmon').encode(
        alt.X('yearmonthdate(date):T',
              axis = alt.Axis(labelAlign='left')),
        alt.Y('average(temp_max):Q'),
    )
ch2 = alt.Chart(source).mark_line(
    opacity = 0.8, stroke = 'teal').encode(
        alt.X('yearmonth(date):T',
              axis = alt.Axis(labelAlign='left')),
```

```
        alt.Y('average(temp_max):Q'),
    )

ch3 = alt.Chart(source).mark_line(
    opacity = 1, stroke = 'crimson').encode(
        alt.X('year(date):T', axis = alt.Axis(labelAlign='left')),
        alt.Y('average(temp_max):Q'),
    )

alt.layer(ch1, ch2, ch3).resolve_scale(
    color = 'independent').resolve_axis(
        'independent'
    )
```

The resulting plot, with the two axes overlaid, is as follows:

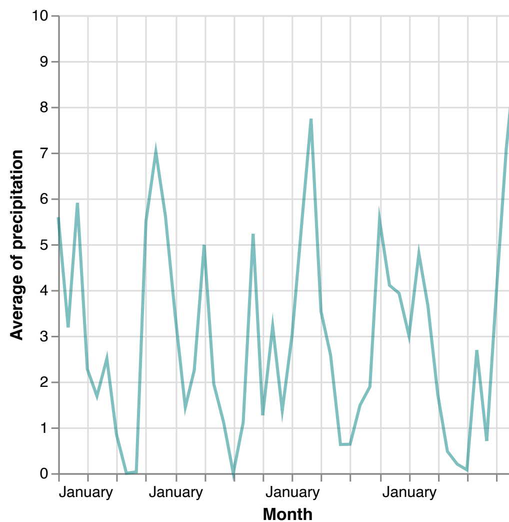


Time unit transformations can also be performed using the *transform_timeunit* function. This function allows assigning a new name to the calculated field, making it reusable.

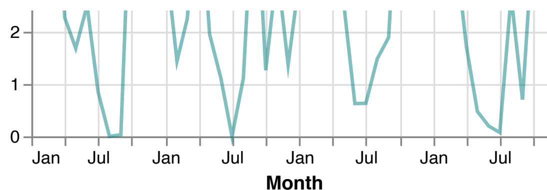
The following example transforms the date to day and uses it afterward:

```
source = data.seattle_weather()

alt.Chart(source).mark_line(
    opacity = 0.5, stroke = 'teal').encode(
        alt.X('monthY:T', title = 'Month',
              axis = alt.Axis(format='%B')),
        alt.Y('average(precipitation):Q'),
    ).transform_timeunit(monthY = 'yearmonth(date)')
```

The result is:



We used a formatting call to define how the month names should appear. If we want the data formatted in different ways, we can modify this field. For example, using the format "%b" would display the months in abbreviated form:



Several parameters can be adjusted to define how labels appear on the axes. You can rotate them, offset the initial or last values (which are typically padded to ensure visibility in the chart), align the labels (relative to the ticks), and further customize the format string (e.g., adding white spaces) as needed. However, it is essential to ensure that the displayed values are easy for users to read.

The format string also has various configurations, which are borrowed from D3.

The specifier string may contain the following directives:

- **%a** - Abbreviated weekday name
- **%A** - Full weekday name
- **%b** - Abbreviated month name
- **%B** - Full month name
- **%c** - Locale's date and time
- **%d** - Zero-padded day of the month as a decimal number [01,31]
- **%e** - Space-padded day of the month as a decimal number [1,31]; equivalent to %_d
- **%H** - Hour (24-hour clock) as a decimal number [00,23]
- **%I** - Hour (12-hour clock) as a decimal number [01,12]
- **%j** - Day of the year as a decimal number [001,366]
- **%m** - Month as a decimal number [01,12]
- **%M** - Minute as a decimal number [00,59]
- **%p** - Either AM or PM
- **%Q** - Milliseconds since UNIX epoch
- **%s** - Seconds since UNIX epoch
- **%S** - Second as a decimal number [00,61]
- **%u** - Monday-based (ISO 8601) weekday as a decimal number [1,7]
- **%w** - Sunday-based weekday as a decimal number [0,6]
- **%y** - Year without century as a decimal number [00,99]
- **%Y** - Year with century as a decimal number

You can also ask Altair to label the axis according to the week of the year, using either Monday-based or Sunday-based weeks. However, the system (neither Altair nor Vega) cannot generate those values, meaning you cannot currently aggregate by the week number. Therefore, some of these formats may be misguiding if used improperly.

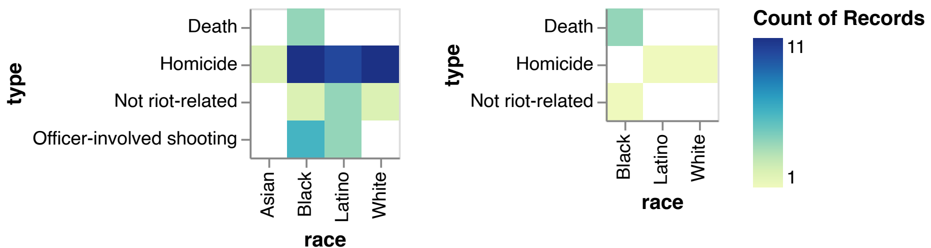
8.6. Filter transformation

The filter transformation removes part of the data based on a specified criterion. To define the criterion, we use the *datum* object, which refers to the input dataset. Each dataset field can be specified using *datum.<fieldname>*.

For example, to plot the events stored in the *la_riots* dataset for comparing author origins and filtering by gender, we can do the following:

```
df = data.la_riots()

ch1 = alt.Chart(df).mark_rect().encode(
    alt.X('race:N'),
    alt.Y('type:O'),
    alt.Color('count():Q')
)

ch1.transform_filter(
    alt.datum.gender == 'Male') | ch1.transform_filter(
          alt.datum.gender == 'Female')
```

This code filters the gender field, with the first plot showing data for males and the second for females.

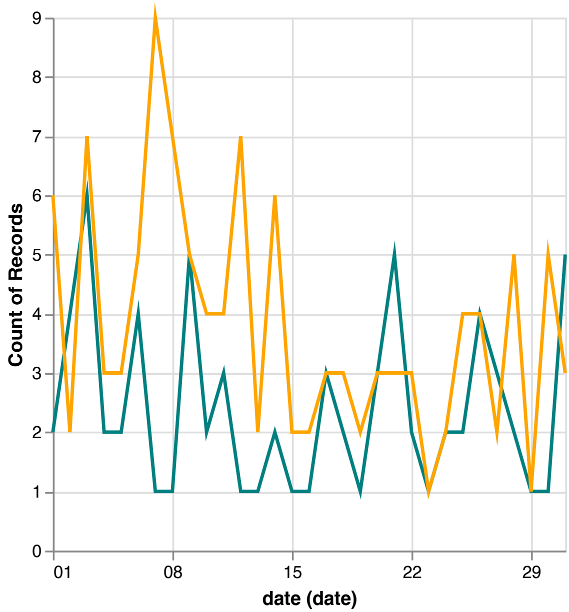


Another example involves creating a line chart to compare the number of flights for each day of the month that originated from Los Angeles (LAX) or Houston (HOU). From the *flights_2k* dataset, we filter only the flights departing from these airports.

```
df = data.flights_2k()

ch1 = alt.Chart(df).mark_line(color = 'teal').encode(
    alt.X('date(date):T'),
    alt.Y('count(destination):O'),
).transform_filter(
    alt.datum.origin == 'LAX'
)

ch2 = alt.Chart(df).mark_line(color = 'orange').encode(
    alt.X('date(date):T'),
```

```
    alt.Y('count(destination):O'),
).transform_filter(
    alt.datum.origin == 'HOU'
)

ch1 + ch2
```

The result is:

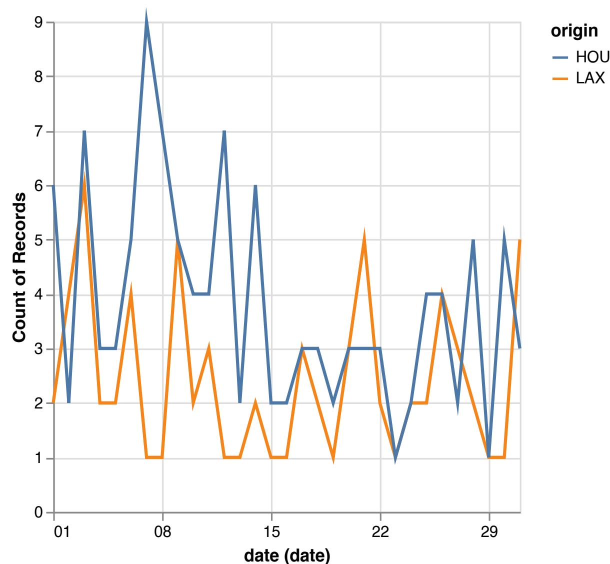


In this case, each chart plots a single line, but since the color is defined at the top level, no legend is generated. If a legend is desired, we can use a workaround by asking Altair to color-code the datasets based on the origin, even though the filter already specifies it. The result is the following code and plot:

```
df = data.flights_2k()
ch1 = alt.Chart(df).mark_line().encode(
    alt.X('date(date):T'),
    alt.Y('count(destination):O'),
    alt.Color('origin:N')
).transform_filter(
    alt.datum.origin == 'LAX'
)
```

```
ch2 = alt.Chart(df).mark_line().encode(
    alt.X('date(date):T'),
    alt.Y('count(destination):O'),
    alt.Color('origin:N')
).transform_filter(
    alt.datum.origin == 'HOU'
)

ch1 + ch2
```

The result is:



Note that the legend now has a title. While various techniques can enhance the visualization, if we want to encode colors based on a parameter with a meaningless name, we can remove the legend title by setting *title = ''*. Additionally, the colors have been automatically selected and may differ from those in the original chart. We can specify a range of valid colors for the values of the selection, as follows:

```
df = data.flights_2k()

ch1 = alt.Chart(df).mark_line().encode(
    alt.X('date(date):T'),
```

```
    alt.Y('count(destination):O'),
    alt.Color('origin:N').scale(
        domain=['LAX', 'HOU'], range=['teal', 'orange'])
).transform_filter(
    alt.datum.origin == 'LAX'
)
ch2 = alt.Chart(df).mark_line().encode(
    alt.X('date(date):T'),
    alt.Y('count(destination):O'),
    alt.Color('origin:N').scale(
         domain=['LAX', 'HOU'], range=['teal', 'orange'])
).transform_filter(
    alt.datum.origin == 'HOU'
)
(ch1 + ch2)
```

However, this can be slightly simplified by reusing a base chart:

```
df = data.flights_2k()

base = alt.Chart(df).mark_line().encode(
    alt.X('date(date):T'),
    alt.Y('count(destination):O'),
    alt.Color('origin:N').scale(
        domain=['LAX', 'HOU'], range=['teal', 'orange'])
)
ch1 = base.transform_filter(alt.datum.origin == 'LAX')
ch2 = base.transform_filter(alt.datum.origin == 'HOU')
(ch1 + ch2)
```

8.7. Lookup transform

Data lookup can be viewed as a straightforward form of data aggregation, though it has distinct characteristics. Looking up data is particularly useful when the information we want to plot is divided across different datasets.

There are several methods to achieve this:

- Merging the data
- Looking up information from another table

Whenever possible, merging the data is generally easier, as numerous Python libraries facilitate efficient data manipulation. This section presents two examples: one that utilizes the *lookup* feature and another that combines both *lookup* and data merging using Pandas.

Previously, we noted that geographic data plots require data in specific formats, such as *geoJSON* files. In this example, we will render a choropleth map of the US, displaying the employment rate per county as recorded in the *unemployment* dataset.

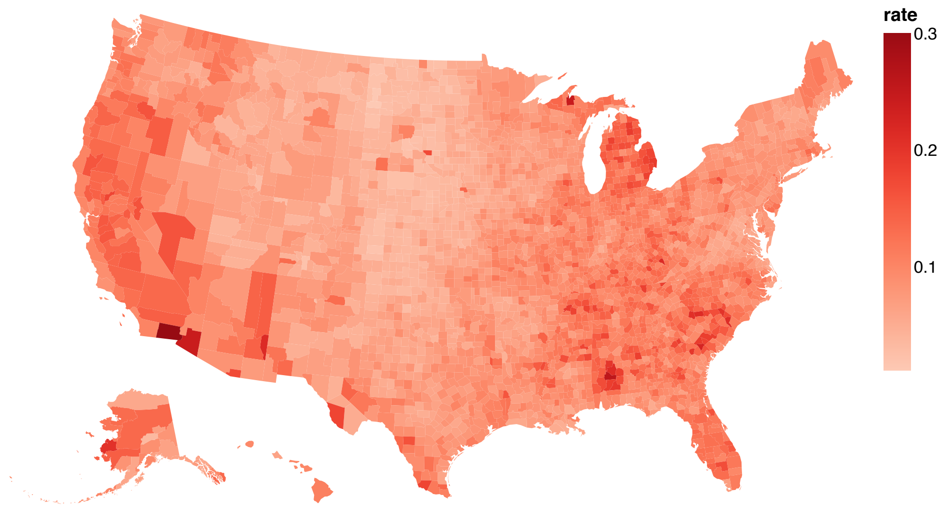
First, we will load the US map representation from the *us_10m* dataset, and then we will perform a *transform_lookup* operation to find the employment rate in the *unemployment* file. This operation has two parameters: the data origin and the search function, which are two different parameters separated by commas:

- **lookup value**: the field to search for in the second file
- **lookupData function call, with parameters**: includes the name of the secondary file, the key in the first file, and the list of fields to look up in the second file

The code is as follows:

```
map = alt.topo_feature(data.us_10m.url, feature = 'counties')
unemp_data = data.unemployment.url

alt.Chart(map).mark_geoshape().encode(
    alt.Color('rate:Q', scale = alt.Scale(scheme = 'reds'))
).transform_lookup(
    lookup = 'id',
    from_ = alt.LookupData(unemp_data, 'id', ['rate'])
).project(
    type = 'albersUsa'
).properties(
    width = 500, height = 300
)
```

The result is the following choropleth map:



In this example, we encounter a more complex problem. We aim to plot the populations of various countries using *gapminder_health_income* dataset. However, this file contains country names, while the file *world_110m* file uses country IDs. To address this, we will find a connection file (world_110m_country_codes.json) and merge it with the gapminder_health_income file. Subsequently, we will plot a map using world_100m dataset and retrieve the population values from the merged file.

First, we need to upload the file that connects the country codes to their names. This can be accomplished by clicking the folder icon on the left side of Google Colab, then either selecting the file upload button or dragging the JSON file directly into the file explorer area.

Next, we will load the uploaded file:

```
world_codes = pd.read_json('world_110m_country_codes.json')
```

For the *gapminder* data, we can load directly from the HTTP address:

```
gapminder = pd.read_csv('https://raw.githubusercontent.com/
vega/vega/master/docs/data/gapminder-health-income.csv')
```

Next, we perform the merging operation. Note that the *gapminder* dataset is on the left, and the country codes are on the right:

```
merged_data = pd.merge(gapminder, world_codes, how='left',
                       left_on='country', right_on='name')
```

Be aware that the input files may contain inconsistencies, resulting in columns with *NaN* values. Additionally, files may include different IDs or names for the same countries. If these issues are not addressed during the data cleaning phase, the final map may exhibit missing data, as we will demonstrate in this example.

After merging, we obtain two DataFrames: one with the geographic data (loaded as shown next) and another with the merged data containing health, income, and population data, as well as other fields storing country names and *IDs*.

To load the geographic data, we only need to call the appropriate function:

```
world_map = alt.topo_feature(data.world_110m.url, 'countries')
```

Now, we can proceed to render health expectancy on the map using the *transform_lookup* feature. The general process involves calling *alt.Chart* with the geometry and making a *transform_lookup* call to retrieve the parameter (in this case *health*) that we want to use as the map color. The syntax for this call would be:

```
transform_lookup(
    lookup='id',
    from_=alt.LookupData(merged_data, 'id', columns)
)
```

Here, *columns* is a list of the columns we want to import. We have chosen multiple columns to include additional information as tooltips:

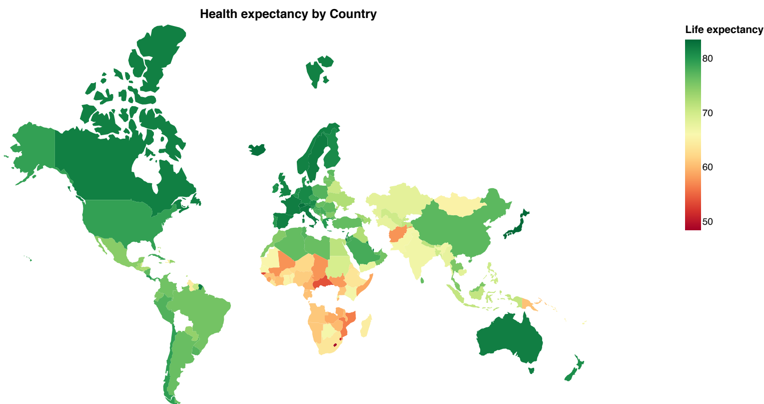
```
columns = ['country', 'health', 'income', 'population']
```

The full implementation is as follows:

```
alt.Chart(world_map).mark_geoshape().encode(
    alt.Color('health:Q',
              scale = alt.Scale(scheme = 'redyellowgreen'),
              legend = alt.Legend(title = 'Life expectancy')),
    tooltip=['country:N', 'health:Q', 'income:Q', 'population:Q']
).transform_lookup(
```

```
    lookup='id',
    from_=alt.LookupData(merged_data, 'id', columns)
).project(
    type='mercator'
).properties(
    width=800,
    height=400,
    title='Health expectancy by Country'
)
```

Note that we use the *mark_geoshape* mark type and encode life expectancy using the color option by querying the *health* field. The result is:



Some countries appear as white, notably Russia along with several countries in central Africa and South America. This occurs due to inconsistencies in the codes within the data sources.

8.8. Regression transform

The regression transform fits a two-dimensional regression model to smooth and predict data. It can fit multiple models for input data and generate new data objects that represent points for summary trend lines.

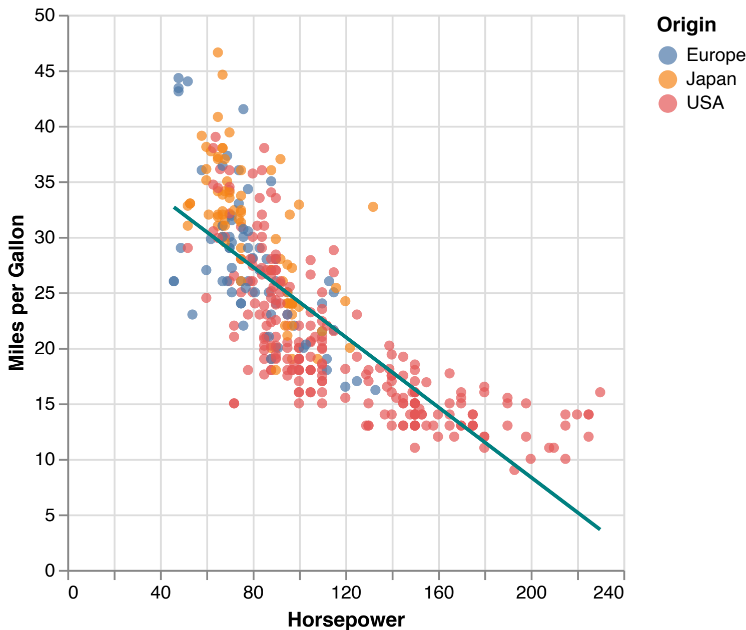
This transformation supports various parametric models, such as linear, logarithmic, polynomial, and quadratic, among others.

The following example demonstrates a linear regression based on the *cars* dataset:

```
cars = data.cars()

ch = alt.Chart(cars).mark_circle().encode(
    alt.X('Horsepower:Q', title = 'Horsepower'),
    alt.Y('Miles_per_Gallon:Q', title = 'Miles per Gallon'),
)

linear_regression = ch.transform_regression(
    'Horsepower',
    'Miles_per_Gallon'
).mark_line(color = 'teal')

(ch.encode(alt.Color('Origin:N')) + linear_regression)
```

In this case, we again used a base chart to generate the regression. However, the concatenation includes a slight modification of the base chart to incorporate the origin color. If this were encoded in the base chart, the second chart would display a second "Origin" legend. Since the color of the line is singular, the legend instead shows an option labeled "Undefined".

The result with this implementation is:



We can add multiple regression lines of different degrees by changing the fitting function using the *method* parameter in the *transform_regression* function. The following example includes a polygon of degree 2:

9
Tips and tricks

9.1. Loading large datasets

By default, Altair limits the number of rows that can be loaded to a maximum of 5000. This restriction helps prevent creating excessively large examples that may crash the browser. However, certain datasets exceed this limit, such as the 10,000-row or 200,000-row flights datasets.

To load larger datasets, you can remove the row limit using the following call:

```
alt.data_transformers.disable_max_rows()
```

However, be cautious when disabling this limit and ensure that everything functions correctly after doing so.

9.2. Adding labels

Altair provides several parameters that can be adjusted, but sometimes these are insufficient for creating the desired chart. Many limitations can be overcome by overlaying multiple charts.

A common example involves adding individual labels to charts. Labeling allows you to provide additional information like takeaways or highlight specific elements. To label all data points, you can overlay a new chart using *mark_text* with appropriate offset displacement to prevent text from overlapping any corresponding data points. For individual labels, you can create a synthetic dataset that includes coordinates and text labels for the desired annotations. These labels can be overlaid on the original chart using a new layer with the *mark_text* function.

For instance, in the example below, we want to add labels indicating “Min” and “Max” to the average lines of minimum and maximum temperatures in Seattle. This is done by creating a custom chart with text annotations.

First, we create two line charts for temperature data:

```
source = data.seattle_weather.url

ch1 = alt.Chart(source,
                title = 'Seattle temperature'
                ).mark_line(color = 'crimson').encode(
    alt.X('month(date):T', title = 'Month'),
    alt.Y('mean(temp_max):Q', title = 'Temperature')
).transform_calculate(
    year = 'year(datum.date)'
).transform_filter(
    alt.datum.year == 2015
)

ch2 = alt.Chart(source).mark_line(color = 'steelblue').encode(
    alt.X('month(date):T', axis =
          alt.Axis(labels = False, title = '', ticks = False)),
    alt.Y('mean(temp_min):Q', title = '')
).transform_calculate(
    year = 'year(datum.date)'
).transform_filter(
    alt.datum.year == 2015
)
```

Note that we filter the data to include only the specified year by extracting the year using the *transform_calculate* operation.

Afterward, we create a third chart containing the text annotations and overlay it onto the others:

```
dfTexts = pd.DataFrame({'x': [10, 40],
                        'y': [4, 22],
                        'text': ['Min', 'Max']})

ch3 = alt.Chart(dfTexts).mark_text(
    fontStyle = 'bold', font = 'Courier', fontSize = 16
```

```
).encode(
    alt.X('x:Q',
          scale = alt.Scale(domain = (0, 50)),
          axis = alt.Axis(labels = False, title = '',
                          ticks = False)),
    alt.Y('y:Q'),
    alt.Text('text')
)
alt.layer(ch1, ch2, ch3)
```

Note the positions of the labels are based on synthetic data, accounting for both the desired label locations and the size of the virtual space in which the plot will be rendered.

Since Altair maps the virtual plot space to match the physical space, we must displace the text marks within the virtual space and ensure that the virtual space is large enough to accommodate them. To achieve this, we customized the axis ranges for the text plot to extend beyond the plotted data points. In this case, our domain spans from 0 to 50, while the data points are plotted at coordinates (10, 4) and (40, 22).

Additionally, we customized the font style and also made it bold.

The result is:

9.3. Customizing axes

As mentioned earlier, we can adjust the axis size by modifying the field *scale* in the *X* field:

```
alt.X('x:Q',
      scale = alt.Scale(domain = (0, 50)),
      axis = alt.Axis(labels = False, title = '', ticks = False)),
```

In this example, the axis is made invisible. If not adjusted, the axis corresponding to the plot would still appear.

Setting the *labels* parameter to false removes the labels, and setting the parameter *ticks* to false removes the ticks from the axis.

This customization can be useful for multiple views, such as small multiples, or when encoding one chart over another chart, such as in graduated symbol maps.

9.4. Saving charts

Charts can be saved in various formats. To save a chart as an image, the following calls can be used:

```
chart.save('chart.png')
chart.save('chart.svg')
```

Depending on the browser, you may need to install certain extensions to generate the files you are working with (for more information on the different options and solutions, refer to the Altair documentation: https://altair-viz.github.io/user_guide/saving_charts.html).

Additionally, you can also save charts in HTML format by using:

```
chart.save('chart.html')
```

To save charts in *JSON* format, use the same function with a *.json* extension:

```
chart.save('chart.json')
```

To embed a *JSON* file in any web page, the *vegaEmbed* library is required (https://github.com/vega/vega-embed).

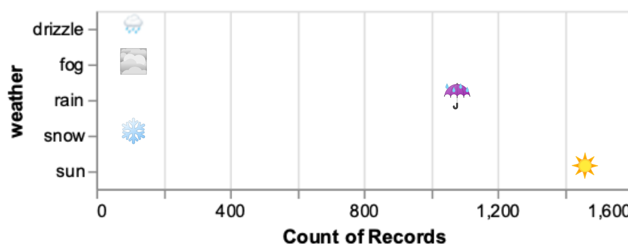
9.5. Plotting graphical elements

There are several ways to plot icons and images in Altair. Early versions did not support real image files, but they could plot emojis and isotypes. The simplest method involves using emojis, which can be encoded as Unicode symbols.

To plot emojis, Altair uses *mark_text* to plot Unicode symbols corresponding to the desired emojis. For supported emojis and their Unicode values, you can refer to the full emojis list at https://unicode.org/emoji/charts/full-emoji-list.html. To add emojis to your code, you can simply copy them from the "Browser" column on the page and paste them into your Colab file. You can then create a simple chart with emojis representing the text by using a DataFrame with emojis or by calculating them in a *transform_calculate*. Here is an example:

```
source = pd.read_csv('https://raw.githubusercon-
tent.com/vega/vega/master/docs/data/weather.csv')

alt.Chart(source).mark_text().encode(
    alt.X('count():Q'),
    alt.Y('weather:N'),
    alt.Text('emoji:N')
).transform_calculate(
    emoji =
    "{'drizzle':'🌧️', 'rain':'☔️', 'sun':'☀️
',    'snow':'❄️', 'fog':'🌫️'}[datum.weather]"
)
```

This results in:



For more detailed icons, an alternative to emojis is using an SVG path, which can be added as a shape property to your plot. An example can be found in the Altair gallery at https://altair-viz.github.io/gallery/isotype.html. Note that SVG paths are required for the isotypes. This can be extremely inconvenient because Altair

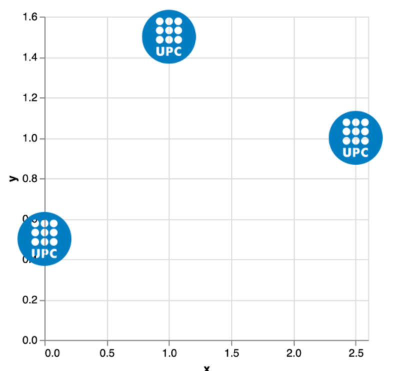
(and Vega) support only simple SVG paths, which are not typically used by most applications to create SVGs.

9.6. Plotting real images

Since version 4.0, Altair supports image data. In this case, the image files do not need to be in SVG format, which can be quite cumbersome. The following example demonstrates how to plot images linked from public websites within a chart.

```
source = pd.DataFrame.from_records([
      {"x": 0., "y": 0.5, "img":
"https://upload.wikimedia.org/wikipedia/commons/
thumb/9/97/Logo_UPC.svg/110px-Logo_UPC.svg.png"},
      {"x": 1., "y": 1.5, "img":  "https://upload.
wikimedia.org/wikipedia/commons/thumb/9/97/
Logo_UPC.svg/110px-Logo_UPC.svg.png"},
      {"x": 2.5, "y": 1, "img":  "https://upload.
wikimedia.org/wikipedia/commons/thumb/9/97/
Logo_UPC.svg/110px-Logo_UPC.svg.png"}
])
alt.Chart(source).mark_image(
    width=100, height=50
).encode(
    alt.X('x'),
    alt.Y('y'),
    url='img'
)
```

The result is:



10
Interaction basics

To enhance data exploration, it is essential to add interaction features to our charts.

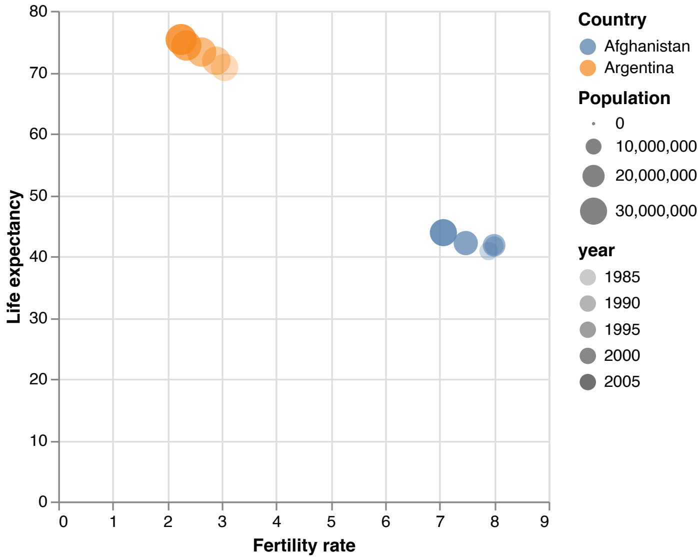
10.1. Basic interaction: Pan and zoom

The simplest way to make a chart interactive is to configure it to receive interaction events. This can be done by defining the chart as *interactive* after the chart's definition. For example, the following code generates a scatter plot of fertility rates for Afghanistan and Argentina across several years after 1980.

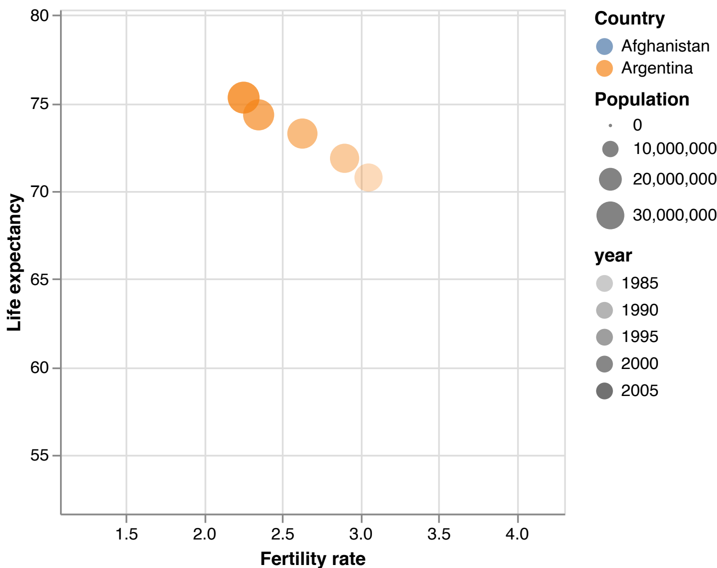
```
df = data.gapminder()

alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q', title = 'Fertility rate',
           scale = alt.Scale(domain = (0,9))),
    alt.Y('life_expect:Q', title = 'Life expectancy'),
    alt.Size('pop:Q', title = 'Population'),
    alt.Color('country:N', title = 'Country'),
    opacity = 'year:O'
).transform_filter((alt.datum.year > 1980)
& ((alt.datum.country == 'Argentina')
| (alt.datum.country == 'Afghanistan')))
```

In this example, each country is coded in a different color, and the years are encoded with varying opacities.

The result is:



To zoom in on a specific set of points, you can add "*.interactive()*" to the chart definition. This method activates zoom and pan options, which can be accessed by pinching on a touch screen or using the mouse wheel to zoom in and out, and by swiping or dragging the mouse to pan. For instance, you could zoom in on Argentina's data and pan across it to explore the points further:



However, this feature offers only a limited range of interactions. To fully unlock the power of interaction, we need to incorporate more advanced actions, such as selection, highlights, and filters. These actions do not require the *interactive()* method.

Altair provides two main approaches to interaction: using widgets to modify view parameters (e.g., sliders to filter elements); and making data points directly selectable and responding to those selections. For more advanced interactions, Altair relies on three main components:

– **The *selection* object**: Captures user interactions via mouse or other inputs (e.g., dropdown menus or radio buttons).

– **The *condition* function**: Changes the visual properties of the chart based on the selection input.

– **The *bind* property**: Establishes a two-way binding screen between the selection and an input element.

We will begin by exploring simple examples of interaction using widgets.

10.2. Basic interaction: Filter based on parameters

Widgets can be used to **filter** data or modify other viewing parameters. To do this, we create parameter objects and bind them to widgets. Parameter objects function like regular variables but can be changed via widget bindings. This is done by declaring a parameter as we would a variable, with the *alt.params* call, which can take an initial value and a binding object.

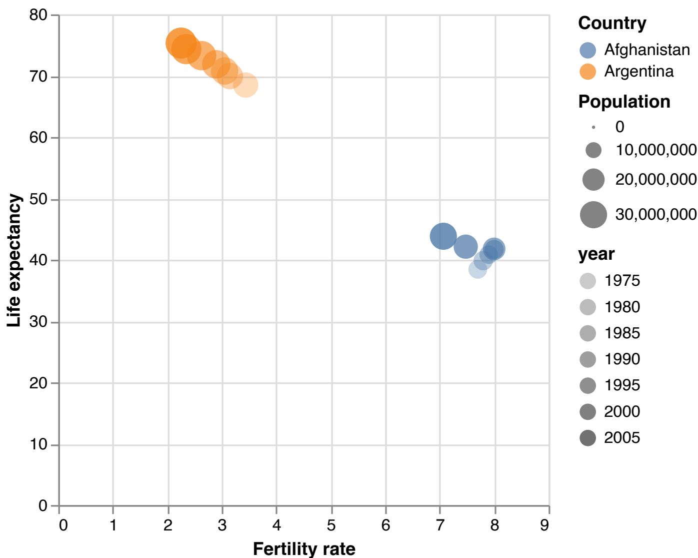
To bind a parameter to a widget, we first declare the widget. For a slider, we use the declaration function *alt.binding_range*, with the parameters: min (minimum value), max (maximum value), step (step size), and name (the widget label).

The following example allows us to filter the visible points of the *gapminder* data based on a parameter controlled by a slider. The parameter will define the years to filter.

```
slider = alt.binding_range(min=1960,
max=2005, step=5, name='year:')
op_var = alt.param(value=1960, bind=slider)
```

```
alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q', title = 'Fertility rate',
          scale = alt.Scale(domain = (0,9))),
    alt.Y('life_expect:Q', title = 'Life expectancy'),
    alt.Size('pop:Q', title = 'Population'),
    alt.Color('country:N', title = 'Country'),
    opacity = 'year:O'
).transform_filter((alt.datum.year > op_var)
& ((alt.datum.country == 'Afghanistan')
| (alt.datum.country == 'Argentina'))).add_params(op_var)
```

By dragging the slider, we can, for example, filter the minimum year to 1970:



We could also use the slider to set parameters such as the opacity of the data points, as in the following example:

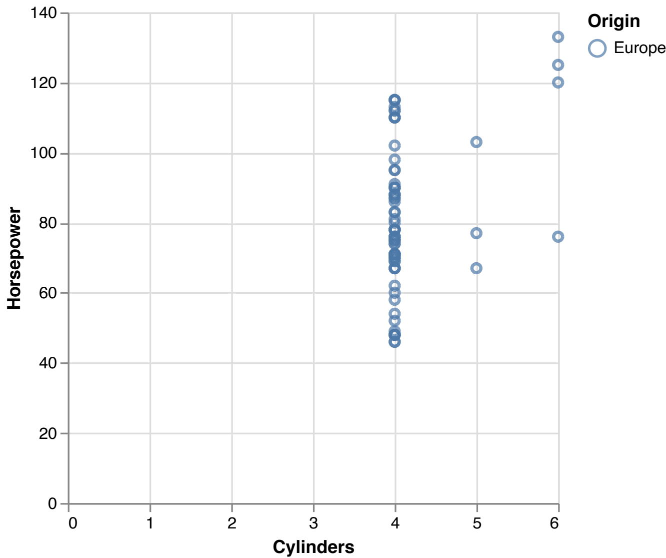
```
slider = alt.binding_range(min=1960,
max=2005, step=5, name='year:')
op_var = alt.param(value=1960, bind=slider)

alt.Chart(df).mark_circle().encode(
```

```
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    alt.Color('country:N'),
    opacity = 'year:O'
).transform_filter((alt.datum.year > op_var)
& ((alt.datum.country == 'Afghanistan')
| (alt.datum.country == 'Argentina'))).add_params(op_var)
```

Another example involves using a radio button to select which data subgroup to display. The cars dataset, which includes cars from Europe, Japan, and the USA, allows us to use a radio button to select which origins to display, as in the following code:

```
df = data.cars()

radio_button = alt.binding_radio(
    labels = ['Europe', 'USA', 'Japan'],
    options = ['Europe', 'USA', 'Japan'],
    name = 'Country')

origin_par = alt.param(value = 'Europe', bind = radio_button)

alt.Chart(df).mark_point().encode(
    alt.X('Cylinders:Q'),
    alt.Y('Horsepower:Q'),
    alt.Color('Origin:N')
).transform_filter(alt.datum.Origin == origin_par
                   ).add_params(origin_par)
```

When Europe is selected, the result looks like this:



Note that since only a single selection is made, the data points are colored uniformly. Later, we will explore how to solve this using conditions.

We have reviewed some interaction techniques that allow us to explore data. However, the most powerful interactions involve manipulating data points directly and using those selections for actions like cross-highlighting. Altair supports direct interaction through two selection methods: selecting individual data points or groups of points. The next section covers selection, followed by an exploration of how conditions are applied. Finally, we will discuss more advanced bindings and other tips.

11
Selection

Selection is one of the most fundamental interaction methods in data visualization. It involves choosing one or multiple items, which are then treated differently.

The simplest action on selected items is highlighting. However, more complex tasks can be performed with Altair, such as filtering and cross selection.

Altair offers two types of selection:

- **Point:** A single data item is selected.
- **Interval:** Multiple items are selected.

11.1. Individual selection

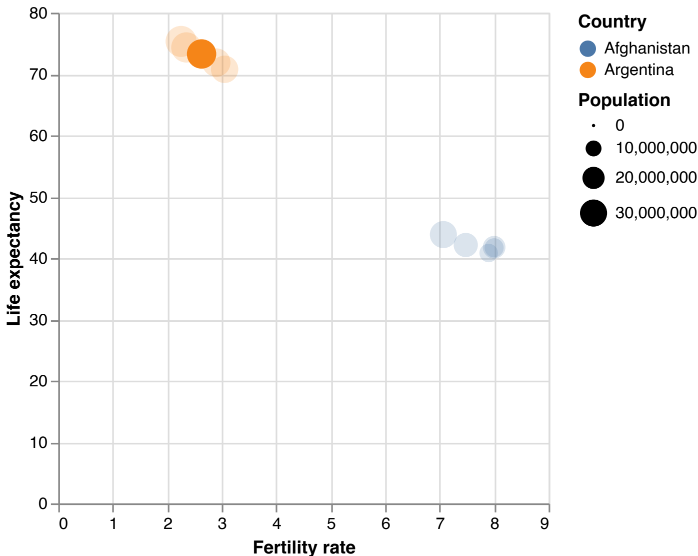
To create a selection tool, two steps are required: declaring a selection object and adding it to the chart object as a parameter.

Individual items can be selected using the *selection_point* object, as shown in the following code:

```
df = data.gapminder()

select_point = alt.selection_point()

alt.Chart(df).mark_circle().encode(
      alt.X('fertility:Q',
      title = 'Fertility rate',
      scale = alt.Scale(domain = (0,9))),
    alt.Y('life_expect:Q', title = 'Life expectancy'),
    alt.Size('pop:Q', title = 'Population'),
    alt.Color('country:N', title = 'Country'),
```

```
    opacity = alt.condition(select_point, alt.
value(1.0), alt.value(0.2))
).transform_filter((alt.datum.year > 1980)
& ((alt.datum.country == 'Afghanistan')
| (alt.datum.country == 'Argentina'))).add_params(select_point)
```

A simple selection by itself does nothing unless the chart is programmed to respond to it. In this case, the opacity of the data points changes based on the selection: Selected data points are fully opaque, while unselected points are assigned an opacity of 0.2. This is handled by the condition call applied to the opacity channel. The result is illustrated below:



If the selection is cleared by clicking elsewhere, the original colors are restored.

For charts with small targets, where precise clicking may be difficult, we can relax the selection condition so that the closest object to the mouse is considered selected at the moment of clicking.

This is done by adding the parameter *nearest* to the selection object and setting it as *True*. Additionally, selections can be triggered without clicking by using the *hover* operation from the library. This can be achieved by adding the parameter *on* to the selection with the value *'mouseover'*, as in the following line:

```
select_point = alt.selection_point(on = 'mouseover',
                                   nearest = True)
```

Although point selection is intended for a single data point, it can return a list of data points if the *Shift* key is held while moving the mouse.

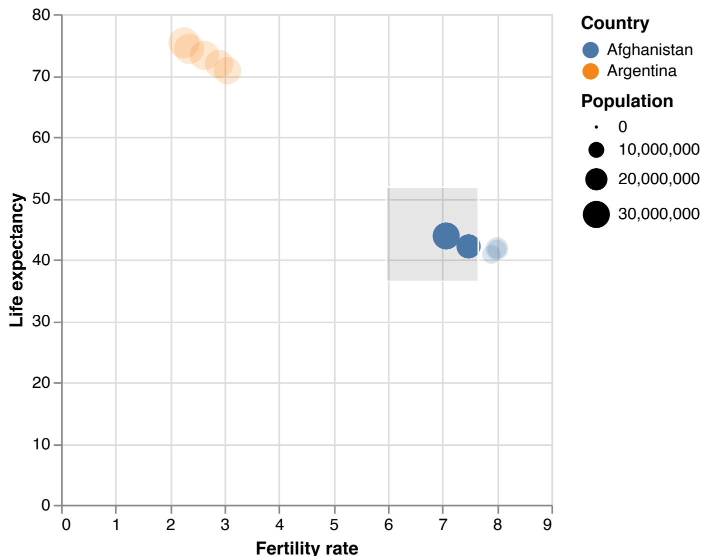
11.2. Interval selection

Multiple selections behave similarly to individual selections. Users can select multiple elements by dragging the mouse (clicking at the start point and dragging). The selection object is defined using *alt.selection_interval* and, as in the previous case, it is added with the *add_params* function call.

By switching the point selection to interval selection, we can replicate he previous behavior for a group of objects by selecting a region (i.e., making those inside the selection fully opaque and those outside semitransparent). This is achieved simply by changing the selection type, while the rest of the code, other than the name, remains the same:

```
select_Interval = alt.selection_interval()
```

Dragging over the chart will produce the following effect:



This selection technique is called **brush** (or *brushing*). A brush is an operation that involves selecting a rectangular region defined by the initial mouse press position and the point where the left button is released.

Selections also allow configurations of default behavior when no element is selected, using the *empty* parameter, which can be set to *'True'* (all elements are selected by default) or *'False'* (no elements are selected). This lets us control the initial appearance of charts before interaction, for instance, when the interaction changes colors.

However, selections can be configured in different ways. For example, we might want to brush along only one axis. This can be done by specifying the selection to be performed on the *'x'* encodings, as shown here:

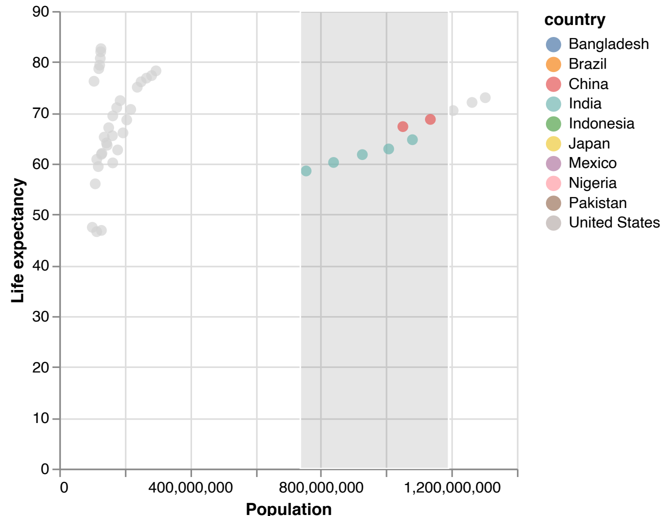
```
df = data.gapminder()

select_interval = alt.selection_interval(
    empty = False, encodings = ['x'])

alt.Chart(df).mark_circle(size = 50).encode(
    alt.X('pop:Q', title = 'Population'),
    alt.Y('life_expect:Q', title = 'Life expectancy'),
    color = alt.condition(select_interval,
                          alt.Color('country:N'),
                          alt.value('lightgray'))
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)
```

In this plot, countries with more than 100 million citizens are displayed. Data points retain their original color unless selected. This is achieved through the parameters in the *selection_interval* function, whereby the first parameter indicates that no selection results in an empty selection, and the second specifies that the selection process works only along the *x*-axis.

If we want no points colored until a selection is made, we can change the empty parameter to *True*.

```
select_interval = alt.selection_interval(
    empty = True, encodings = ['x'])
```

The result is an *x*-axis selection that renders only the selected elements in the proper color:



Note that in a previous example, we could not freely apply the chart's *interactive* property to make interval selection available. This is because the *interactive* property defines a selection mode for scales, allowing users to modify the scales interactively by zooming or dragging.

We can set similar behavior using scale selection instead of the *interactive* property, as follows:

```
select_scales = alt.selection_interval(bind = 'scales')
```

This behavior is similar to that of the *interactive* function but also interacts with the brushing process.

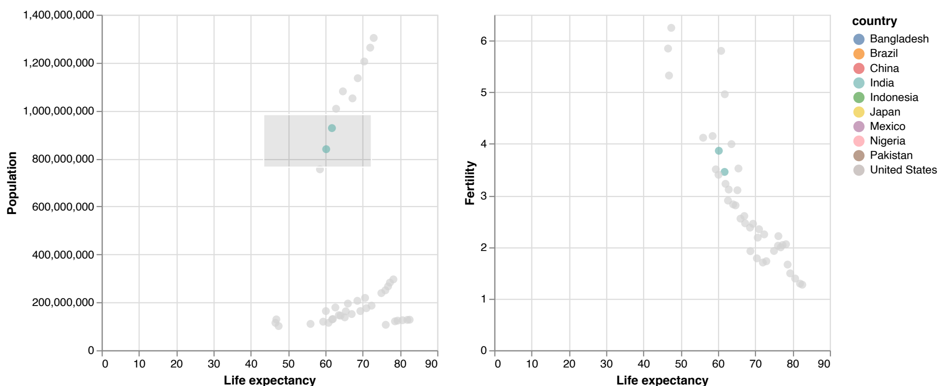
However, the full power of selections is realized when they are applied across multiple charts. This process, called cross-selection or cross-highlighting, results in what is commonly known in visualization as **Coordinated Views** (also referred to as linked or coupled views). Cross-highlighting involves using the selection in one chart to highlight corresponding values in another.

Cross selection can also be achieved using the selector object. For example, if we want to analyze the relationship between fertility rate and life expectancy among populations across different European countries, we can display both charts side

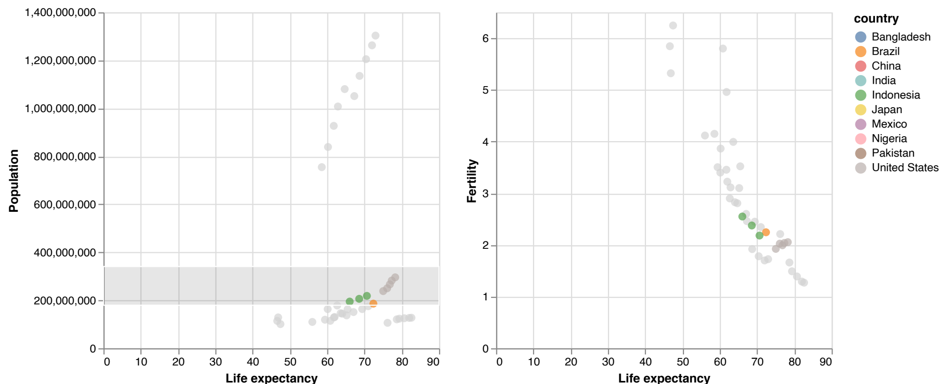
by side and make them share a selection for better data inspection. The following code shows how this can be done:

```
df = data.gapminder()

select_interval = alt.selection_interval(empty = False)

ch1 = alt.Chart(df).mark_circle(size = 50).encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('pop:Q', title = 'Population'),
    color = alt.condition(select_interval,
                              'country:N', alt.value('lightgray'))
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)

ch2 = alt.Chart(df).mark_circle(size = 50).encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('fertility:Q', title = 'Fertility'),
    color = alt.condition(select_interval, 'coun-
try:N', alt.value('lightgray'))
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)

ch1 | ch2
```

And the result, with a selected region:



Here, we selected along one axis in the left chart, and the same data points were highlighted in the right chart, regardless of whether the same coordinates were shared. This is even more apparent when we restrict the selection by the y-encoding. This can be done by adding the parameter *encodings = ['y']* to the selection, which results in:

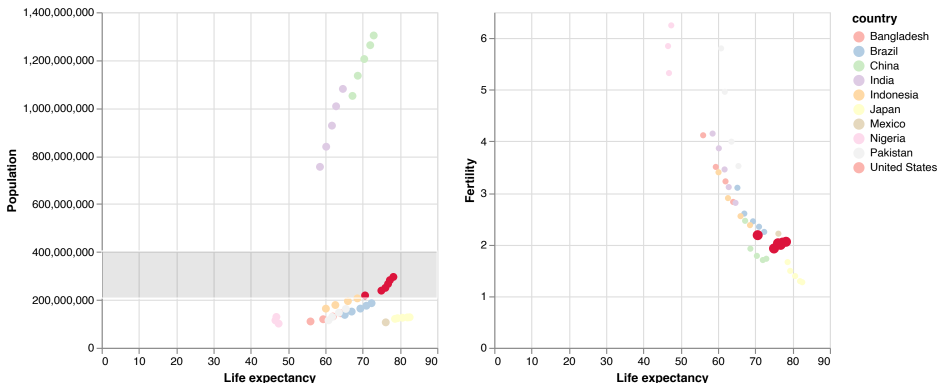


To further enhance the communication of selected items and add more color to the chart, we can retain the original country colors with the nominal palette and change the selected items to crimson. Additionally, since the right chart does not display the boundaries of the selected region (because no such region exists), we can increase the size of the selected items in the second chart. This can be done by adding a second condition. Below are the changes in the selection definition and conditions:

```
df = data.gapminder()

select_interval = alt.selection_interval(
    empty = False, encodings = ['y'])

ch1 = alt.Chart(df).mark_circle(size = 50, opacity = 1.0).encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('pop:Q', title = 'Population'),
    color = alt.condition(
        select_interval, alt.value('crimson'),
        alt.Color('country:N',
```

```
                        scale=alt.Scale(scheme = 'pastel1'))),
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)

ch2 = alt.Chart(df).mark_circle(size = 50, opacity = 1.0).encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('fertility:Q', title = 'Fertility'),
    color = alt.condition(
        select_interval, alt.value('crimson'),
        alt.Color('country:N',
                        scale=alt.Scale(scheme = 'pastel1'))),
    size = alt.condition(
        select_interval, alt.value(80), alt.value(30))
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)

ch1 | ch2
```

Note that we made additional adjustments to improve the communication. This includes using a pastel color palette (*pastel1*), which is less intense, making the crimson selection color stand out more clearly. The result is:



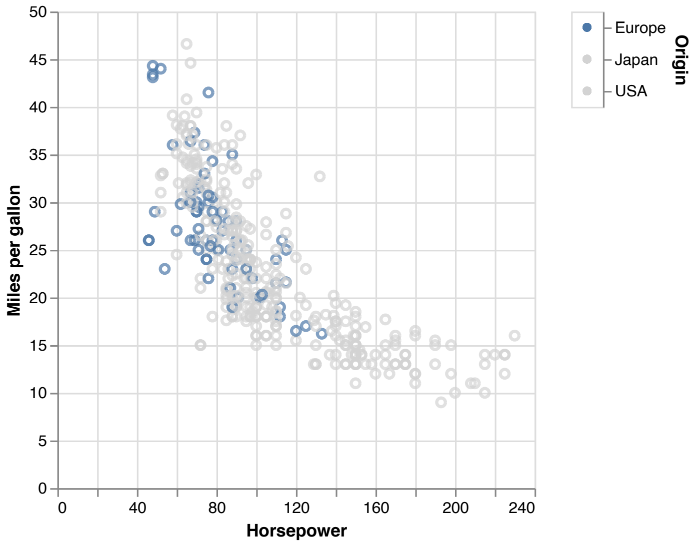
11.3. Selecting by fields or encodings

We can customize the selection by focusing on either fields or encodings. For example, using the cars dataset, we might be interested in selecting based on the origin of the cars. To achieve this, we need a widget that displays this information. This can be done by creating a legend that encodes the information and also serves as an input for the section process. This legend is represented as a small chart with three items corresponding to each origin value. Since we have three different values encoded in different colors, we can set the selector object to respond to the color encoding of the item clicked:

```
df = data.cars()

select_point = alt.selection_point(encodings = ['color'])

color = alt.condition(select_point,
                      alt.Color('Origin:N', legend = None),
                      alt.value('lightgray'))

scatter = alt.Chart(df).mark_point().encode(
    alt.X('Horsepower:Q'),
    alt.Y('Miles_per_Gallon:Q').title('Miles per gallon'),
    color = color,
    tooltip = 'Name:N'
).add_params(select_point)

legend = alt.Chart(df).mark_circle().encode(
    alt.Y('Origin:N').axis(orient='right'),
    color = color,
).add_params(select_point)

scatter | legend
```

Here, the color value is determined outside the chart definition, using the *alt.condition* function, and it is then applied inside it to define the field *color*. To avoid confusion, it is good practice to give a separate name to the variable that holds the result of the condition. Additionally, we added the tooltip field to show detailed data (in this case the item's name) when hovering over a data point. By defining the color property before using it in the chart, we can reuse its definition.

The result, with European cars selected, looks like this:



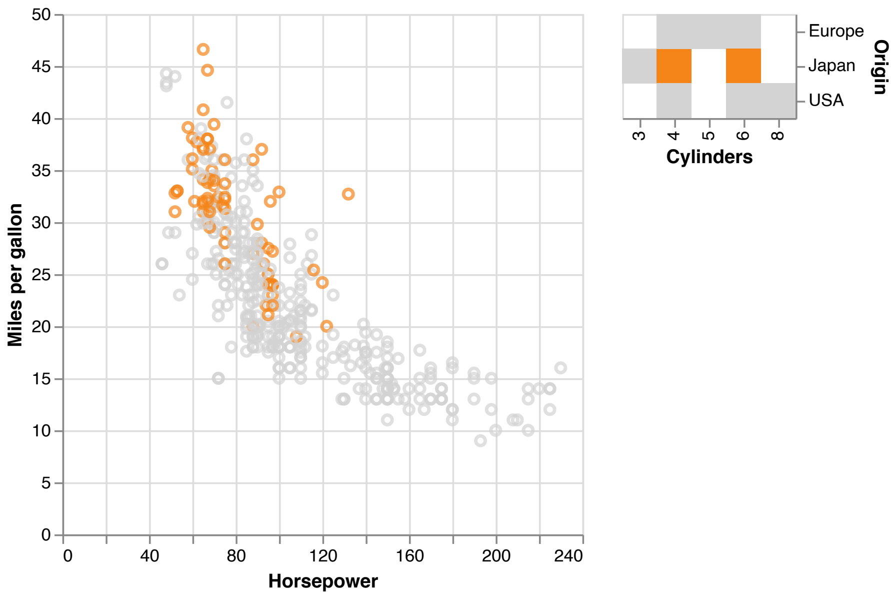
A more complex selection could involve multiple fields, such as the car's origin and the number of cylinders. Since there are various combinations of cylinders and origin countries, we can create a more advanced selector, such as a matrix that shows all valid combinations. By clicking elements in the matrix, the user can select cars that match the chosen properties. Additionally, we can enable multiple selections, allowing users to hold the Shift key to select multiple options from the legend:

```
df = data.cars()

select_point = alt.selection_point(-
fields = ['Origin', 'Cylinders'])

color = alt.condition(select_point,
                      alt.Color('Origin:N', legend = None),
                      alt.value('lightgray'))

scatter = alt.Chart(df).mark_point().encode(
    alt.X('Horsepower:Q'),
    alt.Y('Miles_per_Gallon:Q').title('Miles per gallon'),
    color = color,
    tooltip = 'Name:N'
).add_params(select_point)
```

```
legend = alt.Chart(df).mark_rect().encode(
    alt.X('Cylinders:O'),
    alt.Y('Origin:N').axis(orient='right'),
    color = color,
).add_params(select_point)

scatter | legend
```

In this case, the selection is based on fields, not encodings.

The result, when selecting all Japanese cars with 4 or 6 cylinders, looks like this:



12 Binding interactions to user input

We have previously demonstrated how widgets can be linked to user interaction, thereby allowing the modification of variable values that affect visualizations. In this section, we describe the available widgets and provide examples of how they can be bound to visualizations. Note that the images depict only the chart portion, since the widget itself is not captured when exporting chart images, as we did here.

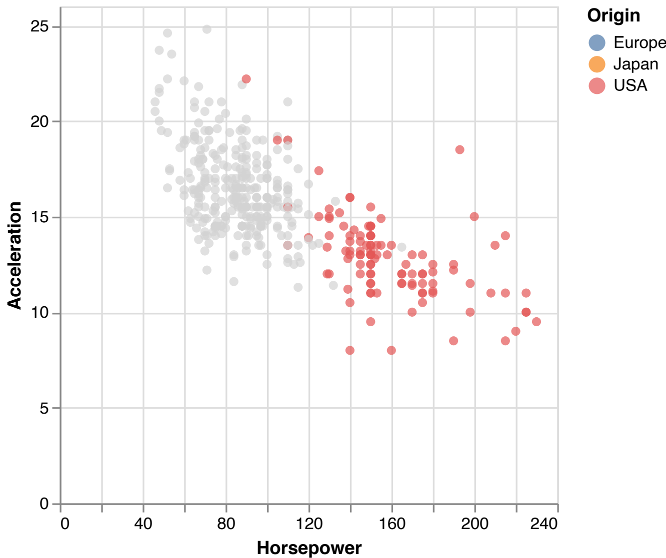
12.1. Sliders

Sliders are input widgets used to select a value within a specified range. The slider definition (*alt.binding_range*) allows users to set the value range (minimum and maximum), the step, and an initial value, if needed. In the following example, a slider is used to select cars based on the number of cylinders, in a plot that compares horsepower and acceleration:

```
df = data.cars()

slider = alt.binding_range(min=3, max=8, step=1,
                           name='Cylinders: ')

cyls = alt.selection_point(fields = ['Cylinders'],
                           bind = slider)

color = alt.condition(cyls,
                      alt.Color('Origin:N'),
                      alt.value('lightgray'))

alt.Chart(df).mark_circle(filled = True).encode(
    alt.X('Horsepower:Q'),
```

```
    alt.Y('Acceleration:Q'),
    color = color,
    tooltip = 'Name:N'
).add_params(cyls)
```

The result shows that all cars with 8 cylinders come from the US, as can be seen here:



12.2. Dropdown menus

We can also make selections using **dropdown** menus. In the following example, we use the *gapminder* dataset and include region names. The original data encodes world regions as a *cluster* variable in the data. First, we add the region names to the dataset using the *transform_lookup* function. Then, we create a selection object based on a dropdown menu with these region names. Finally, we filter the data to show only the year 2005.

The first part of the code declares the DataFrame that maps cluster IDs to their respective names and defines the selection object.

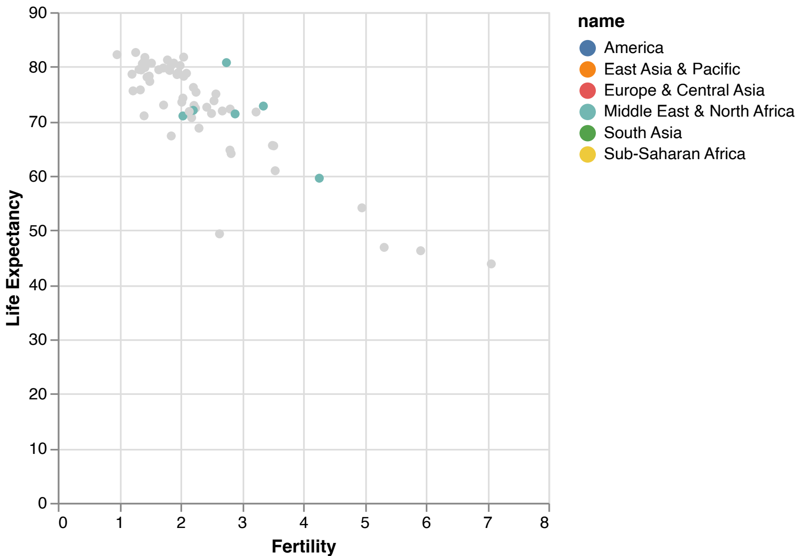
```
source = data.gapminder()

clusters = pd.DataFrame([
```

```
    {"id": 0, "name": "South Asia"},
    {"id": 1, "name": "Europe & Central Asia"},
    {"id": 2, "name": "Sub-Saharan Africa"},
    {"id": 3, "name": "America"},
    {"id": 4, "name": "East Asia & Pacific"},
    {"id": 5, "name": "Middle East & North Africa"},

])

cluster_dropdown = alt.binding_select(
    options = ['South Asia', 'Europe & Central Asia',
               'Sub-Saharan Africa', 'America',
               'East Asia & Pacific', 'Middle East & North Africa'
    ]
)

dropSelect = alt.selection_point(
    fields = ['name'],bind = cluster_dropdown,
    name = 'Region')
```

Next, we create the chart and add the selection to it:

```
base = alt.Chart(source).mark_circle(opacity = 1.0).encode(
    alt.X('fertility:Q', title = 'Fertility'),
    alt.Y('life_expect:Q', title = 'Life Expectancy'),
    color = alt.condition(dropSe-
lect, 'name:N', alt.value('lightgray'))
).transform_lookup(
    lookup = 'cluster',
    from_ = alt.LookupData(
        data = clusters, key = 'id', fields = ['name']
    )
).add_params(dropSelect)

base.transform_filter(alt.datum.year == 2000)
```

The result, with the *Middle East & North Africa* region selected, is the following:



We can further exploit this approach by allowing users to navigate through different years in the dataset. To do this, we add another selection condition, enabling the user to choose the year they want to display.

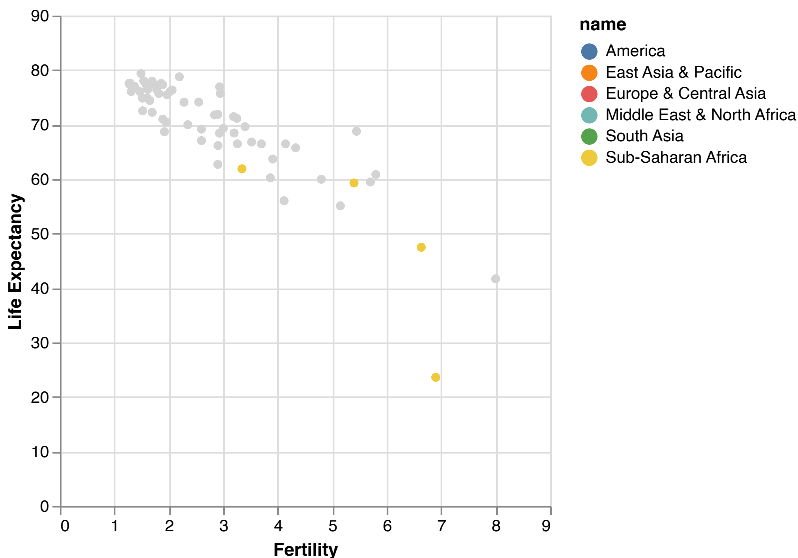
This requires adding a second selection and adjusting how the data are rendered. For unselected years, the data will be made transparent.

Here is the code for this behavior (the *clusters* DataFrame declaration is intentionally omitted):

```
cluster_dropdown = alt.binding_select(
    options = ['South Asia', 'Europe & Central Asia',
               'Sub-Saharan Africa', 'America',
               'East Asia & Pacific', 'Middle East & North Africa'])

dropSelect = alt.selection_point(
    fields = ['name'], bind = cluster_dropdown, name = 'Region')

input_year = alt.binding_range(min = 1960, max = 2005, step = 5)

yearSelect = alt.selection_point(
    fields = ['year'], bind = input_year, name = 'Year:')

alt.Chart(source).mark_circle(opacity = 1.0).encode(
```

```
    alt.X('fertility:Q', title = 'Fertility'),
    alt.Y('life_expect:Q', title = 'Life Expectancy'),
    color = alt.condition(
        dropSelect, 'name:N', alt.value('lightgray')),
    opacity = alt.condition(
        yearSelect, alt.value(1.0), alt.value(0.0))
).transform_lookup(
    lookup = 'cluster',
    from_ = alt.LookupData(
        data = clusters, key = 'id', fields = ['name']
    )
).add_params(dropSelect, yearSelect)
```

Thus, selecting the year 1990 and the Sub-Saharan Africa region would yield the following plot:



Alternatively, we could achieve a similar effect by applying a filter transformation based on the selected year. The corresponding chart code would be:

```
alt.Chart(source).mark_circle(opacity = 1.0).encode(
    alt.X('fertility:Q', title = 'Fertility'),
```

```
    alt.Y('life_expect:Q', title = 'Life Expectancy'),
    color = alt.condition(
        dropSelect, 'name:N', alt.value('lightgray')),
).transform_lookup(
    lookup = 'cluster',
    from_ = alt.LookupData(
        data = clusters, key = 'id', fields = ['name']
    )
).add_params(dropSelect, yearSelect).transform_filter(yearSelect)
```

However, this filtering method removes part of the data and results in not all years being plotted identically, due to the axes changing after the year 2000. Specifically, the y-axis (Life expectancy) ranges from 0 to 80 prior to the year 2000 and shifts to 0 to 90 thereafter). The x-axis (Fertility) also changes, (the range decreases to 8 children per woman from 9, after year 2005) . This unexpected behavior makes visual comparisons difficult. To avoid it, we can manually set the axis scales, as done in previous examples.

12.3. Other widgets

Other data bindings include radio buttons and checkboxes.

The following example uses **radio buttons** to select which country is displayed. Note that we define a color domain to ensure that the countries are consistently represented by the same color palette.

```
options = ['Europe', 'Japan', 'USA']
labels = [option + ' ' for option in options]

cars = data.cars()

input_dropdown = alt.binding_radio(
    options=options + [None],
    labels=labels + ['All'],
    name='Region: '
)
selection = alt.selection_point(
    fields=['Origin'],
    bind=input_dropdown,
)
```

```
alt.Chart(cars).mark_point().encode(
    alt.X('Horsepower:Q', scale = alt.Scale(domain = (0,240))),
    alt.Y('Miles_per_Gallon:Q',
          title = 'Miles per Gallon',
          scale = alt.Scale(domain = (0,50))),
    color=alt.Color('Origin:N').scale(domain=options),
).add_params(
    selection
).transform_filter(
    selection
)
```

In this example, we fixed the domain of both axis scales to prevent them from changing when the region filter is applied.

We can also control what is plotted on the axes using user input. In the following example, the x-axis is populated with data selected from a dropdown widget.

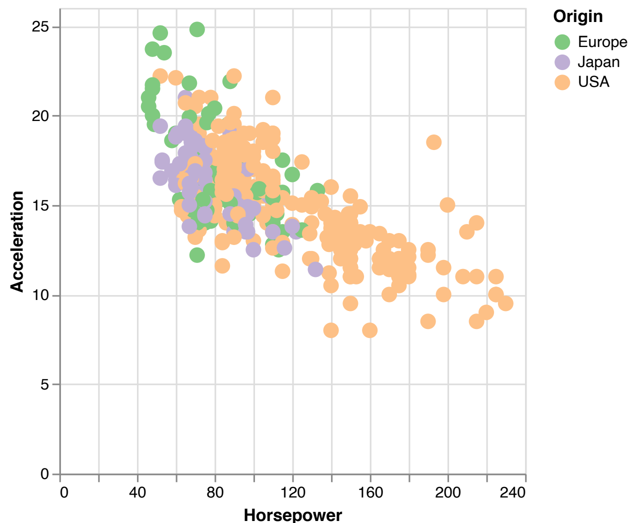
```
dropdown = alt.binding_select(
    options=['Horsepower', 'Displacement',
             'Weight_in_lbs', 'Acceleration'],
    name='X-axis column '
)
xcol_param = alt.param(
    value='Horsepower',
    bind=dropdown
)
alt.Chart(data.cars.url).mark_circle().encode(
    alt.X('x:Q').title(''),
    alt.Y('Miles_per_Gallon:Q', title = 'Miles per Gallon'),
    alt.Color('Origin:N')
).transform_calculate(
    x=f'datum[{xcol_param.name}]'
).add_params(
    xcol_param
)
```

In this code, the fields are explicitly defined in the dropdown, and when the user makes a selection, the selected text is transformed into a new value in the dataset using *transform_calculate*. The new column, labeled as *x* in the code, is then used as the values encoded in the X channel.

The following example demonstrates the use of a **checkbox**, which increases the size of the points when checked:

```
df = data.cars()

check = alt.binding_checkbox( name='Large points')
checkbox_selection = alt.param(bind=check)

alt.Chart(df).mark_circle(filled = True, opacity = 1.0).encode(
    alt.X('Horsepower:Q'),
    alt.Y('Acceleration:Q'),
    color = alt.Color('Origin:N').scale(scheme = 'accent'),
    size = alt.condition(checkbox_selection,
                         alt.value(100), alt.value(40)),
    tooltip = 'Name:N'
).add_params(checkbox_selection)
```

When the code is executed, a checkbox appears at the bottom of the chart when it is displayed, and, when checked, the resulting chart is the following:



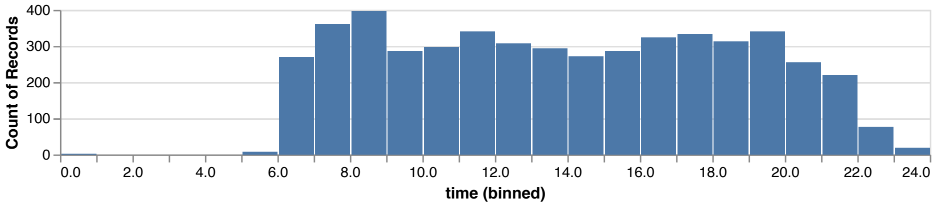
12.4. Responsive charts

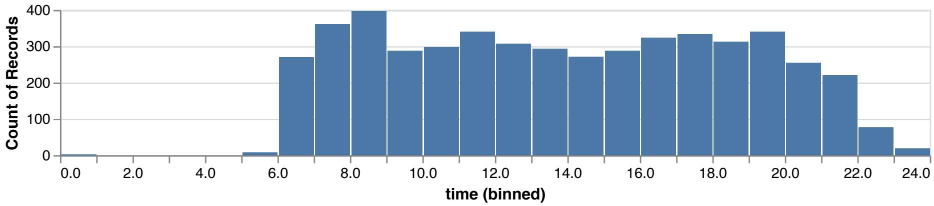
In addition to adjusting a chart's appearance based on user selections, since version 4, charts can also respond to user interactions. We have already discussed setting the *content* value for width and height (though this feature does not work properly in Google Colab, it works as expected in other environments).

Another interactive feature is the ability to modify the behavior of a histogram based on the user's interaction, such as brushing. In the following example, taken from Altair's webpage, the top histogram changes based on a selection made in the bottom histogram:

```
source = data.flights_5k.url

brush = alt.selection_interval(encodings = ['x'])

base = alt.Chart(source).transform_calculate(
    time = 'hours(datum.date) + minutes(datum.date) / 60'
).mark_bar().encode(alt.Y('count():Q')
).properties( width = 600, height = 100)


detail = base.encode(
    alt.X('time:Q', bin = alt.Bin(maxbins = 30, extent = brush),
    scale = alt.Scale(domain = brush)
  )
)

overview = base.encode(alt.X('time:Q',
           bin = alt.Bin(maxbins = 30)),
  ).add_params(brush)

alt.vconcat(detail, overview)
```

The result might look like this:





12.5. Using widgets in creative ways

The combination of selection and hover functionality can be used in various ways. In this example, inspired by the "Multi-line tooltip" in the Altair library, we enhance the visualization by adding text and a rule to the points near the mouse cursor. We plot the maximum and minimum temperatures from the Seattle weather dataset. To achieve this effect, the chart creates hidden tooltips and points that become visible only when the mouse cursor hovers near them. Note the clever use of the *nearest* and hover options.

```
nearest = alt.selection_point(nearest=True, on='mouseover',
                        fields=['monthC'], empty=False)

chMax = alt.Chart(source,
                title = 'Seattle temperature'
                ).mark_line(color = 'crimson').encode(
    alt.X('monthC:T', title = 'Month'),
    alt.Y('mean(temp_max):Q', title = 'Temperature')
).transform_calculate(
    year = 'year(datum.date)',
    monthC = 'month(datum.date)'
).transform_filter(
    alt.datum.year == 2015
)

chMin = alt.Chart(source).mark_line(color = 'steelblue').encode(
```

```
    alt.X('monthC:T', axis =
          alt.Axis(labels = False, title = '', ticks = False)),
    alt.Y('mean(temp_min):Q', title = '')
).transform_calculate(
    year = 'year(datum.date)',
    monthC = 'month(datum.date)'
).transform_filter(
    alt.datum.year == 2015
)
```

The code for drawing the tooltips uses a condition to provide information relevant to the **details-on-demand** aspect of the visualization. This is implemented as three layers: one for the points on the curves; another for the text; and a third for the vertical rules:

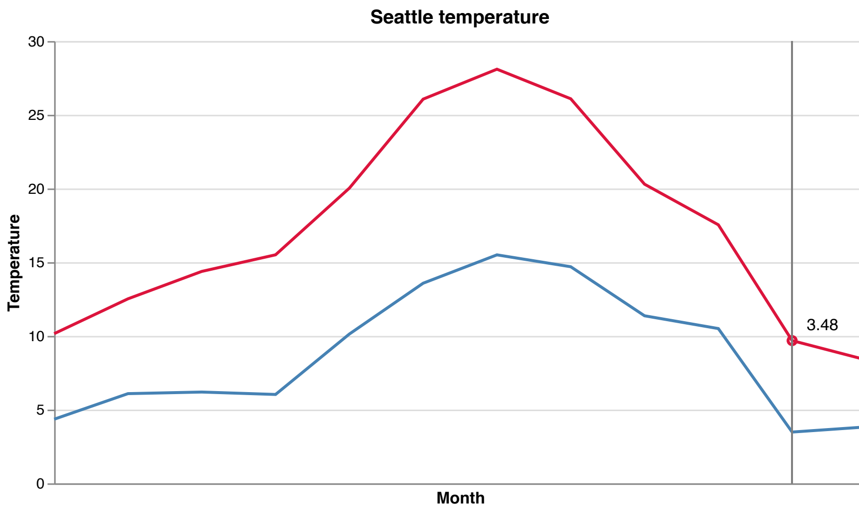
```
selectorMax = chMax.mark_point().encode(
    opacity=alt.value(0),
).add_params(
    nearest
)

pointsMax = chMax.mark_point(color = 'crimson').encode(
    opacity=alt.condition(nearest, alt.value(1), alt.value(0))
)

textMax = chMax.mark_text(align='left', dx=10, dy=-10).encode(
    text=alt.condition(nearest, 'mean(-
temp_min):Q', alt.value(' '))
)

rulesMax = alt.Chart(source).mark_rule(color='gray').encode(
    alt.X('monthC:T', title = 'Month'),
).transform_calculate(
    year = 'year(datum.date)',
    monthC = 'month(datum.date)'
    ).transform_filter(
    nearest
).transform_filter(
    alt.datum.year == 2015
)
```

Note how the selection uses the calculated chart variable monthC, and this value must be calculated for all the necessary charts, including the *rules*. Additionally, the same filters must be applied; in this case, the data shown is restricted to the year 2015.

The final chart composition is achieved using the *layer* function:

```
alt.layer(
    chMax, chMin, selectorMax, pointsMax, rulesMax, textMax
).properties(
    width=550, height=300
)
```

The resulting chart is shown next:

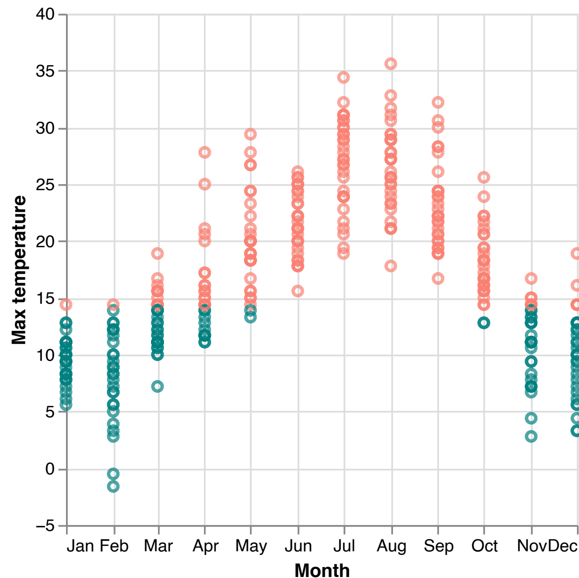


If we want to add text to the other lines, additional work is required. Furthermore, fine-tuning is necessary to ensure that labels do not display too many decimal places or overlap, which can make them difficult to read.

Selections can also be incorporated into expressions. For instance, we can create a slider to define a range of values. In the following example, we take the maximum temperature values from the *seattle_weather* dataset and highlight those that fall below a certain threshold, as defined by a slider:

```
source = data.seattle_weather()

temp_slider = alt.binding_range(
    min = 0, max = 100, step = 1, name = 'Temp Cut: ')
sel_temp = alt.selection_point(
    value = 'Temp', fields = ['tmpCut'],
    bind = temp_slider, init = {'tmpCut': 50})

alt.Chart(source).mark_point().encode(
    alt.X('month(date):T', title = 'Month'),
    alt.Y('temp_max:Q', title = 'Max temperature'),
    color = alt.condition(alt.datum.temp_max < sel_temp.tmpCut,
                          alt.value('teal'), alt.value('salmon'))
).transform_calculate(year = 'year(datum.date)'
).transform_filter(alt.datum.year == 2014).add_params(sel_temp)
```

The result, when the slider is set to 14, appears as follows:



13
Compound charts

Altair provides several ways to display multiple charts, and we have already encountered different approaches throughout the documentation.

The most basic approaches use the operators '+', '|', and '&':

- The '+' operator overlaps two charts.
- The '|' operator arranges two charts side by side.
- The '&' operator paces two charts one on top of the other.

These operators correspond to the following function calls:

- **alt.layer**: This function is equivalent to the '+' operator and accepts any number of charts to overlay. Its parameters are the chart names,.
- **alt.hconcat**: This function allows users to arrange multiple charts horizontally.
- **alt.vconcat**: This function places multiple charts vertically.

For layered charts, the drawing order is from the first to the last, one on top of the other, meaning that subsequent charts may occlude marks from previously drawn charts.

In addition to these basic methods of chart layouts, two specialized approaches for displaying multiple charts are:

- **Repeated charts**: These charts are designed to display multiple charts in a vertical or horizontal layout, where the only difference is a modification of one or more encodings.

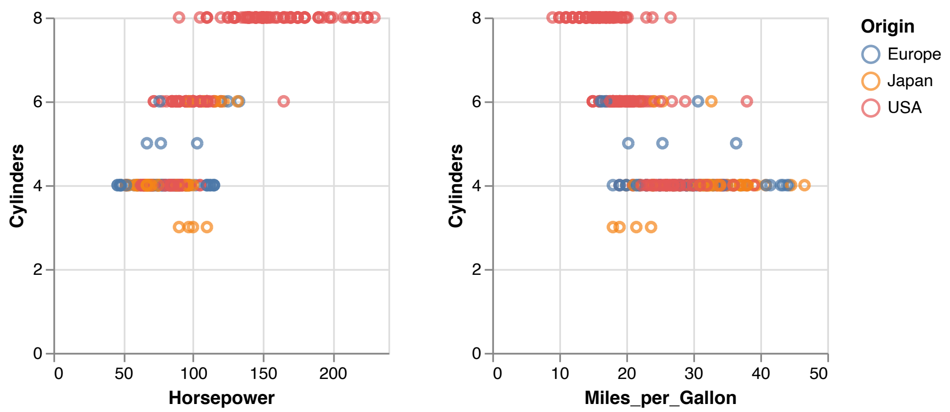
- **Faceted charts**: These charts produce multiple views of a dataset, where each chart represents a subset of the data.

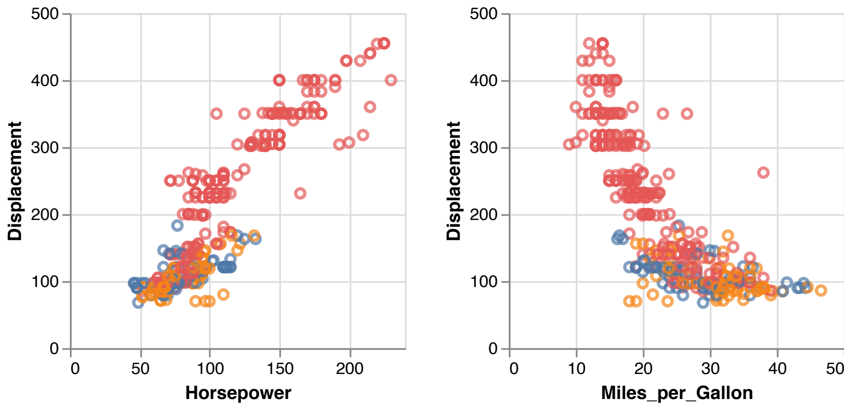
13.1. Repeated charts

The function that provides this feature is *repeat*, which must be defined in two steps. First, specify what information must be repeated (either column, or row, or both) in the encoding part of the chart definition, as well as what type of encoding to use (e.g., quantitative). Second, modify the chart by adding the *repeat* function, which specifies the parameters for each dimension. The following example illustrates this method:

```
cars = data.cars.url

alt.Chart(cars).mark_point().encode(
    alt.X(alt.repeat('column'), type = 'quantitative'),
    alt.Y(alt.repeat('row'), type = 'quantitative'),
    color = 'Origin:N'
).properties(width = 200, height = 200).repeat(
    row = ['Cylinders', 'Displacement'],
    column = ['Horsepower', 'Miles_per_Gallon']
).interactive()
```

The result will be a set of four charts that compare the values of *Cylinders* and *Displacement* with *Horsepower* and *Miles per Gallon* variables, as depicted here:





Although a combination of vertical and horizontal layouts can achieve the same result, the code would be more cumbersome.

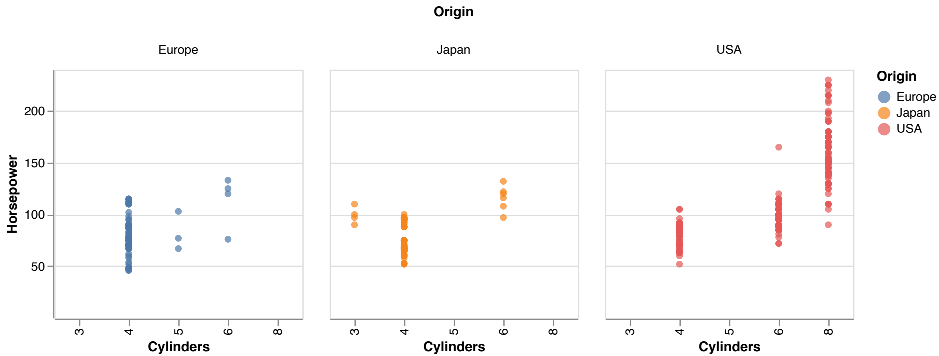
For compositions using multiple layers, aligning them and sharing equivalent scales (when depicting the same data on some axes) is necessary to facilitate reading and comparing values.

13.2. Faceted charts

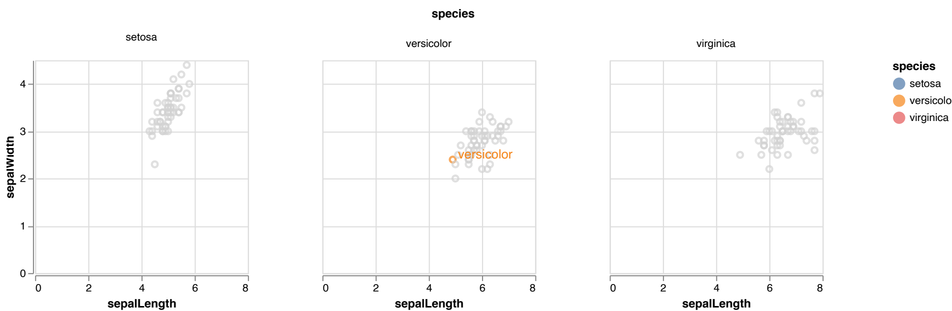
Multiple facets of the same chart can be created using horizontal or vertical concatenation with different filters applied to each chart. However, the *facet* operation simplifies this process when the filtering operation can be expressed easily. For example, we can plot the horsepower of the *cars* dataset against the number of cylinders, faceted by origin:

```
cars = data.cars.url

alt.Chart(cars).mark_circle().encode(
    alt.X('Cylinders:O'),
    alt.Y('Horsepower:Q'),
    color = 'Origin:N'
).properties(width = 200, height = 200).facet(
    column = 'Origin:N'
).interactive()
```

This provides the following output:



In this specific case, the same result could be achieved using the *column* parameter in the chart encoding. However, the *facet* method can build compound layouts of more complex charts, such as those with selection operations, as shown in the following example:



In this case, each chart consists of a combination of two charts: one that displays text upon hovering and another that shows the points. To create these, we define a base chart, which is then used to create one chart with text marks identifying the item and another chart for the points. Both respond to a condition that either de-emphasizes (for points) or renders (or makes transparent) the text:

```
dataset = data.iris()

hover = alt.selection_point(on='mou-
seover', nearest=True, empty='none')
```

```
base = alt.Chart(dataset).mark_point().encode(
    alt.X('sepalLength:Q'),
    alt.Y('sepalWidth:Q'),
    color = alt.condition(
        hover, alt.Color('species:N'), alt.value('lightgray')),
).properties(
    width=200,
    height=200
).add_params(hover)

point = base.mark_point().add_params(hover)

text = base.mark_text(
    align='left', dx=5, dy=-5, fontSize = 12).encode(
    alt.Text('species:N'),
    opacity = alt.condition(hover, alt.value(1), alt.value(0))
)

alt.layer(point, text).facet('species:N')
```

We can also repeat the charts in both dimensions, as in the following example:

```
df = data.cars()

alt.Chart(df).mark_circle().encode(
    alt.X(alt.repeat("column"), type='quantitative'),
    alt.Y(alt.repeat("row"), type='quantitative'),
    alt.Color('Origin:N')
).properties(
    width=150,
    height=150
).repeat(
    row=['Horsepower', 'Acceleration', 'Miles_per_Gallon'],
    column=['Miles_per_Gallon', 'Acceleration', 'Horsepower']
)
```

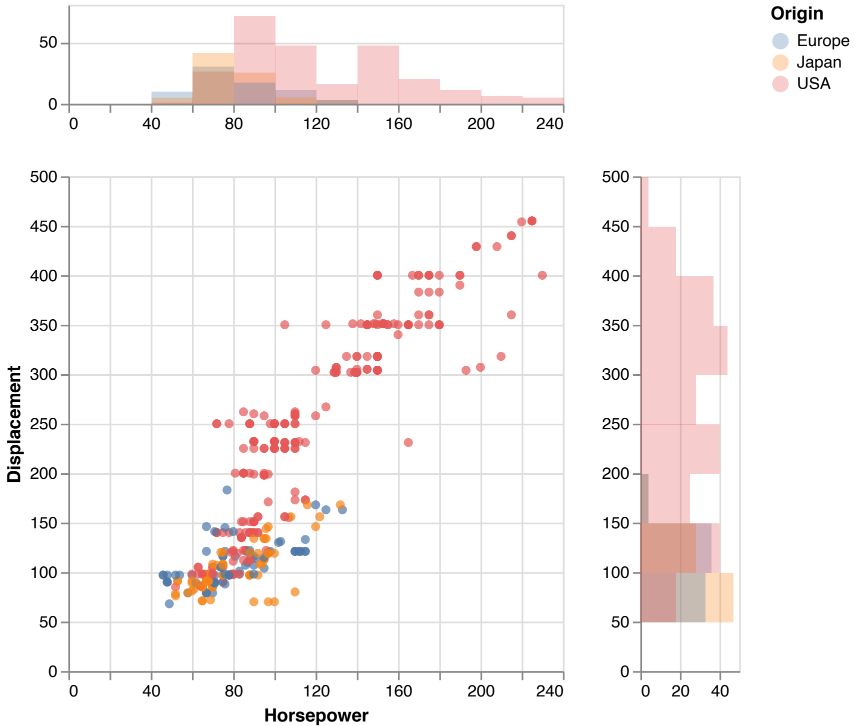
In this case, the result is a 3x3 arrangement of charts with the columns and rows explicitly indicated.

Complex compound charts can also be created in Altair by combining charts of different types. A common example in scientific fields is the scatter plot with marginal histograms. This can be easily achieved in Altair by combining a scatter plot and two bar charts positioned appropriately. The following example implements this type of chart for two parameters of the *cars* dataset:

```
cars = data.cars()

base = alt.Chart(cars)
base_bar = base.mark_bar(opacity=0.3, binSpacing=0)

xscale = alt.Scale(domain=(0, 240))
yscale = alt.Scale(domain=(0, 500))

points = base.mark_circle().encode(
    alt.X('Horsepower').scale(xscale),
    alt.Y('Displacement:Q').scale(yscale),
    alt.Color('Origin'),
)

top_hist = (
    base_bar
    .encode(
        alt.X('Horsepower:Q')
            .bin(maxbins=20,
                 extent=xscale.domain).stack(None).title(''),
        alt.Y('count()').stack(None).title(''),
        alt.Color('Origin:N'),
    )
    .properties(height = 60)
)

right_hist = (
    base_bar
    .encode(
        alt.Y('Displacement:Q')
            .bin(maxbins=20,
                 extent=yscale.domain).stack(None).title(''),
        alt.X('count()').stack(None).title(''),
        alt.Color('Origin:N'),
    ).properties(width = 60)
```

```
)

top_hist & (points | right_hist)
```

The previous code displays the following visualization:



When layering charts, it is sometimes necessary to use different color schemes or axes. By default, Altair will use only one of the schemes and scales declared by the charts. To preserve both, we need to use the *resolve_scale* function with the appropriate parameters to maintain the color schemes (*color = 'independent'*) or axes scales (e.g., *x = 'independent'*). We have already seen examples of both in earlier sections.

14 Advanced maps

We previously looked at basic static map visualizations. However, for some datasets, we may need to interact differently with choropleth maps. For example, imagine plotting a value on maps that can change over the years, utilizing a slider to select the year. This can be challenging to achieve using *alt.Chart().mark_geoshape* with geometry as input, since the data used to color the map is obtained through a *look_up*, which provides only one value per country.

One solution is to treat the plot as one that renders the indicator for the selected year while gathering the geometry of the countries.

A complex example of this can be found at the following website: https://www.kaggle.com/labdmitriy/kaggle-survey-2019-map-mini-dashboard-altair/notebook

A simpler version that renders the data of the *gapminder* dataset for selected years can be designed as follows.

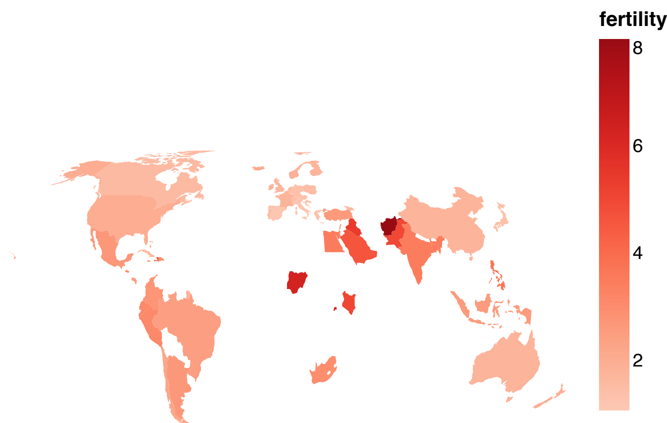
First, we create the imports and load the files that match country names with country codes. Then, we read and merge the data and country code files:

```
geom = alt.topo_feature(data.world_110m.url, 'countries')

corresp = pd.read_json('world_110m_country_codes.json')
df = data.gapminder()

merged = pd.merge(df, corresp, how = 'left',
                  left_on = 'country', right_on = 'name')

merged['id'] = merged['id'].fillna(-1)
merged['id'] = merged['id'].astype(int)
```

Next, we create the visualization using the *merged* file as input and looking up the geometry in the *geom* file.

```
input_slider = alt.binding_range(max = 2000, min = 1980, step = 5)

selection = alt.selection_point(
    fields = ['year'], bind = input_slider, name = 'Year ')

alt.Chart(merged).transform_filter(selection).transform_lookup(
    lookup = 'id',
    from_ = alt.LookupData(geom, 'id'),
    as_ = 'geom',
    default = 'Other'
).transform_calculate(
    geometry = 'datum.geom.geometry',
    type = 'datum.geom.type'
).mark_geoshape(
).encode(
    alt.Color('fertility:Q', scale = alt.Scale(scheme = 'reds')),
).add_params(selection)
```

The result appears as follows (note that the interactive visualization will include a slider at the bottom for selecting the year):



As in previous examples, some countries appear white, for two possible reasons: 1) The *gapminder* dataset lacks data for those countries; or 2) the *gapminder* file

and the file that maps codes to country names may use different names for the same country. It is crucial to ensure that the data are processed correctly before creating such visualizations.

In the following example, we create a graduated symbol map with obesity data for the US. The dataset appears as follows:

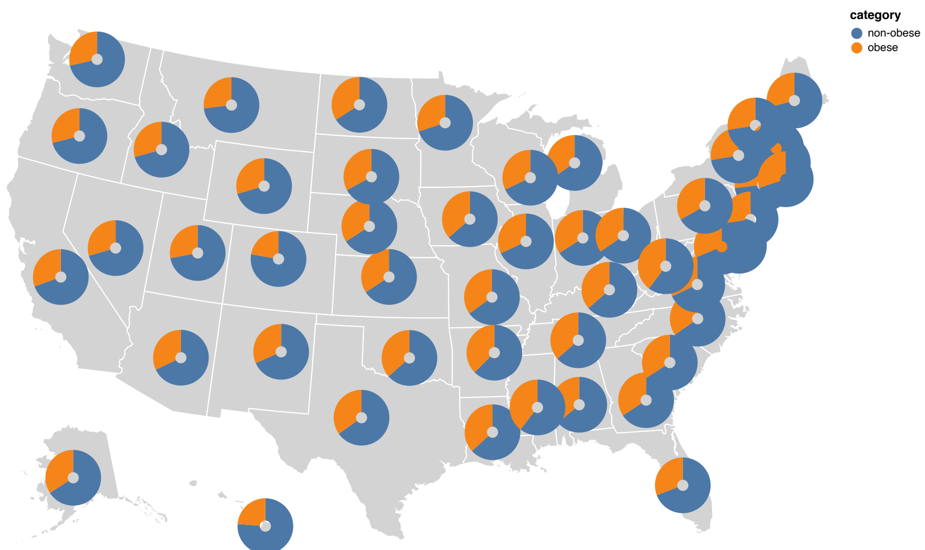
	state	id	latitude	longitude	percentage	category
0	Alabama	1	32.318231	-86.902298	63.8	non-obese
1	Alabama	1	32.318231	-86.902298	36.2	obese
3	Alaska	2	63.588753	-154.493062	65.8	non-obese
4	Alaska	2	63.588753	-154.493062	34.2	obese

We then create a pie chart for each state and layer them on top of the map. The code is:

```
states = alt.topo_feature(data.us_10m.url, 'states')

base = alt.Chart(states).mark_geoshape(
    fill='lightgray',
    stroke='white'
).properties(
    width=800,
    height=500
).project('albersUsa')

pies = []
for i in new_df['state'].unique():
    pie = alt.Chart(new_df[new_df['state'] == i]).mark_arc(
        innerRadius=5,
        outerRadius=25
    ).encode(
        theta=alt.Theta(field='percentage', type='quantitative'),
        color=alt.Color(field='category', type='nominal'),
        tooltip=['state', 'percentage']
    ).encode(
        longitude='longitude:Q',
        latitude='latitude:Q',
    )
    pies.append(pie)
```

```
charts = alt.layer(base, *pies)
charts.show()
```

The result is as follows:



15 Interactive visualization of very large datasets

Altair and Vega face two challenges regarding large datasets: 1) limitations on the number of rows, and 2) performance issues. The fixed row limitation can be removed using *alt.data_transformers.disable_max_rows()*, as explained previously. However, large datasets may still cause display problems, occasionally resulting in disconnections in the Colab runtime and thus leading to incomplete data displays.

One solution is to generate the chart in *HTML* instead of rendering it directly. Opening the generated file typically resolves display issues. However, we can also work interactively with large datasets in Colab using Vegafusion. Vegafusion (https://vegafusion.io/) is an external project aimed at providing server-side scaling for the Vega library.

Although some examples are provided on the website, not all function correctly in Colab. The following approach does.

First, install *vegafusion['embed']* and *vegafusion_jupyter*.

```
!pip install vegafusion['embed']
!pip install vegafusion_jupyter
```

To use the Vegafusion widget necessary for handling data, inform Colab of its usage with the following code:

```
from google.colab import output
output.enable_custom_widget_manager()
```

Note that this code must be enabled for each session.

Next, enable the Vegafusion widget with the following code:

```
import pandas as pd
import altair as alt
import vegafusion as vf

vf.enable_widget()
```

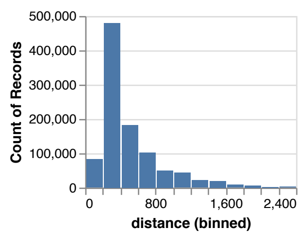
This code works in the background to perform numerous data transformations required by Altair, using efficient Rust implementations.

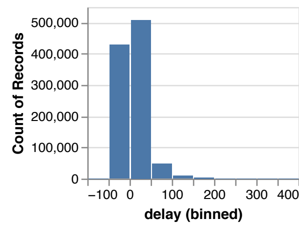
The following example code (from the Vegafusion website) illustrates 1 million flights in an interactive visualization:

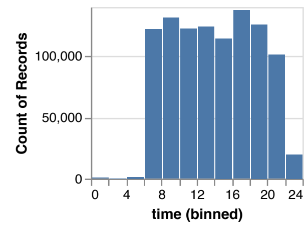
```
flights = pd.read_parquet(
    "https://vegafusion-datasets.s3.amazo-
naws.com/vega/flights_1m.parquet"
)

brush = alt.selection(type='interval', encodings=['x'])

# Define the base chart, with the common parts of the
# background and highlights
base = alt.Chart().mark_bar().encode(
    x=alt.X(alt.repeat('column'), type='quan-
titative', bin=alt.Bin(maxbins=20)),
    y='count()'
).properties(
    width=160,
    height=130
)

# gray background with selection
background = base.encode(
    color=alt.value('#ddd')
).add_selection(brush)

# blue highlights on the selected data
highlight = base.transform_filter(brush)
```

```
# layer the two charts & repeat
chart = alt.layer(
    background,
    highlight,
    data=flights
).transform_calculate(
    "time",
    "hours(datum.date)"
).repeat(column=["distance", "delay", "time"])
chart
```

The result is as follows:







16
Moving forward

The goal of this book is to provide a comprehensive introduction to the capabilities of Altair, a versatile and intuitive library for data visualization in Python. We began by describing how to input data and progressed through the creation of basic charts and complex interactive visualizations. Altair's declarative syntax and seamless integration with Pandas make visualization accessible to various users.

Altair's strength lies in its ability to simplify the creation of high-quality visualizations with minimal code. By focusing on the visual properties of data rather than the mechanics of visualization, Altair enables users to convey insights effectively and efficiently. We also introduced methods for handling large-scale datasets interactively.

Altair's website features numerous examples of different charts that were not covered in this book, and it is continuously updated with new versions.

Moreover, the use of Altair can be complemented by other dashboard creation tools, which may facilitate development of web applications. Various tools are available for this purpose, including Panel, Dash, and Streamlit, to name a few. Among these, Streamlit (https://streamlit.io/) is particularly straightforward, enabling the development of web applications through the integration of multiple components. Streamlit provides components for Altair charts, as well as for other charting technologies such as Plotly, ECharts, and several components for specific visualizations, such as trees or graphs.