Declarative Visualization with Vega-Altair 5

COURSE NOTES

Pere-Pau Vázquez | Information Visualization – GECD | 2023 – 2024

Contents

1. Introduction

Altair is a declarative visualization library for Python. In its current version it only supports Python 3.6 and above, due to the deprecation of previous versions of Python. Altair had its initial release in July 2016.

From the architectural point of view, Altair generates Vega code, which is then executed in a JavaScript environment. Vega (and its lighter version: Vega-Lite) is a declarative language for interactive graphics developed by researchers of the University of Washington. Vega Lite specification can be checked through https://vega.github.io/vega-lite/.

There are other alternative libraries for visualization in Python, but most of them have important limitations such as:

- Too many programming required.
- Not enough support for interaction.
- Steep learning curve.
- Not enough power.

As a result, the developers of Altair focused on creating something that was simple, and powerful at the same time. If the goal is having something with the maximum flexibility, one should turn to something such as D3 over JavaScript.

1.1 ALTAIR VERSIONS

Altair officially upgraded to version 5 in 2023, and the current version is 5.1.1 in August 2023, and it is the default version you have in Google Colab. The current documentation can be accessed through the webpage: https://altair-viz.github.io/.

Note that, recently, the name of the library has changed to Vega-Altair due to a complaint by a company also named Altair.

1.2 DECLARATIVE LANGUAGE

From the point of view of programming, there are two main programming paradigms: imperative and declarative.

The **imperative** paradigm requires the user to specify exactly **how** the tasks accomplish the needs. The user must provide all the steps that accomplish the result.

The **declarative** paradigm focuses on specifying **what** needs to be done, and the way it is realized remains as internal details of the system.

Using a declarative language often has advantages in terms of usability since the amount of code required to write is less. But it has other advantages too, such as that the solutions are easier to specify in terms of *what* needs to be done instead of *how*.

1.3 ALTERNATIVES

There are other alternatives to get the same work done in Python. For instance, Seaborn is an easy-to-use library that is programmed using the imperative paradigm, while Bokeh is a declarative library that can also be used. However, both of them are much less powerful than other alternatives such as *Matplolib*, which, again is imperative, or *Plotly*.

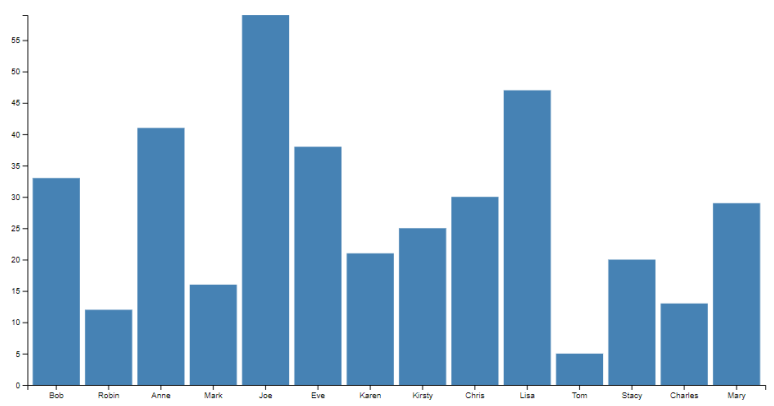
As already mentioned, there are other alternatives outside Python, such as D3, which is an extremely powerful library, over JavaScript. However, working with D3 to design visualizations from scratch, require a much larger number of lines than for doing the same work using Altair.

For example, to create a simple bar chart in D3, we would require a several dozens of lines…

```
.bar { fill: steelblue; }

</style>
<body>

<!-- load the d3.js library -->
<script src="//d3js.org/d3.v4.min.js"></script>
<script>

// set the dimensions and margins of the graph
var margin = {top: 20, right: 20, bottom: 30, left: 40},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

// set the ranges
var x = d3.scaleBand()
          .range([0, width])
          .padding(0.1);
var y = d3.scaleLinear()
          .range([height, 0]);

// append the svg object to the body of the page
// append a 'group' element to 'svg'
// moves the 'group' element to the top left margin
var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

// get the data
d3.csv("sales.csv", function(error, data) {
  if (error) throw error;

  // format the data
  data.forEach(function(d) {
    d.sales = +d.sales;
  });

  // Scale the range of the data in the domains
  x.domain(data.map(function(d) { return d.salesperson; }));
  y.domain([0, d3.max(data, function(d) { return d.sales; })]);
```

```
  // append the rectangles for the bar chart
  svg.selectAll(".bar")
      .data(data)
    .enter().append("rect")
      .attr("class", "bar")
      .attr("x", function(d) { return x(d.salesperson); })
      .attr("width", x.bandwidth())
      .attr("y", function(d) { return y(d.sales); })
      .attr("height", function(d) { return height - y(d.sales); });

  // add the x Axis
  svg.append("g")
      .attr("transform", "translate(0," + height + ")")
      .call(d3.axisBottom(x));

  // add the y Axis
  svg.append("g")
      .call(d3.axisLeft(y));

});
```

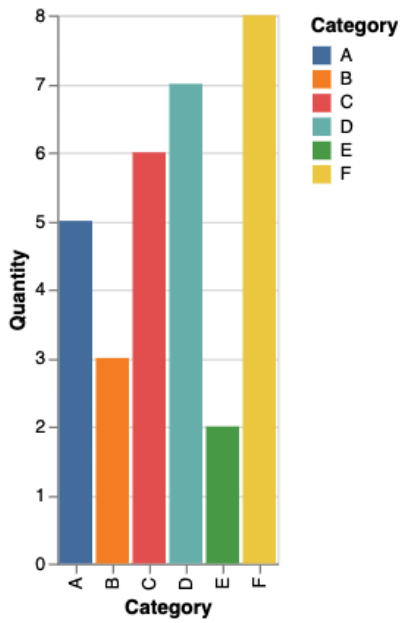
The code above generates the following chart:



On the other hand, building a simple bar chart with Altair requires only less than a dozen lines…

```
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]})
alt.Chart(data).mark_bar().encode(
    x='Category:N',
    y='Quantity:Q',
    color='Category:N'
)
```

The previous code generates the following chart:



2. Altair library

Altair is a library that works as a wrapper of Vega-Lite for Python. As such, it has a series of data structures and methods that make this translation to Vega-Lite syntax transparent to the users.

The main process to perform when using altair consists on loading or generating the data, and creating a chart or a series of charts that display the information.

Typically, developers will iterate in the design process by first creating one or more simple charts, then combining them in multiple views, and finally, designing the interaction methods.

2.1 BASIC CHART DESIGN

The simplest way to create a chart in altair is by calling the *Chart* function, which receives as parameter the source data. It has two important methods that we can concatenate: *mark_**, and *encode*. The first one has several flavors to define the type of mark we are going to use (e.g., *mark_bar* is used when we want a bar chart). The second, lets us specify the aspect of the marks in the chart as well as how they are laid out.

This is shown in the following Figure.

To create different types of charts, we will change the marks how they are laid out. The *encode* function, besides determining what goes in which axis, also lets us define the visual configuration of marks (e.g., colors or palettes, size, etc.).

Altair lets us create more complex charts, as well as multiple charts that are linked. This will be explained later in this tutorial.

Now that we have an idea of what a visualization in altair looks like, we dive deeper. We will start with the specification of data: how data is read in altair, as well as some modifications we can do to better fit our needs.

2.2 VISUALIZATION PIPELINE

The common process you have to follow when designing visualization applications typically 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 is related to the problem(s) to solve. The second and third are not addressed with altair, but other tools such as Open Refine or Data Wrangler may be very helpful. Some libraries of Python may also be used to clean the data.

Vega-Altair will let you deal with steps 4 to 6, although up to different extents: We are not going to be able to design the marks with full flexibility, but a quite wide range of possibilities are allowed with the different options provided for channel and marks customization. The interaction will also be limited to a certain number and types of interactions, but they will be enough for a large set of problems.

3. Data specification

The data is specified to each top-level chart object using a dataset encoded in one of three ways:

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

Pandas Dataframe are two-dimensional size-mutable tabular data structure with labeled axes (rows and columns). See Pandas' documentation for more details on creating those datasets if required (https://pandas.pydata.org). For example, we can create a simple dataset using the following code:

```
import altair as alt
import pandas as pd

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

Data is a class that can be used to specify data using JSON-style records. To create the same dataset using Data, we should 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 main difference between the two encodings is the ability of Altair for extracting the data type from the DataFrame. For Data, we will need to specify the types of the different elements (though we can override the detected type from the DataFrame using the same procedure to specify them in the Chart construction.

We can also input data form an url provided that the data is stored in a JSON file. The following code will do the work:

```
import altair as alt
from vega_datasets import data
url = data.cars.url
```

Like in the previous case, we will need to specify the data types because those cannot be extracted from the file pointed by the URL string.

There are two common conventions 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 within the table as *values*

An example of wide-form is provided next:

```
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]})
print(wide_form)
```

```
         Date    AAPL   AMZN    GOOG
0  2007-10-01  189.95  89.15  707.00
1  2007-11-01  182.22  90.56  693.00
2  2007-12-01  198.08  92.64  691.48
```

On the contrary, 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',
                                   '2007-10-01', '2007-11-01', '2007-12-01'],
                          'company': ['AAPL', 'AAPL', 'AAPL',
                                      'AMZN', 'AMZN', 'AMZN',
                                      'GOOG', 'GOOG', 'GOOG'],
                          'price': [189.95, 182.22, 198.08,
                                    89.15,  90.56,  92.64,
                                    707.00, 693.00, 691.48]})
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
    6  2007-10-01    GOOG  707.00
    7  2007-11-01    GOOG  693.00
    8  2007-12-01    GOOG  691.48
```

In the documentation, the authors state that Altair works better with the long form version. We can specify that the long-form is used in the creation call, as depicted next.

```
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 method *transform_fold*.

```
alt.Chart(wide_form).transform_fold(
    ['AAPL', 'AMZN', 'GOOG'],
    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 require to generate several charts (e.g. one for each company in this example) and use layering to plot them together, such as in the following example.

```
import altair as alt
import pandas as pd


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'
)

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

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

ch1 + ch2 + ch3
```

We will talk on compound charts later in these course notes.

Finally, Altair supports 4 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) by verbosely (e.g. "temporal") or shorthand (e.g. "T") in the *encode* method of the chart.

An example with explicit encoding:

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

Explicit encoding:

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

EXERCISES

Exercise 1. Check the contents of the Vega dataset co2_concentration.json . Plot with the x variable 'Date' as quantitative and y with variable 'CO2' as quantitative. What happens? Now try to use the date as a temporal variable? What happens if you use the date as a categorical variable?

Exercise 2. Create a scatterplot using the dataset gapminder.json with the X axis as the population, and the Y axis as the life expectancy. Now use the X axis as the life expectancy, and the Y axis as the fertility rate.

Exercise 3. Compare the previous chart with another one where the marks are depicted as circles. Make another experiment with the charts as squares.

Exercise 4. With the previous example, now encode in the circle mark the population of the country. Compare the sizes using circles, squares, and points.

Exercise 5. The last plot shows all the data in a single chart, but this includes every year. In order to make sense of them, color code the years.

4. Marks

In order to create a visualization, we must transform the data into visual representations. These representations have two different kinds of parameters, the geometric element we use, and its visual properties. The first is called **mark**, while the second is called **channel** or **visual variables**.

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

4.1 BASIC MARKS

There are eight basic mark types:

- Arc: Used to encode pie charts and donut charts

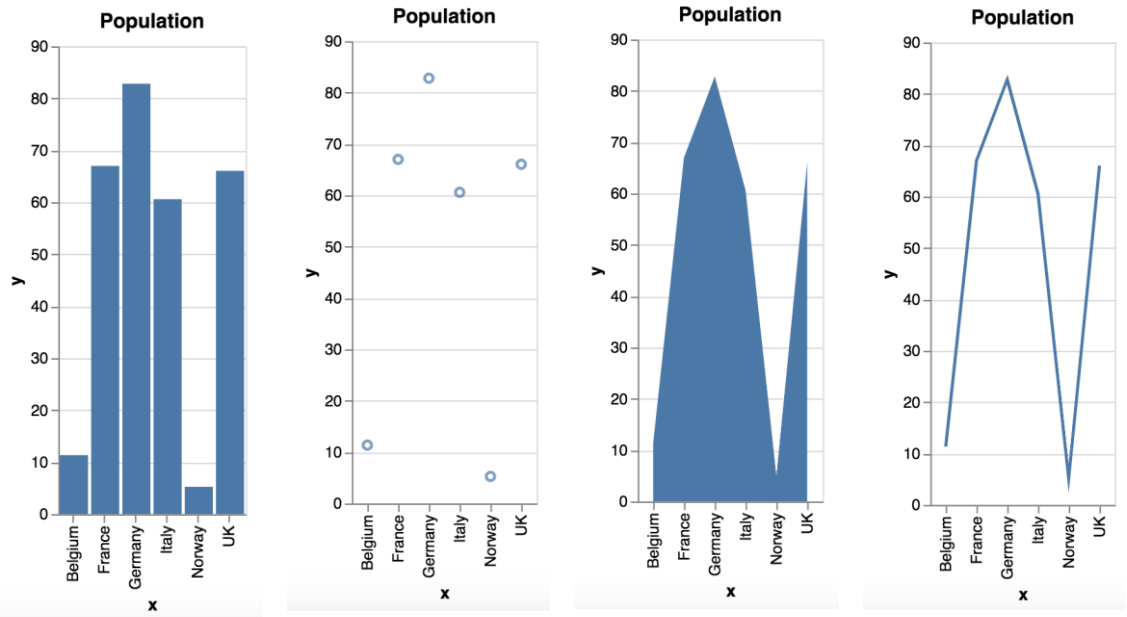
- Area: Used to plot filled area charts
- Bar: All sorts of bar charts and histograms
- Circle: Needed for scatterplots
- Geoshape: Element used to encode spatial data
- Image: Used in scatterplot, to use images instead of points
- Line: For line charts
- Point: It is used for scatterplots, but has the option of change its shape
- Rect: A filled rectangle, usually used to draw heatmaps
- Rule: A vertical or horizontal line spanning the axis
- Square: It is basically a scatterplot with square marks
- Text: Used to show 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 more complex elements are also available (see next section).

The marks are defined through the use of the *mark_** method, where the symbol "*" must be replaced with the mark name. Thus, for example, to design a bar chart, we will use the method *mark_bar()* of the *Chart* type. The following example builds a bar chart showing the population of a set of countries:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar().encode(
    x='x',
    y='y'
).properties(title = 'Population')
```

By changing the mark to point, area, or line, you will have these four variants of the same chart:

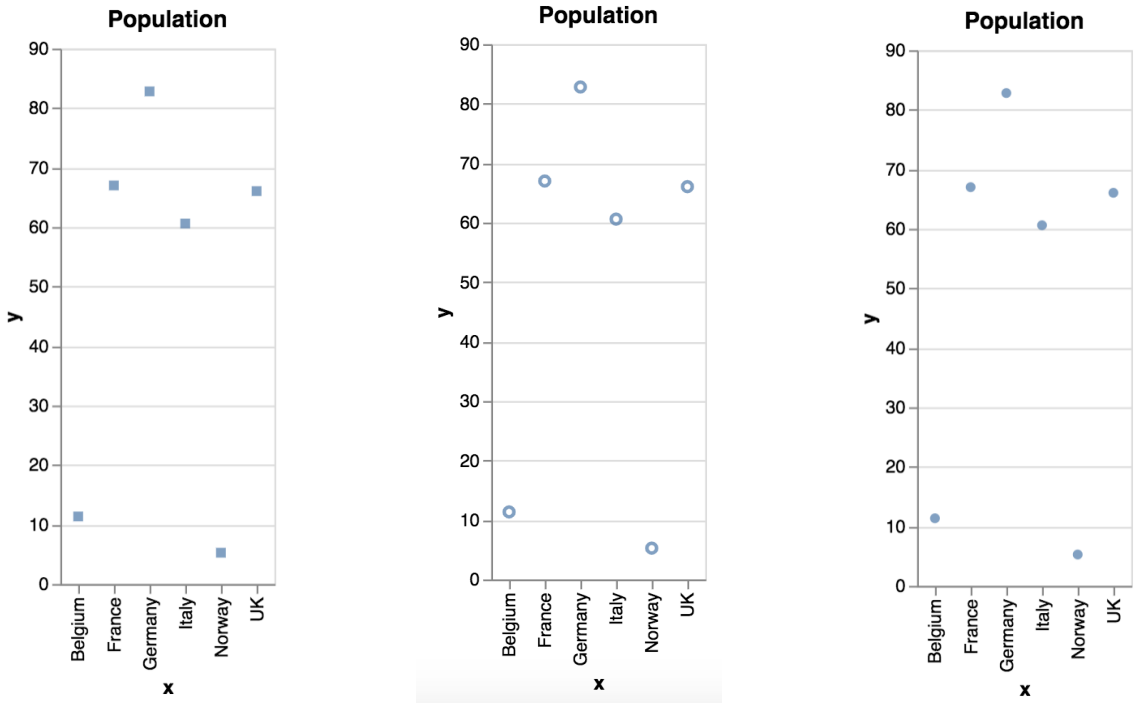


Note that the fact that Altair lets you encode the data in a certain way does not imply that the result is adequate. From the previous examples, only the two leftmost charts are correct, with the first one being better than the second. The rightmost charts may induce the user to read the data as having some continuity from one country to the other, and this is wrong. Therefore, we must carefully design the charts so that they are expressive.

Moreover, the fact that Altair uses a certain name for a mark does not imply that formally, the chart uses that mark. For example, line charts use points as marks, and lines are connecting the points. This can be intuitively understood if we take into account that in a line chart, the datapoints (i. e., the values of the datapoints) do not map directly to line lengths, but to point positions. The lengths of lines depend on the values of two consecutive datapoints. The same applies to *mark_area*, for example.

Points, squares and circles can also be used to generate very similar scatterplots. Actually, when we talk about shapes, we will see that the different marks collide partially with the visual channels.

The previous dataset visualized with squares, points, and circles, for comparison, is shown in the following figure.

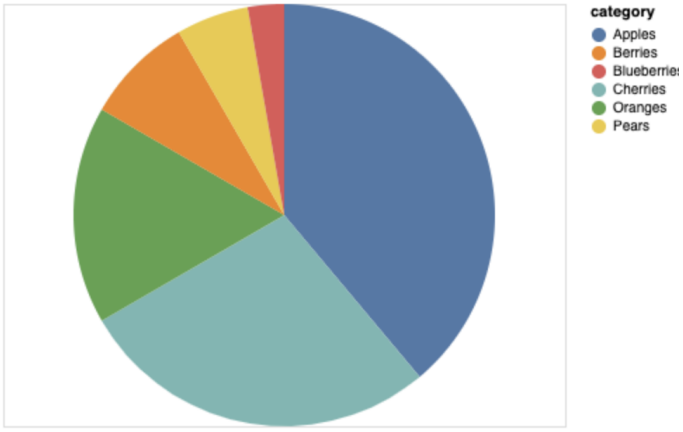


Another, very popular visualization technique is the pie chart. Pie charts were added to altair very recently, in version 4.2. Though they are highly controversial and many visualization practitioners will not use them, they appear in many infographics.

The most basic version that altair supports is the regular pie chart:

```
import pandas as pd
import altair as alt

source = pd.DataFrame({"category": ['Apples', 'Cherries', 'Oranges',
                                    'Berries', 'Pears', 'Blueberries'],
                       "value": [14, 10, 6, 3, 2, 1]})

alt.Chart(source).mark_arc().encode(
    theta=alt.Theta(field="value", type="quantitative"),
    order = alt.Order('value', sort = 'descending'),
    color=alt.Color(field="category", type="nominal"),
)
```

The result with this code would be this one:



But pie charts come in different flavors, such as donut charts, where the chart has a whole in the center. To create a donut chart, we only have to define the size of the hole, as an *innerRadius* parameter that can be passed to the arc mark as a configuration value, such as in the following example:

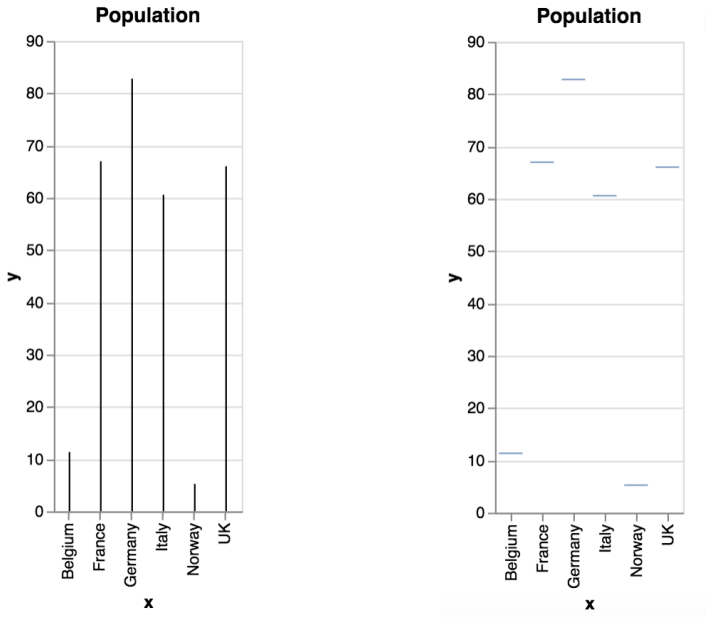
```
alt.Chart(source).mark_arc(innerRadius = 80).encode(
    theta=alt.Theta(field="value", type="quantitative"),
    color=alt.Color(field="category", type="nominal"),
)
```

Which would result in the following chart:

The arc mark also allows for more complex charts such as the radial chart, that will be visited later.

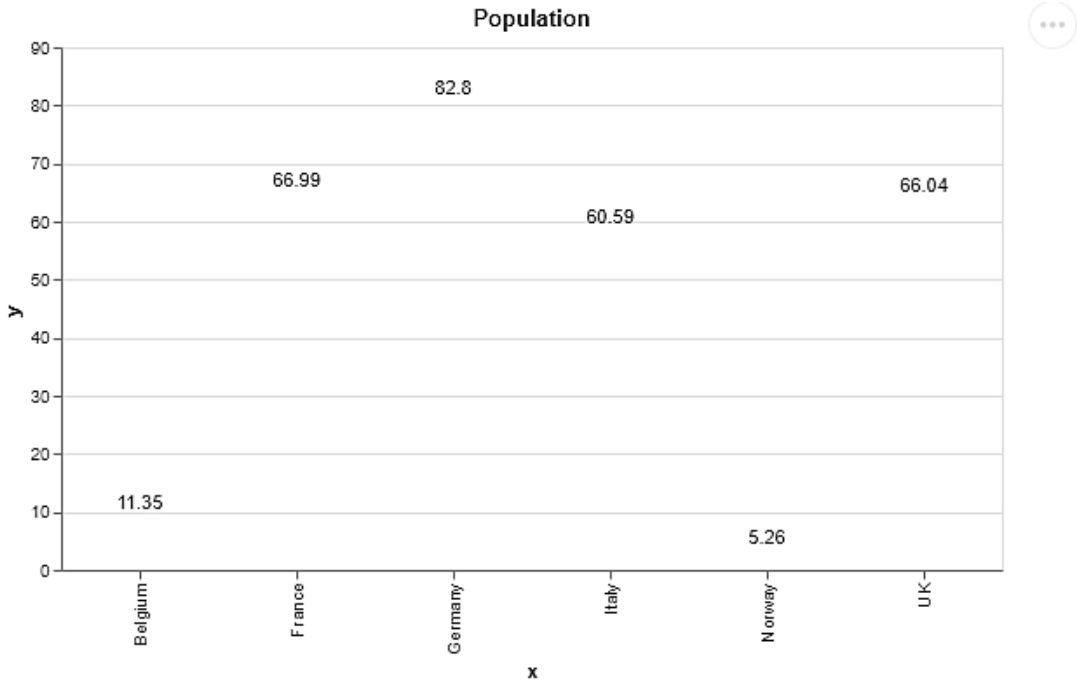
There are two other basic types that may result not so familiar: rule and tick. The first one is typically used to create markers (e.g., reference lines for a certain

value), although can be used to create charts on their own, as shown next. Tick is another element that is often used to highlight reference values. Again, we can compare them simply by showing how they plot the same data in a chart.



Text mark is highly useful to illustrate values, either directly in a chart, or as a tooltip. We can use the text mark in the previous example to illustrate the population values:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_text().encode(
    x='x',
    y='y',
    text = 'y'
).properties(title = 'Population')
```



We will deal with the other objects in more advanced sessions.

4.2 COMPOSITE MARKS

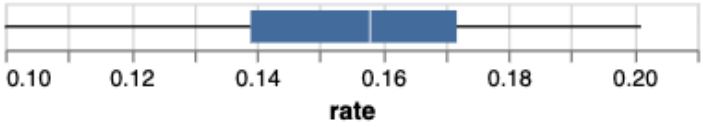
Besides the basic marks, there are other specific types:

- 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.

```
import altair as alt
from vega_datasets import data

source = data.obesity()

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

The result would be as follows:

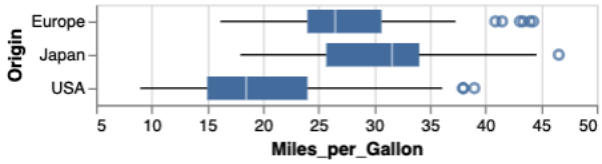


The default version of boxplots follows Tukey's approach, that has whiskers that span from the smallest data to the largest data within the range [Q1 – k * IQR, Q3 + k * IQR] where Q1 and Q3 are the first and third quartiles and IQR is the interquartile range (Q3-Q1). The constant can be explicitly specified 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:

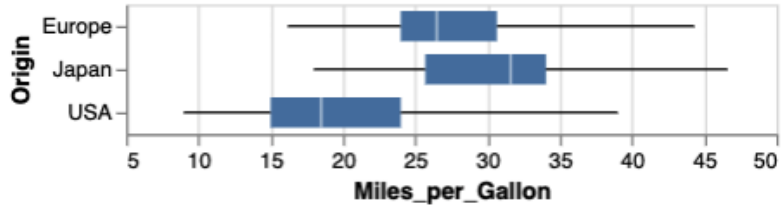
```
source = data.cars()

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

As shown in this result:



If we modify the extent to the min-max range, the whiskers will take over all the range, as shown here:

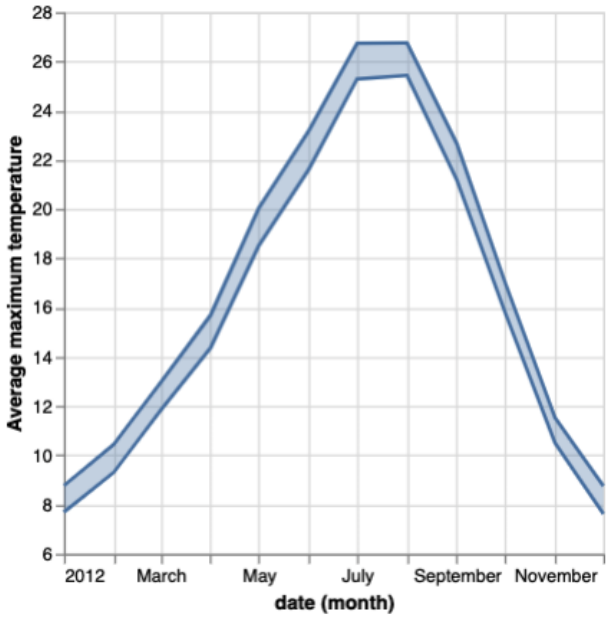
```
source = data.cars()

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



An example with error bands is shown here, where the average of maximum temperatures reached per month in Seattle are displayed as a range:

```
source = data.seattle_weather()

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

The result is shown as:



5. Channels

The visual properties of marks are called channels. Altair has several ways to modify channels, including position, shape, or color, to name a few. To define and modify their default values, we either change their visual properties by using the different *markRef* values, or set the position by defining some fields such as the x and *y* locations.

5.1 CHANNEL ENCODING

There are several aspects of the marks that can be defined/modified. First, the position, that can be defined using the following channels:

- x: X coordinates of the marks, or width of horizontal bars (and area marks).
- y: Y coordinates of the marks, or 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: the GPS longitude parameter for geographical charts
- latitude: the latitude value for geographical charts
- longitude2: second longitude parameter for ranges in geographical charts
- latitude2: second latitude value for ranges in geographical charts
- xError: the x-axis error value
- yError: the y-axis error value
- xError2: the second x-axis error value
- yError2: the second y-axis error value
- xOffset: offset to the x position
- yOffset: offset to the y position
- theta: the start arc angle
- theta2: the end arc angle (given in radians), used for non-complete pie charts (e.g., Pacman-like)

To modify the aspect of the channels, we have the following modifiers:

- angle: to define the angle of the mark, as in pie charts
- color: default color of the mark
- fill: color that fills the mark (has higher precedence than color)
- fillOpacity: float indicating the opacity [0..1]
- opacity: opacity of the mark
- radius: the radius of the mark (for radial charts)
- shape: for point marks, shape can be:
    - circle, square, cross, diamond, triangle up, triangle down, triangle right, or triangle left (other shapes include arrow or wedge)
- size: the size of the shape. For point, circle and square, it will be the pixel area of the marks.
- stroke: the stroke of the mark
- strokeDash: stroke style of the mark (for line charts, basically)
- strokeOpacity: the opacity of the line
- strokeWidth: width of the line

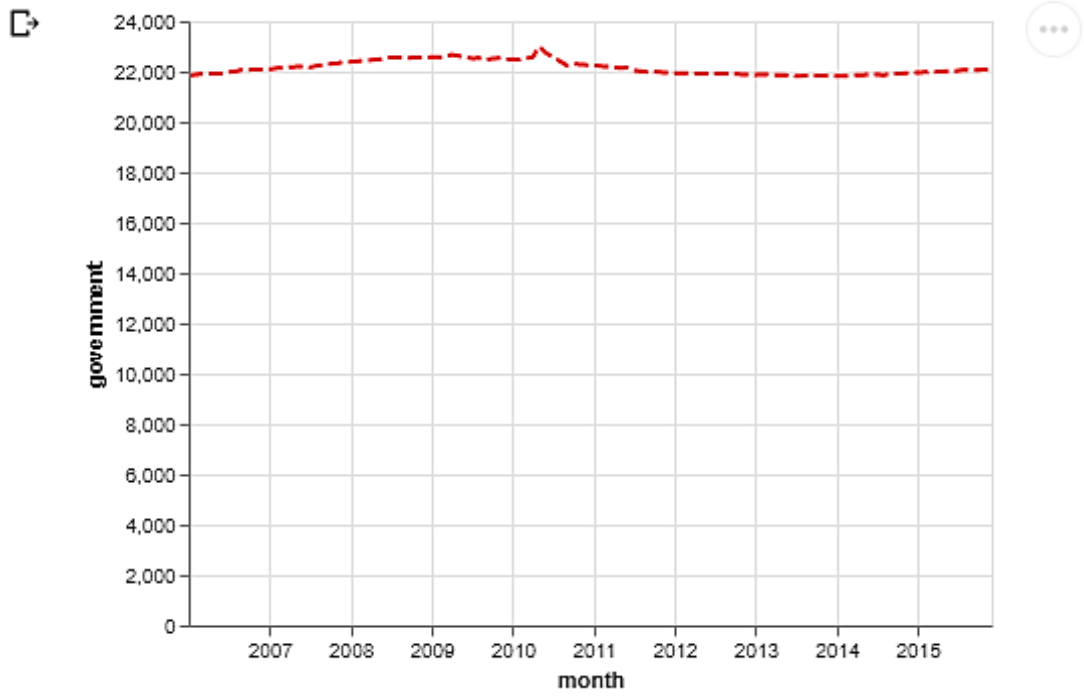
Other specific encodings apply for text and tooltips:

- text: text to use for the mark
- tooltip: tooltip value (does not necessarily show a single field, can be a set of fields)

There are many other modifiers that can help configuring the chart, such as ("detail", that can be used to group elements, "order" to sort the elements of a channel, etc.).

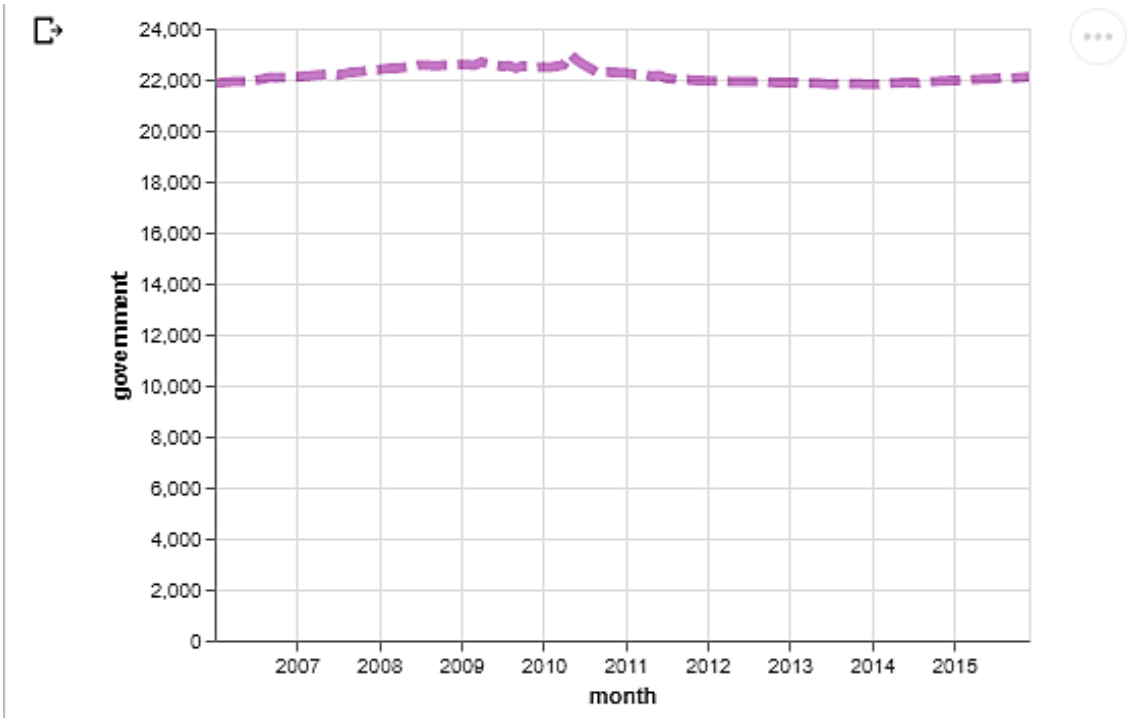
In the following example we use the color and the strokeDash to modify the plot:

```
df = data.us_employment.url

alt.Chart(df).mark_line(strokeDash=[7,3], color = 'red').encode(
    x='month:T',
    y='government:Q'
)
```



As stated, the *stroke* property has more precedence than the *color* property. In the following chart, we use both, besides changing the stroke width.

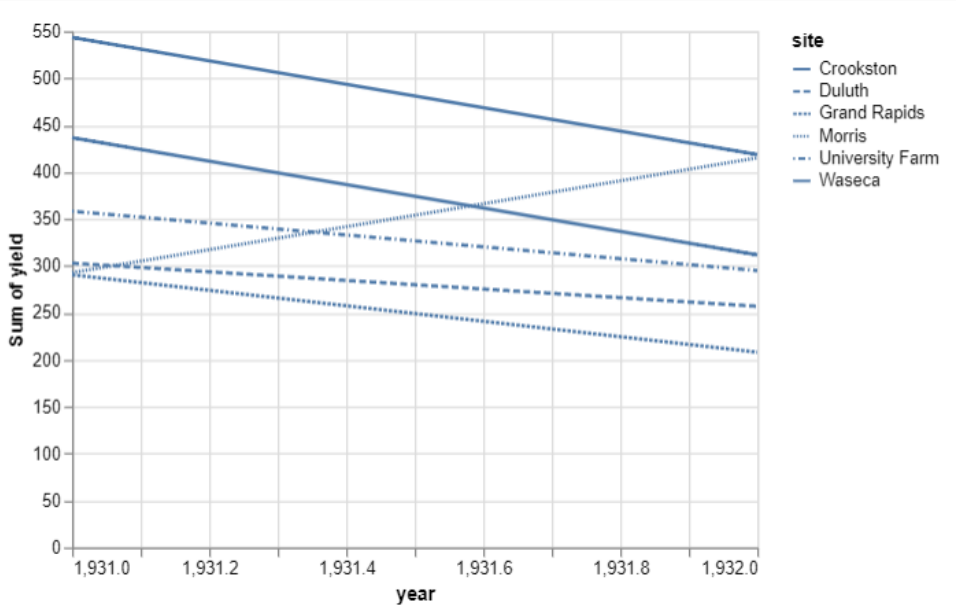
```
df = data.us_employment.url

alt.Chart(df).mark_line(
    strokeDash=[15,5], color = 'red', stroke = 'purple',
    strokeWidth = 5, strokeOpacity = 0.5
    ).encode(
    x='month:T',
    y='government:Q'
)
```



In the following example, we use the parameter to apply different styles to the different sources of barley in the chart.

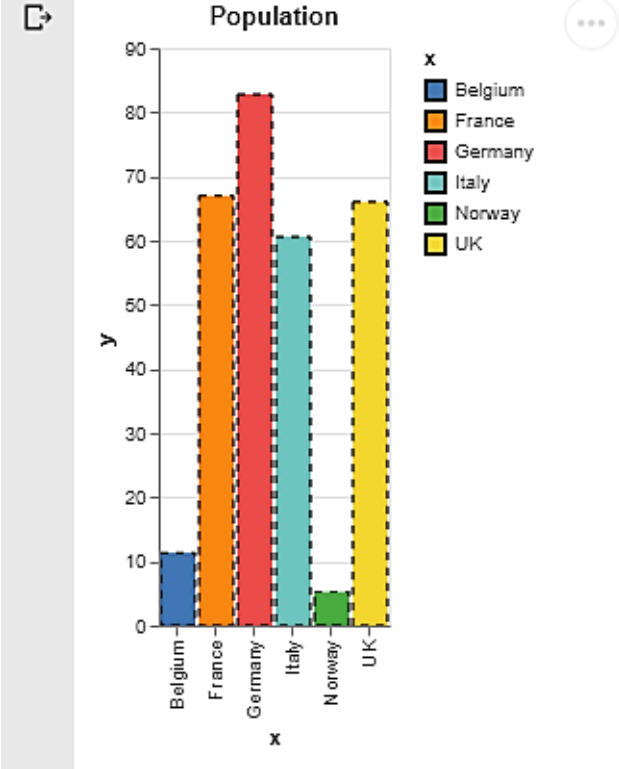
```
import altair as alt
from vega_datasets import data

source = data.barley()

alt.Chart(source).mark_line().encode(
    x=alt.X('year'),
    y='sum(yield):Q',
    strokeDash='site',
)
```

The result would be:



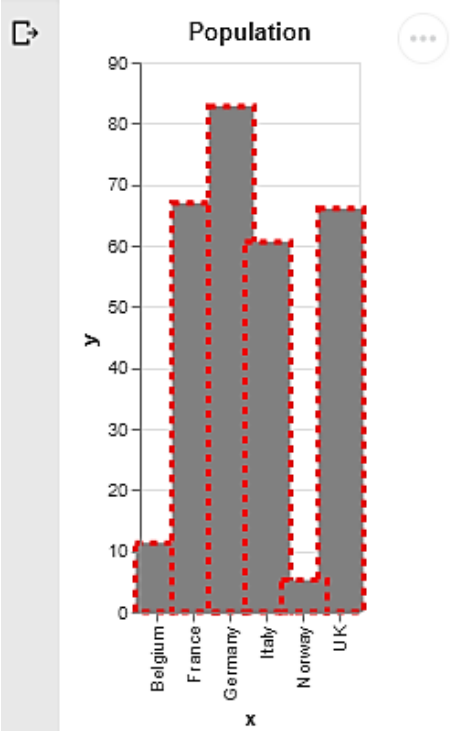
The stroke color can also be used with other marks, such as bars, as in the following example:

```
    data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                               'UK', 'Italy', 'Norway'],
                         'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
    alt.Chart(data).mark_bar(stroke = 'black', strokeDash=[4,4]).encode(
        x='x',
        y='y',
        color = 'x:N'
    ).properties(title = 'Population')
```



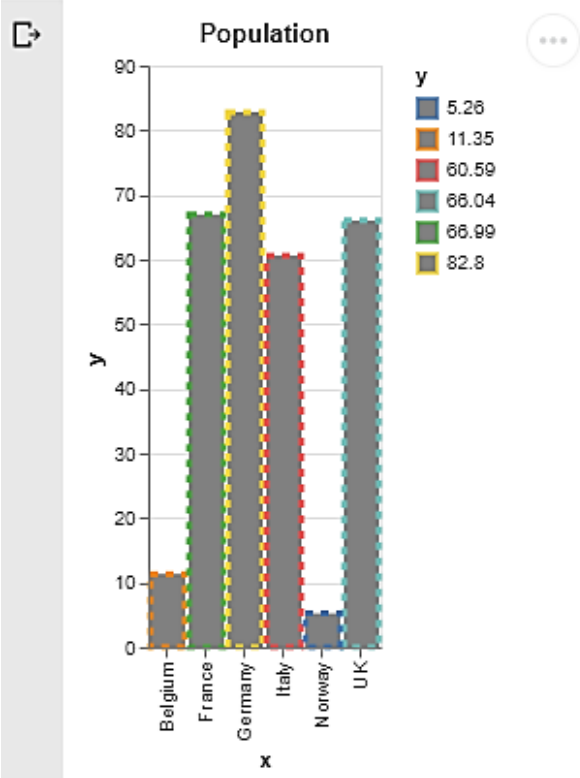
Note that we can also generate plots that are not easy to read, for instance, using colors that do not contrast enough, or by making the marks overlap unnecessarily:

```
data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar(color = 'pink', fill = 'gray',
                         stroke ='red', strokeDash=[4,4],
                         strokeWidth=3, width=25).encode(
    x='x',
    y='y'
).properties(title = 'Population')
```



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

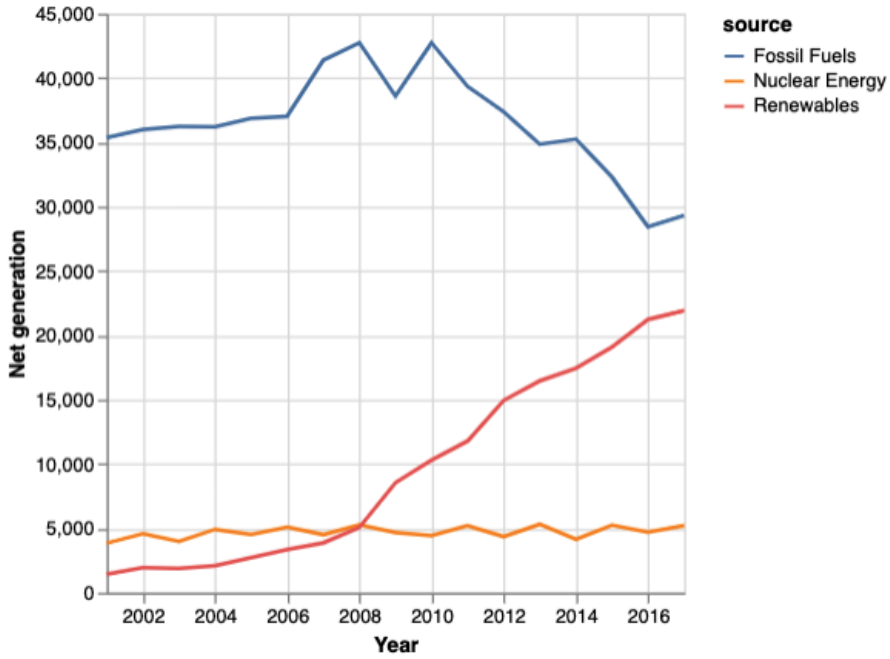
```
data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar(color = 'pink', fill = 'gray',
                         stroke ='red', strokeDash=[4,4],
                         strokeWidth=3).encode(
    x='x',
    y='y',
    stroke = 'y:N'
).properties(title = 'Population')
```



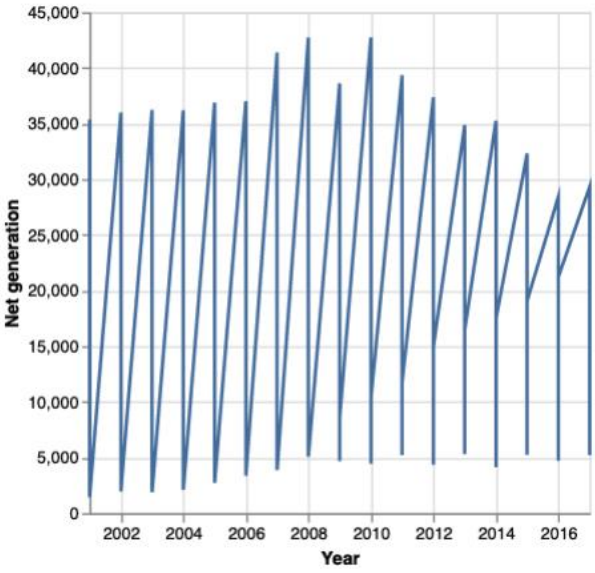
Grouping data is necessary in many visualizations. A channel named *detail* lets us group elements without requiring to map the fields to any visual properties. For example, if we want to plot the energy sources of Iowa electricity, we can draw the following line chart:

```
alt.Chart(source).mark_line().encode(
    x=alt.X("year(year)").title("Year"),
    y=alt.Y("net_generation:Q").title("Net generation"),
    color = 'source'
)
```

Which renders the sources of electricity in different colors:



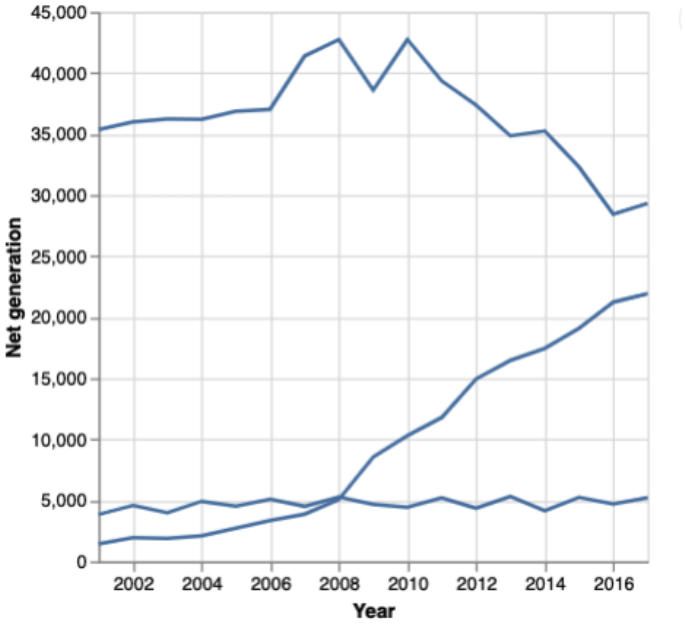
The different sources have been separated, and plot in different colors thanks to the "color = 'source'" channel. However, we might be interested in plotting all lines using the same color. Removing this line would generate the following chart:



Here, no distinction on the sources is made, and therefore, the system interprets all the data as one source. We could separate the sources by using the *detail* field:

```
import altair as alt
from vega_datasets import data

source = data.iowa_electricity()

alt.Chart(source).mark_line().encode(
    x=alt.X("year(year)").title("Year"),
    y=alt.Y("net_generation:Q").title("Net generation"),
    detail = 'source'
)
```

This would generate three lines with the same color:



Obviously, if we want to further distinguish the lines, we would need some label, which can be added in as an overlaid view, for example. But sometimes the detail will be useful by itself, because other elements help us distinguish the different individual groups.

EXERCISES

Exercise 1. Use the Vega dataset wheat.json . Plot with the x variable 'year' as temporal and y with variable 'wheat' as quantitative. Use a line plot, with a dashed pink line of width 3.

Exercise 2. Use the same dataset to produce a scatterplot with gray triangles filled with blue.

Exercise 3. Increase the size of the previous triangles.

Exercise 4. With the cars dataset, render a scatterplot of the acceleration versus the horsepower, with the points encoded as red crosses with black stroke, and their size proportional to the horsepower.

Exercise 5. Modify the previous plot so that the marks are circles with an opacity of 0.25 and the outline is the same color than the filling color.

5.2 CHANNEL OPTIONS

Channels can also be configured with extra options that perform operations on the data, such as aggregation, sorting, or binning. As might be expected, the operations that can be applied depend on the type of data.

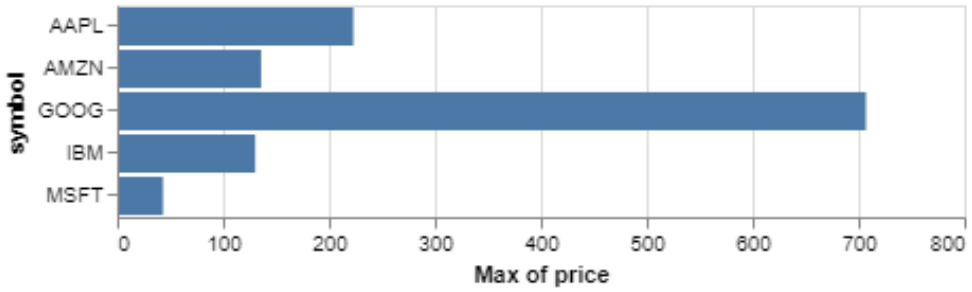
Options of x and y encodings:

- aggregate: An aggregation function is applied to the field, such as mean, sum, median, etc.
- axis: Modifies the properties of the axes.
- bandPosition: Relative position on a band of a stacked, binned, time unit or band scale. For example, the marks will be positioned at the beginning of a band if this parameter is set to value zero, and at the middle, if it is set to value 0.5.
- bin: It is used as a flag for binning quantitative fields.
- field: it is a required field (unless we are using the aggregate function of *count*) that defines the name of the field from which to pull the data values. It can also be an object that defines iterated values from the repeat operator. Dots and brackets can be used to access nested objects. If the field name contains dots or brackets that do not represent nested values, those have to be escaped with \\.
- impute: An object defining the properties of the *impute* operation to be applied.

- scale: Can be used to scale properties proportional to the data. If it is disabled, the data is directly encoded.
- sort: Defines the sort order of the encoded field.
- stack: Used to stack values of x or y if they encode values of continuous domains.
- timeUnit: The time unit (e.g., year, yearmonth, month, hours) for a temporal field.
- title: Defines a title for the field.
- type: The type of measurement ("quantitative", "temporal", "ordinal", "nominal", "geojson") for the encoded field or constant value (datum).

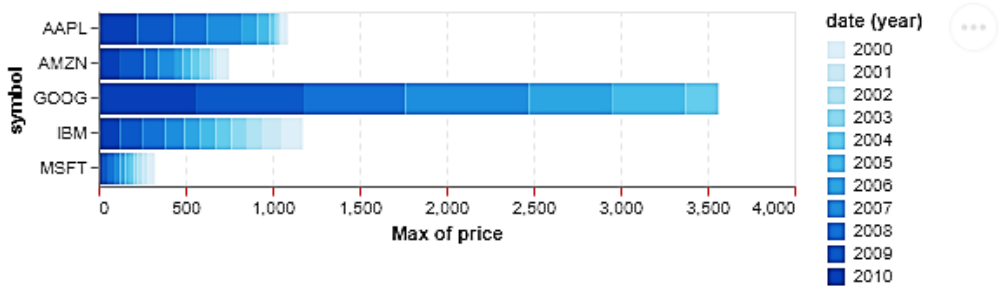
A very common operation in X and Y encoding consists in aggregating data. This can be achieved through a set of functions that let us make calculations such as maximum, minimum, average, counts, etc. If we want to calculate the maximum price of each company (defined as 'symbol' in the *stocks* dataset), we can ask Altair to calculate its maximum from the field itself (we put the values in x to improve the space usage):

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    x='max(price):Q',
    y='symbol:N',
)
```



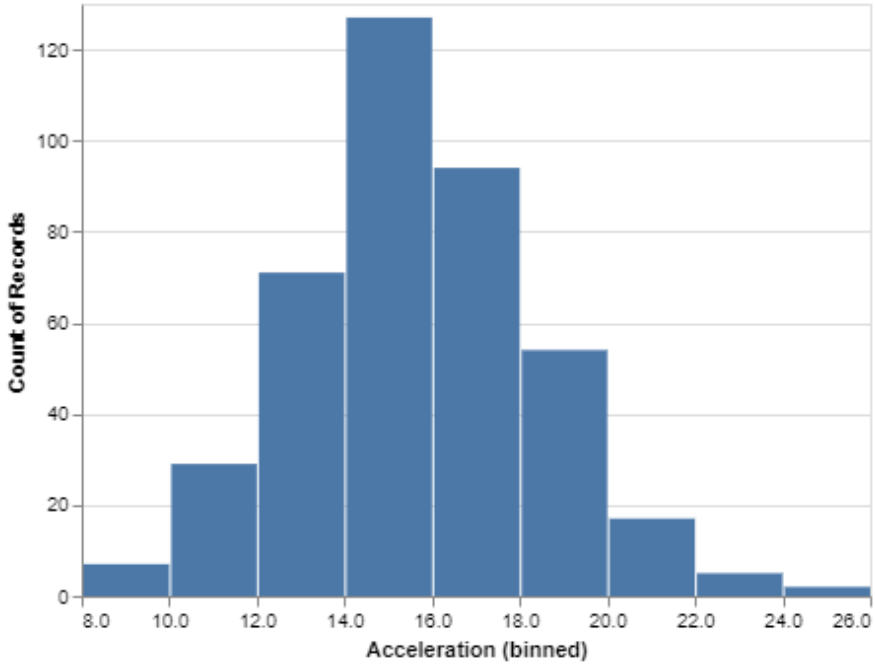
If we want to check the maximum price stock per year, we can separate per years the following way, which will create a stacked bar chart with the maximum stock price of each element as a component of the bar:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

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



In order to build histograms, we can use the *bin* option. In the following example, we separate the cars per acceleration:

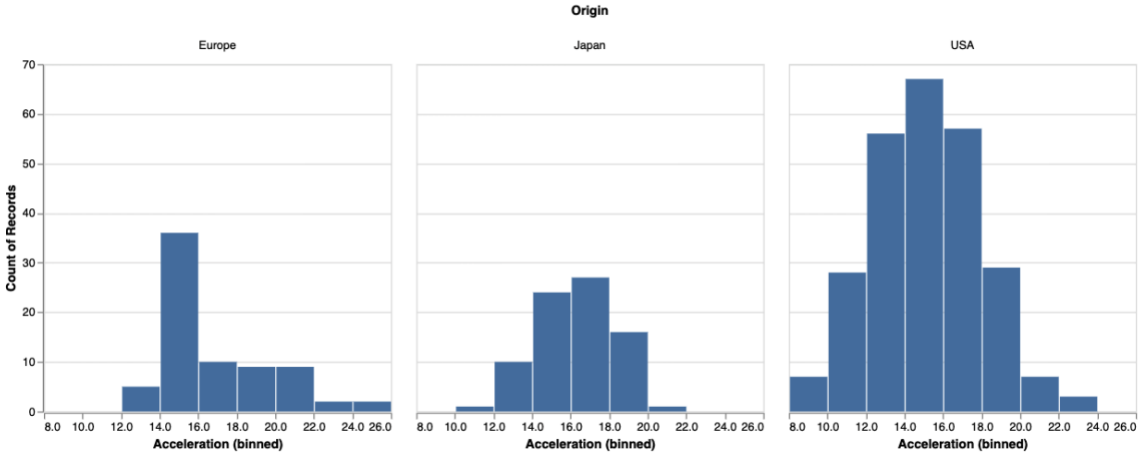
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

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



We can separate them per origin, and add a column for each origin by adding the option *column* to the plot:

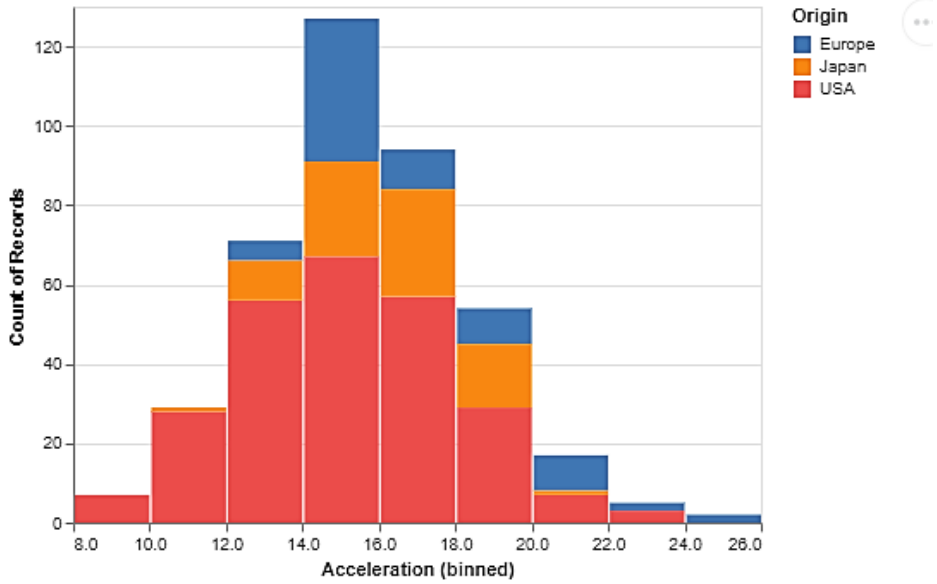
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

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

This will generate three bar charts.



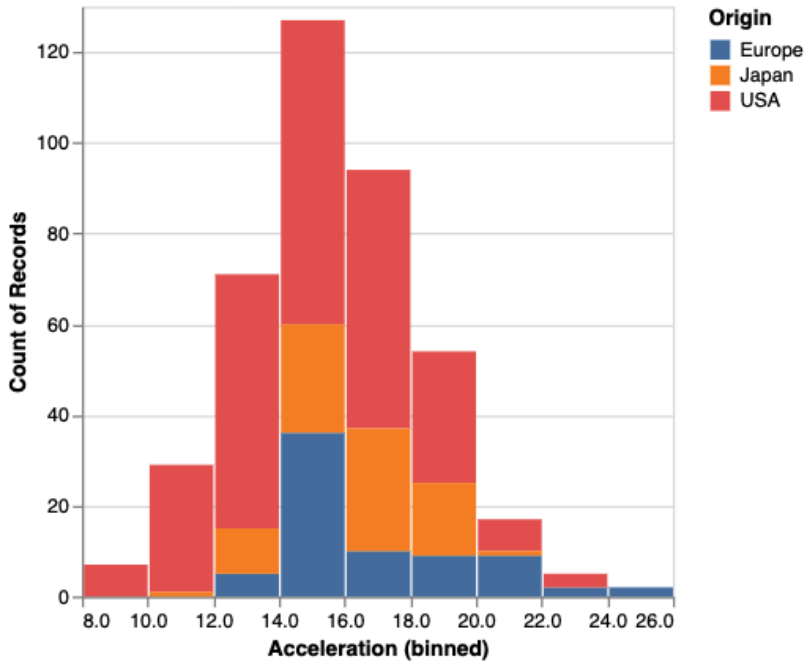
We could have stacked the bars:

```
import altair as alt
import pandas as pd
from vega_datasets import data

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 and specify which kind of sorting we want, for example, if we want the stacks to be sorted by the origin field values in ascending order we could set the sort option for the color encoding:

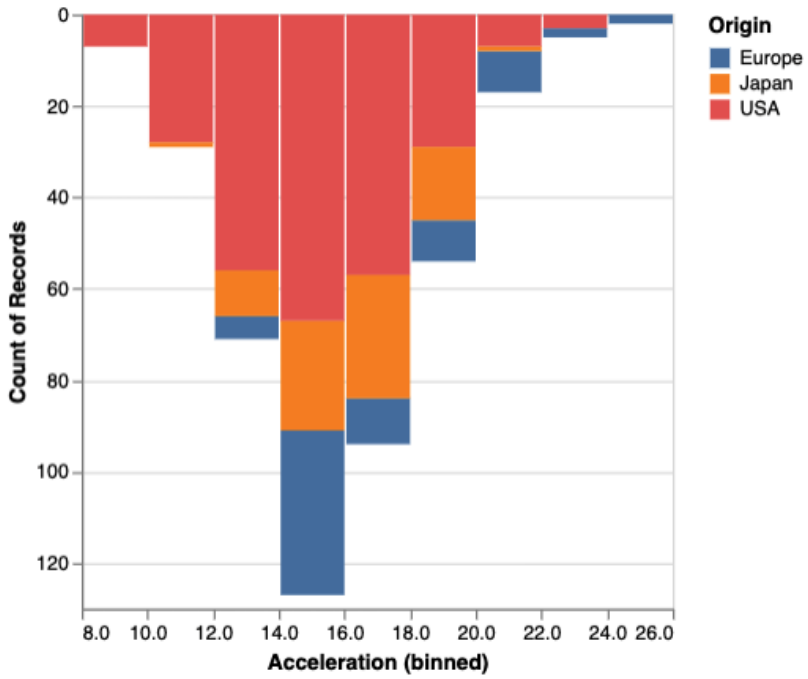
```
alt.Chart(df).mark_bar().encode(
    alt.X('Acceleration:Q', bin = True),
    alt.Y('count():Q'),
    alt.Color('Origin:N', sort = 'ascending')
)
```



We can also sort the Y data, for example, 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'),
    color = 'Origin:N',
)
```

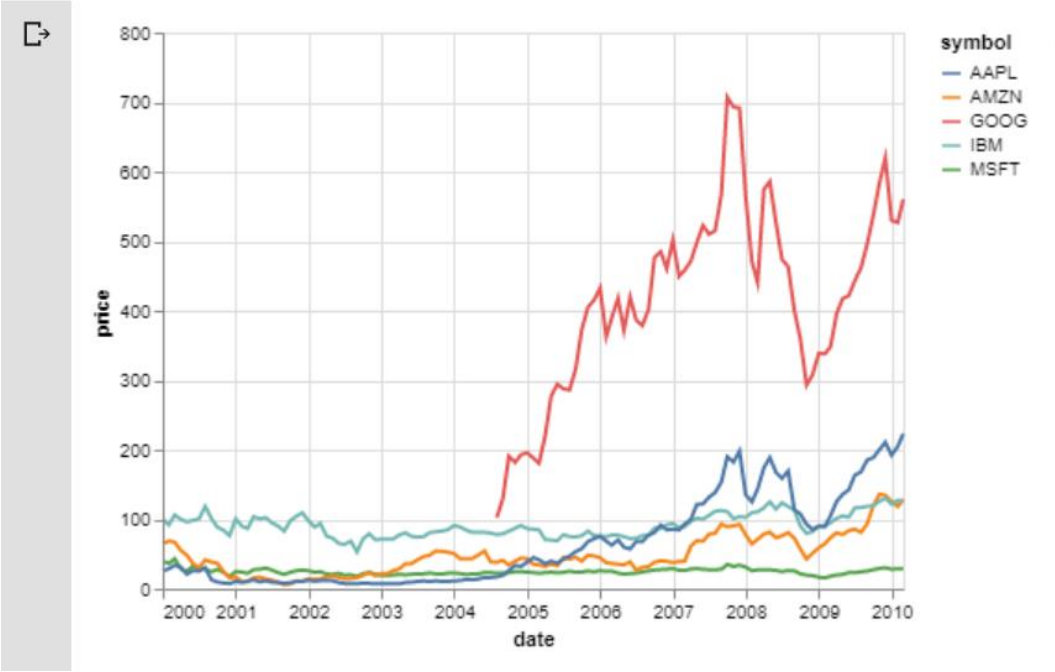
Which will set as the origin of the Y axis, the top of the chart:



Although we must avoid, in general, the use of non-standard axis.

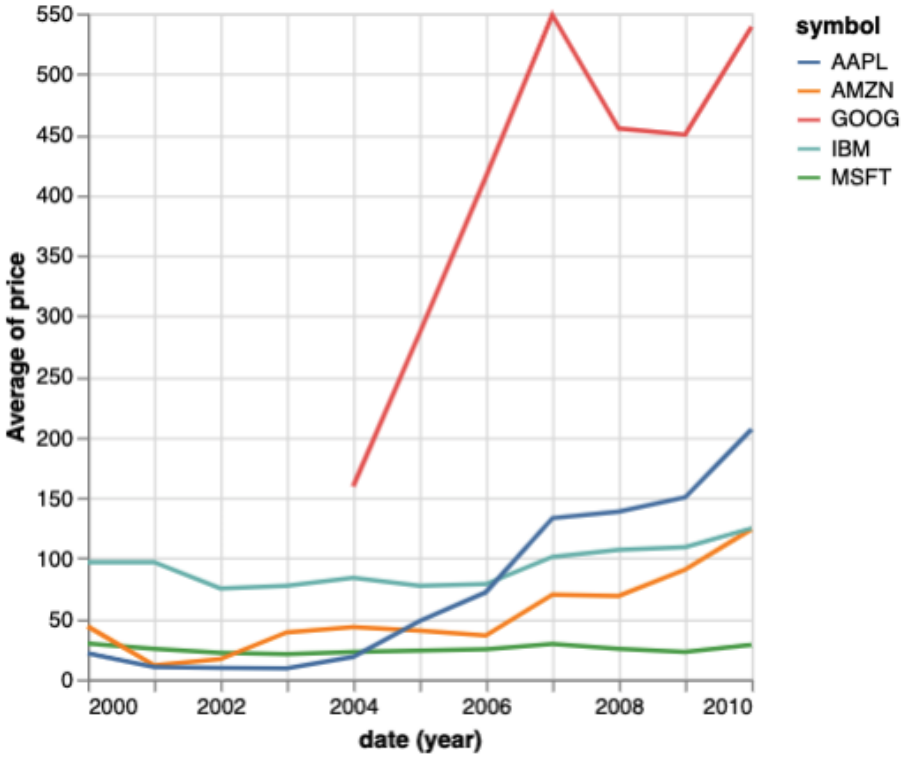
We can also bin time by using the *timeUnit* option, which allows us to group data in different time size slots. For example, the *stocks* dataset can be visualized with the fine grain data:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_line(
    ).encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)
```



Or it can be visualized by averaging the values yearly:

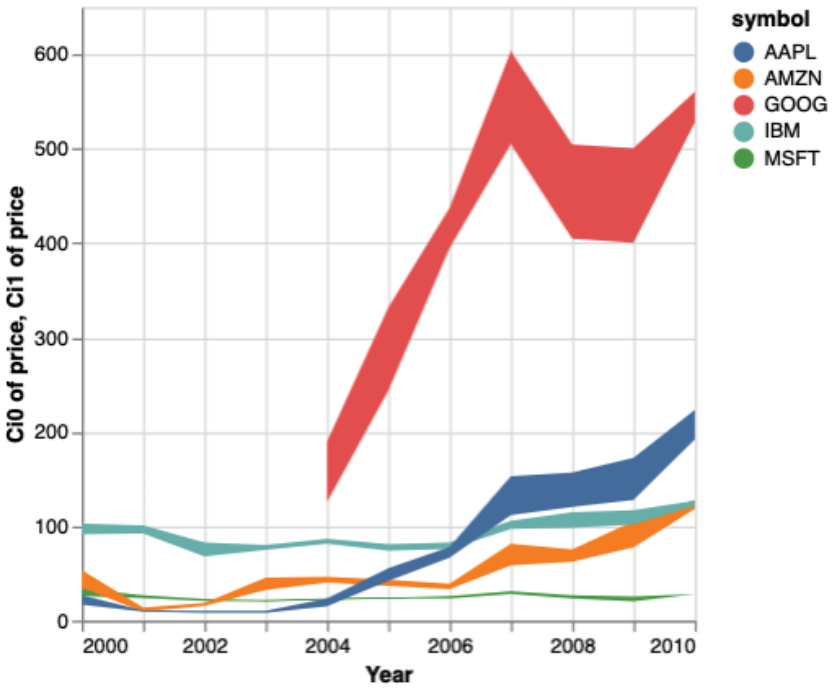
```
df = data.stocks.url

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, so we could add a confidence interval to the function, using *ci0* and *ci1* and encoding the initial and final values of the intervals in the Y axis, by using the options *y* and *y2*:

```
df = data.stocks.url

alt.Chart(df).mark_area().encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('ci0(price):Q'),
    y2 = alt.Y2('ci1(price):Q'),
    color = 'symbol:N'
)
```

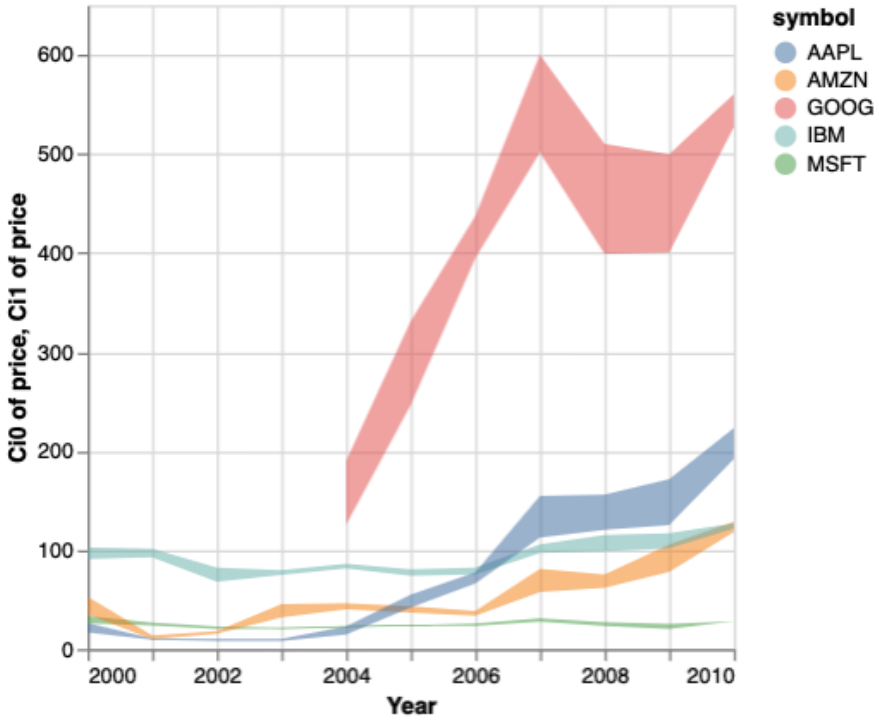
Note that now we use the area mark. And since there are values that overlap, in some regions it is not possible to see exactly where the different values start and end.



We can improve this by changing the opacity:

```
alt.Chart(df).mark_area(opacity = 0.5).encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('ci0(price):Q'),
    y2 = alt.Y2('ci1(price):Q'),
    color = 'symbol:N'
)
```

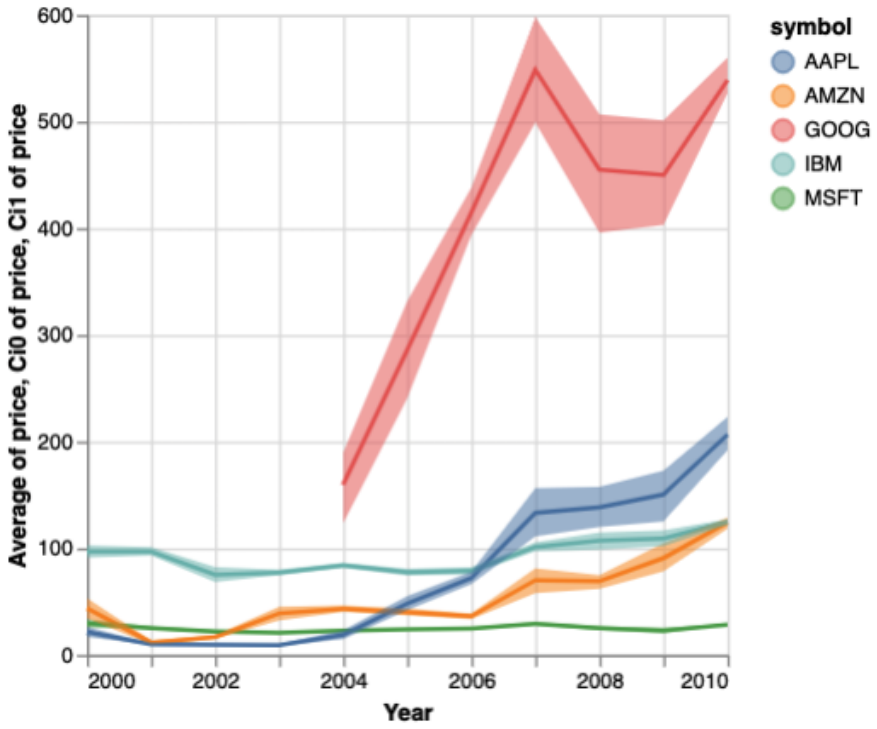
Which results in a chart that can be better interpreted.



We can also combine both charts by overlapping them as we saw previously, and the result would be:

```
df = data.stocks.url

ch1 = alt.Chart(df).mark_line().encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('average(price):Q'),
    color = 'symbol:N'
)

ch2 = alt.Chart(df).mark_area(opacity = 0.5).encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('ci0(price):Q'),
    y2 = alt.Y2('ci1(price):Q'),
    color = 'symbol:N'
)

ch1 + ch2
```

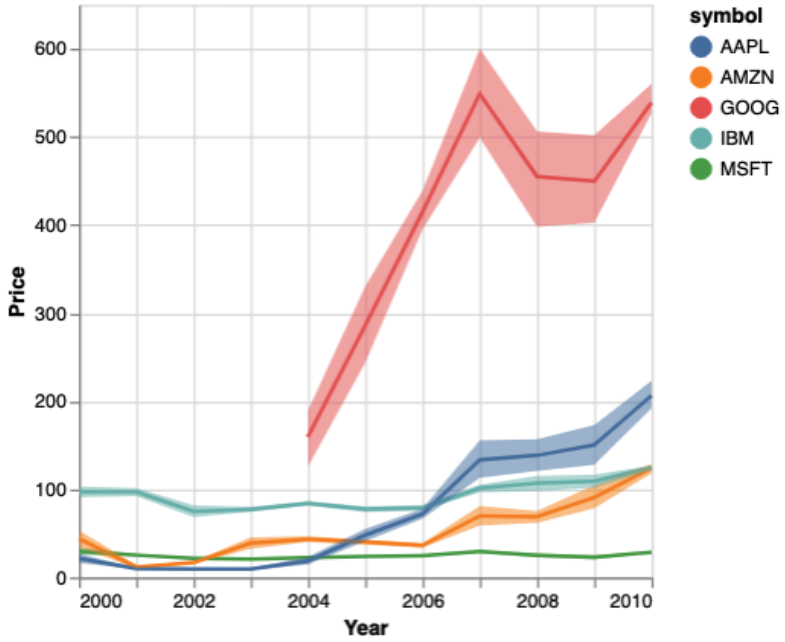
And the result:



We can polish the result by ensuring that axes are properly labelled and that the symbols in the legend are rendered opaque. The following changes would allow it:

```
ch1 = alt.Chart(df).mark_line().encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('average(price):Q').title("Price"),
    color = alt.Color('symbol:N'),
)

ch2 = alt.Chart(df).mark_area(opacity = 0.5).encode(
    x = alt.X('date:T', timeUnit = 'year').title("Year"),
    y = alt.Y('ci0(price):Q').title(""),
    y2 = alt.Y2('ci1(price):Q').title(""),
    color = alt.Color('symbol:N', legend=alt.Legend(symbolOpacity = 1.0)),
)

(ch1 + ch2)
```

With the following result:

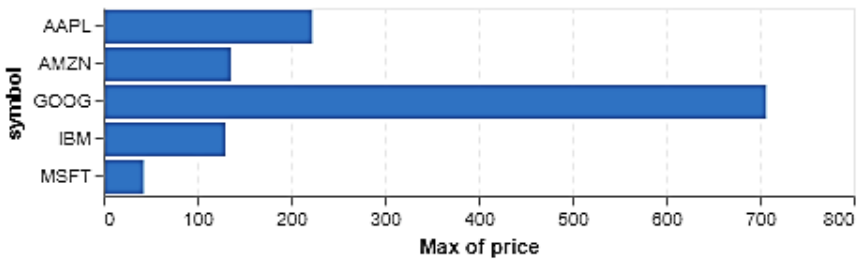


5.3 CUSTOMIZATION OPTIONS

If we need to add other configurations, we can use the alternative naming for the encodings: *alt.X* for the X axis, and *alt.Y* for the Y axis. Those are functions that accept several parameters (separated by commas) that can be used to further configure the properties.

If we want to change axis properties, such as adding a dashing style to the grid in the chart, we can do it the following way:

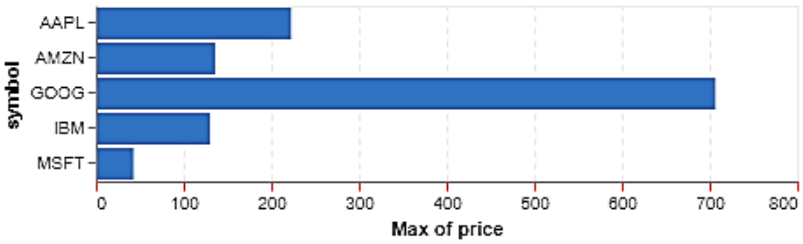
```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q', axis = alt.Axis(gridDash = [4,3])),
    y='symbol:N',
)
```



Note that we use the same syntax for the grid dashing than for the strokes.

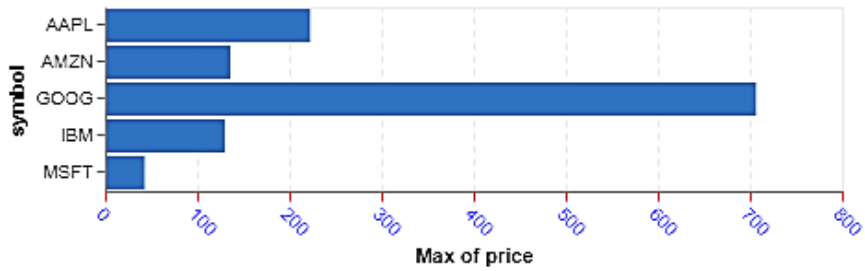
In the same manner, we can also add other features, such as colors to the ticks:

```
df = data.stocks.url

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



Or we can even modify how the labels are shown:

```
df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q',
          axis = alt.Axis(gridDash = [4,3], tickColor='red',
                          labelColor='blue', labelAngle=45)),
    y='symbol:N',
)
```



There are all sorts of customizations that can be made with these parameters. When you start typing on the editor, the options will appear on a floating window.

But with a great power comes a great responsibility: Like in the previous case, the fact that we can do many modifications to the original layout does not mean that these will be appealing to the user. We have to make sure that we do not

add extraneous embellishments that prevent the user to properly perceive the information we are plotting.

There are other customization options that affect the whole layout, instead of just a single object of the chart. Size of the chart can be changed with the *width* and *height* properties.

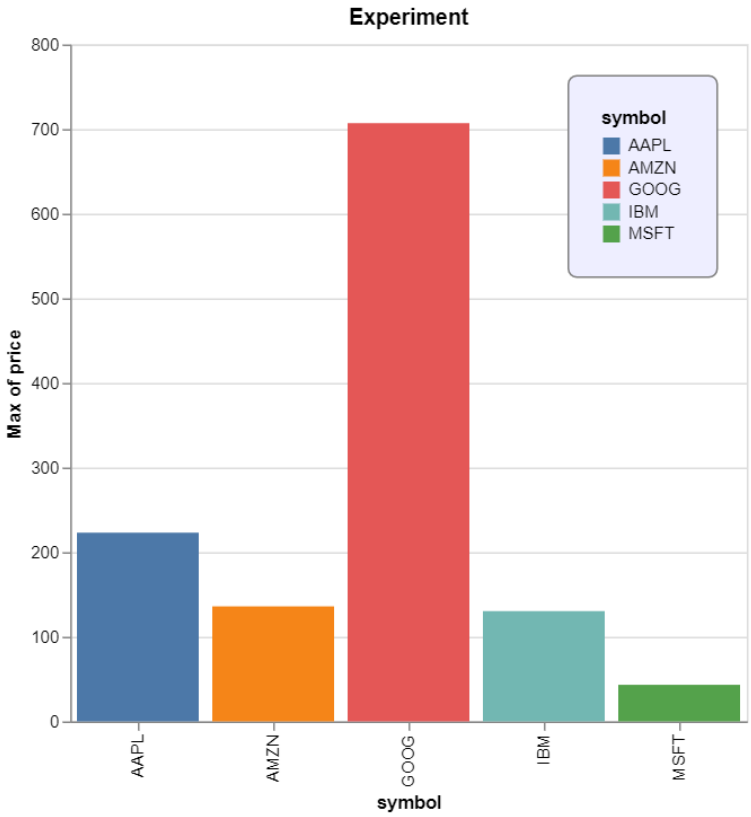
```
source = data.stocks.url

alt.Chart(source).mark_bar().encode(
    x='symbol:N',
    y='price:Q',
    color = 'symbol:N'
).properties(width = 100, height = 200)
```

We can also change the title with the *title* property, that receives a string.

Finally, other useful configurable elements are those of the legend. We can change the title, stroke color, fill color, padding, and orientation, by using the *configure_legend* function, as can be seen in the following example:

```
df = data.stocks.url

alt.Chart(df).mark_bar().encode(
    alt.X('symbol:N'),
    alt.Y('max(price):Q'),
    alt.Color('symbol:N')
).properties(
    width = 400, height = 400, title = 'Experiment'
).configure_legend(
    strokeColor = 'gray',
    fillColor = '#EEEEFF',
    padding = 20,
    cornerRadius = 5,
    orient = 'top-right'
)
```

That would have as a result the following chart:



Changing the title of the legend would require a little circumvention, since it takes the name of the axis. We can, however, disable it with the option *title = null*.

There is another option for the *width* and *height* values, make them depend on the size of the HTML page or container. In order to do so, you only need to change the *width* value to *container*. This will adjust the size to the available, given by the HTML page. The advantage of this feature is that the size of the charts will adapt to window resizing. Note that, however, this function does not seem to run well in Google Colab.

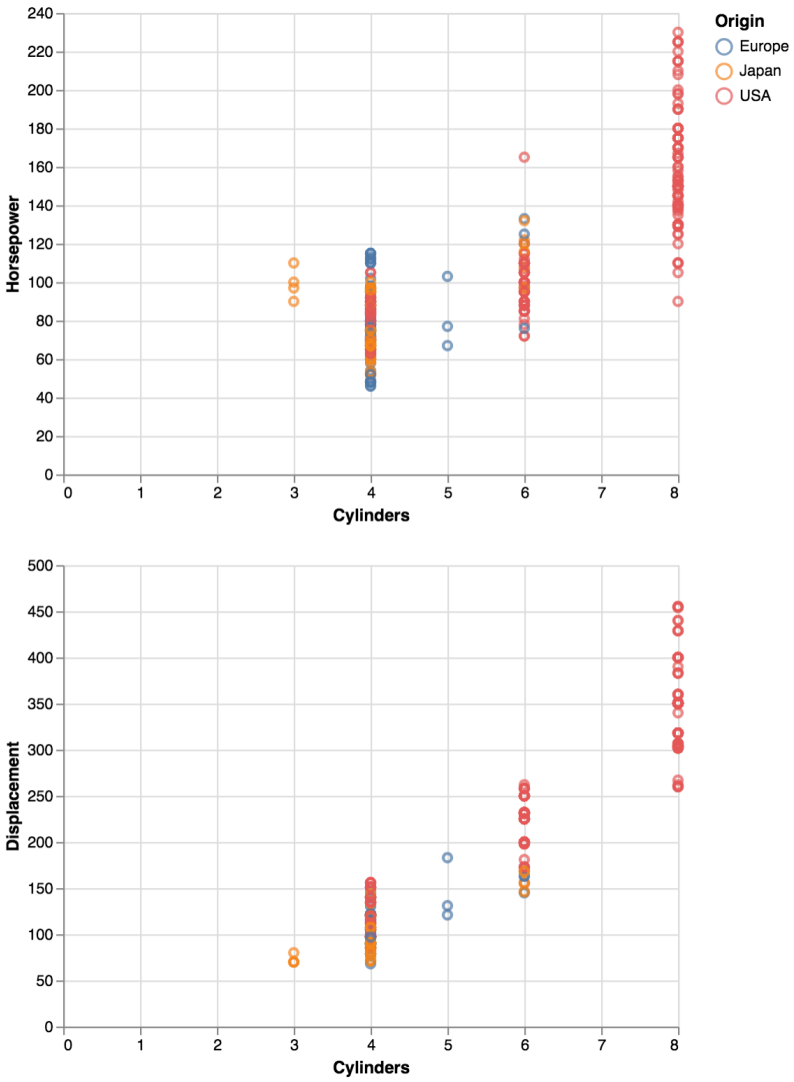
5.4 MULTIPLE CHARTS: SIMPLE COMBINATIONS

We already saw the '+' operator to plot two charts one in top of the other. There are a number of ways to combine charts, such as by using layers. In this section, we want to provide some more examples of simple chart combinations.

If we want to generate more than one chart, we can do it with the '&' operator. This will plot two charts one on top of the other. We can do it by naming the two different charts, or using the parenthesis to group the plotting method:

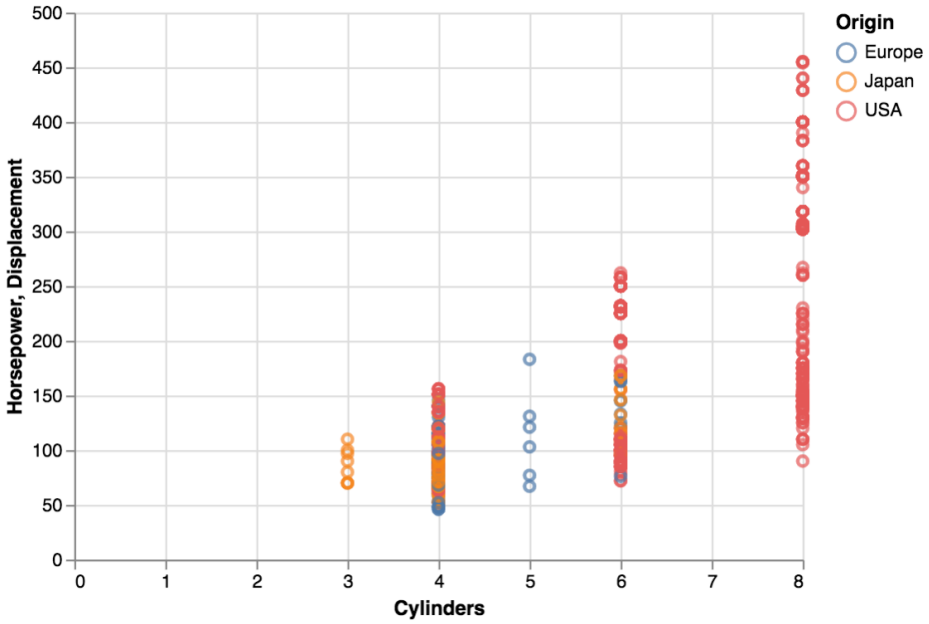
```
df = data.cars()

(alt.Chart(df).mark_point(shape = 'circle')).encode(
    alt.X('Cylinders:Q'),
    alt.Y('Horsepower:Q'),
    alt.Color('Origin:N')
) & alt.Chart(df).mark_point(shape = 'circle').encode(
    alt.X('Cylinders:Q'),
    alt.Y('Displacement:Q'),
    alt.Color('Origin:N')
)
```

Which will generate:



Note that the legend is shared.

But we can also overlay one on top of the other, by substituting the '&' symbol for a '+', and the result would be:



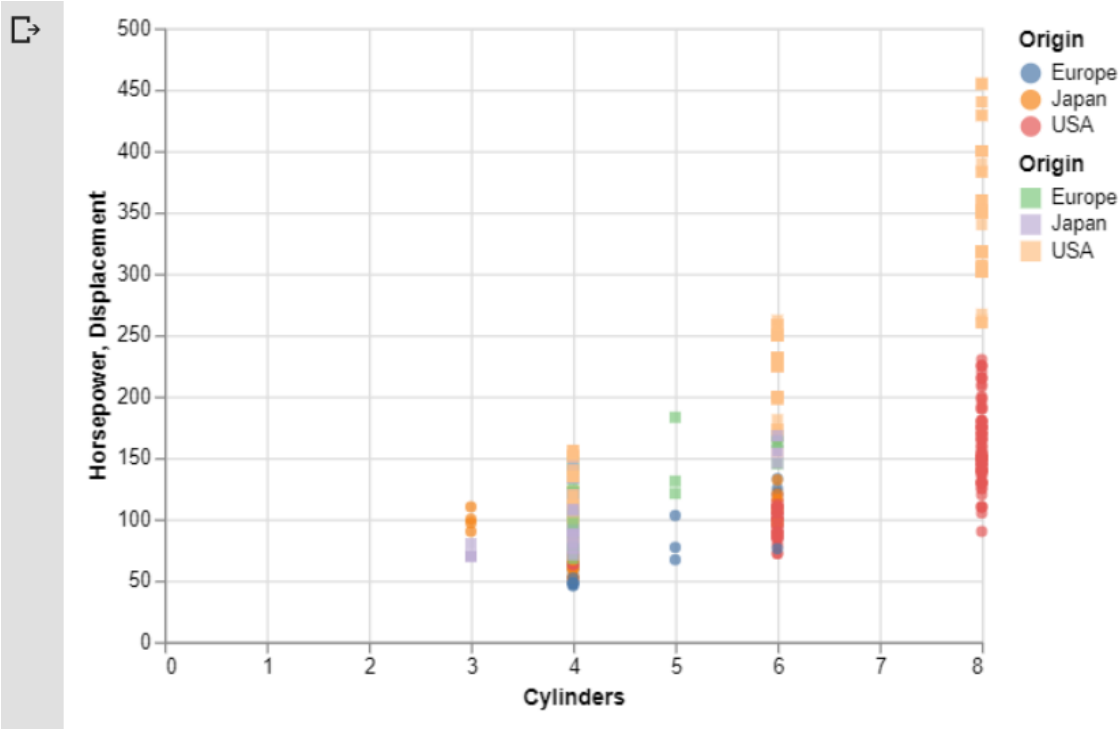
In this case, there is ambiguity since we are using color encodings of the data that are shared among both plots. If they are indicating different data, marks must be different, to avoid confusion.

It is possible to change the marks of one of the charts, to facilitate the separation. And it is also possible to change the color scheme of the charts, but since it is a property that it is by default shared among all the charts that are plot on top of each other, it is necessary to explicitly ask Altair to treat the colors independently. This can be achieved by using a combination of the *scale* property and *resolve_scale* method to make the colors independent.

In the following example of one of the charts takes squares as shape, and the other circles. Then, we set the color scheme of one of the charts as *accent* color scheme (you need to check the Vega documentation to see the available color schemes (https://vega.github.io/vega/docs/schemes/). Then, we ensure that both schemes are treated independently:

```
df = data.cars()

first= alt.Chart(df).mark_circle(
    ).encode(
    x='Cylinders:Q',
    y=alt.Y('Horsepower:Q'),
    color = 'Origin:N'
    )
second = alt.Chart(df).mark_square(
    ).encode(
    x='Cylinders:Q',
    y=alt.Y('Displacement:Q'),
    color=alt.Color('Origin:N', scale=alt.Scale(scheme='accent'))
)

(first + second).resolve_scale(color='independent')
```

And the result is:

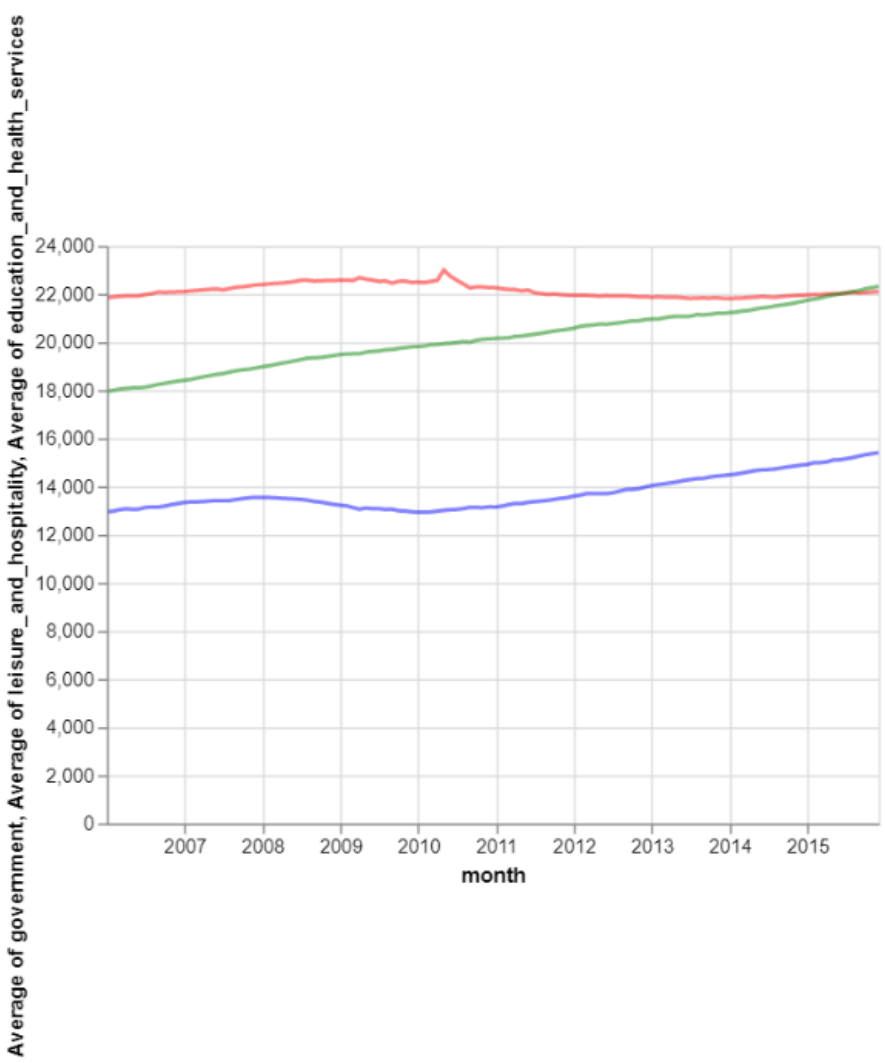


Layers can be used to visualize wide form data, such as the *stocks* dataset:

```
df = data.us_employment()

base = alt.Chart(df).mark_line(
    color = 'red',
    strokeWidth = 2, strokeOpacity = 0.5
    ).encode(
    x='month:T',
    y='average(government):Q'
)

alt.layer(
    base,
    base.encode(y='average(leisure_and_hospitality):Q',
                color = alt.value('blue')),
    base.encode(y='average(education_and_health_services):Q',
                color = alt.value('green')))
```

Layers are the equivalent to the '+' operators on charts, but let you add more than two elements. In this case, since the data is wide form, we do not have a row per entry, and therefore, if we want to visualize all the columns, we would need to specify them one by one.

The result of the previous code is:

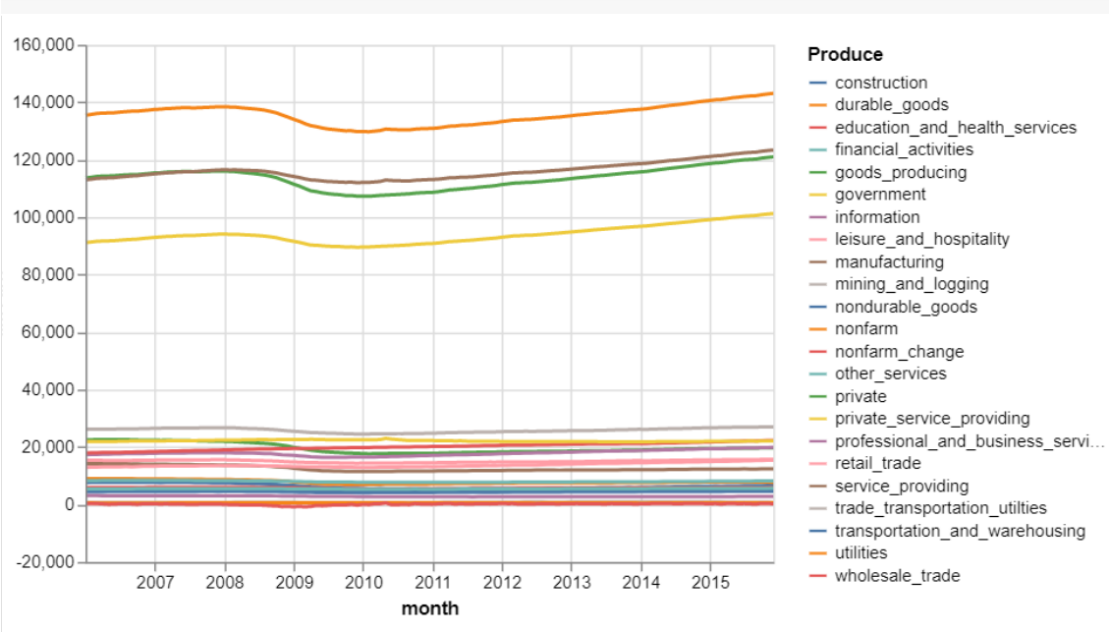


This is far less convenient than when data is in long form.

If we want to render the whole column set, we can melt the data using the function from pandas, and then render it as if it was long form:

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

alt.Chart(df2).mark_line().encode(
    x='month:T',
    y='amount:Q',
    color = 'Produce',
)
```

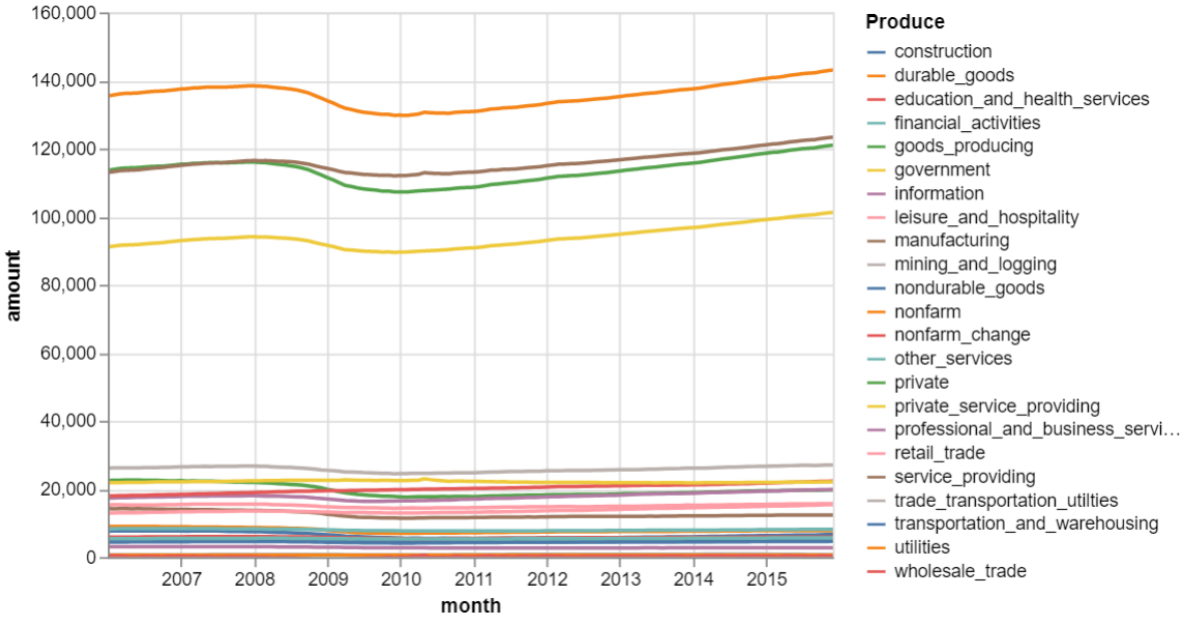
This would result in the following chart:



The previous chart is rendering negative values, which makes the Y axis span until the -20000 value. We can clip this by defining the domain of the axis 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(
    x='month:T',
    y=alt.Y('amount:Q',scale = alt.Scale(domain=(0,150000))),
    color = 'Produce'
)
```

By adding the extra option of *clip* to *True*, the values negative values (that would be plotted in spite of the fact that the axis would start at 0, will be clipped. The result is shown in the following Figure.



EXERCISES

Exercise 1. Use the cars Vega dataset and plot a bar chart that counts the number of cars with each quantity of cylinders.

Exercise 2. Use the cars Vega dataset and plot a bar chart that shows the maximum displacement of the cars from each origin.

Exercise 3. Use the cars Vega dataset and plot a bar chart that shows the maximum displacement of the cars from each origin.

Exercise 4. Use the cars Vega dataset and plot chart that shows the average miles per gallon of the cars from each origin.

Exercise 5. Use the stocks Vega dataset and plot a bar chart that shows the average stock price for each company.

Exercise 6. Use the stocks Vega dataset and plot a bar chart that shows the average stock price for each company.

Exercise 6. Use the stocks Vega dataset and plot a chart that shows the average stock price for each company per year.

Exercise 7. With the cars Vega dataset, plot a histogram of the cars with different miles per gallon, stacked per origin, inverting the current sorting.

Exercise 8. Create a line plot that shows the price evolution of the yield barley in the different sites from the barley dataset.

Exercise 9. Create a bar chart with the maximum yield value of each variety using the barley dataset.

Exercise 10. Create a bar chart with the average yield value for each site using the barley dataset.

Exercise 11. Create a line chart using the cars dataset, with the average displacement of the cars per year, with different lines for each origin.

Exercise 12. Create a line chart using the cars dataset, with the average miles per gallon per year.

Exercise 13. Improve the previous chart by adding the confidence intervals of the miles per gallon. Plot both charts together.

Exercise 14. Create another line chart, also with the cars dataset, with the average horsepower per year, with confidence intervals, separated per origin, and render both of them separately.

Exercise 15. Render both charts on top of each other. Resolve the ambiguities.

Exercise 16. Render the first three columns of the employment dataset with three different colors.

Exercise 17. Add a rule on the average of the employment values for those data.

Exercise 18. Plot the monthly average of the all employment types of the employment dataset.

Exercise 19. Plot as a rule the average of the all employment types of the employment dataset.

Exercise 20. Use the dataset "Nivell academic de la població per sexe de la ciutat de Barcelona" of year 2018 from the BCN Open Data web, and show the number of people in the city of each academic degree.

Exercise 21. Show the number of people holding each academic degree ("Nivell academic") of the Horta-Guinardó neighborhood for a concrete month.

Exercise 22. Compare the higher academic degrees (high school and university) of each neighborhood in Barcelona for a certain year.

Exercise 23. Plot the evolution of the academic degree for men and women in Barcelona.

Exercise 24. Plot the evolution of the academic degree for men and women in a selected neighborhood in Barcelona.

Exercise 25. Show how the university degree has evolved in Barcelona.

6. Charts

We have already seen how the mark selection determines the type of chart, e.g. points will generate scatter plots, while bars will generate bar charts.

In this section we will overview these different types of charts, as well as more elaborated ones.

6.1 BASIC CHART TYPES

The simplest charts have already been presented:

- Scatterplot: It can be created by encoding the mark as a point (*mark_point*), and its coordinates with the x and *y* fields.
- Bar chart: It requires the use of the mark as a bar (*mark_bar*) and the x field encodes the variable and the *y* coordinate its value.
- Line chart: The mark must be configured as a line (*mark_line*), the x must contain the first field, and the *y* coordinate the second.
- Area chart: It is equivalent to the previous one, where the mark is configured as an area (*mark_area*).

6.2 VARIATIONS OVER SIMPLE CHARTS

There are all sorts of small modifications that can be done to the basic plots, such as changing the values of the axis, for example to make them appear as percentages:

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

Otherwise, the Y axis would have values such as 0.018, instead of 1.8%.

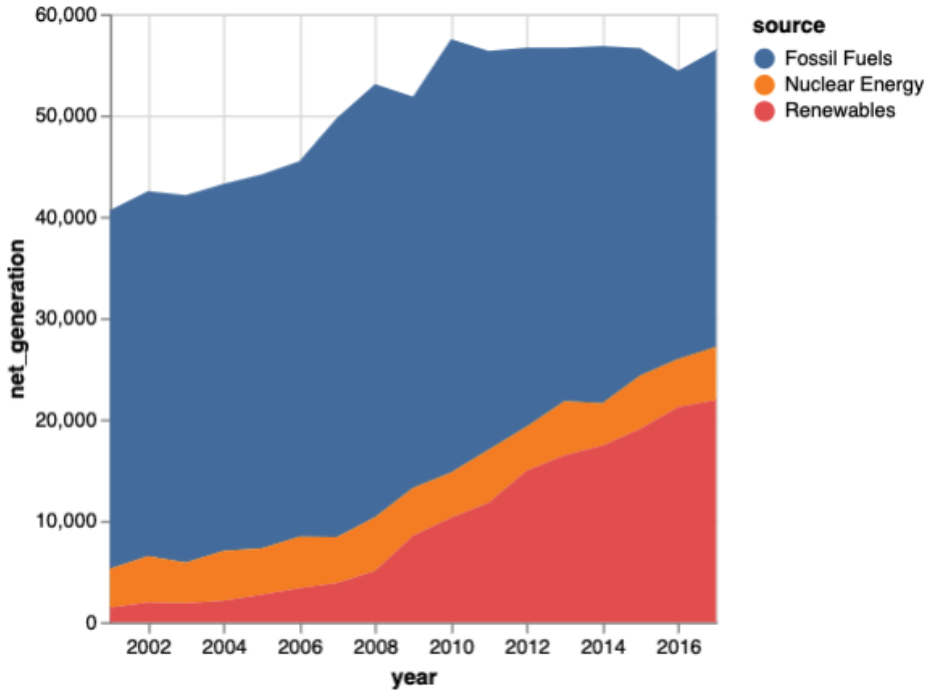
We can also add points to the line charts, just by adding the option *point=True* to the mark properties.

Or we can configure the line thickness by adding the option *size* to the encoding and making it change according to a certain variable, such as the one encoded in the Y axis.

We have already seen area charts in the previous document. These can be configured in two different ways: stacked and not stacked. The default is

stacked. However, this may lead to difficulties in calculating ratios. For example, if we show the electricity production sources for Iowa, we would get:

```
source = data.iowa_electricity.url

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

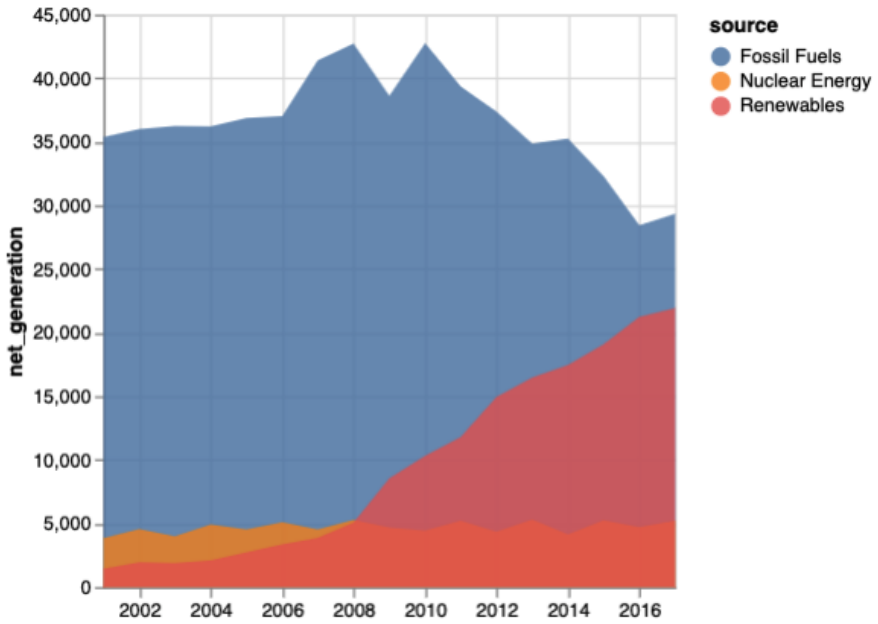


But if we want to compare them side by side, we should specify that they are not stacked:

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

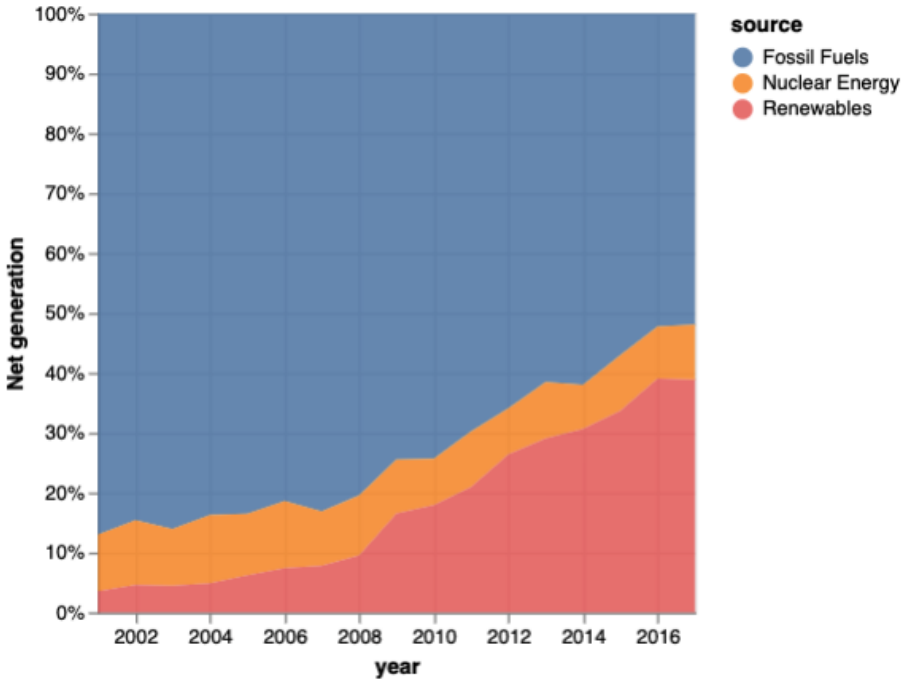
But now they overlap. So we should add some sort of transparency to provide visual comparison:

```
alt.Chart(source).mark_area(opacity = 0.8).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', stack = False),
    alt.Color('source:N')
)
```



But another possibility is to stack the data, but making them normalized, so that the relative ratios are easier to appreciate:

```
source = data.iowa_electricity.url

alt.Chart(source).mark_area(opacity = 0.8).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q', stack = 'normalize').title('Net generation'),
    alt.Color('source:N')
)
```



Note that, by keeping a certain level of transparency, we make the grid lines visible, and thus, visually evaluating magnitudes is easier. If the render is totally opaque, these lines are not visible.

We could also represent each category in a different chart, by creating the so-called Trellis chart. In order to do so, we only have to ask Altair to assign a different row per category:

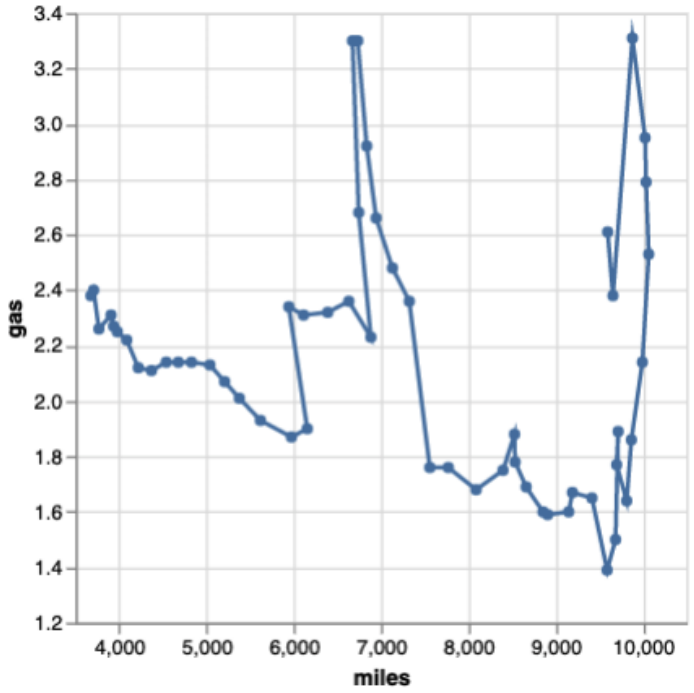
```
source = data.iowa_electricity.url

alt.Chart(source).mark_area(opacity = 0.8).encode(
    alt.X('year:T'),
    alt.Y('net_generation:Q').title('Net generation'),
    alt.Color('source:N'),
    alt.Row('source:N')
)
```

Note that, although Trellis charts can be of great utility for visual comparison, this current example, configured as is, is not the best one, since one would need to scroll up and down to get a good impression of all the charts.

Connected line chart with ordering (also known as connected scatterplot): Sometimes it may be useful to draw a scatterplot where the lines between points do not follow the same order than the x values. We can solve this using a regular line plot with ordering:

```
driving = data.driving()

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

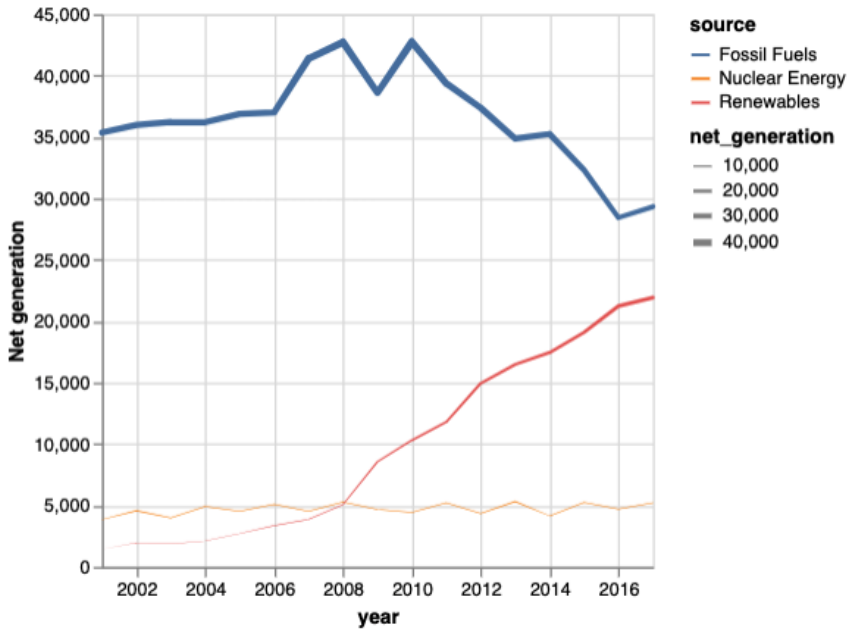
The result would be:



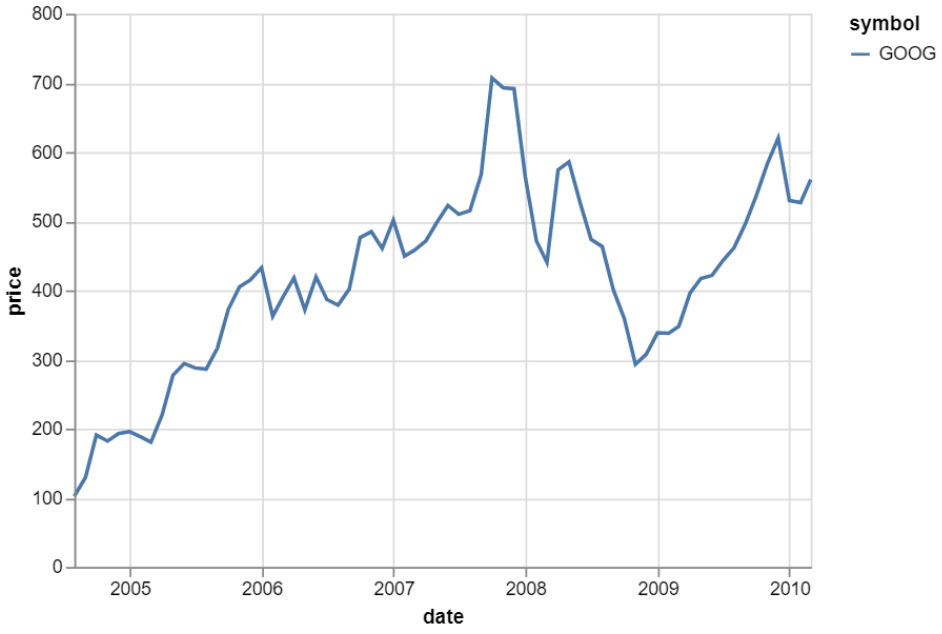
We can also modify the line chart with the use of the *trail* channel and making the trail to have different width according to a variable:

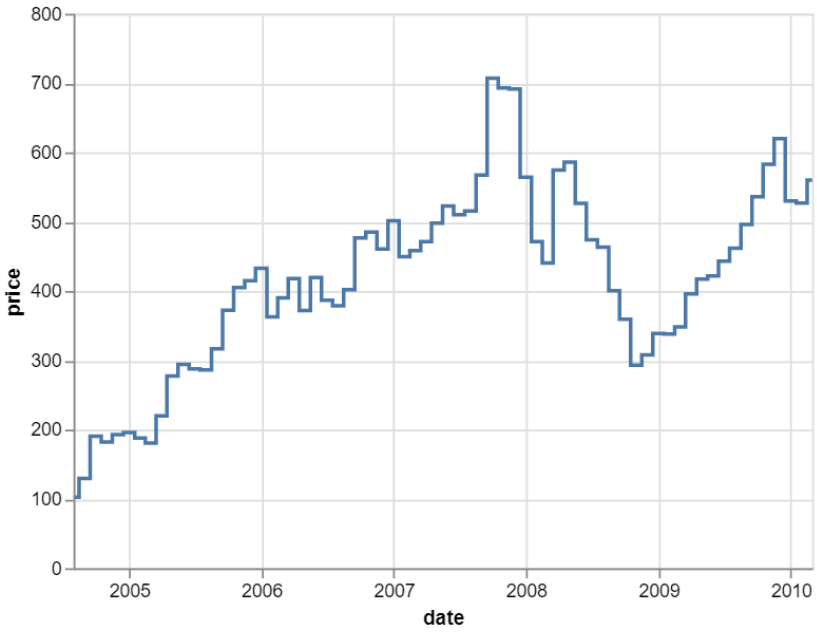
```
source = data.iowa_electricity.url

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

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



Another way to configure line plots is by changing the way the points are connected through the *interpolate* option of the *mark_line* function. If we interpolate the previous stock value of Google company (using the *stocks* dataset) with linear or step, we get the following two different charts:



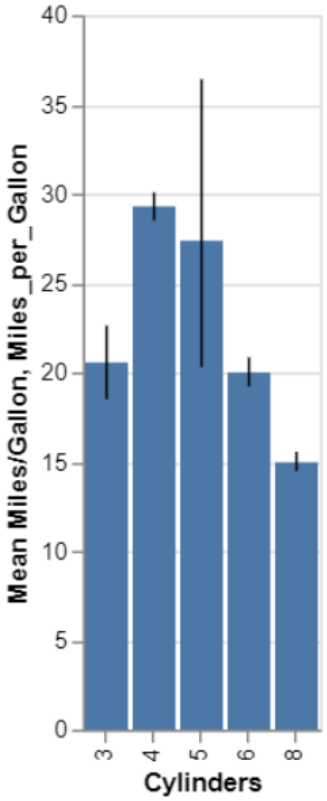


Other interpolation options include *linear*, *linear-closed*, *step*, *step-before*, *step-after*, *basis*, *basis-open*, *basis-closed*, *cardinal*, *cardinal-open*, *cardinal-closed*, *bundle*, and *monotone*.

We can also enrich bar charts with extra information such as the error bars. This can be simply implemented by creating a layer that contains the error bars for the dataset, as in the following example, where we used the *cars* data:

```
base = alt.Chart(cars).mark_bar().encode(
    x='Cylinders:O',
    y=alt.Y('mean(Miles_per_Gallon):Q', title='Mean Miles/Gallon')
)

errorbars = alt.Chart(cars).mark_errorbar(extent = 'ci').encode(
    x='Cylinders:O',
    y='Miles_per_Gallon:Q',
)

base + errorbars
```

The result is:



7. Advanced chart types

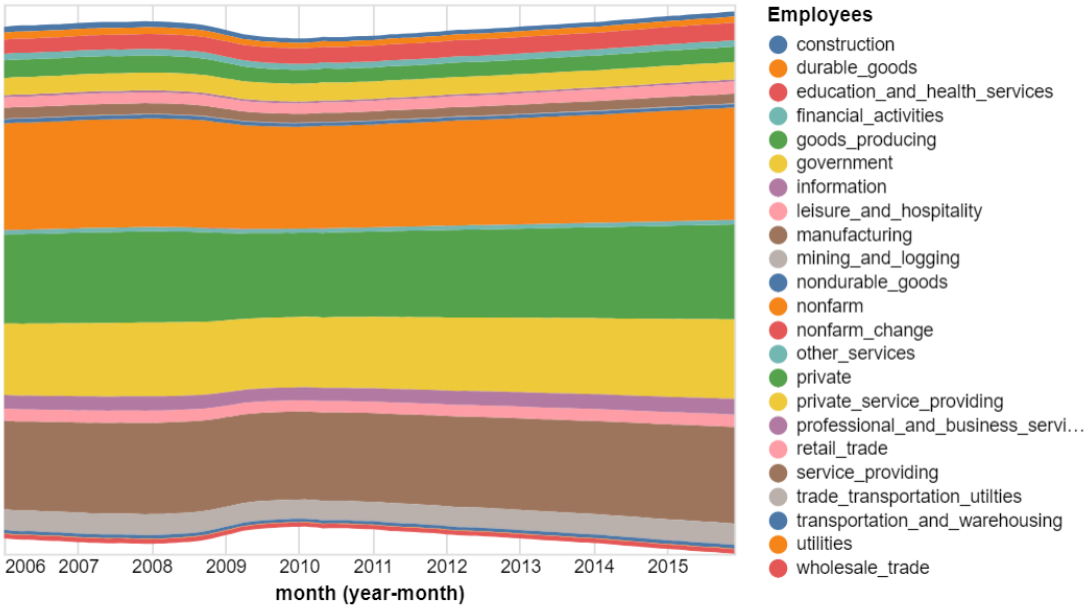
Besides those simple charts and its derived versions, there are other, more sophisticated charts that are focused on showing a certain type of information.

Streamgraphs are special versions of area charts whose goal is to **represent large quantity of categories whose values change along the time**. Streamgraphs emphasize variation along the time, and thus, the most important features to communicate are the evolution of the different elements, more than the exact amounts. The main features of a streamgraph are the fact that areas are stacked, and that they are distributed above and below the X axis.

In Altair, these charts can be obtained by asking the system to stack the elements on 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'
    )
)
```

The result is:



Note that in this case, since the emphasis is put in the time changes, it is less necessary to put the vertical axis values, and therefore it has been removed.

Box plots are plots used **to represent statistical information**. More concretely, it groups the upper and lower quartiles inside a box, with a whisker that commonly indicates the median. In Altair, we only need to use the mark boxplot (*mark_boxplot*), as in the following example:

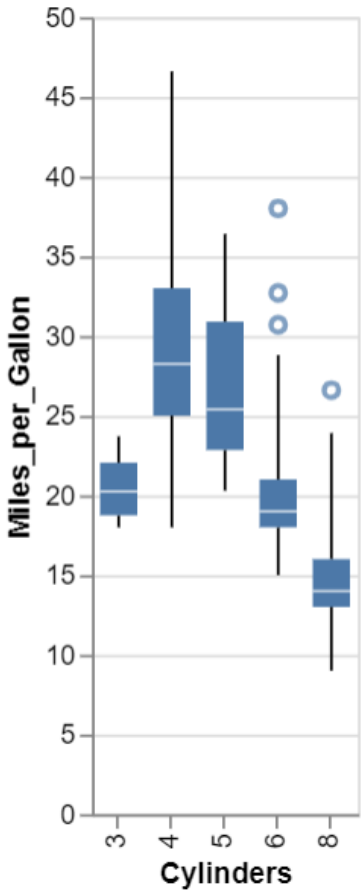
```
import altair as alt
from vega_datasets import data

source = data.cars()

alt.Chart(source).mark_boxplot().encode(
    x='Cylinders:O',
    y='Miles_per_Gallon:Q'
)
```

The default way to indicate outliers in Altair is by using points. Outliers are defined as points that are at a distance larger than 1.5 from the interquartile range (or IQR: the middle 50% of the samples). IQR is calculated as the difference between the 75th and 25th quartiles: IQR = $Q_3 - Q_1$. This threshold can be adjusted by using the *extent* property.

If we extend it to 3, as in the following code:

```
    alt.Chart(source).mark_boxplot(extent = 3.0).encode(
        x='Cylinders:O',
        y='Miles_per_Gallon:Q'
    )
```

The result would be:

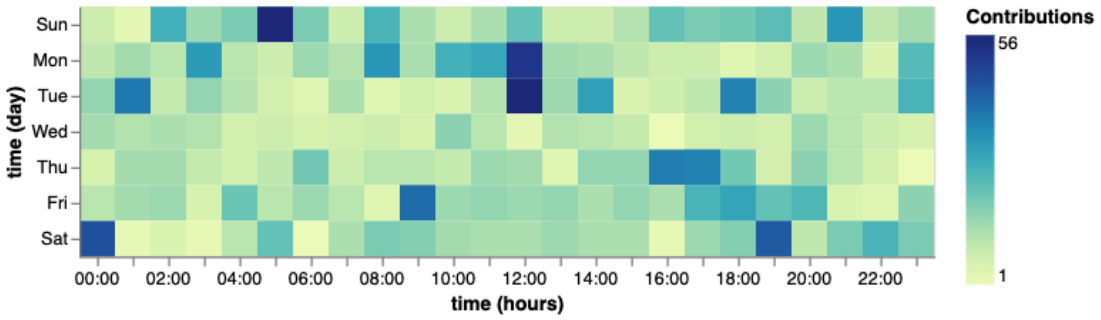


We could even ignore the outliers totally by defining the extend to be equal to the maximum – minimum values: *extent = 'max – min'*.

A commonly used chart is the **heatmap**, which is a chart used to **visualize complex data in a tabular format**. In the following example, we show how many contributions to Github are made each day of the week, and each hour range. The data can be obtained 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(title='Contributions')
)
```

And the result will be the following heatmap:

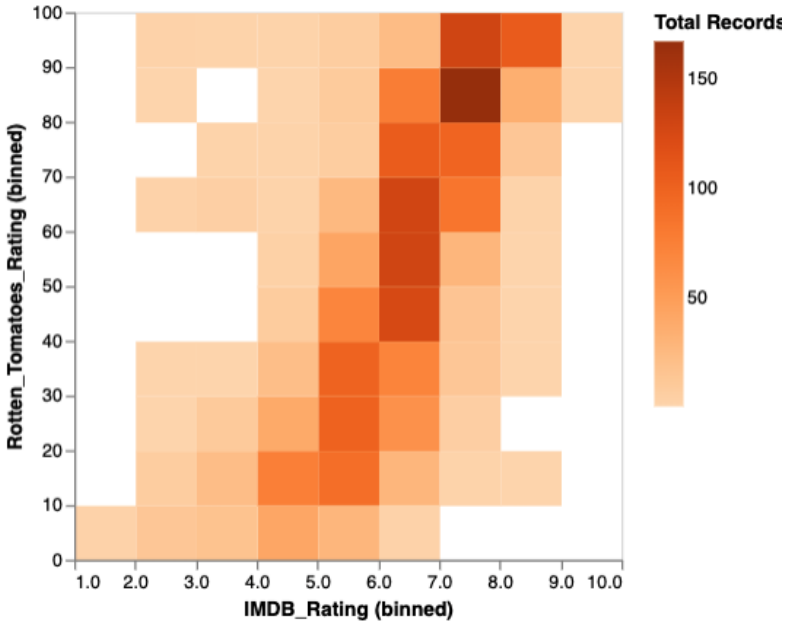


Note that we changed the legend title in the *color* field to ensure that the name makes sense. Otherwise, it would have read "Sum of count".

The next example shows the ratings of several movies in 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(scheme = 'oranges')
        .legend(title = 'Total Records')
)
```

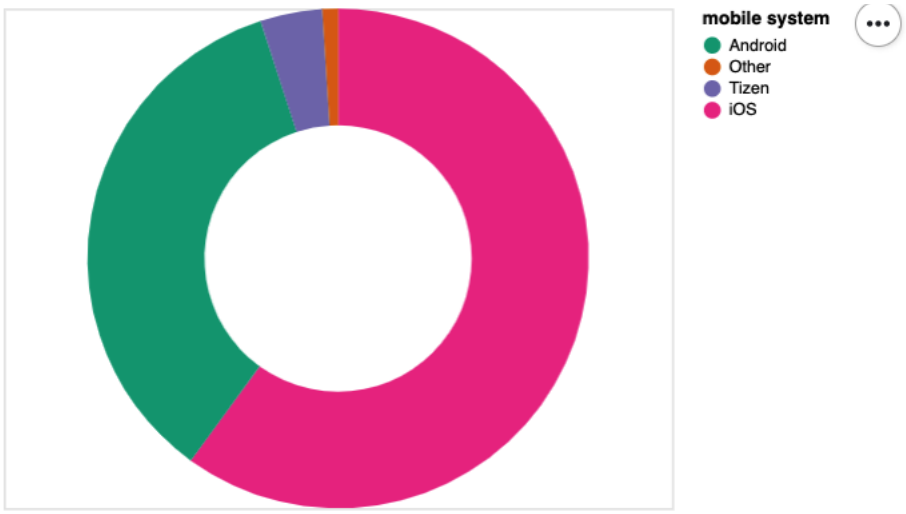
And the result is:



Donut charts are a modified version of pie charts where the pie has a hole. Therefore, they have the shape of a torus or a donut. They can be encoded using the arc mark, with the theta angle as usual, but defining an inner radius as a configuration parameter. An example follows:

```
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 one (dark2). The different color schemes that can be used are described in Vega's page: https://vega.github.io/vega/docs/schemes/.



Radial charts are a specific version of pie charts where quantities are encoded in the radius, instead of the angle. They can be implemented by using the theta and theta2 parameters, for the beginning and end of each arc, plus the radius parameter for the length. The following example shows how:

```
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]})

baseChart = 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"),
)
baseChart
```

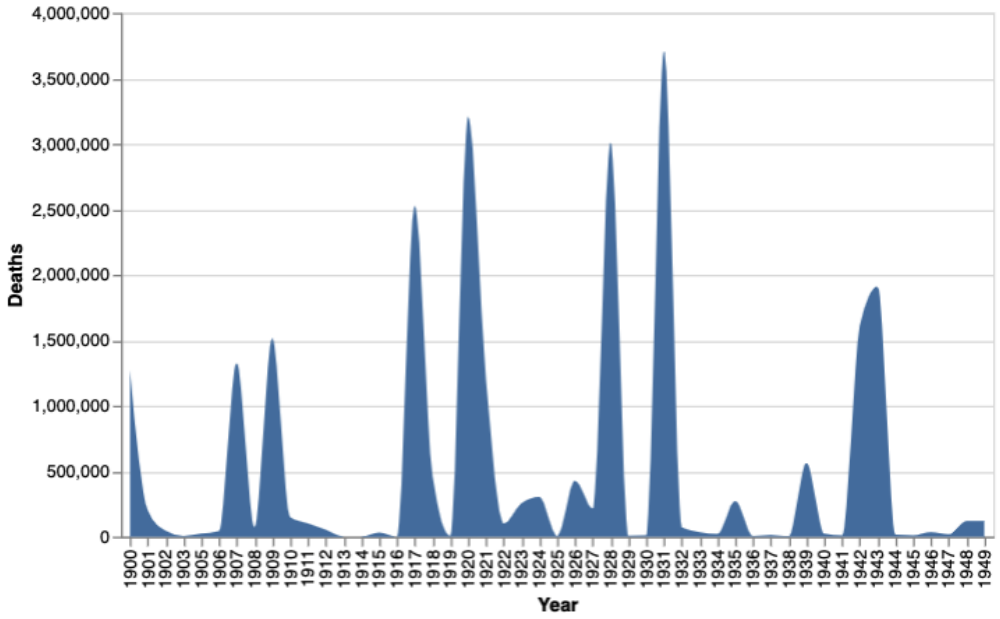
Note how, in this case, the values of the angles are contained in the dataframe. We could do it in other ways. The result is here:

Horizon graphs are a modified version of area charts with the goal of **reducing the vertical space required for the plot**. They use a combination of clipping, overlaying, and transparency to achieve the effect. They may be very useful, but are otherwise more difficult to read, and users typically require a certain amount of training.

Let's imagine that we have a plot such as the following one:

```
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)
```

Where we plot the deaths caused by natural disasters from the dataset *disasters* before 1950.



We can see that the data has very high values together with very low values, and thus, it requires a high amount of vertical space. In order to save space, horizon graphs use the following strategy. Horizon graphs work as follows: The Y axis is clipped by the center, and the data in the top part is overlaid to the bottom as a second chart, changing the coding to denote that the quantities are higher than the ones depicted in the background. This process is repeated several times, with each process saving half of the vertical space, at the cost of more overlay.

We can generate horizon graphs in Altair using the following technique. We plot the original chart, but clipped to the 2M casualties. Then, we overlay a second chart with the original data modified (subtracting the 2M amount) and with a different color.

This is what we did in the following example:

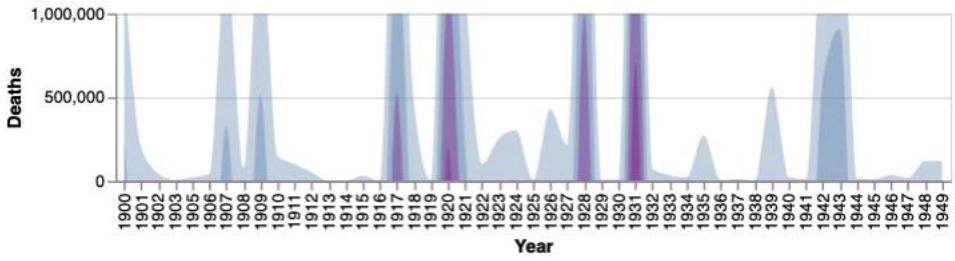
```
df = data.disasters()

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

The process described for the creation of the horizon chart can be iterated several times. In the following example, we have iterated 3 times, and we changed the colors for the third and fourth iteration:

```
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
```

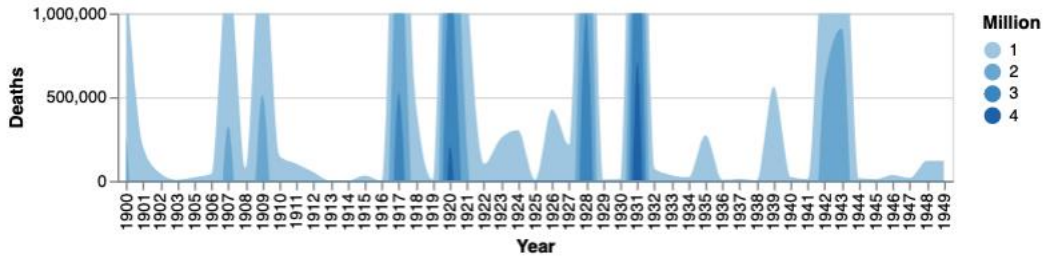
We also reduced the vertical size of the chart to 100 units, and the result is:



We can further customize the chart to make it more understandable. In the following example, we have created a new variable so that we can automatically assign the color according to a quantitative palette, and we changed the names of some calculated variables to make them easier to interpret:

```
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:

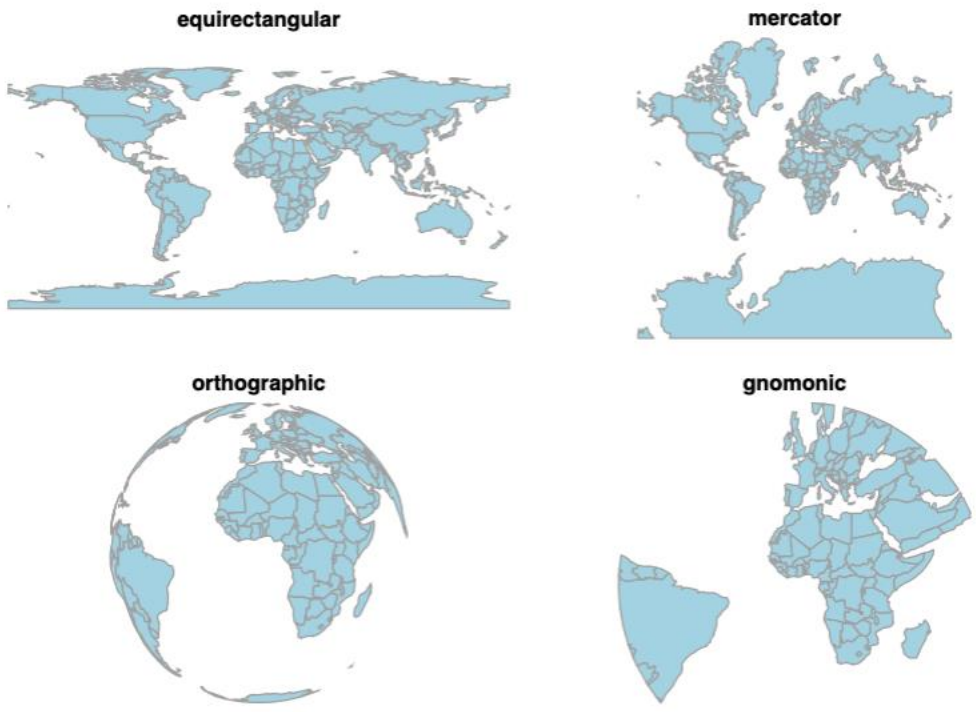


Maps are used to visualize **all kinds of information that is linked to geographic positions**, from routes, to flows, or demographic or economic datasets, the use of maps is widespread. With Altair, they can be generated using the *mark_geoshape* type. However, its usage is more cumbersome than other marks. First, we need to get a dataset that encodes the geometry of the regions to plot. This is typically obtained using a *geojson* file. Then, we need to connect the dataset with the data to represent, since the captured data commonly only stores country names (or any other geographic entity, such as a province, a city, and so on…). As a result, in order to combine this information, we will often require a *lookup* operation, that will be dealt with later.

The initial examples, only show data that is encoded in the same geographic file. In the following example, we can see 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', 'mercator', 'orthographic', 'gnomonic']
charts = [base.project(proj).properties(title=proj)
          for proj in projections]

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

The result would be:



In the following example, we plot the US map, and overlay a circular mark that shows the number of airports per state. In this case, we will have two sources of

information, but these will not be connected. The strategy is the following one: we draw the map using the *mark_geoshape*. Then, the airport positions are encoded as longitude and latitude in another chart. Then, both charts are overlaid.

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')
```

In order to add the airports information, we gather the data from the *airports* dataset, and calculate their average longitude and latitude by grouping per state (all these information bits are stored in the *airports* dataset). By calculating the aggregation, we can count, and by calculating the average in longitude and latitude of their positions, we can generate a 2D point in the map to represent those quantities in size-graded circles.

```
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 would be:

Note that we are using a particular projection, *Albers*, which is a conic, equal area map projection which has as feature that, though scale and shape are not preserved, distortion is minimal between the standard parallels. It is the projection used in some institutions such as the US Census Bureau. The concrete version of the Albers projection used here only contains the US.

Moreover, take into account that Altair has mapped the longitude and latitude that are given in the second chart to the proper positions in the map, even if Alaska is drawn in a different position than its real situation with the AlbersUSA projection. If we use the Mercator projection, we would get the positions right (e.g. now Puerto Rico appears where it should). However, this projection is intended to show the whole world, and our geographic data only contains the US states. Thus, most of the map does not show any countries.

Finally, another remark is that plotting first the points and the map, renders a wrong configuration.

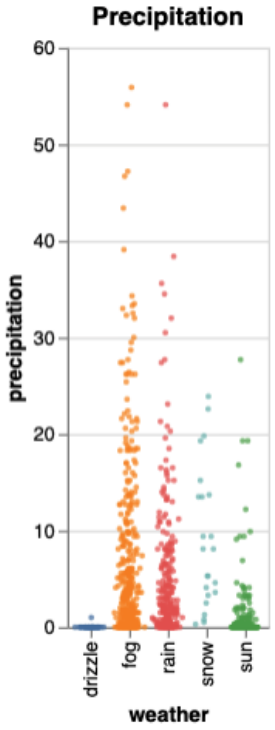
Some more advanced maps are shown in the section devoted to data transformations, more concretely, using the method *transform_lookup*.

You can also create strip charts by using the offset parameters.

The following example shows the different precipitation amounts in Seattle, for the different types of weather conditions:

```
source = data.seattle_weather()



alt.Chart(source, title='Precipitation').mark_circle(size=8).encode(
    alt.X("weather:N"),
    alt.Y("precipitation:Q"),
    xOffset="jitter:Q",
    color=alt.Color('weather:N').legend(None)
).transform_calculate(
    # Generate Gaussian jitter with a Box-Muller transform
    jitter="sqrt(-2*log(random()))*cos(2*PI*random())"
)
```

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



Many other examples of simple and compound examples can be found in the webpage.

8. Data transformations

8.1 BASICS

Very commonly, we need to transform the input data to generate visualizations. The most obvious way is to transform the data using Pandas data transformations. This will give you the highest flexibility and power to manipulate input data.

However, Altair is able to load data from *json* files or a *csv* file, or an *url*. In those cases, the modification with Pandas is less suitable.

Altair allows to specify data transformations within the chart specification itself, by providing a set of *transform_** functions. Some of the most relevant are:

- *transform_aggregate* creates a new data column by aggregating an existing column.
- *transform_bin* creates a new data column by binning an existing column.
- *transform_calculate* creates a new column by using arithmetic expressions on an existing column.

- *transform_filter* selects a subset of the input data.
- *transform_fold* to convert wide-form data to long-form data directly.
- *transform_lookup* performs a one-sided join of two datasets based on a lookup key.
- *transform_pivot* transforms long-form to wide-form data
- *transform_stack* will let you compute values related to stacked versions of encodings
- *transform_timeunit* discretizes a date by a time unit (day, month, year, etc.)

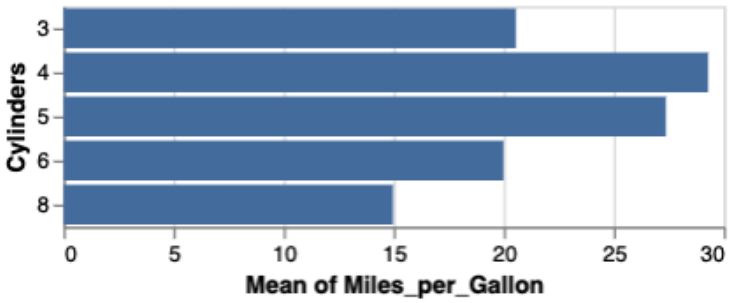
8.2 AGGREGATE TRANSFORMS

We have already seen some aggregate transforms. Altair provides two different ways of calculating aggregations: within the encoding itself (choosing an encoding based on a certain value) or using an aggregate transform.

The aggregate property of a field definition can be used to compute aggregate summary statistics (e.g. median, min, max) over groups of data. Whenever at least one field in the specified encoding channels contains an aggregate function, the resulting visualization will show aggregate data. In this case, all fields without aggregation function specified are treated as group-by fields in the aggregation process.

We have already seen this in previous examples. If we want to render the cars dataset and show the average displacement of cars of different number of cylinders, we can do it simply with an aggregation using the *mean* function on the second field:

```
cars = data.cars.url

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



This code is equivalent to expressing the x field as:

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

The *transform_** functions let us calculate new columns with certain operations. In this case, *transform_aggregate* could be used to compute the same result, by calculating a new column with the aggregated values. It can be achieved this way:

```
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 function *transform_aggregate* can have three options: the output field name to use for each aggregated field, the field to aggregate and the operation to perform. There is a large number of operations including: count, sum, mean, variance, stdev, median, q1, q3, ci0, ci1, min, max.

8.3 BIN TRANSFORMS

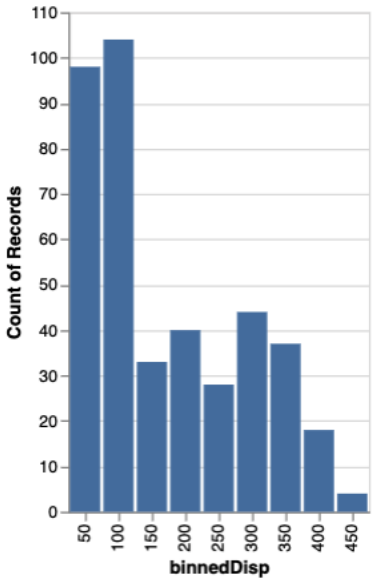
Like in the previous case, we can create bin transforms through the encoding and by explicitly defining it. So, the following code:

```
import altair as alt
from vega_datasets import data

cars = data.cars()

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

Is equivalent to:

```
alt.Chart(cars).mark_bar().encode(
    x='binnedDisp:O',
    y='count()'
).transform_bin(
    'binnedDisp', field = 'Displacement'
)
```

And both of them result in the following chart:

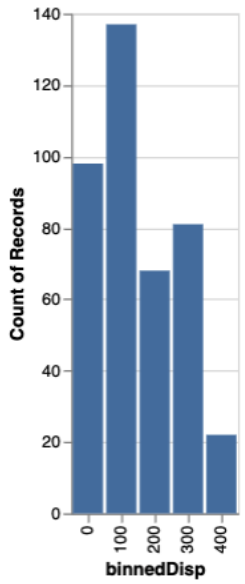


With the only difference being the name of the X axis (that can be changed in any case).

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

```
cars = data.cars.url

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

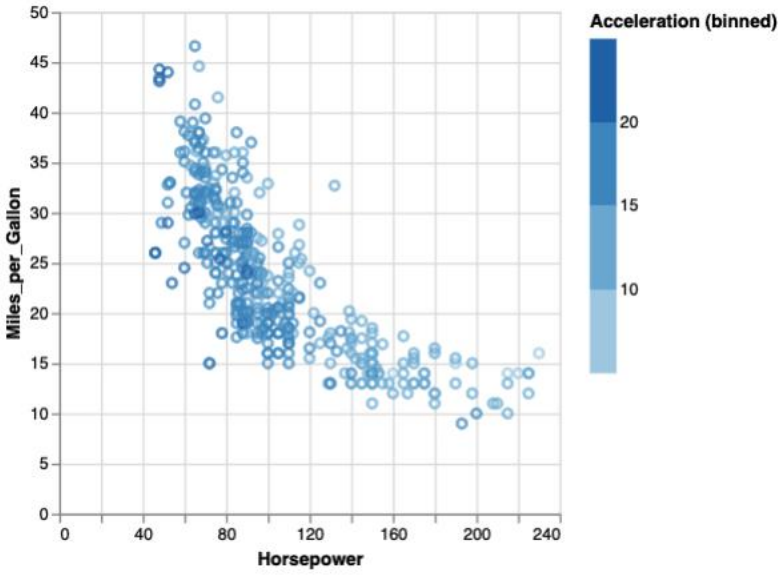
Which would render:



But we can also 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))
)
```

That would render:

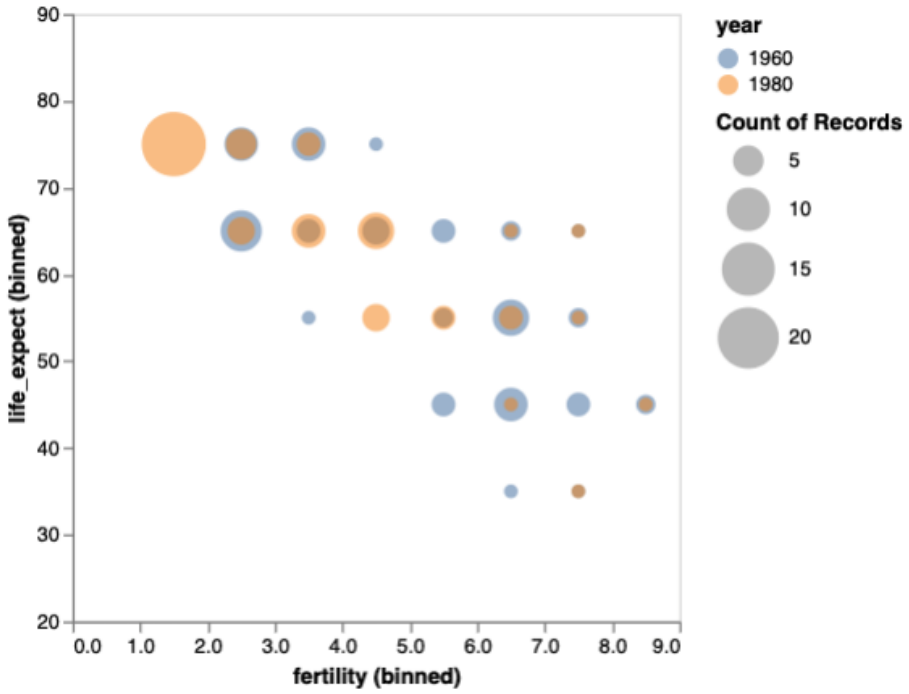


The advantage of the top-level transform is that the same named field can be used in multiple places in the chart if desired. Note the slight difference in binning behavior between the encoding-based binnings (which preserve the range of the bins) and the transform-based binnings (which collapse each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

An example of bin transforms that create a specific version of plot are **binned scatterplots**. **Binned scatterplots** can be used to simplify the look of a scatterplot and analyze the density. They can be used to **show the non-parametric relationship between two variables**. Creating binned scatterplots in Altair is as simple as creating a scatterplot and adding the option *bin* as true for the variables we want to aggregate. The following example compares the fertility rate and life expectancy for all the countries in the *gapminder* dataset for years 1960 and 1980. Again, we use selection here, 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:



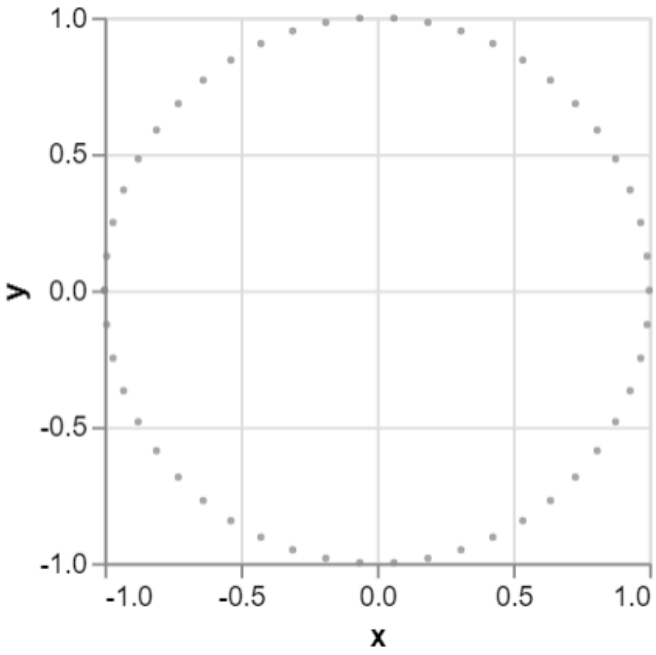
8.4 TRANSFORMING DATA THROUGH CALCULATIONS

We can create new data from the input by using some arithmetic calculations. This can be achieved with the function *transform_calculate*. This allows us to create a new column (or more than one) with the arithmetic operations we determine. For example, if we want to draw a set of circles in a circular layout, we can calculate the circle positions by using the sine and cosine functions:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'angle': range(51)})

alt.Chart(data).mark_circle(size=7, color = 'gray').encode(
    x='x:Q',
    y='y:Q',
    order='angle:Q',

).transform_calculate(
    x='cos(PI*(datum.angle -25 ) / (25.) )',
    y='sin(PI*(datum.angle -25 ) / (25.) )'
).properties(width = 200, height = 200)
```

Where x and y are the names of the new fields that were added, and their values depend on the sine and cosine of variable angle, that was created as an array from 0 to 100.

The result will be:



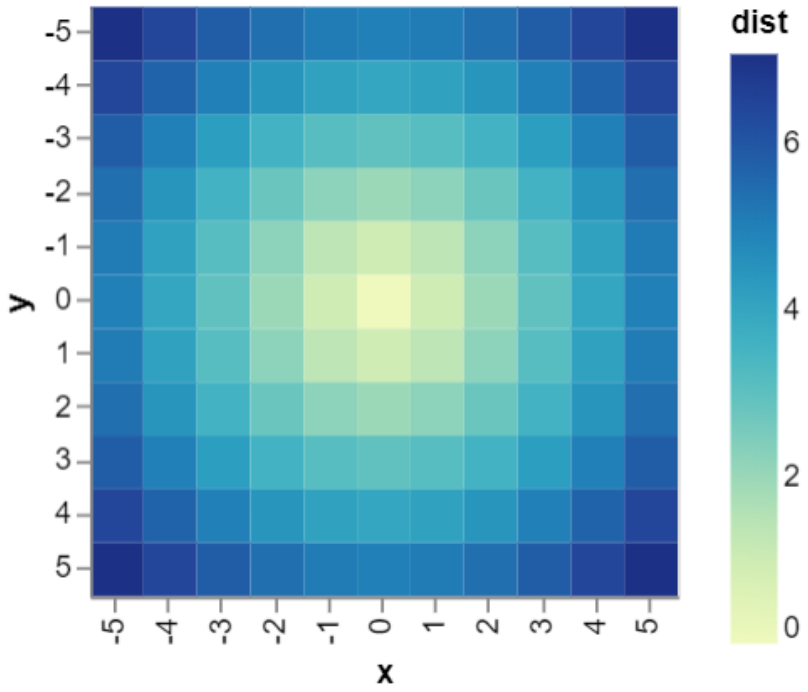
Take into account that the sizes of the plot are decided by Altair, so if we do not impose a square shape, in this case, the chart appears deformed.

To calculate new data, Altair uses expressions that can get as input the current data set. The data can be referred to with the name *datum*.

In the following example, we are going to create a table with elements from (-5, -5) to (5, 5). These values are the coordinates of the pixels, and we will plot a heatmap based on those data. We will use *transform_calculate* to compute the distances of the pixels to the center. Therefore, we will perform two different jobs: First, the distances from each pixel to the center are calculated by deriving a new column that contains the Euclidean distance of the pixels to the point (0, 0), and then a heatmap is plotted that takes these new calculated values to encode the color:

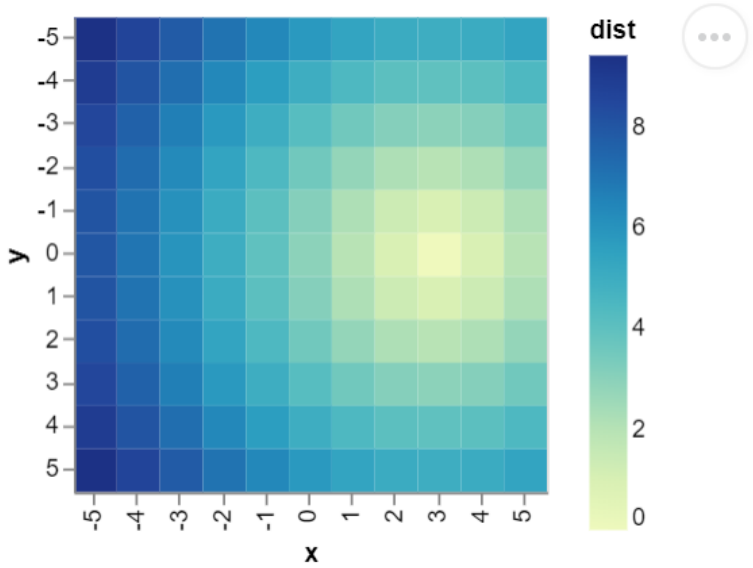
```
import altair as alt
import numpy as np
import pandas as pd

# Compute a 2D grid with the distances to the center
x, y = np.meshgrid(range(-5, 6), range(-5, 6))

# Convert this grid to columnar data expected by Altair
derived = pd.DataFrame({'x': x.ravel(),
                        'y': y.ravel()})

alt.Chart(derived).mark_rect().encode(
    x='x:O',
    y='y:O',
    color = 'dist:Q',
).transform_calculate(
    dist='sqrt(datum.x*datum.x + datum.y*datum.y)',
).properties(width = 200, height = 200)
```

The result is:



We can further work upon this function. For instance, we could calculate new columns and use them in the generation of the *dist* value. Let's imagine that we want to shift the center 3 pixels to the right. We can do it by generating a new column (x2) calculated as subtracting 3 to the values of x – 3. This column will be called x2. Then, we use it in the *sqrt* calculation (with the name *datum.xShift*). The code and result appears below:

```
alt.Chart(derived).mark_rect().encode(
    x='x:O',
    y='y:O',
    color = 'dist:Q',
).transform_calculate(
    xShift = 'datum.x-3',
    dist='sqrt(datum.xShift*datum.xShift + datum.y*datum.y)',
).properties(width = 200, height = 200)
```



8.5 TIME MANIPULATIONS

There are different ways to aggregate information based on time units. Whenever we have temporal data, we can aggregate it using time units, that work as functions over a temporal field. Some of the relevant functions are:

- *year*, *yearmonth*, *yearmonthdate*, which will aggregate per year, per year and month, or per year, month and day of the month. This means that aggregation operations will happen over the data that shares the above mentioned properties.
- *month*, *monthdate*: will aggregate data from the same month or the same month and day.
- *date*: will calculate aggregate functions based on the day of month (i.e., 1 - 31).
- *day*: takes into account the day of week.

There are other units that take into account seconds or minutes. Just check the documentation to see which values for the *timeUnit* function are available

(https://altair-viz.github.io/user_guide/transform.html#timeunit-transform). Take into account that we may be aggregating data that makes no sense. For example, we may aggregate temporal data from different years. It is up to you to see whether this makes sense or not.

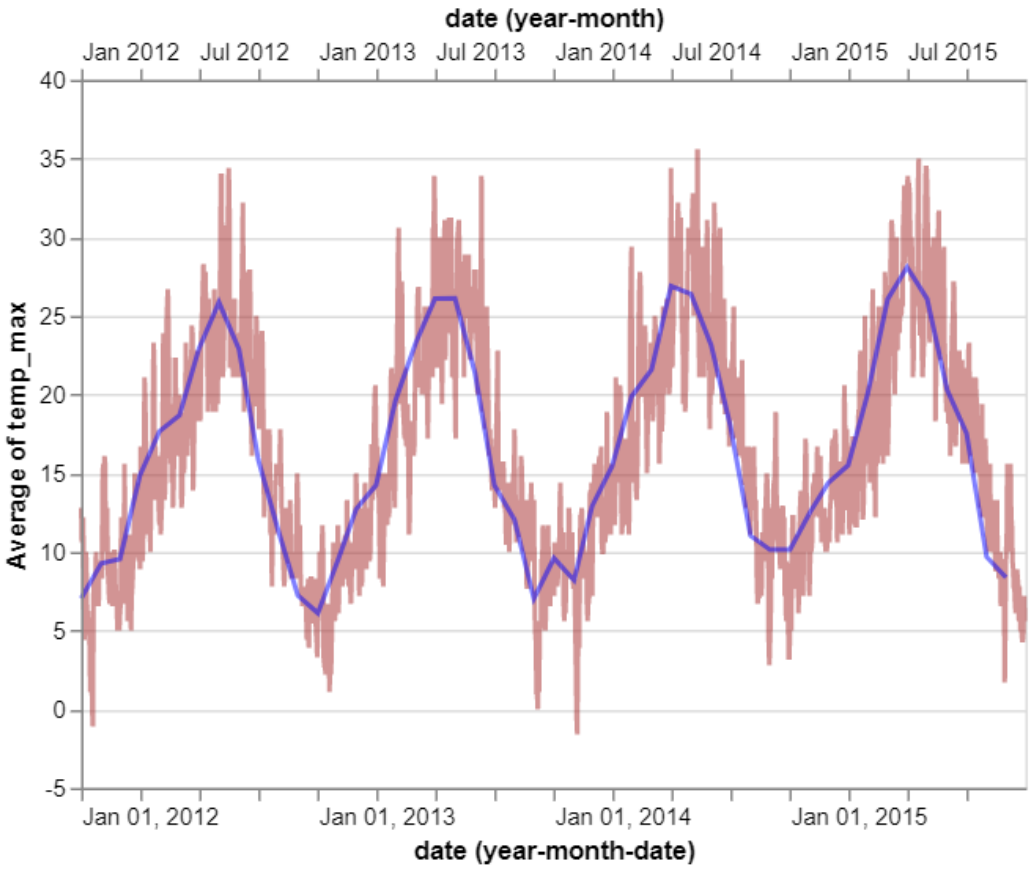
The following example uses the *weather.csv* dataset from the Vega datasets, uploaded manually:

```
df = pd.read_csv('weather.csv')

ch1 = alt.Chart(df).mark_line(
    opacity = 0.5, stroke='brown').encode(
        x=alt.X('yearmonthdate(date):T', axis = alt.Axis(labelAlign='left')),
        y='average(temp_max):Q',
).transform_filter(alt.datum.location=='Seattle')


ch2 = alt.Chart(df).mark_line(
    opacity = 0.5,stroke = 'blue').encode(
        x=alt.X('yearmonth(date):T', axis = alt.Axis(labelAlign='left')),
        y='average(temp_max):Q',
).transform_filter(alt.datum.location=='Seattle')


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

The plot shows two different encodings of the dataset, where in one case we average per month, and the other, per day in the month.

To ensure we are calculating the data properly, the axes are independent, so you can see the range of time that is used for the plots.

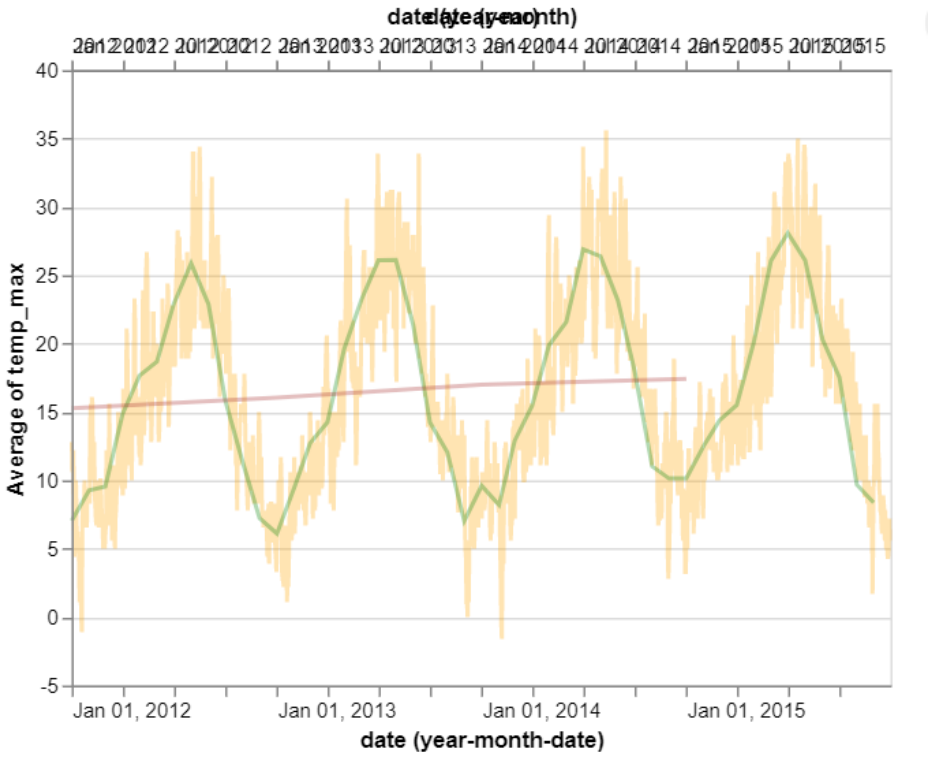
The result is shown next:



We can also show how aggregating per year would result. In this case, we plot the three charts. Note that if we ask Altair to resolve the three axis as independent, two of them are overlaid:

```
ch1 = alt.Chart(df).mark_line(opacity = 0.3, stroke='orange').encode(
    x=alt.X('yearmonthdate(date):T', axis = alt.Axis(labelAlign='left')),
    y='average(temp_max):Q',
).transform_filter(alt.datum.location=='Seattle')


ch2 = alt.Chart(df).mark_line(opacity = 0.3,stroke = 'green').encode(
    x=alt.X('yearmonth(date):T', axis = alt.Axis(labelAlign='left')),
    y='average(temp_max):Q',
).transform_filter(alt.datum.location=='Seattle')


ch3 = alt.Chart(df).mark_line(opacity = 0.3,stroke = 'brown').encode(
    x=alt.X('year(date):T', axis = alt.Axis(labelAlign='left')),
    y='average(temp_max):Q',
).transform_filter(alt.datum.location=='Seattle')

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

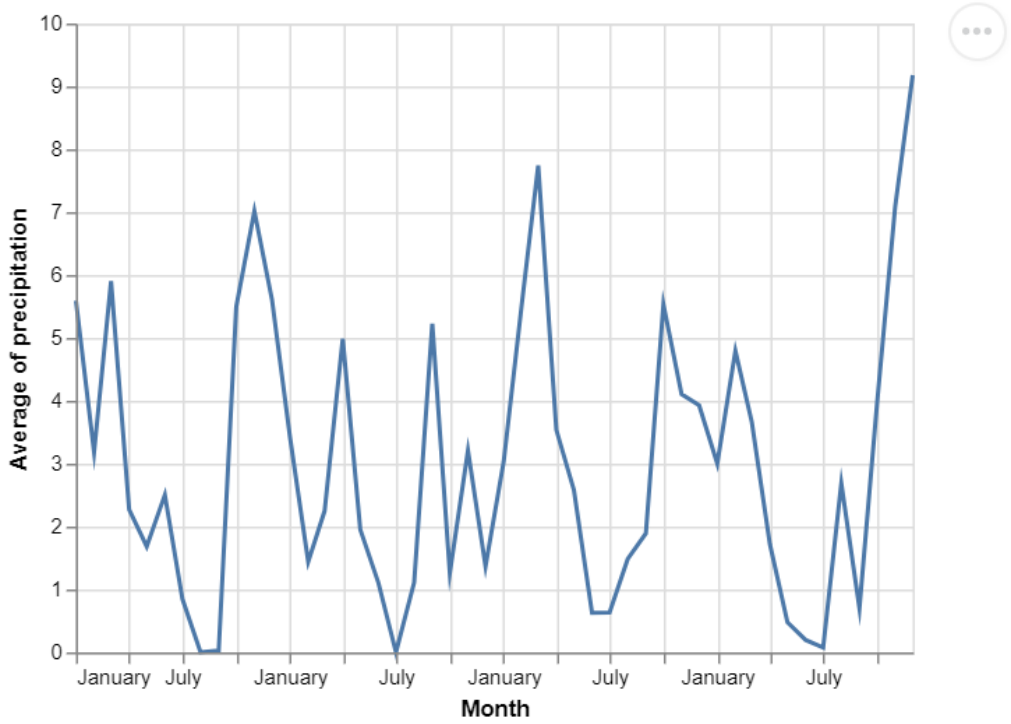
The result, with the two top axis on top of each other is:



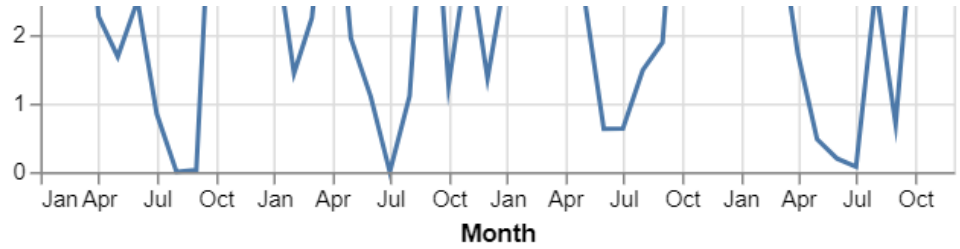
Time unit transformations can also be carried out using the transform_timeunit function. This can be used to assign a new name to the calculated field so that it can be reused.

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

```
alt.Chart(df).mark_line().encode(
    alt.X('monthY:T', title = 'Month', axis=alt.Axis(format="%B")),
    y = 'average(precipitation):Q',
).transform_timeunit(monthY = 'yearmonth(date)')
```



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



There are several parameters you can play in order to define the way labels appear in the axes. You can rotate them, you can offset the initial or last value (which typically are padded so that they appear in the chart), you can align the labels (always referring to the ticks) and you can further play with the format string (e.g. adding white spaces) if you need to.

The format string also has different forms, that are actually borrowed from D3 because the final code will be converted to D3 code. Some relevant values can be:

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 - the 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 event ask the axis to plot the week of the year (either in Monday-based week or Sunday-based week, although the system (neither Altair nor Vega) can generate those values, i.e. you cannot (for the moment) aggregate by number of week. So some of those formats may be misguiding if you use them improperly.

8.6 FILTER TRANSFORMATION

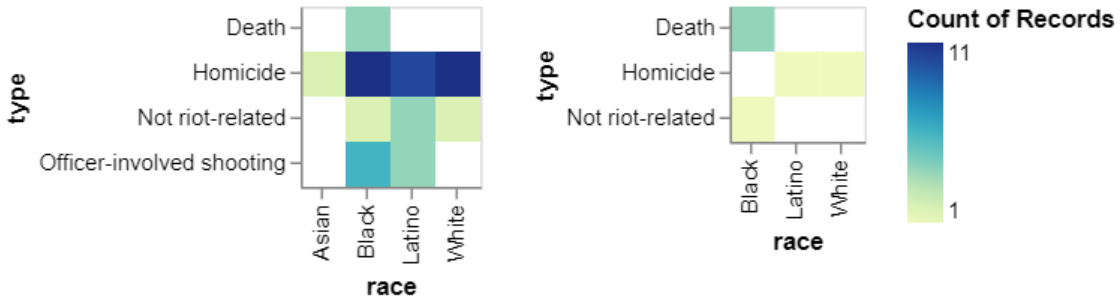
This transformation eliminates part of the data following a specified criterion. In order to specify the criterion, we can use the *datum* object, that refers to the

input dataset. Each of the fields of the dataset can be specified using *datum.<fieldname>*.

For example, if we want to plot the events stored in the *la_riots* dataset to compare the origin of the authors and we want to see whether they were a male or female, we can do the following:

```
import altair as alt
from vega_datasets import data

df = data.la_riots()


ch1 = alt.Chart(df).mark_rect().encode(
    x='race:N',
    y='type:O',
    color = 'count():Q',
)

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

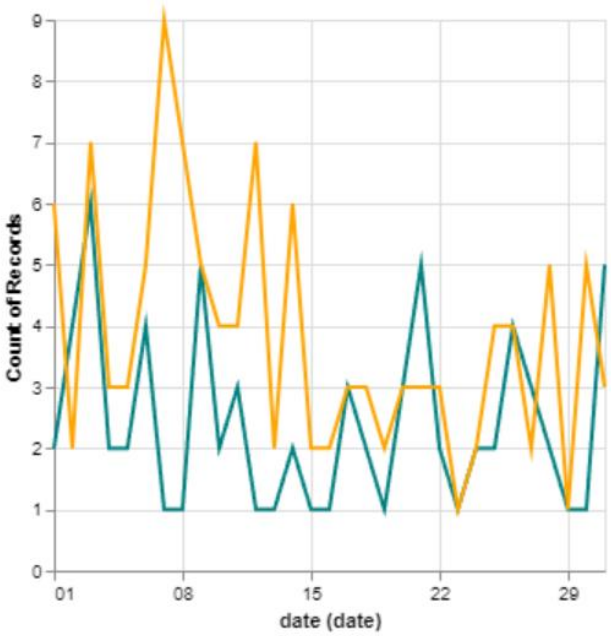
This way, we are filtering the gender and the first plot will show the data corresponding to males, and the second to females.



Another example shows a line chart where we want to compare the number of flights each day of the month that have as origin Los Angeles (LAX) or Houston (HOU). From the dataset, named *flights_2k*, we only select those of the mentioned 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 would be:

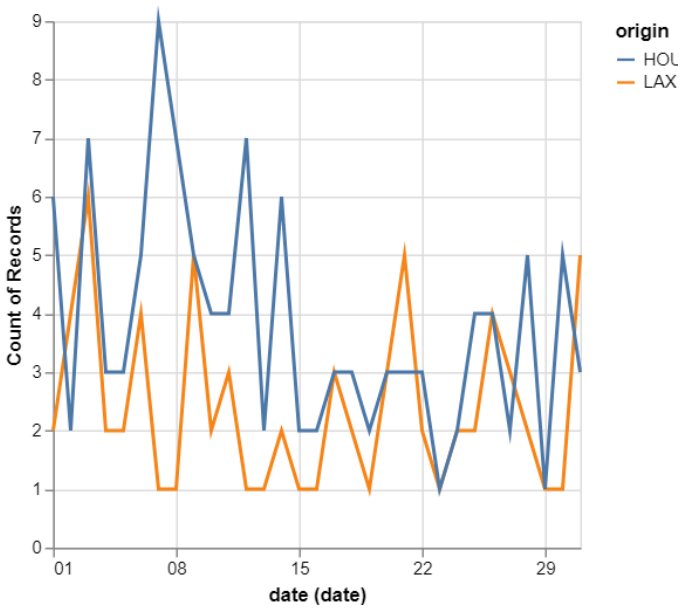


Since each chart plots a single line, and the color is defined at top level configuration, the result is that we have no legend. If we want a legend, we can do a small trick, we can ask Altair to color code the datasets based on the

origin, even though we have explicitly defined it in the filter. This would result in 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 would be:



Note that the legend now has a title. We can do all sorts of tricks using this approach, but if we want to do the color encoding based on a parameter whose name does not mean anything, we can delete the title of the legend by configuring it as *title* = ''. Besides, now the colors have been automatically selected, and are not the ones of the original chart. We can change this by defining a range of valid colors for the values of the selection, like this:

```
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)
```

But 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

We can consider data lookup as a simple example of data aggregation, but it has its own characteristics. Looking up data is useful when we have the information we want to plot divided into different datasets.

There are several ways to solve this:

- Merging the data
- Looking up for information in another table

Whenever possible, merging the data is typically easier, since several Python libraries at our disposal provide ways to manipulate data efficiently. We will present two different examples, one that includes the *lookup* feature, and another that combines the *lookup* and the data merging using pandas.

We already saw previously that, for geographic data plots, we need the data in some particular format, such as a *geojson* file.

In this example, we are going to render a choropleth map of the US with the employment rate per county, as stored in the *unemployment* dataset.

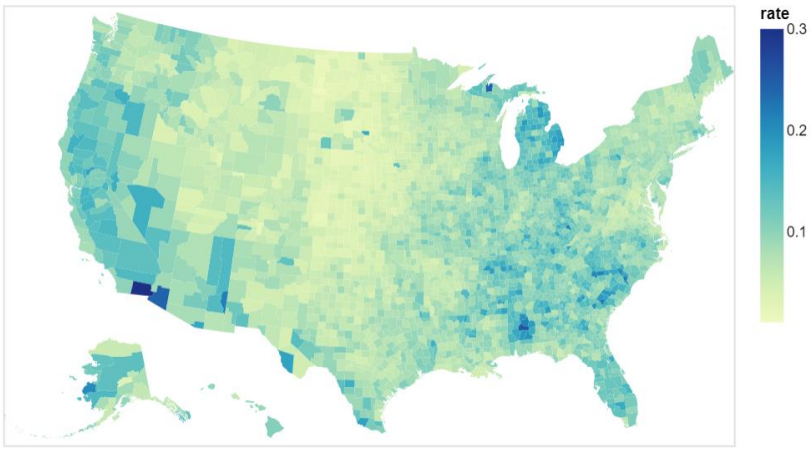
First, we will load the US map representation from the *us_10m* dataset, and then, we will look up for the employment rate in the *unemployment* file by performing a *transform_lookup* operation. This operation has two parameters, the data origin, and the search function, which are stated as two different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

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

The code is as follows:

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

alt.Chart(counties).mark_geoshape().encode(
    color='rate:Q'
).transform_lookup(
    lookup='id',
    from_=alt.LookupData(unemp_data, 'id', ['rate'])
).properties(
    projection={'type': 'albersUsa'},
    width=500, height=300
)
```

The result is the following choropleth map:



In this other example, we face a more complicated problem. We want to plot the population of the different countries using the data of *gapminder_health_income*. However, this file does contain country names, while the file *world_110m* does have country *id*s. What we are going to do is to find a connection file (*world_110m_country_codes.json*) and merge this with the *gapminder_health_income* file. Then, we are going to plot a map, using *world_100m* dataset, and lookup the population values from the merged file.

First, we need to upload the file that connects the country codes with their names to the system:

```
from google.colab import files

uploaded = files.upload()
```

Then, we load the uploaded files:

```
corresp = pd.read_json(
    io.StringIO(uploaded['world_110m_country_codes.json'].decode('utf-8')))
df = data.gapminder_health_income()
```

And perform the merging operation. Note that we take the *gapminder* dataset on the left, and the country codes on the right:

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

print(merged)
```

If we print the merged data, we can see that there is something strange with the ids:

```
                  country  income  health  ...  code     id                  r
0             Afghanistan    1925   57.63  ...    AF    4.0          Afghanis
1                 Albania   10620   76.00  ...    AL    8.0              Alba
2                 Algeria   13434   76.50  ...    DZ   12.0              Alge
3                 Andorra   46577   84.10  ...   NaN    NaN
4                  Angola    7615   61.00  ...    AO   24.0               Ang
5      Antigua and Barbuda   21049   75.20  ...   NaN    NaN
6               Argentina   17344   76.20  ...    AR   32.0            Argent
```

They are floating point values with some *NaN* values. Since the *ids* in the geographic data are integers, we need to transform them to integers. This can be done again using Pandas Dataframe:

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

Note that we first change the *NaN* values to -1. This way, we ensure that changing the type to integer does not raise an error.

Once we have done this, we end up with two Dataframes, one with the geographic data (loaded as it is shown next), and another one with the population data which contains country names and *ids*.

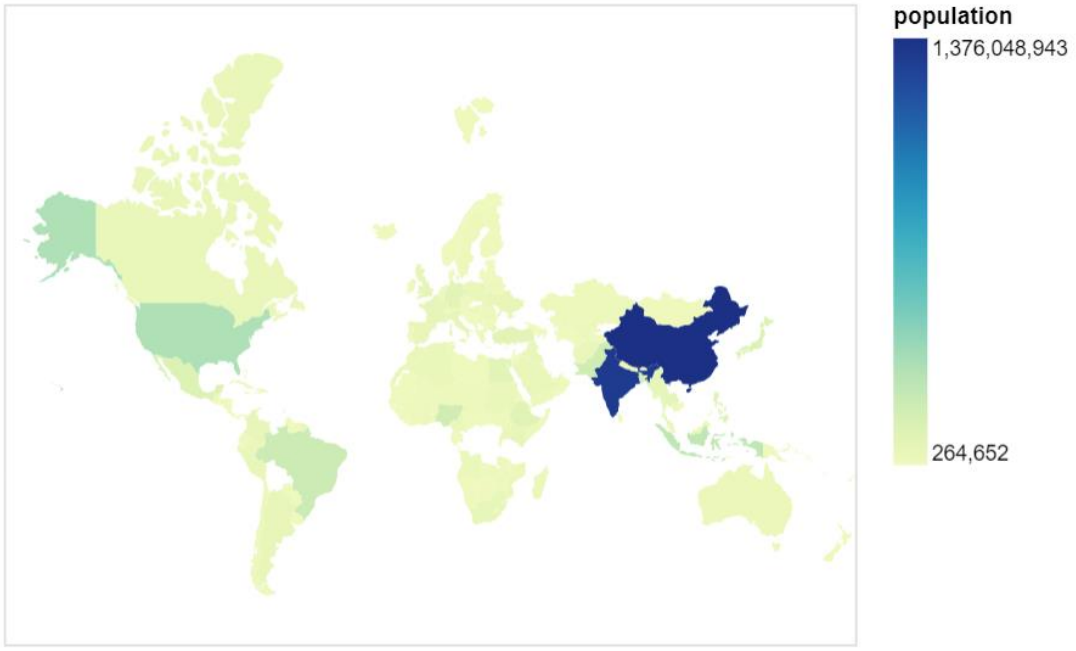
To load the geographic data, we only need to call the proper function:

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

Now, we can proceed to render the population in the map using the *transform_lookup* feature. The implementation is as follows:

```
alt.Chart(geom).mark_geoshape().encode(
    color='population:Q',
).transform_lookup(
        lookup='id',
    from_=alt.LookupData(data=merged, key='id', fields=['population'])
).project(type='mercator')
```

Note that we use the *mark_geoshape* mark type, and that we encode the population in the color option, by querying over the *population* field. The result would be:



Note that some of the countries appear as white, notably Russia and some countries in the center of Africa and South America. This is probably caused by the country codes not appearing properly in the files. Don't forget that we had some errors in the merged file.

8.8 REGRESSION TRANSFORM

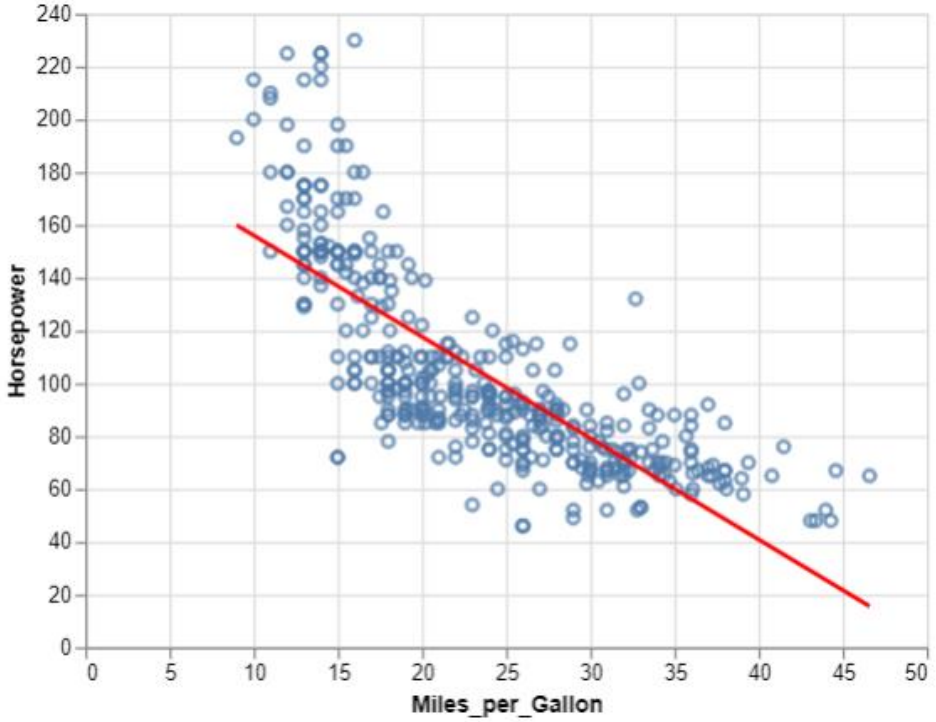
The regression transform may fit a two-dimensional regression model to smooth and predict data. The transform can fit multiple models for input data and generate new data objects that represent points for summary trend lines.

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

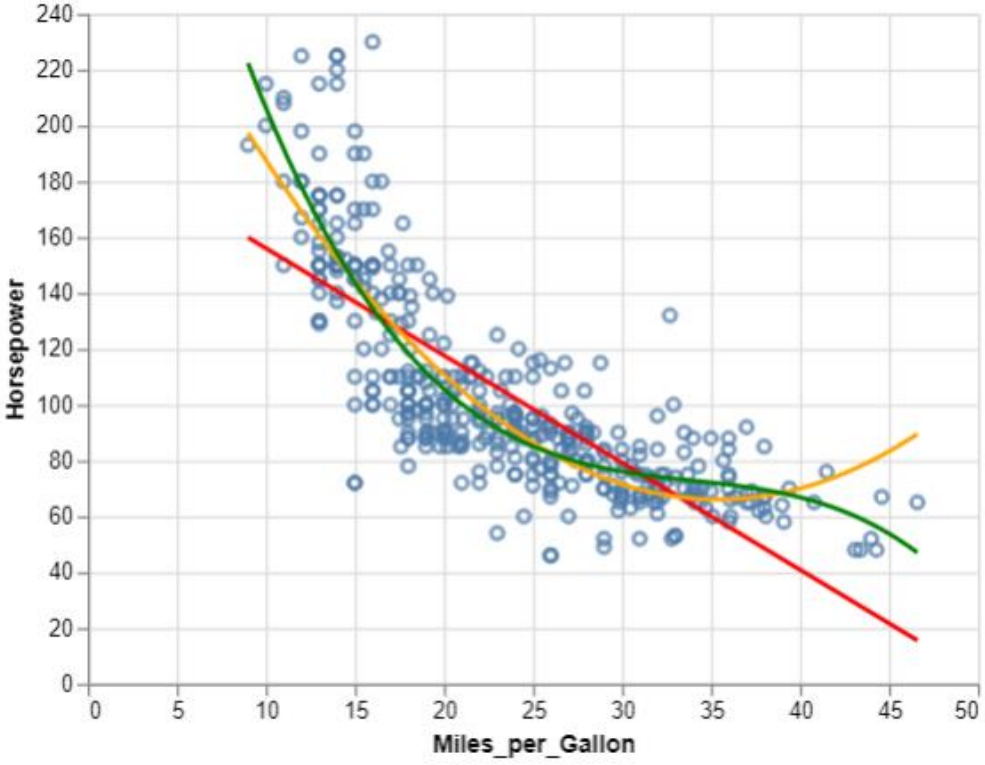
The following example shows a linear regression based on the *cars* dataset:

```
import altair as alt
from vega_datasets import data

cars = data.cars()

ch = alt.Chart(cars).mark_point().encode(
    x='Miles_per_Gallon:Q',
    y='Horsepower:Q',
)


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

ch + linear_regression
```

The result would be:



We can add multiple regression lines of different degrees by changing the fitting function using the *method* parameter in the *transform_regression* function:



9. Tips and Tricks

9.1 LOADING LARGE DATASETS

Altair has a limited number of rows by default that it can load. The maximum number allowed is 5000 rows. This means that some datasets, such as the flights datasets of 10k and 200k rows cannot be loaded.

This limitation is set by default to avoid creating very large examples that might crash the browser.

You can drop the limitation using the following call:

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

However, take into account that this is delicate, and ensure everything is working afterwards.

9.2 ADDING TEXT

Altair has a number of parameters that can be tuned, and this sometimes is quite limited, not allowing you to create the sort of chart you would like.

There are however several workarounds to these limitations, and most of them come from combining several charts onto each other.

A good example is the addition of labels over the chart. You can simply add labels by creating a synthetic dataset that contains coordinates and text labels and plotting them over the desired chart using an overlay and the *mark_text* mark.

In the following example, we want to add a text indicating Min and Max to the average lines of minimum and maximum temperatures in Seattle. We create a custom chart with the texts.

For the first two charts (temperatures), we create two line charts:

```
df = data.seattle_weather()


ch1 = alt.Chart(df).mark_line(color='crimson').encode(
    x=alt.X('month(date):T'),
    y='average(temp_max):Q',
).transform_calculate(
    year='year(datum.date)').transform_filter(alt.datum.year == 2014)

ch2 = alt.Chart(df).mark_line(color='dodgerblue').encode(
    x=alt.X('month(date):T',
            axis = alt.Axis(labels=False, title='', ticks = False)),
    y='average(temp_min):Q',
).transform_calculate(
    year='year(datum.date)').transform_filter(alt.datum.year == 2014)
```

Note that we filter the data to belong to the year, and in order to do this, we extract the year with a *transform_calculate* operation.

Then, we create a third chart with the text and overlay to the others:

```
df2 = pd.DataFrame({'x': [10, 40],
                    'y': [4, 22],
                    'text': ['Min', 'Max']})


ch3 = alt.Chart(df2).mark_text(fontStyle='bold', font='Helvetica').encode(
    x = alt.X('x:Q', scale=alt.Scale(domain=(0,50)),
              axis = alt.Axis(labels=False, title='', ticks = False)),
    y = 'y:Q',
    text = 'text'
)

alt.layer(ch1, ch2, ch3)
```

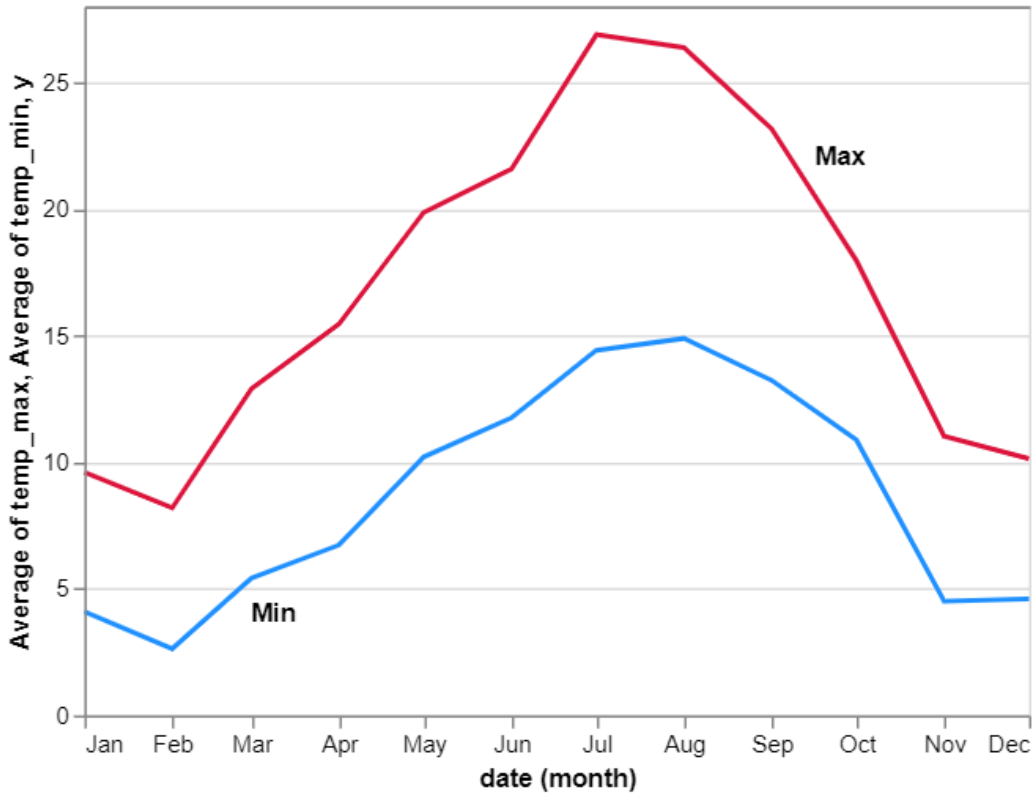
Note the data of the positions is synthetic and takes into account the positions we want to put the labels, as well as the size of the virtual space we will be plotting to.

Since Altair will map the virtual space of the plot to make all of them match the same physical space, we need to displace the marks (texts) from the virtual space and ensure that the virtual space covered is big enough. In order to do this, we customized the axis sizes for the text plot to cover a domain larger than

the positions we have plot the data. In our case, our domain will go from 0 to 50, and the data is plot at (10, 4) and (40, 22).

We also customized the font type and its aspect (bold).

The result is:



9.3 CUSTOMIZING AXES

As already mentioned before, we can define the size of the axes by modifying the field *scale* in the *X* field:

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

Note that we also adjust the axis to make it disappear. Otherwise, the axis corresponding to the plot will also appear.

By setting the *labels* parameter as false, labels will not appear, and by setting the parameter *ticks* as false, will remove the ticks from the axis.

9.4 SAVING CHARTS

We can save the charts in different ways. For saving the chart as an image format, the following calls are allowed:

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

You may need to install certain extensions to generate the files depending on the browser you are working with (you can check the different options and solutions through https://altair-viz.github.io/user_guide/saving_charts.html).

Besides, we can also save the charts in HTML format by calling:

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

We can also save the charts in JSON format by using the same function call and writing a *json* extension to the file:

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

To embed the json file in any web page, we need to use the *vegaEmbed* library (https://github.com/vega/vega-embed).

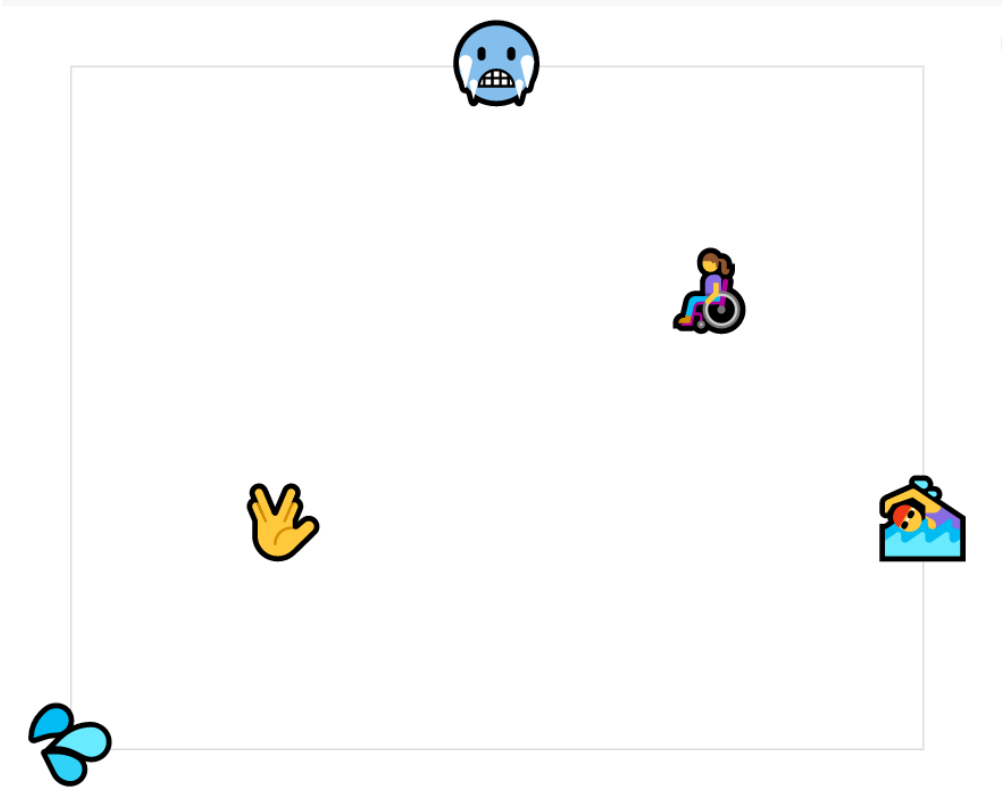
9.5 PLOTTING IMAGES (OR SOMETHING SIMILAR)

The simplest way to plot images is to use emojis.

Vega can plot as text (using *mark_text*) unicode symbols that correspond to emojis. You can get the symbols from https://unicode.org/emoji/charts/full-emoji-list.html, for example by copying (with copy-paste) from the third column in the page (the column named "browser"), and paste in your Colab file. As a result, we can define simple charts where the text will be represented by emojis:

```
import altair as alt
import pandas as pd

source = pd.DataFrame([
      {'x': '0', 'y': '0', 'text': '💨'},
      {'x': '10', 'y': '10', 'text': '🖖'},
      {'x': '20', 'y': '30', 'text': '😭'},
      {'x': '30', 'y': '20', 'text': '🦽'},
      {'x': '40', 'y': '10', 'text': '🏚️'},
    ])


alt.Chart(source).mark_text(size=45, baseline='middle').encode(
    alt.X('x:Q', axis=None),
    alt.Y('y:Q', axis=None),
    text = 'text'
).properties(width=500, height=400)
```

The result would be:

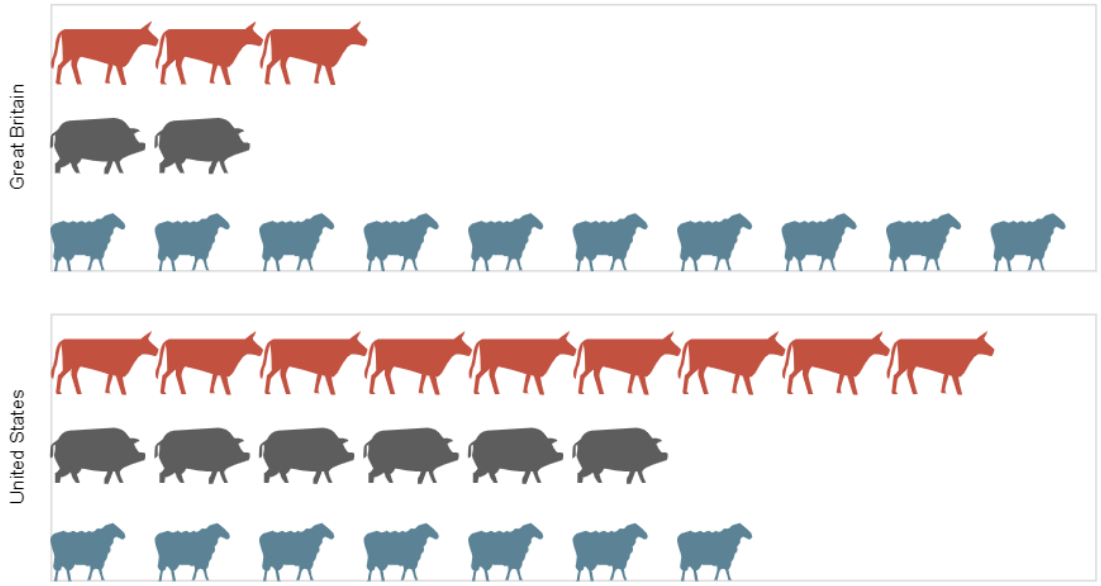


The alternative to using emoji images consists into getting an SVG path for the figure you want to plot and add it to the plot as a shape property.

The following code does so. Note that this is the example in https://altair-viz.github.io/gallery/isotype.html and the data definition has been scrapped for the code. The svg paths are also cut. Refer to the previous link for the full code:

```
domains = ['person', 'cattle', 'pigs', 'sheep']

shape_scale = alt.Scale(
    domain=domains,
    range=[
        'M1.7 -1.7h-0.8c0.3 -0.2 0.6 -0.5 0.6 -0.9c0 -0.6 -0.4 -1 -1 -1c-0.6 0 -1 0.4 -1 1c0 0
        'M4 -2c0 0 0.9 -0.7 1.1 -0.8c0.1 -0.1 -0.1 0.5 -0.3 0.7c-0.2 0.2 1.1 1.1 1.1 1.2c0 0.2
        'M1.2 -2c0 0 0.7 0 1.2 0.5c0.5 0.5 0.4 0.6 0.5 0.6c0.1 0 0.7 0 0.8 0.1c0.1 0 0.2 0.2 0
        'M-4.1 -0.5c0.2 0 0.2 0.2 0.5 0.2c0.3 0 0.3 -0.2 0.5 -0.2c0.2 0 0.2 0.2 0.4 0.2c0.2 0 0
    ]
)

color_scale = alt.Scale(
    domain=domains,
    range=['rgb(162,160,152)', 'rgb(194,81,64)', 'rgb(93,93,93)', 'rgb(91,131,149)']
)

alt.Chart(source).mark_point(filled=True, opacity=1, size=100).encode(
    alt.X('x:O', axis=None),
    alt.Y('animal:O', axis=None),
    alt.Row('country:N', header=alt.Header(title='')),
    alt.Shape('animal:N', legend=None, scale=shape_scale),
    alt.Color('animal:N', legend=None, scale=color_scale),
).transform_window(
    x='rank()',
    groupby=['country', 'animal']
).properties(width=550, height=140)
```

The result is:



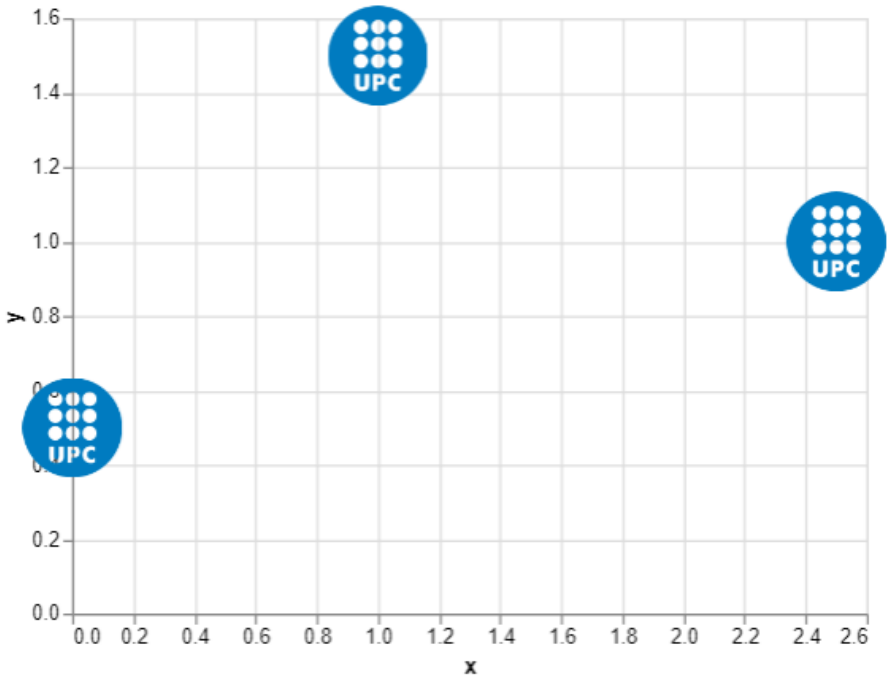
The way Altair (and Vega) are implemented seem to support only simple SVG paths that are not the usual way to create SVGs in most applications. There are some ways you can extract (or fuse) all the paths to a single one from a regular SVG file you can find in the Internet. GIMP provides an import tool that lets you fuse the paths. However, the generated file does not seem to be exempt from problems, and the experiments tried for now seem not successful.

9.6 PLOTTING REAL IMAGES

From version 4.0, altair now supports image plots. It is not necessary to have them as SVGs, which can be quite cumbersome. In the following example we plot some images in 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(
    x='x',
    y='y',
    url='img'
)
```

The result is:



10. Interaction basics

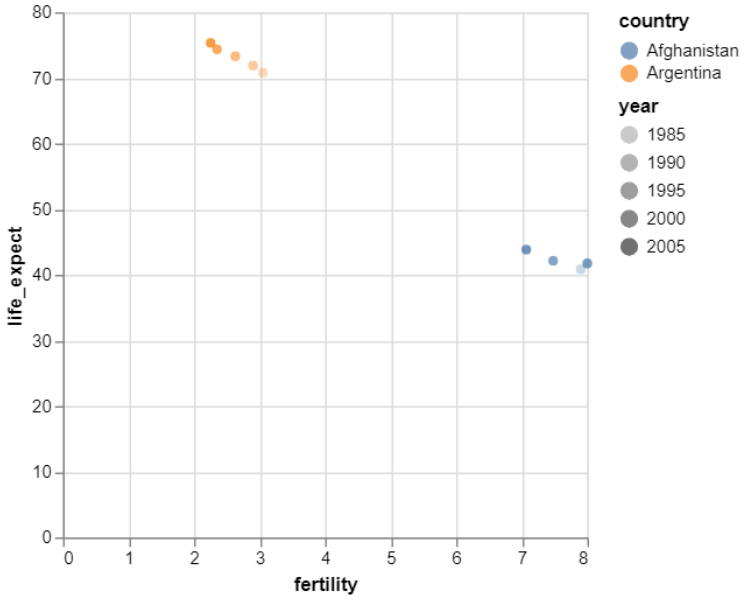
To further allow data exploration, it is necessary to add interaction features to our charts.

10.1 BASIC INTERACTION: PAN AND ZOOM

The first step to make the chart interactive is to determine that it can receive interaction events. This is carried out simply by defining it as *interactive* after the chart definition. For example, the next code generates a scatterplot of fertility rate for two countries, Afghanistan and Argentina, for the several years after 1980.

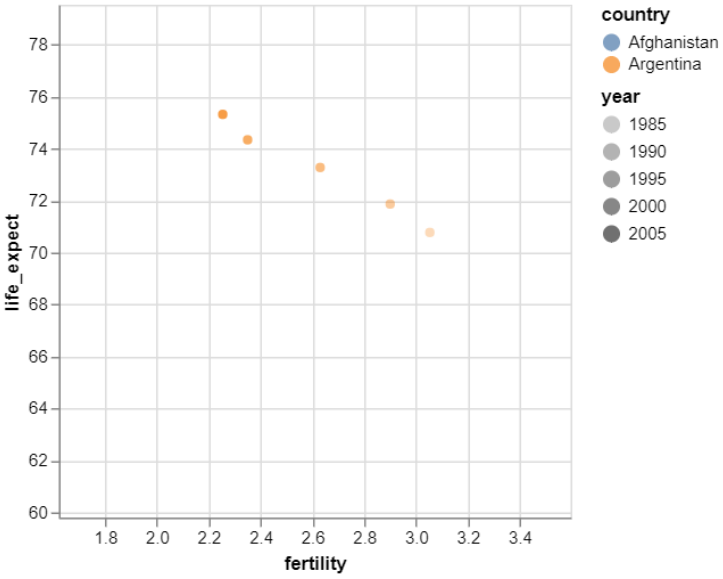
```
df = data.gapminder()

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 > 1980)
& ((alt.datum.country == 'Afghanistan')
| (alt.datum.country == 'Argentina')))
```

The result would be:



Note that we coded each country in a different color, and the years are also encoded with different opacities.

If we want to get closer to the set of points, we can do it by adding ".interactive()" to the chart definition. This method activates zoom and pan options, that can be accessed by pinching (or mouse wheel) to zoom in and out and mouse dragging to pan. So, this would allow a further exploration to the Argentina data by zooming in and panning there:



But with this, the possibilities are still quite limited. To unleash the true power of the interaction we need to incorporate actions such as selection, and make something happen upon selection. We can work on two different ways in altair: using widgets to change view parameters (e.g., a slider to filter out elements), or making the datapoints selectable, and act upon those selections. For this more complete case, the interaction in Altair is built upon three main blocks:

- The *selection* object, which is the one in charge of capturing interactions from the mouse or through other inputs (such as a dropdown or a radio button) to interact with the chart.
- The *condition* function: To make the selections have some effect, we need to change the visual properties according to the selections. This is carried out using a function that takes the selection input and changes elements of the chart based on that input.
- The *bind* property of a selection establishes a two-way binding between the selection and an input element of the chart.

First, we are going to deal with simple examples of interaction that depend on widgets.

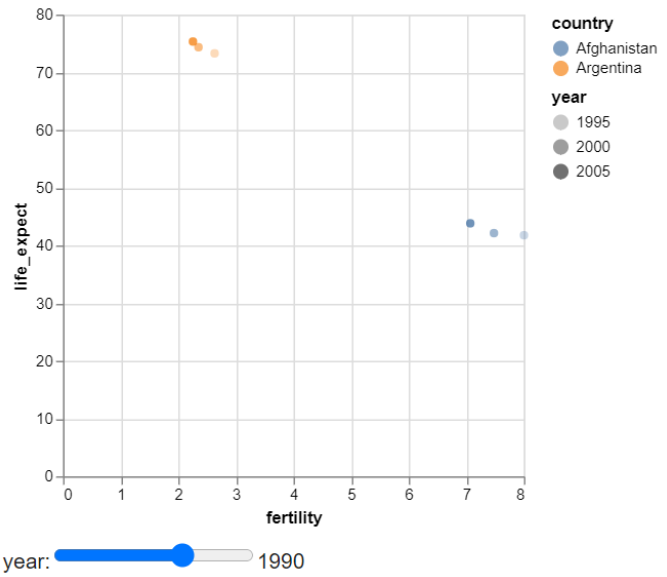
10.2 BASIC INTERACTION: FILTER BASED ON PARAMETERS

Widgets alone can be used to filter out data or change other viewing parameters. To do so, we need to create parameter objects and bind them to the widget. Parameter objects are like regular variables, but they can be changed through the binding to a widget. This is achieved by declaring a parameter as we would a variable, with the *alt.param* call, that can receive an initial value and a binding object. Of course, to bind the parameter to a certain widget, we first need to declare the widget. For a slider, the declaration function will be *alt.binding_range* with the parameters: min (minimum value), max (maximum value), step (step size), and name (defines the label of the widget).

The following example lets us filter the visible points of the gapminder data based on the parameter that can be modified with the slider. The parameter will define the years to be filtered.

```
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)
```

By dragging the slider, we can filter the minimum year to 1990, for example:



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

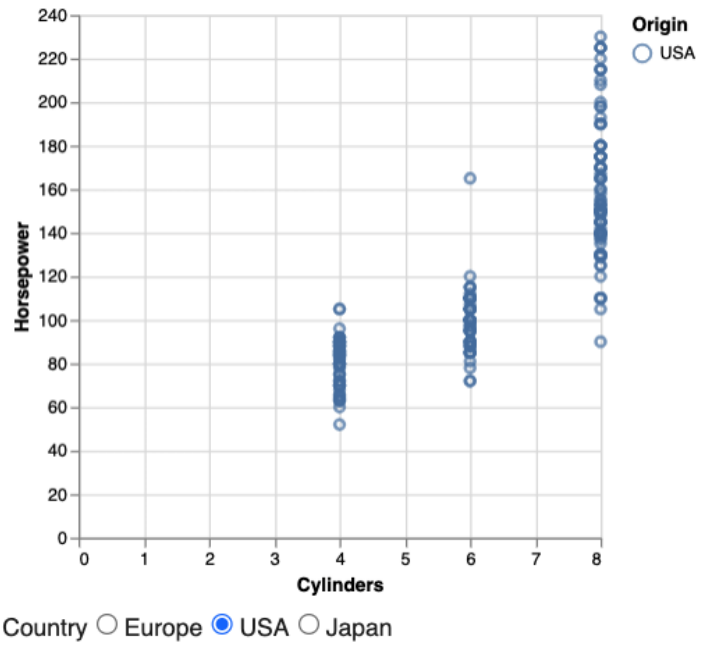
```
slider = alt.binding_range(min=0, max=1, step=0.05, name='opacity:')
op_var = alt.param(value=0.1, bind=slider)

alt.Chart(df).mark_circle(opacity=op_var).encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    alt.Color('country:N'),
).transform_filter((alt.datum.year > 1960)
& ((alt.datum.country == 'Afghanistan')
| (alt.datum.country == 'Argentina'))).add_params(op_var)
```

Another example is shown in the following code, where we use another widget, the radio button, to select which subgroup of data is shown. In this case, we

have the cars dataset, that has cars of three origins: Europe, Japan, and USA. In the following chart, we select which ones are painted through a radio button:

```
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)
```

And the result will be:



Note that now, since only a single selection is performed, the colors of the datapoints are always the same. We will see later how this can be solved using conditions.

We have seen some interaction techniques that allow us to explore the data. However, the most powerful interactions happen throughout the manipulation of the datapoints themselves, and using those selections for cross-highlighting, for example. Altair enables direct interaction with charts through two different selection processes that allow us to select either a single datapoint, or a group of

points. The following section deals with selection, and the next one will present how conditions are used. Finally, we will deal with more advanced bindings and other tips.

11. Selection

Selection is one of the most basic interaction methods in Visualization. By selection we mean the task of choosing one or multiple items that are then treated differently.

The simplest task one can do over selected items its highlighting. But other, more complex tasks can be done with Altair, such as filtering, cross selection, and so on.

Altair has three types of selection:

- Point: Only one data item is selected
- Interval: Multiple items are selected

11.1 INDIVIDUAL SELECTION

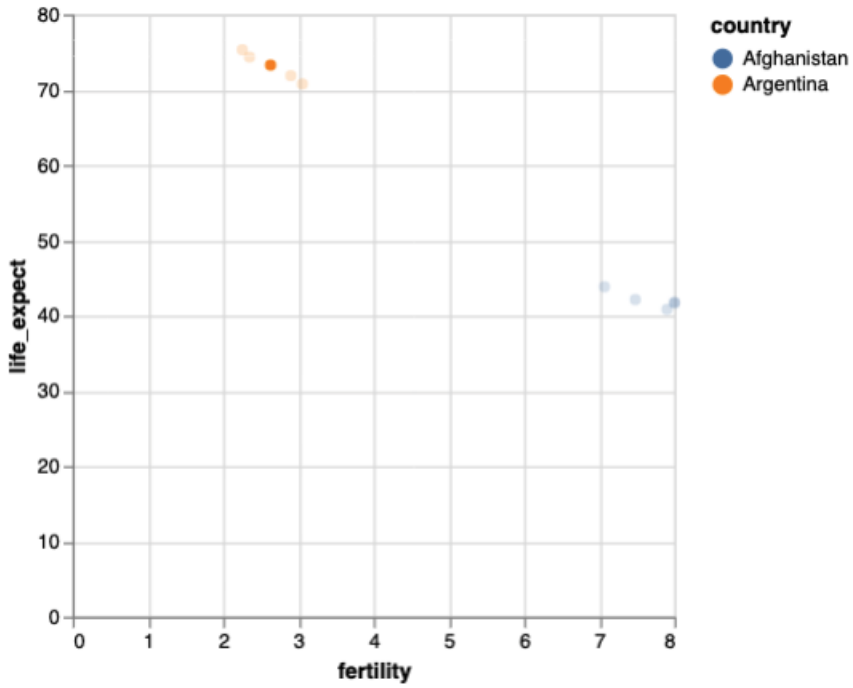
To create a selection tool, we need to perform two steps. We need to declare a selection object, and we need to add it to the chart object as a parameter.

We can select individual items by setting the selection_point object, with the following code:

```
df = data.gapminder()

select_point = alt.selection_point()

alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    alt.Color('country:N'),
    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)
```

Note that a simple selection will do nothing if we don't make the chart react to the selection. In this case, we change the opacity of the datapoints according to the selection: if a datapoint is selected, it is opaque, otherwise, it is assigned

an opacity of 0.2. This happens in the condition call assigned to the opacity channel. The result is illustrated here:



If we clean the selection, by clicking elsewhere, we will recover the original colors.

For charts like the previous one, where clicking an object may require precision, we can relax the selection condition so that we are not forced to click inside the object, but the interface counts as clicked the closer object to the mouse position at the moment of clicking.

This can be achieved by modifying the selection construction adding the parameter *nearest* of the selection, and set it as *True*. We can also make selections without clicking. The library also supports the *hover* operation. It can be defined 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 the point selection is intended for a single datapoint, it can also return a list of datapoints if we use the *Shift* button when moving the mouse.

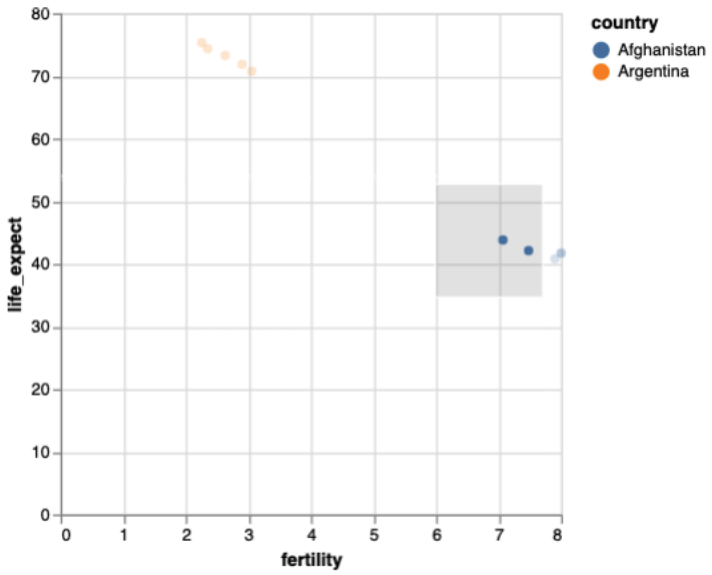
11.2 INTERVAL SELECTION

Multiple selections behave like individual selections. The user can select multiple elements by dragging the mouse (clicking on the initial point, and dragging). The selection object can be defined using *alt.selection_interval*. And, like in the previous case, adding the object with the *add_params* function call.

By changing the previous point selection to an interval selection, we may mimic the previous behavior for a region of objects (i. e., making those fully opaque and the ones outside the selection, semitransparent. This is simply achieved by changing the selection type:

```
select_point = alt.selection_interval()
```

And dragging over the chart, will produce the following effect:



Of course, it would make more sense using a name like selected_points or similar, since in this case, more than one datapoint can be selected.

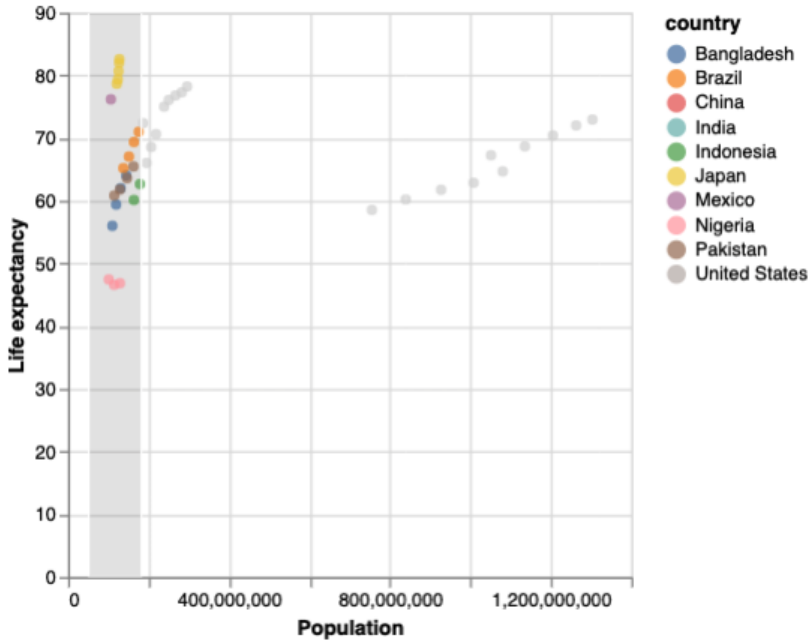
This selection technique is called *brush* (or *brushing*). A *brush* is an operation that selects a rectangular region defined by the initial mouse press position and point at which the left button is released.

Selections also provide some parameters to define the default behavior if no element is selected, through the *empty* parameter, that we can set to 'True' (will consider all elements selected) or 'False' (will consider no element is selected). This way, we can set the initial look of charts before we start interacting with them, if the interaction, for example, changes colors.

But the selections can be also configured differently. For example, we might need to brush only in one axis. This can be achieved by asking the selection to be performed on the 'x' encodings, as shown here:

```
df = data.gapminder()

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

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

The result is a selection in the X axis that will render in the proper color only the selected elements:



Note that, in a previous example, we cannot freely add the *interactive* property of the chart to make interval selection available. The problem lies on the fact that the *interactive* property actually defines a selection mode, which is the selection of scales. That is, the *interactive* mode lets the user interactively modify the scales (in scale, by zooming, or in position by dragging).

We can set such behavior using the selection of scales instead of the *interactive* property the following way:

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

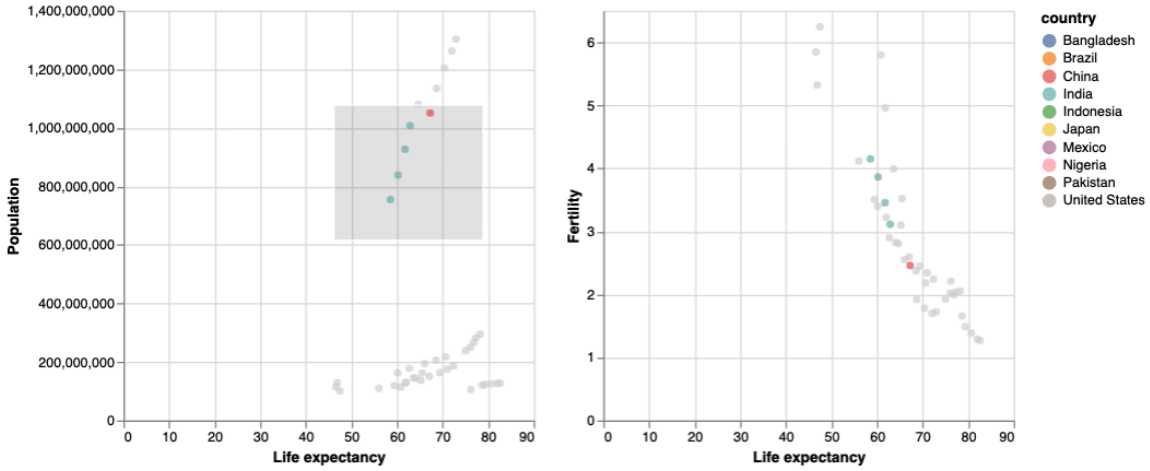
This way, the behavior will be the same than with the *interactive* function. However, this also interacts with the brushing process.

But where we can get the most of selections is when they are used for cross selection. That is, using the selection in one chart to highlight values in another one.

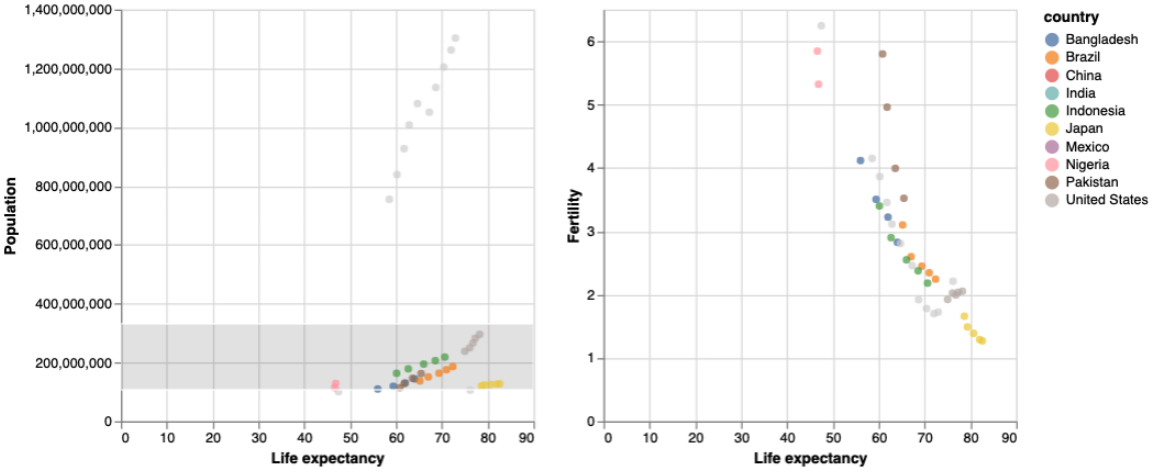
Cross selection can also be achieved using the selector object. For example, if we want to analyze the relation of fertility rate and life expectancy with the population in different European countries, we can plot both charts side by side and, to better inspect the data, make them share the selection. The code could be like this:

```
df = data.gapminder()

select_interval = alt.selection_interval(empty = False)

ch1 = alt.Chart(df).mark_circle().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().encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('fertility:Q', title = 'Fertility'),
    color = alt.condition(select_interval, 'country: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:



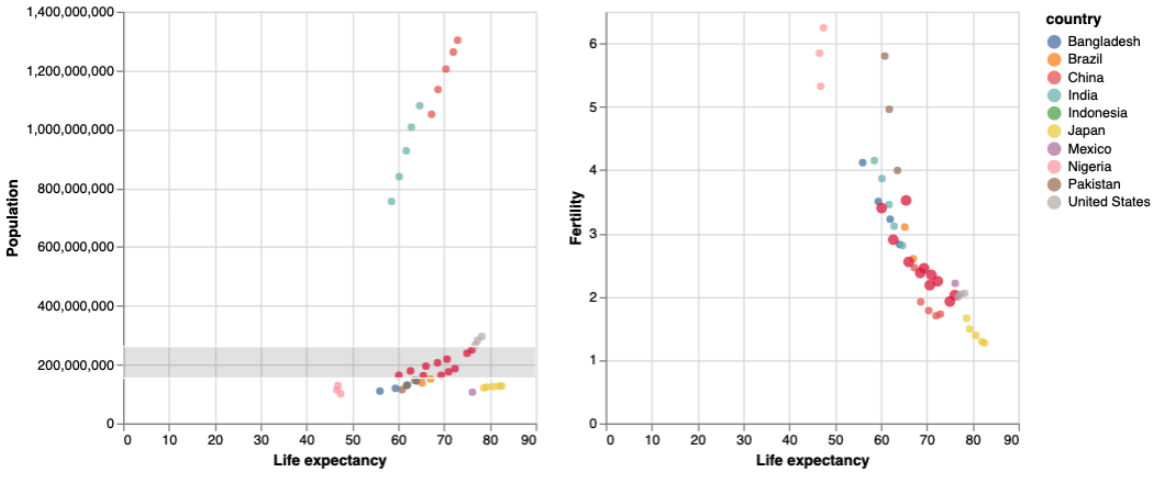
Note that we selected in one axis in the left chart, and the same datapoints were highlighted in the other chart, independently of whether the same coordinates were shared or not. This can more clearly be seen if we restrict the selection to the Y encodings. This can be achieved adding the parameter *encodings = ['y']* to the selection, and the result would be like:



To further communicate the selected items, and to make the chart more colorful, we will keep the original country colors with the nominal palette, and change the selected ones to crimson. Moreover, since the right chart does not show the boundaries of the selected region (as there is not a selection region in this sense), we will increase the size of the selected items. We can do this by adding a second condition. Note the changes in the selection definition as well as the conditions:

```
df = data.gapminder()

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

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

ch2 = alt.Chart(df).mark_circle().encode(
    alt.X('life_expect:Q', title = 'Life expectancy'),
    alt.Y('fertility:Q', title = 'Fertility'),
    color = alt.condition(select_interval, alt.value('crimson'),
                          'country:N'),
    size = alt.condition(select_interval, alt.value(60), alt.value(30))
).transform_filter((alt.datum.year > 1980)
& (alt.datum.pop > 100000000)
).add_params(select_interval)

ch1 | ch2
```

The result would be like this:



11.3 SELECTING BY FIELDS OR ENCODINGS

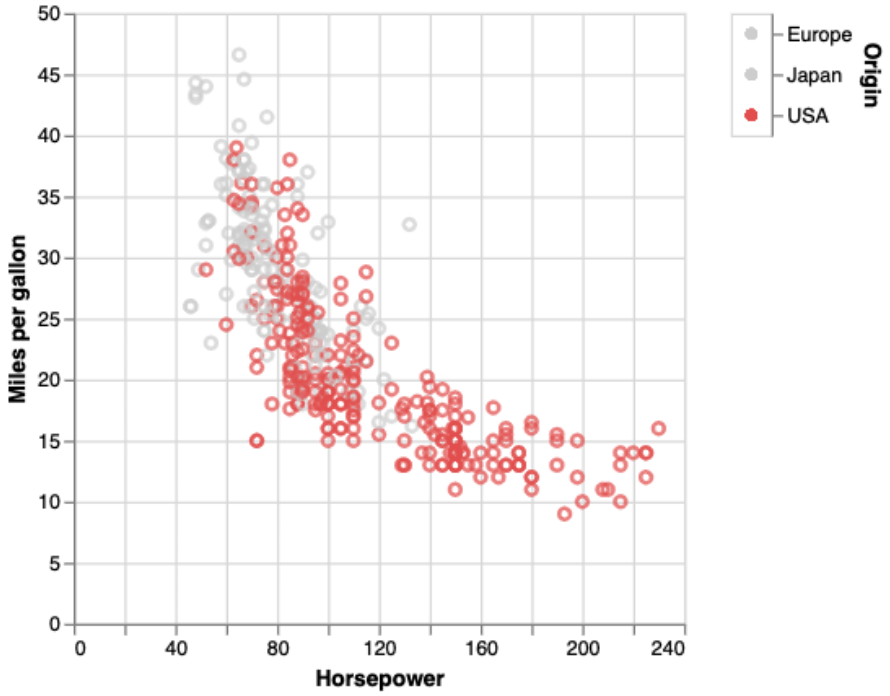
We can customize the selection by thinking on what we are interested on selecting. The two options are fields and encodings. For example, if we take the cars dataset, we may be interested into selecting based on the origin of the cars. In order to do so, we need some widget that provides this information. This can be achieved by creating a legend that encodes this information, and at the same time acts as input to the section process. This legend is created as a small

chart with three items, one per each origin value. Since we have three different values, and those are encoded in different colors, we can ask the selector object to grab the encoding of the element we are clicking, in this case, its color:

```
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
```

Note that we added a new field: tooltip. It can be used to show the details of the data (in this case the name of the item) upon mouse hover. Moreover, we defined the color property before we use it in the chart, so that we can reuse its definition.

The result, with the USA cars selected would be:

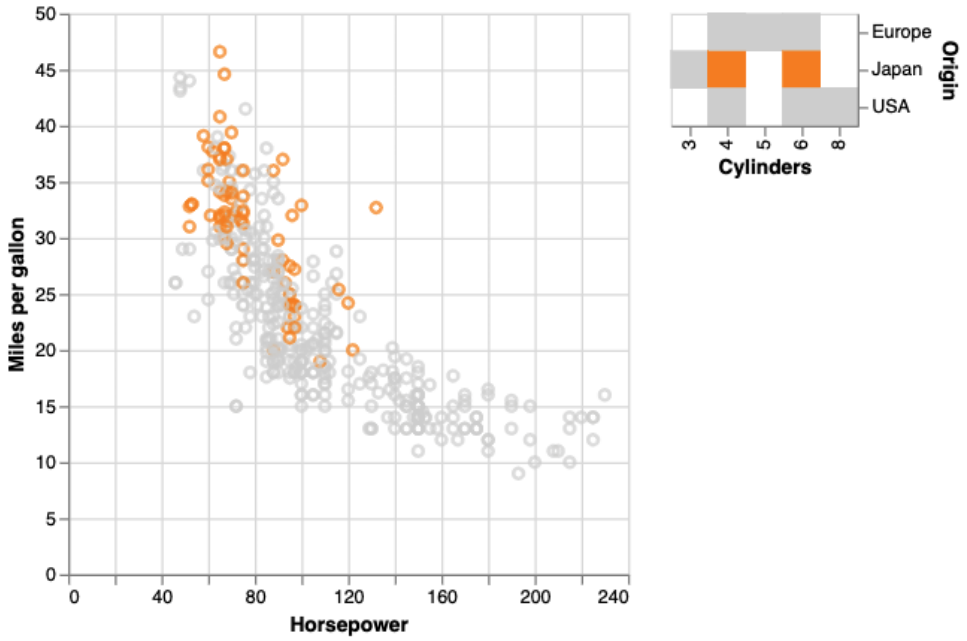


A more complex selection could include more than one field, for instance the origin and the number of cylinders. Since there are different combinations of cylinder numbers and countries, we can create a more complex selector. In this case, a matrix with values for each valid combination. By clicking on the elements of the matrix, we can select the cars that fulfill the properties. In this case, we have also defined a multiple selection, so that the user can click onto many elements of the selection legend (holding the Shift button):

```
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, if we select all the Japan cars with 4 or 6 cylinders would be:



12. Binding interactions to user input

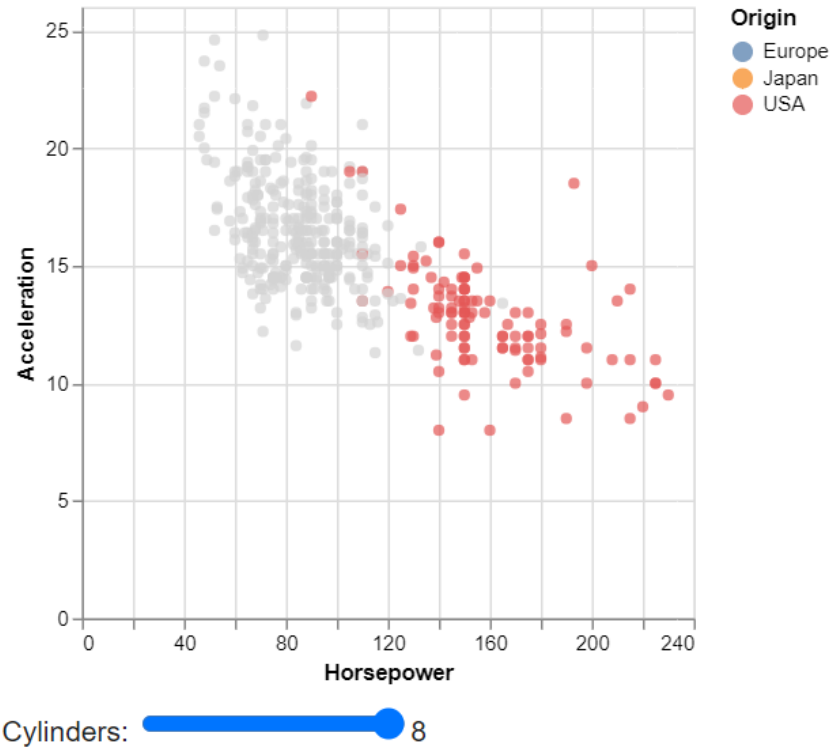
We have already seen that widgets can be linked to user interaction to change variable values that will affect the visualizations. We hereby describe all the widgets available and show some examples on how they can be bound to the visualization.

12.1 SLIDERS

Sliders are input widgets intended to select a value in a range. The slider definition (*alt.binding_range*) allow us to set the value ranges (minimum and maximum), the step, and set an initial value (if needed). In the following example, we want to select the cars with a certain number of cylinders on a plot that shows the horsepower and acceleration. The selection is made through a slider:

```
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 cars with 8 cylinders all come from the US, as can be seen here:



12.2 DROP-DOWN MENUS

We can also select based on dropdown menus. In the following example, we take the *gapminder* dataset, and add the names of the regions. The original data has world regions encoded as the *cluster* variable in the data. The first thing we need to do is to add the names to the dataset, by using the already-known *transform_lookup* function. Then, we add a selection object based on a dropdown menu that has these names. Finally, we filter the data to take into account only the 2005 year.

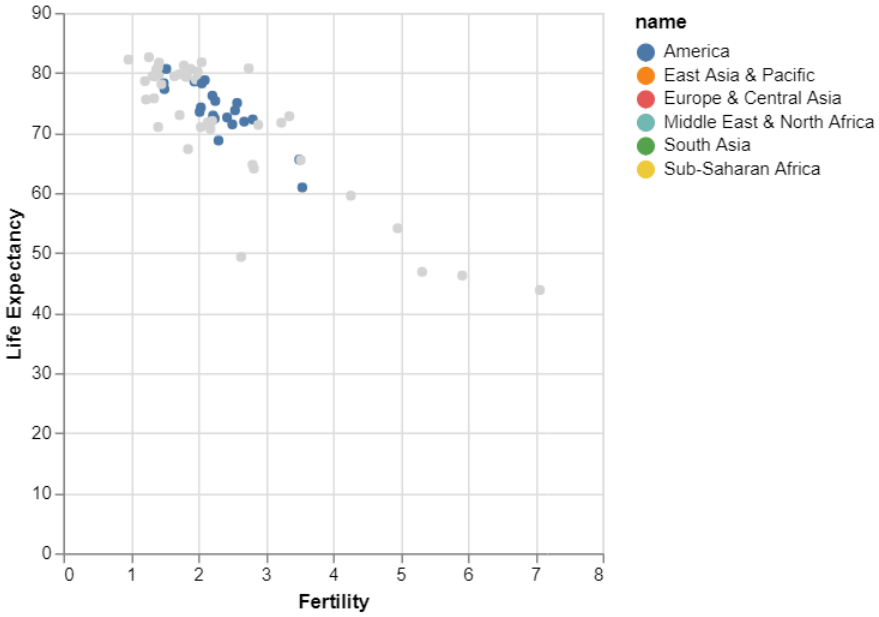
The first part of the code declares the DataFrame that maps the ids of the clusters to their names, and defines the selection object.

```
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'
                                 )
```

Now, 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(dropSelect, '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 == 2005)
```

The result, with the *America* region selected would be:



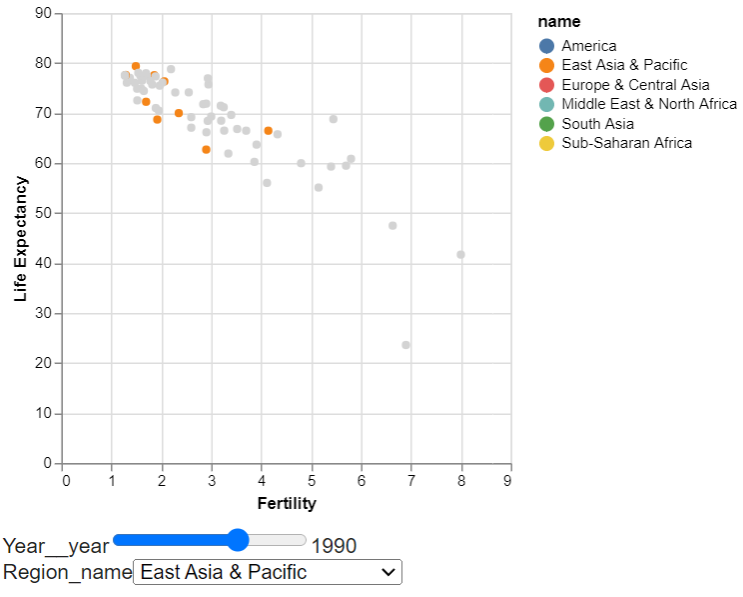
We can further exploit this approach if we want to navigate all the years in the data. We can do so by adding another selection condition that lets the user choose the year she wants to display.

This requires adding a second selection, and modifying a little bit how the data is rendered. In this case, we make all the data not belonging to the selected year as transparent.

The code that implements the behavior is here (we purposely omit the *clusters* DataFrame declaration):

```
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)
```

The result, if we select the year 1990 and the East Asia and Pacific region, appears in the following plot:



Note that we can obtain slightly the same effect using a filter transformation applied to the result of the year selection. The code (of the chart definition) would be:

```
alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = alt.condition(regionSelect, 'name:N',
                          alt.value('lightgray')),
).transform_lookup(
    lookup='cluster',
    from_=alt.LookupData(
        data = clusters, key='id', fields=['name'])
    ).add_selection(
        regionSelect
        ).add_selection(
            yearSelect
            ).transform_filter(yearSelect)
```

However, in this case, since the filtering removes part of the data, not all the years are painted equally. More concretely, the vertical axis changes after year 2000 (has a domain from 0 to 80 previous years, and 0 to 80 after), and the horizontal axis too. This is an unexpected behavior that we must avoid, since it makes it difficult for the user to make visual comparisons if the frame of reference is not constant. As a result, this approach, at least for this dataset, would not be suitable.

12.3 OTHER WIDGETS

Other data bindings are radio buttons and checkboxes.

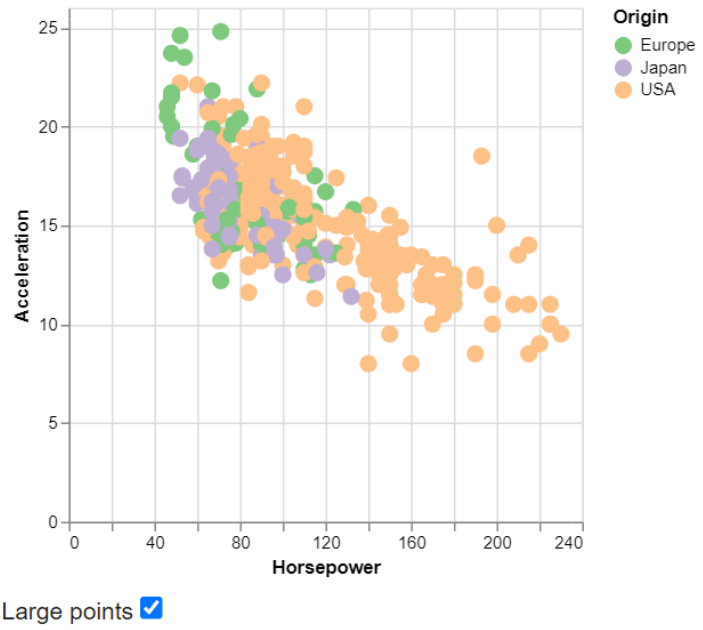
The following example uses radio buttons to decide which country is displayed. Note that we use a domain for the color to ensure the countries are always painted with the same palette.

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

cars = data.cars()

input_dropdown = alt.binding_radio(
    # Add the empty selection which shows all when clicked
    options=options + [None],
    labels=labels + ['All'],
    name='Region: '
)
selection = alt.selection_point(
    fields=['Origin'],
    bind=input_dropdown,
)

alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    # We need to set a constant domain to preserve the colors
    # when only one region is shown at a time
    color=alt.Color('Origin:N').scale(domain=options),
).add_params(
    selection
).transform_filter(
    selection
)
```

The following example illustrates the use of a checkbox, that is used to increase the size of the points:

```
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)
```

The result, when the code is executed, appears like this:



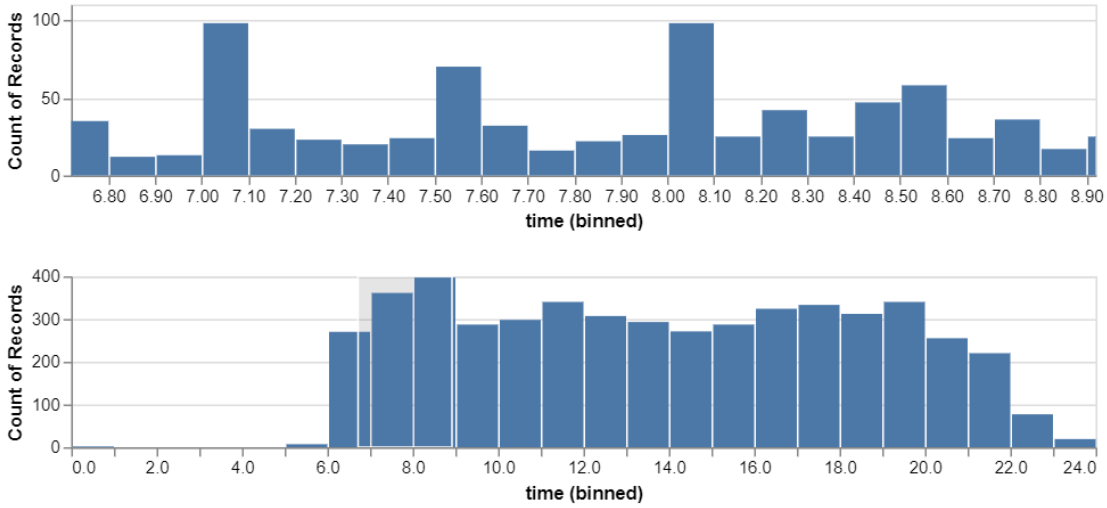
12.4 RESPONSIVE CHARTS

Besides changing the look and feel of some charts according to selection, from version 4 charts can also respond to some interactions. We already saw the *content* value for width and height (though not working currently in Google Colab, you can have such a chart in another environment).

Another possibility is to change the behavior of a histogram according to an interaction, for example a brushing. In this example, taken from altair's webpage, we can see how to change the top histogram according to a selection in the bottom one:

```
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 may look like this:



12.5 USING WIDGETS IN CREATIVE WAYS

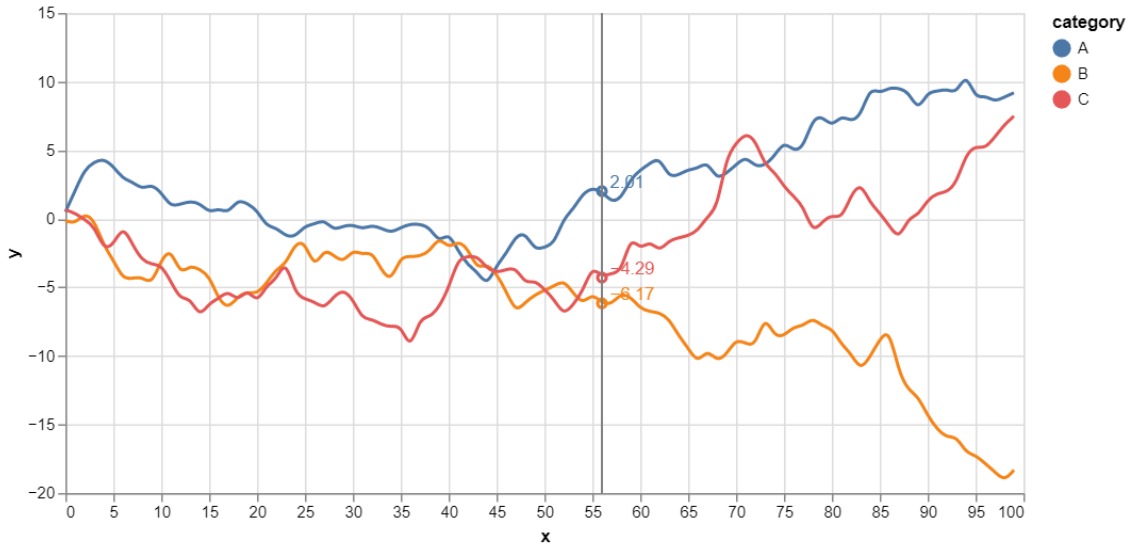
The combination of selection and hover can be used in several ways, such as the example "Multi-line tooltip" in the Altair library, where the plot creates a number of hidden tooltips that are rendered visible only when the mouse is hovering close to them. Note the clever usage of *nearest* and hover.

```
# Create a selection that chooses the nearest point & selects based on x-value
nearest = alt.selection_point(nearest=True, on='mouseover',
                        fields=['x'], empty=False)

# The basic line
line = alt.Chart(source).mark_line(interpolate='basis').encode(
    x='x:Q',
    y='y:Q',
    color='category:N'
)

# Transparent selectors across the chart. This is what tells us
# the x-value of the cursor
selectors = alt.Chart(source).mark_point().encode(
    x='x:Q',
    opacity=alt.value(0),
).add_params(
    nearest
)
```

Then, the code to draw the tooltips, uses the condition to add the information that would correspond to the "details-on-demand" aspect of the visualization. These are implemented as three layers, one for the points on the curves, another for the text, and another one for the vertical rules:

```
# Draw points on the line, and highlight based on selection
points = line.mark_point().encode(
    opacity=alt.condition(nearest, alt.value(1), alt.value(0))
)

# Draw text labels near the points, and highlight based on selection
text = line.mark_text(align='left', dx=5, dy=-5).encode(
    text=alt.condition(nearest, 'y:Q', alt.value(' '))
)

# Draw a rule at the location of the selection
rules = alt.Chart(source).mark_rule(color='gray').encode(
    x='x:Q',
).transform_filter(
    nearest
)

# Put the five layers into a chart and bind the data
alt.layer(
    line, selectors, points, rules, text
).properties(
    width=600, height=300
)
```

The result is shown next:

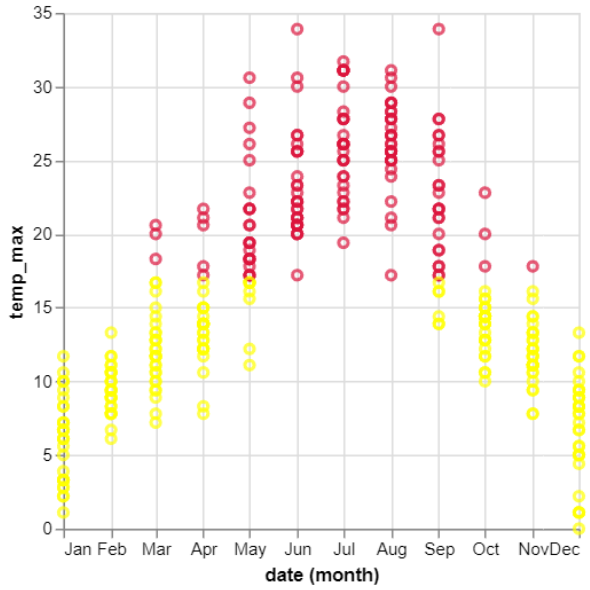


However, this approach, that is completely unaware of the data distribution, may cause some labels to overlap, and then, reading them might be difficult.

Selections can also be used in expressions. For example, we can create a slider to define a range of values. In the following example, we highlight the maximum temperature values of the *seattle_weather* dataset that are under a certain value expressed through a slider:

```
source = data.seattle_weather()

temp_slider = alt.binding_range(
    min = 0, max = 100, step = 1, name = 'tmpCut: ')
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'),
    alt.Y('temp_max:Q'),
    color = alt.condition(alt.datum.temp_max < sel_temp.tmpCut,
                          alt.value('yellow'), alt.value('crimson'))
).transform_calculate(year = 'year(datum.date)'
).transform_filter(alt.datum.year == 2013).add_params(sel_temp)
```

The result is:



Although it is not currently possible to directly select the field you want to plot in a chart according to a widget (e.g., defining a channel, such as 'x' with a parameter obtained from a selection), there is a workaround using the transform_calculate function, as shown in the following example taken from the documentation.

```
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(
    x=alt.X('x:Q').title(''),
    y='Miles_per_Gallon:Q',
    color='Origin:N'
).transform_calculate(
    x=f'datum[{xcol_param.name}]'
).add_params(
    xcol_param
)
```

In this case, what the code is doing is explicitly declaring the names of the fields in the dropdown and, upon user selection, it transforms the text into a new value in the dataset through transform_calculate. Then, it uses the new column, named x, as the values encoded in channel X.

13. Compound charts

There are several ways of displaying multiple charts in Altair.

The most basic approaches are the use of the operators '+', '|', and '&'. The first one will overlap two charts. The second one, lays two charts side to side, and the third one draws the two charts one on top of the other.

The operators correspond to function calls:

- alt.layer: is the equivalent to the '+' operator. However, the function call, whose parameters are the names of the charts to overlay, accepts any number of charts.
- alt.hconcat: lets the user place multiple charts horizontally.
- alt.vconcat: places multiple charts vertically.

For layered charts, the drawing order is from the first to the last, and they are drawn on top of each other. This means that subsequent charts may occlude the marks of the previously drawn.

Besides these basic methods of chart layouts, there are two especially designed for multiple charts:

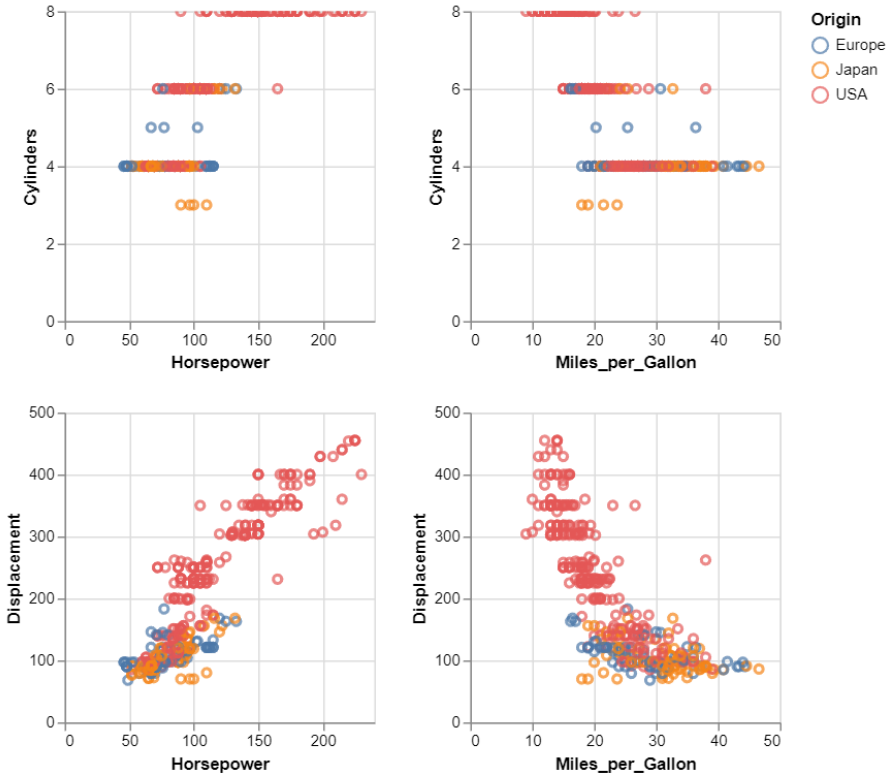
- Repeated charts: Are intended to draw multiple charts in vertical or horizontal layout, where the only change between them is the modification of one or more encodings.
- Faceted charts: Their objective is to produce multiple views of a dataset where for each chart, the represented information is a subset of the data.

13.1 REPEATED CHARTS

The function that provides this feature is *repeat*. It must be defined in two steps. First, in the encoding part of the chart definition, we set up what information must be repeated (either column, or row, or both), and the type of encoding used (e.g. quantitative). Second, we modify the chart by adding the *repeat* function, that specifies the parameters applied to each dimension. The following example illustrates the 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 4 charts, where the *Cylinders* and *Displacement* values will be compared with the *Horsepower* and *Miles per Gallon* variables. as depicted here:



The same result can be obtained by a combination of vertical and horizontal layouts. However, the code is more cumbersome.

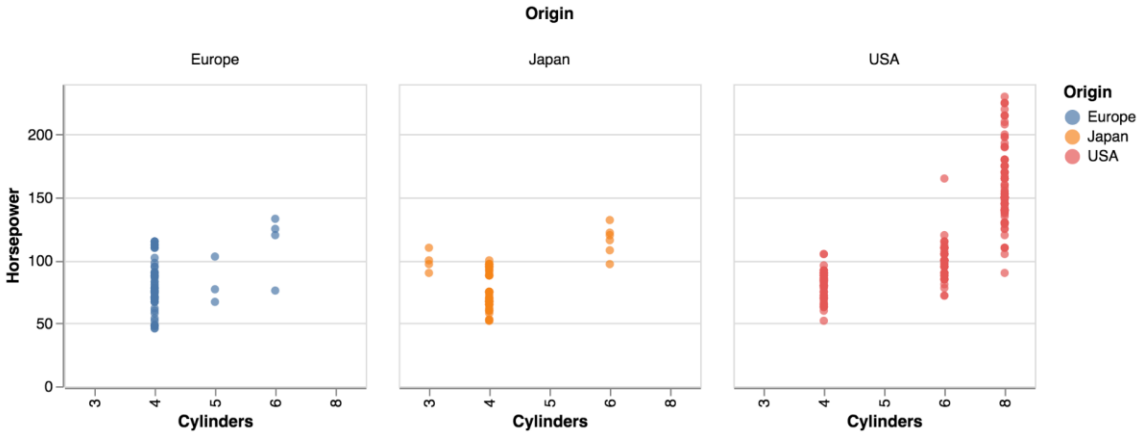
13.2 FACETED CHARTS

Laying multiple facets of the same chart can be also done with a horizontal or vertical concatenation where different filters are applied to each of the

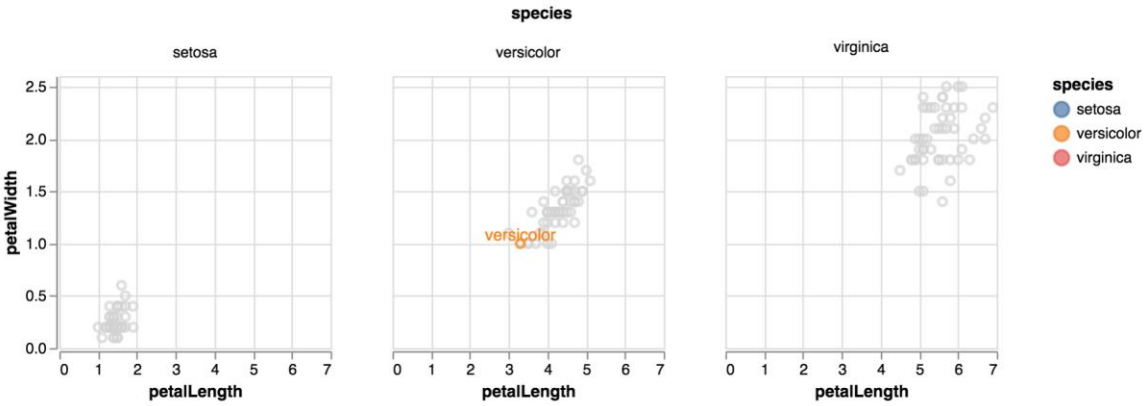
individual charts. However, the *facet* operation makes it slightly easier if the filtering operation can be expressed easily. For example, we can plot the horsepower of the *cars* dataset against the number of cylinders, and faced them by origin:

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

The result would be:



For this concrete case, we could also get the same result using the parameter *column* in the chart encoding. However, the *facet* method can build compound layouts of more complex charts, such as ones with selection operations as the following one:



Where each chart is a combination of two charts, one that shows the text upon hovering, and another that shows the points. To create both, we have defined a base chart, and used it to create the one that uses as marks the text identifying the item, and another for the points. Both of them respond to a condition that either de-emphasizes (points) or renders (or makes transparent) the text:

```
iris = data.iris()

hover = alt.selection_single(on='mouseover', nearest=True, empty='none')

base = alt.Chart(iris).encode(
    x='petalLength:Q',
    y='petalWidth:Q',
    color=alt.condition(hover, 'species:N', alt.value('lightgray'))
).properties(
    width=180,
    height=180,
)

points = base.mark_point().add_selection(
    hover
)

text = base.mark_text(dy=-5).encode(
    text = 'species:N',
    opacity = alt.condition(hover, alt.value(1), alt.value(0))
)

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

Faceting can be done in rows and columns, and both at the same time, as the following example:

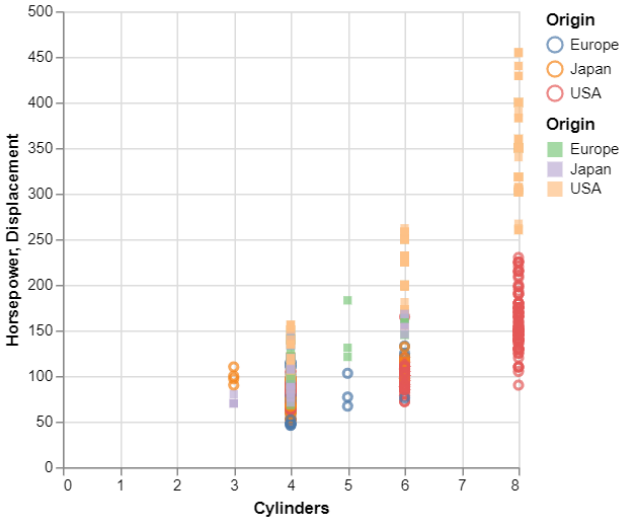
```
import altair as alt
from vega_datasets import data
cars = data.cars.url

alt.Chart(cars).mark_circle().encode(
    x = 'Cylinders:O',
    y = 'Horsepower:Q',
    color = 'Origin:N'
).properties(
    width=200,
    height=200
).facet(
    row='Origin:N',
    column='Cylinders:O'
).transform_filter((alt.datum.Cylinders > 3)
  & (alt.datum.Cylinders < 7))
```

In this case, the result is a 3x3 arrangement of charts, since we filtered the number of cylinders to reduce the output plots for convenience.

Note that, when creating composed charts, if we want to use different color schemes, besides telling altair the color scale to use for each chart, we also need to ensure that altair choses only one of those for both. This can be achieved using the *resolve_scale* parameter, as noted earlier. This applies both when layering, and when faceting. An example of layering would be this one:

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

alt.layer(first, second).resolve_scale(color = 'independent')
```

The result will preserve both color scales:



Though note that we are using double axis in the Y direction, to encode two different values. This is a practice that does not work properly in many cases. In any case, if necessary, we can ask altair to set dual axes by defining the *resolve_scale* property with an independent y, as follows:

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

alt.layer(first, second).resolve_scale(color = 'independent').resolve_scale(
    y='independent'
)
```

This will label the Y axis on the left and right independently.

14. Advanced Maps

For some visualizations, you may need to interact in a different way with choropleth maps. For example, let's imagine that we want to plot a value on maps that can change through the years and we want to have a selector of the

year through a slider. This can be difficult to accomplish by generating a map through the *alt.Chart().mark_geoshape* with the geometry as input, since the data to color the map is gathered through a *look_up*, which does only give a value per country.

A way to solve this is to think the plot as a plot that renders the indicator of the proper year, and gathers the geometry of the countries.

You can find a complex example in the following website:

https://www.kaggle.com/labdmitriy/kaggle-survey-2019-map-mini-dashboard-altair/notebook

A simple version, that renders the data of the *gapminder* dataset for some years could be designed as follows.

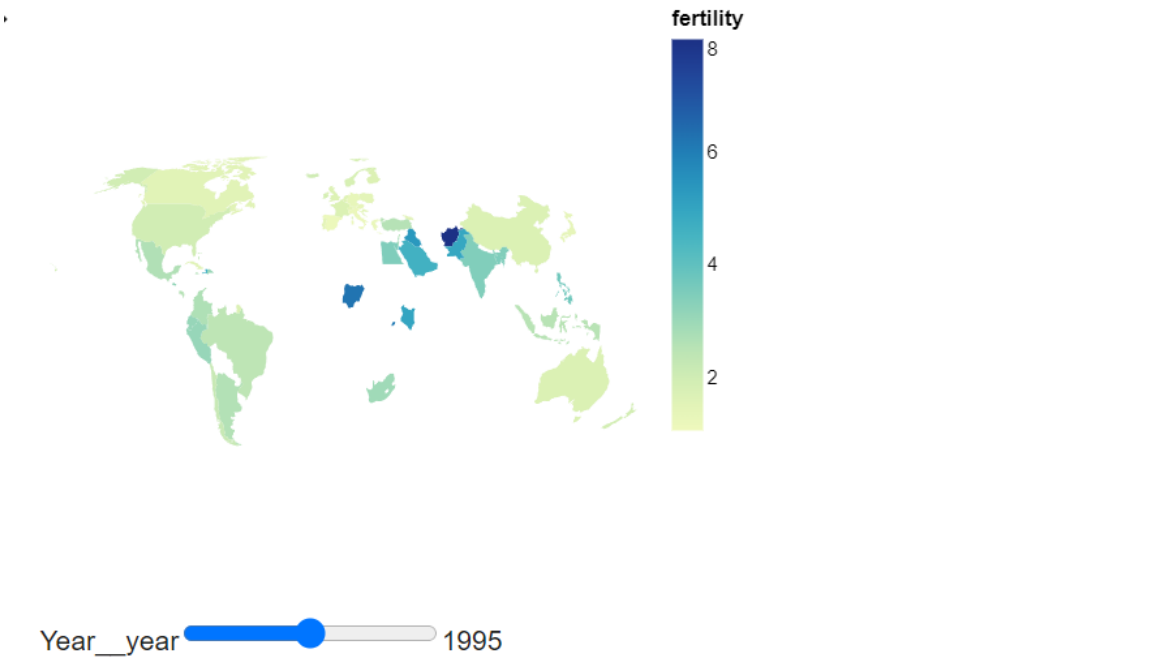
First, we create the imports and load the files that match country names with country codes. Then, we read and merge the data and the country codes files:

```
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)
```

Now we create the visualization taking as input the *merged* file and looking up the geometry in the *geom* file.

```
input_slider = alt.binding_range(max = 2000, min = 1990, 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(
    color = 'fertility:Q'
).add_params(selection)
```

The result is something like this:



Note that some countries appear as white. There are two reasons that may be the cause: a) the gapminder dataset does not have the data on those, or b) the gapminder file and the file that maps codes to country names may have different names for the same country.

15. Interactive visualization of very large datasets

Altair, and Vega, have two issues regarding to large datasets: a) the limitation in the number of rows, and b) the performance. The fixed number of rows can be removed using `alt.data_transformers.disable_max_rows` as explained previously. However, large datasets may still cause problems in the display, and sometimes the Colab runtime disconnects and the data is not displayed.

There are some ways to solve this, one may be generating the chart in html instead of rendering it. Opening the generated file will typically work. However, there is a way that lets us work interactively in Colab with such large datsets: using Vegafusion. Vegafusion (https://vegafusion.io/) is an external project aimed to provide scaling at server side of the Vega library.

Despite some examples are shown in the webpage, not all of them work properly in Colab. The following approach does.

First, you need to install vegafusion['embed'] and vegafusion_jupyter.

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

```
!pip install vegafusion_jupyter
```

To use the Vegafusion widget, necessary to handle the data, you need first to tell Colab that they are going to be used, with this code:

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

Note that this code needs to be enabled for each session.

Then, you need to enable the widget of vegafusion with the following code:

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

vf.enable_widget()
```

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

The following example code (from the Vegafusion webpage) shows 1M flights in an interactive visualization:

```
vf.enable_widget()

flights = pd.read_parquet(
    "https://vegafusion-datasets.s3.amazonaws.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='quantitative', 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 like this: