
# Data Visualization with Altair

Pere-Pau Vázquez | 2021

# Contents

# 1. Introduction

Altair is a declarative visualization library for Python. In its current version (4.1 as of April 2020), it only supports Python 3.6, due to the deprecation of previous versions of Python.

From the architectural point of view, Altair generates Vega code, which is then executed in a JavaScript environment.

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.

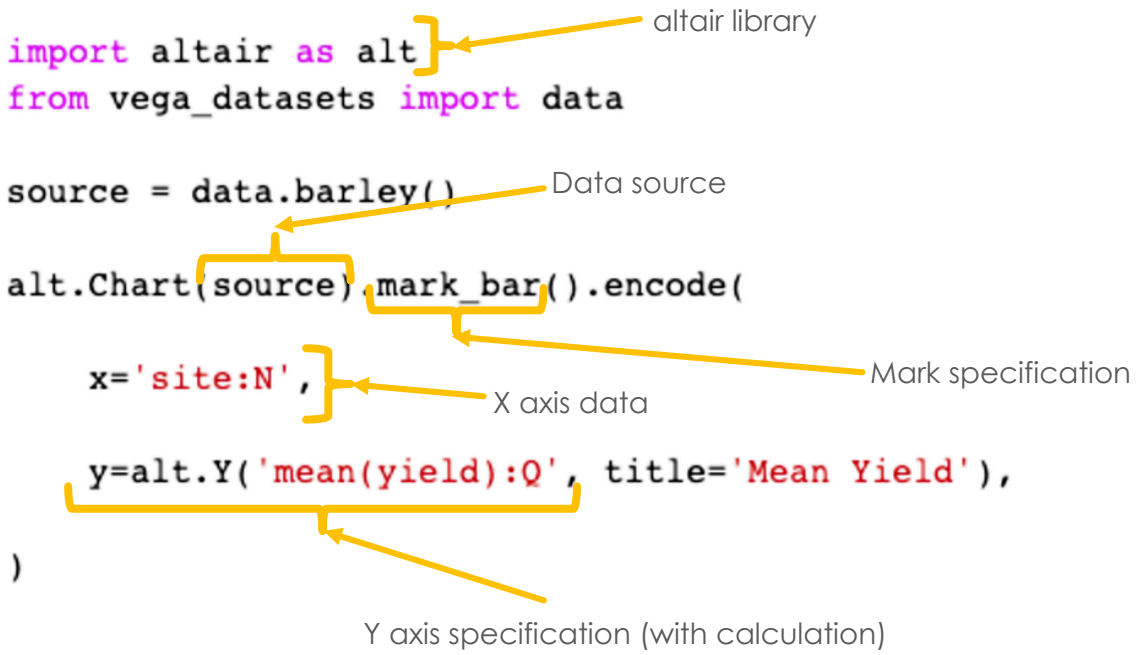
There are other alternatives to get the same work done in Python. For instance, Seaborn is an easy to use library that is programmed using the imperative paradigm, while Bokeh is a declarative library that can also be used. However, both of them are much less powerful than other alternatives such as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

# 2. Altair basics

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.

## 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'
)
```

The inverse transformation is called *pivot* transformation and was incorporated to altair in version 4.0

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'
)
```

# 4. Visualization Pipeline

The common process you have to follow when designing visualization applications includes the following steps:

- Understanding the problem
- Gathering data
- Cleaning data
- Visual encoding design
- View design
- Interaction design
- Evaluation

In the first step, it is necessary to understand what is the type of data we can work with, and which are the questions that the users are asking themselves about the data. This is a crucial step to successfully design any visualization application.

After that, we need to obtain the datasets. Sometimes, these will be available from Open Data websites such as gapminder.org or the New York City council. However, oftentimes, we need to capture or generate the data ourselves. So this may be a complex, time-consuming process.

The data do analyze will commonly have many artifacts. From missing entries, to format issues, we need to cleanse the data properly. So, for most of the examples provided in the exercises proposed, the first step you need to do is to analyze the data using some software package such as Open Refine, and ensure that the input is correct. For the datasets in the *vega_datasets* module, you can assume they are already clean.

Once the data is clean, and we have a rough idea on what are the types of questions the users want to answer with the data, we can start thinking on the visual encodings, as well as the interaction techniques we want to develop to solve the questions. Note that the solution must follow the *Visualization Mantra*: Overview first, zoom and filter, details on demand. Therefore, the initial view to think of is the one providing an overview of the data. To do this, we need to think of what visual encodings are going to be used. These have two main parts: marks, and visual variables. The marks are the geometric entities that we will use to represent each data unit. The visual variables (or channels) are the attributes (e.g. color, opacity…) that will modify the marks to make the design expressive. In the following sections (*Marks*, *Channels*, *Charts*), we will deal with the different methods to encode data in Altair.

Typically, in any visualization system, several views will be needed. Sometimes an overview and detail, sometimes focus and context… Depending on the problem at hand, we will use one of those techniques. This implies designing different views for the aspects of the data we want to analyze. After the visual encodings, we will deal with the different methods of designing multiple views in the section named *Facets*.

Finally, to provide effective data exploration, we need to design and implement interaction methods that provide different operations for data manipulation, such as selection, zooming, filtering, and so on… We will deal with those in section *Interaction*.

# 5. 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.

## 5.1 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
- 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
- Square: It is basically a scatterplot with square marks

Besides these basic marks, that can be used for most of the common charts, other marks are more specific:

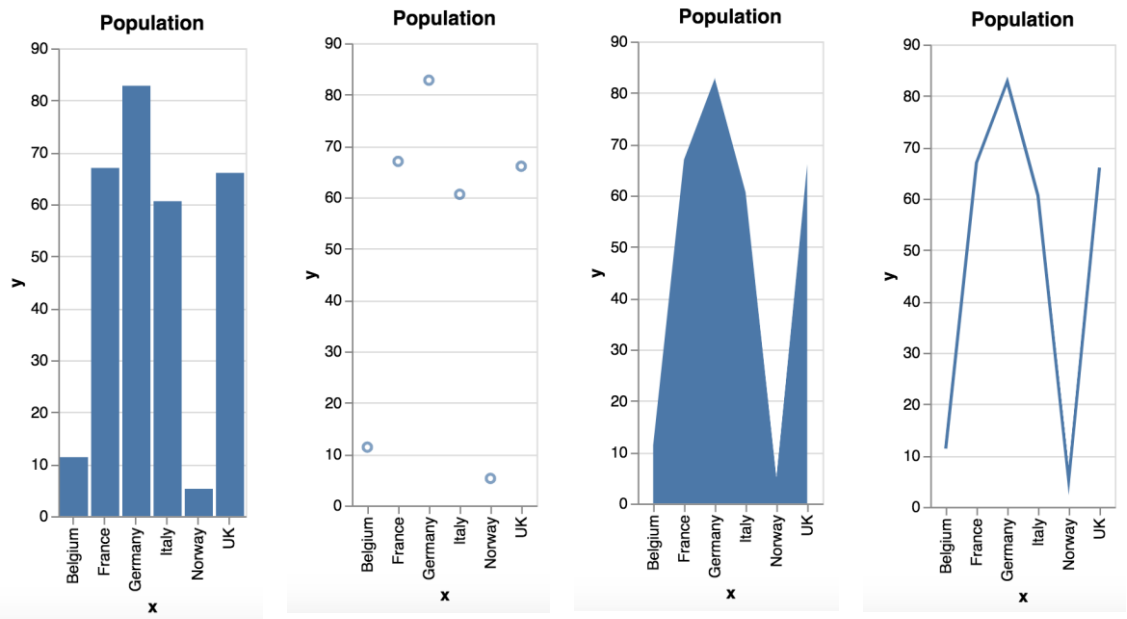
- Geoshape: To plot a geographic shape
- Rule: A vertical or horizontal line that spans the whole axis. Typically used to indicate reference values
- Text: Can be used to draw a text onto a chart
- Tick: Used to paint a vertical or horizontal tick mark. It can be used to draw strip plots

The basic types serve to encode simple elements in charts.

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')
```

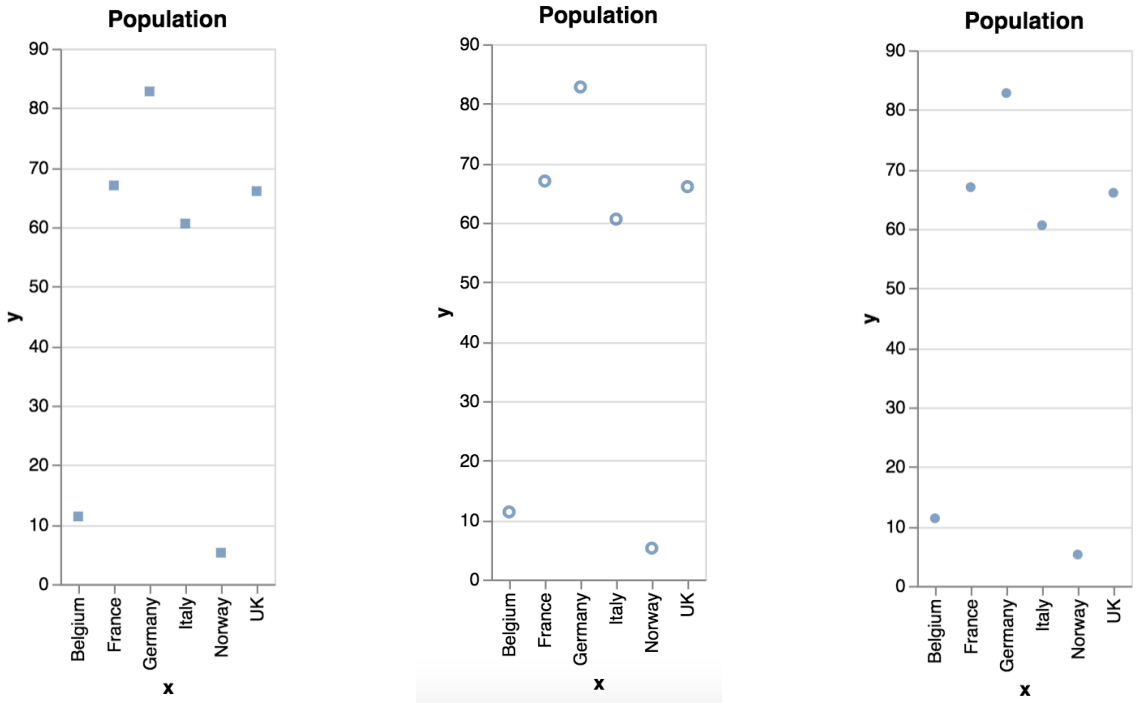
By changing the mark to point, area, or line, you will have these four variants of the same chart:




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.

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.

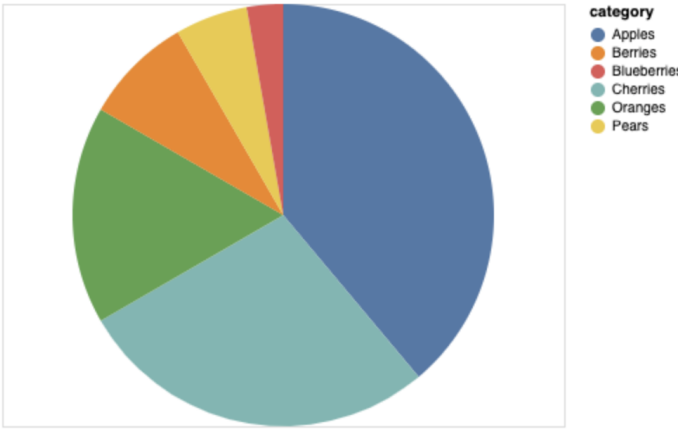



Another, very popular visualization technique is the pie chart. Pie charts were added to altair very recently, in version 4.2. Though they are highly controversial and many visualization practitioners will not use them, they appear in many infographics.

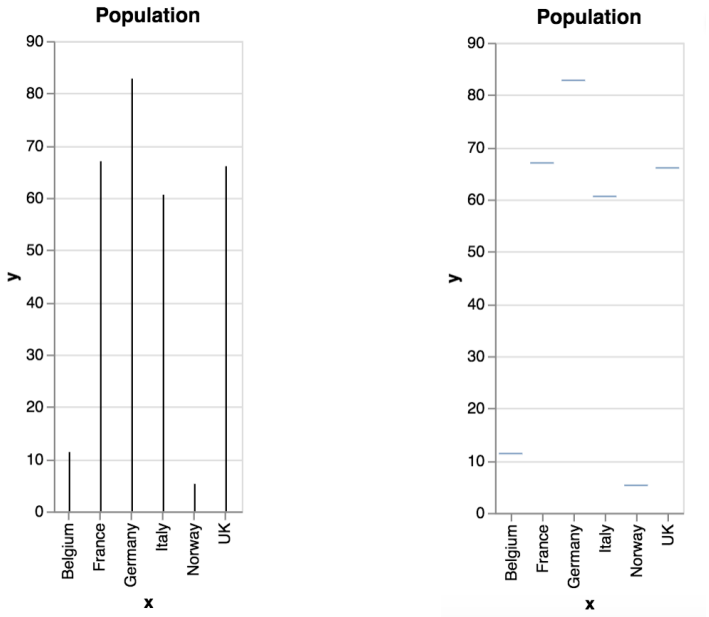
The most basic version that altair supports is the regular pie chart:

```
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:




There are two other basic types that may result not so familiar: rule and tick. Again, we can compare them simply by showing how they plot the same data in a chart.



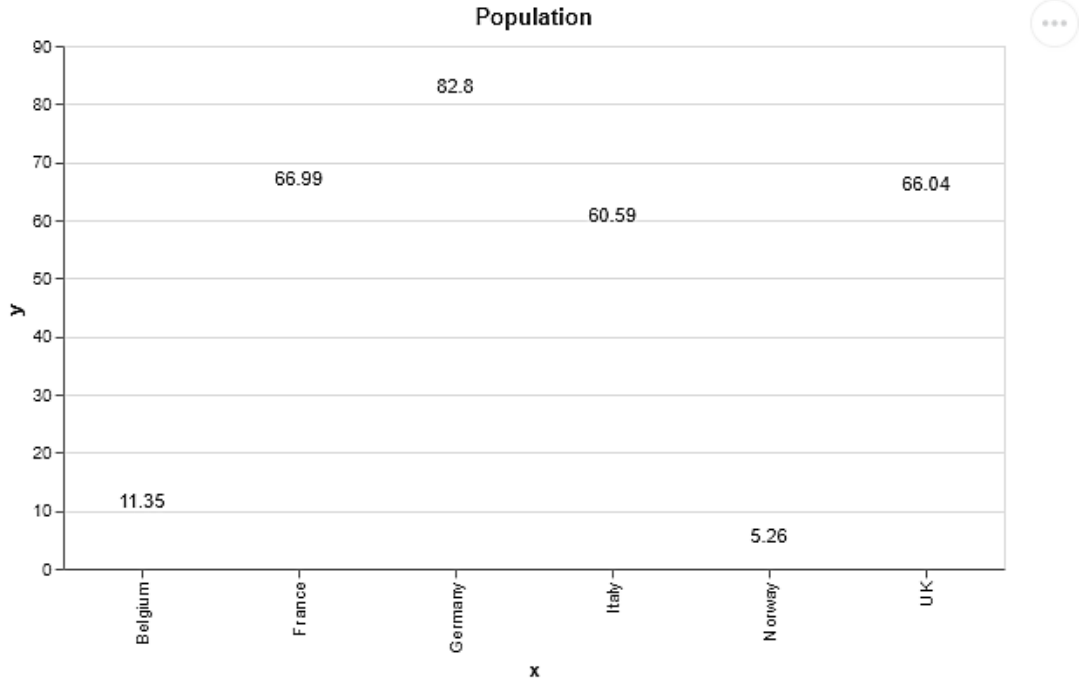

## 5.2 SPECIFIC MARKS

Besides the basic marks, there are three other specific types:

- Geoshape: A geographic shape.
- Rule: a vertical or horizontal line spanning the axis.
- Text. A scatter plot with points represented by text.

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.

For completeness, let's point that beside these simple marks, Altair also provides some compound marks:

- Box plot: To generate box plots.
- Error band: A continuous band around a line.
- Error bar. An error bar around a point.

Again, we will work on these later in the course.

# 6. Channels

The visual properties of marks are called channels. Altair has several ways to modify channels. The first one is shape, that, as we saw in the previous section, can be modified using the different *markRef* properties.

## 6.1 VISUAL PROPERTIES

Some relevant channel modifiers are:

Channels. Color:

- color: default color of the mark
- fill: color that fills the mark (has higher precedence than color)
- fillOpacity: float indicating the opacity [0..1]
- filled: boolean indicating whether the mark is filled
- opacity: float indicating the overall opacity [0..1]
- strokeOpacity: float indicating the stroke opacity [0..1]

For defining the shape and position, Altair provides these channel modifiers:

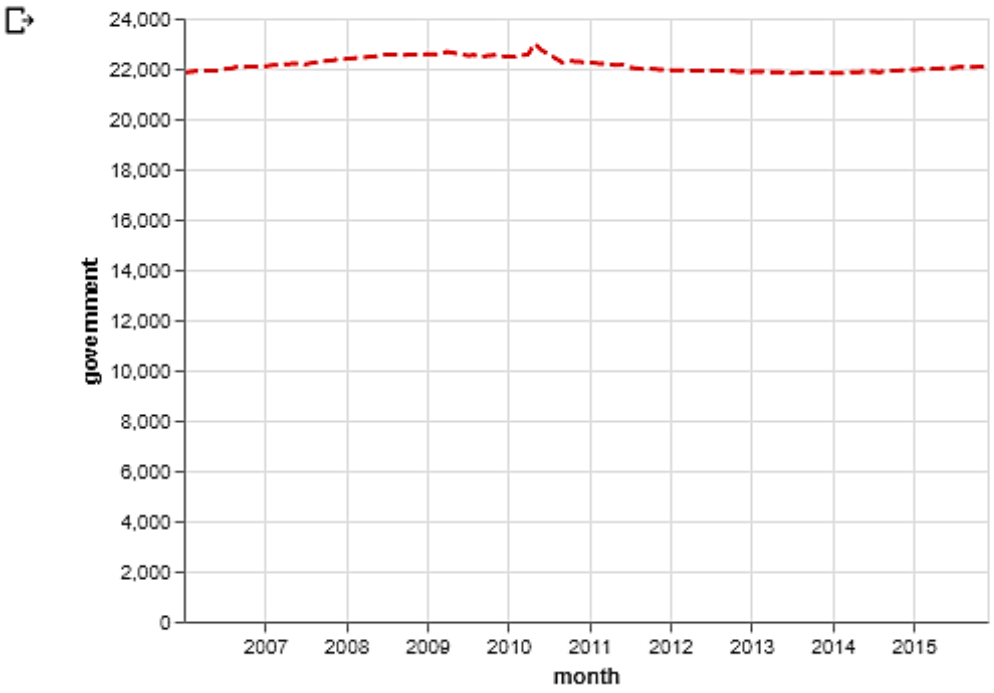
- height: height of the marks
- shape: for point marks, shape can be:
    - circle, square, cross, diamond, triangle up, triangle down, triangle right, or triangle left
- Other shapes: arrow, wedge, triangle
- A custom SVG path (defined in a rectangle between -1 and 1)
- size: the size of the shape. For point, circle and square, it will be the pixel area of the marks.
- 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)
- width: width of the marks.
-

Other properties of the marks refer to the strokes:

- stroke: Default color for the stroke. It has higher precedence than default color (defined using config.color)
- strokeDash: An array of alternating stroke and space lengths, for creating dashed or dotted lines, that may depend on the encoding.
- strokeWidth: The width of the stroke, in pixels.
- thickness: thickness of the tick mark.
- tooltip: Tooltip text to show upon mouse hover over the object.

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

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

df = data.us_employment.url

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



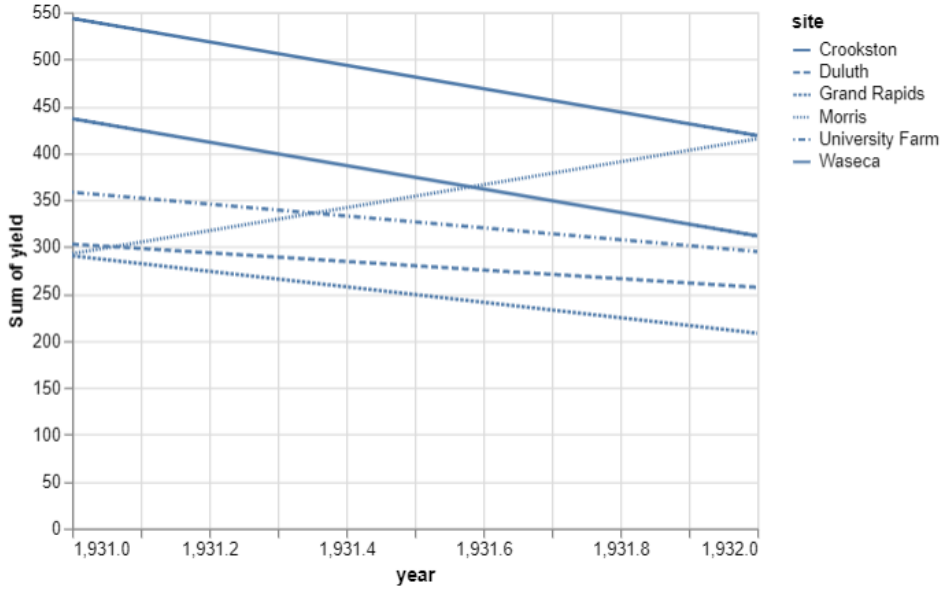

As stated, the *stroke* property has more precedence than the *color* property. In the following chart, we use both, besides changing the stroke width.

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

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.

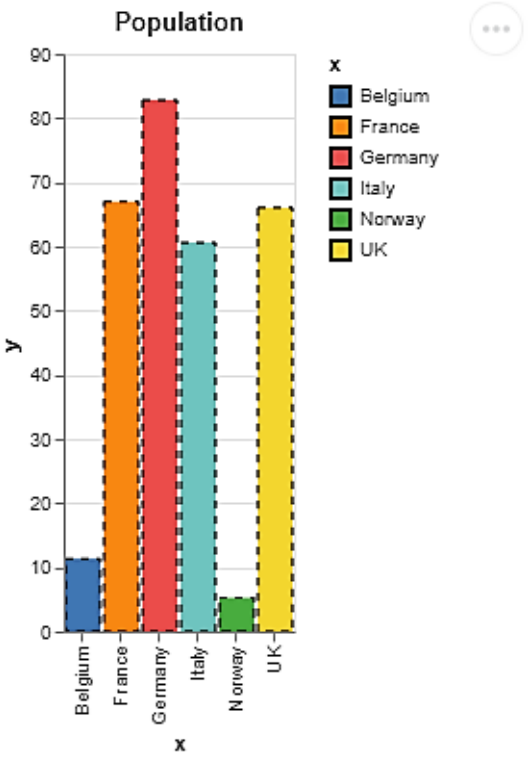
```
import altair as alt
from vega_datasets import data

source = data.barley()

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

The result would be:



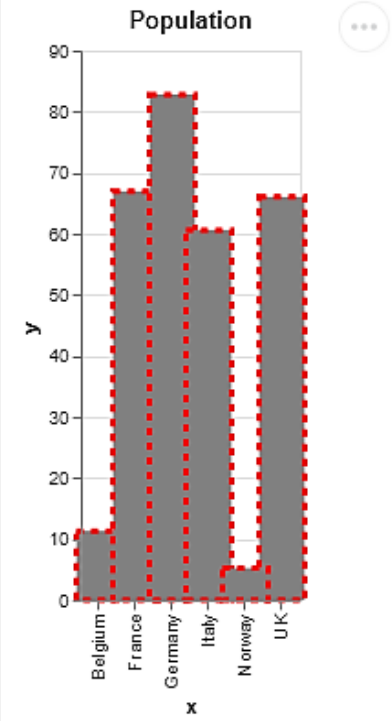

The stroke color can also be used with other marks, such as bars, as in the following example:

```
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(stroke = 'black', strokeDash=[4,4]).encode(
    x='x',
    y='y',
    color = 'x:N'
).properties(title = 'Population')
```



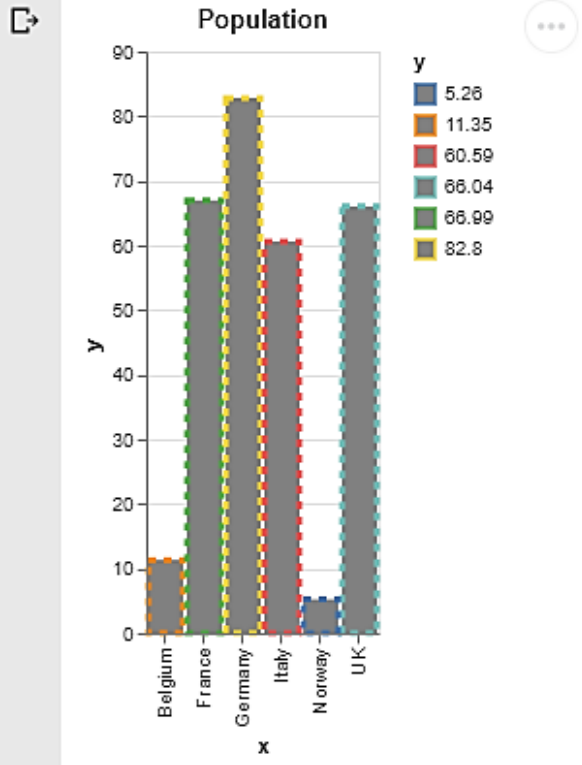

Note that we can also generate plots that are not easy to read, for instance, using colors that do not contrast enough, or by making the marks overlap unnecessarily:

```
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(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:

```
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(color = 'pink', fill = 'gray',
                         stroke ='red', strokeDash=[4,4],
                         strokeWidth=3).encode(
    x='x',
    y='y',
    stroke = 'y:N'
).properties(title = 'Population')
```




## 6.2 AGGREGATION AND BINNING

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

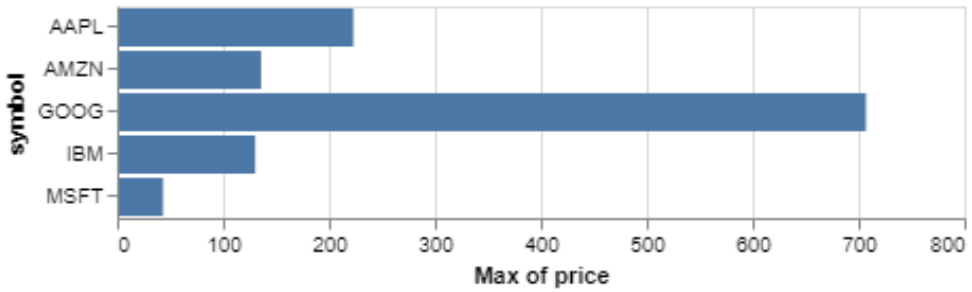
Options of *x* and *y* encodings:

- aggregate: An aggregation function is applied to the field, such as mean, sum, median, etc.

- axis: Modifies the properties of the axes.
- bin: It is used as a flag for binning quantitative fields.
- 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.
- title: Defines a title for the field.

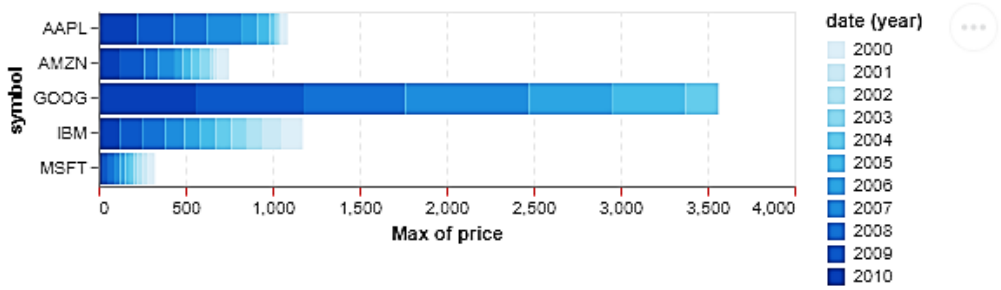
Aggregating data is simple enough, if we want to calculate the average price of each company (defined as 'symbol' in the *stocks* dataset), we can ask Altair to calculate its average from the field itself (we put the values in x to improve the space usage):

```
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',
)
```



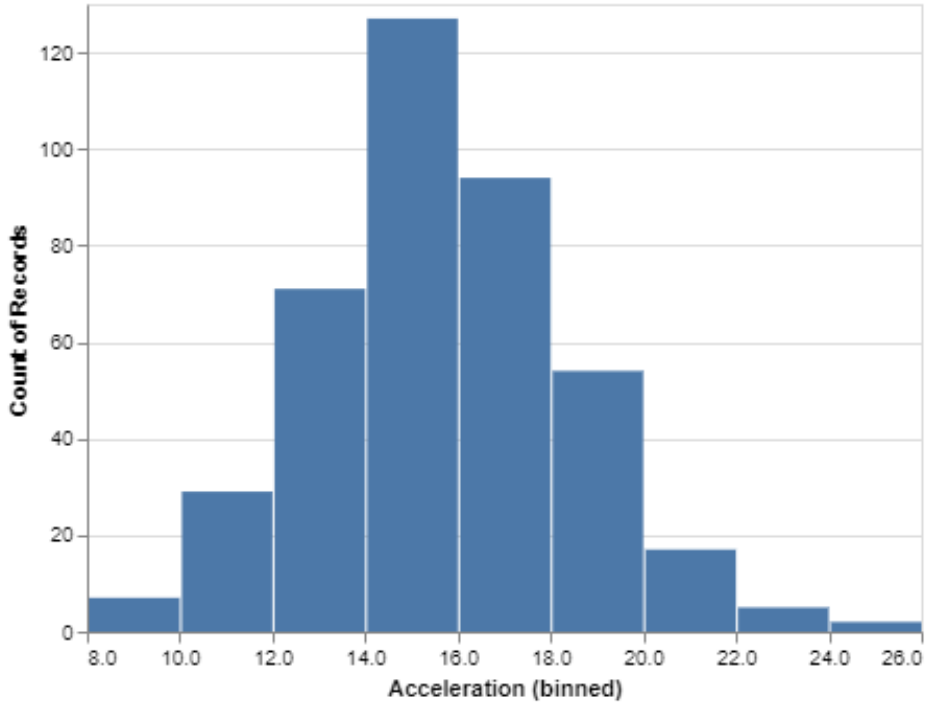

If we want to check the maximum price stock per year, we can separate per years the following way:

```
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'
)
```

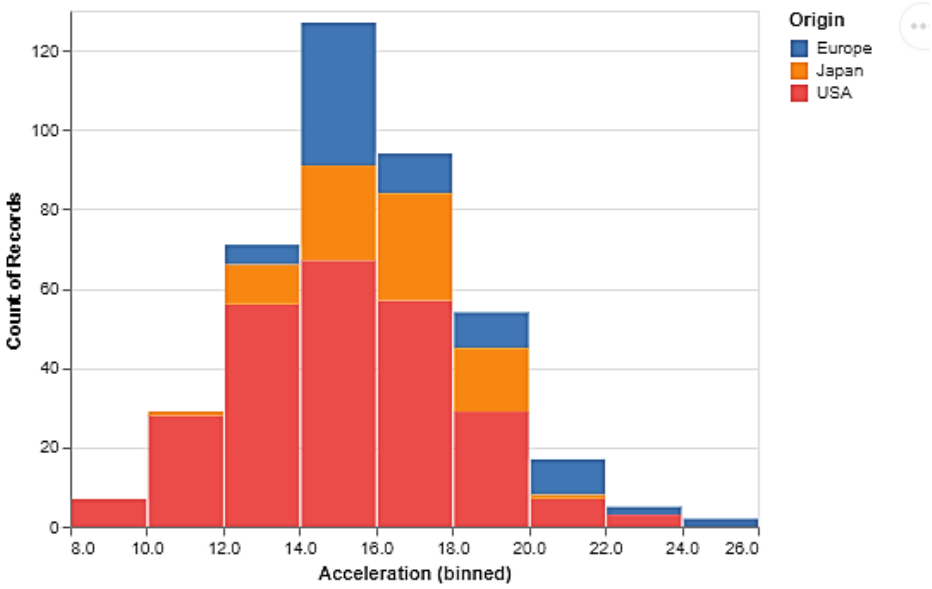



We can separate them per origin, and add a column for each origin by adding the option *column* to the plot:

```
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'
)
```

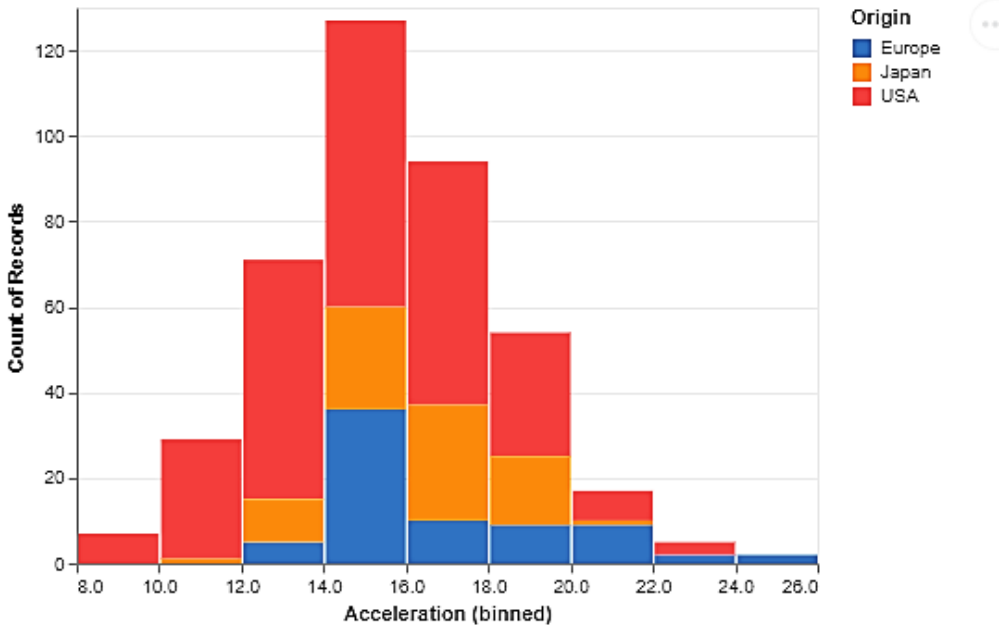
This will generate three bar charts. We could have stacked the bars:

```
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'
)
```



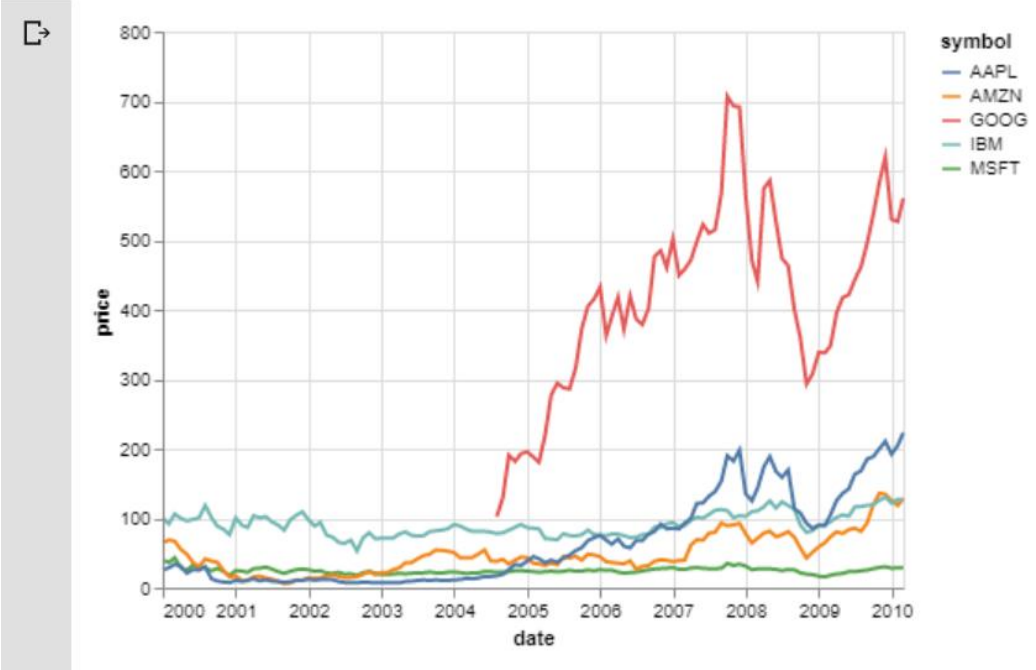

If we want to sort the segments of the bars, we can use the *sort* option:

```
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',
    order=alt.Order('Origin:N', sort='ascending')
)
```



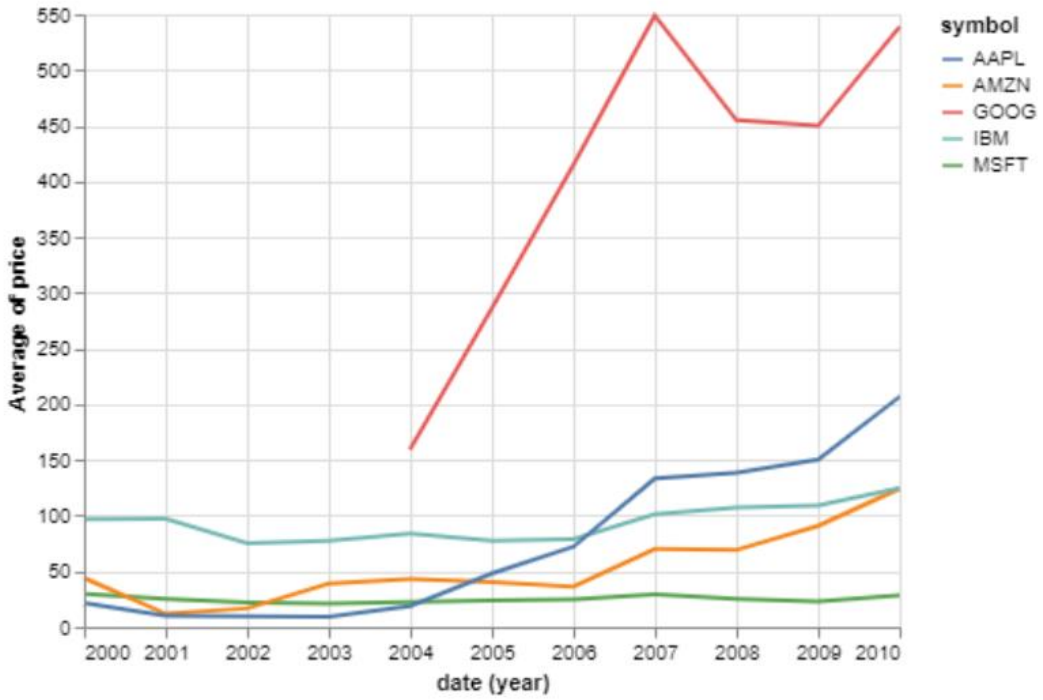

We can also bin time by using the *timeUnit* option, which allows us to group data in different time size slots. For example, the *stocks* dataset can be visualized with the fine grain data:

```
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:

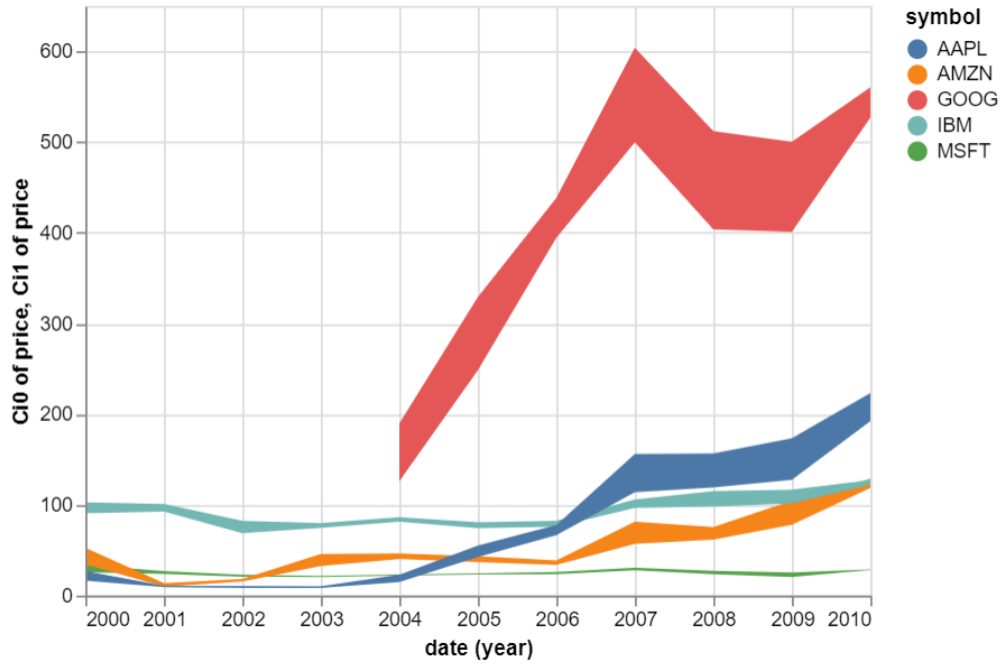
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_line(
    ).encode(
    x=alt.X('date:T', timeUnit='year'),
    y='average(price):Q',
    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*:

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

df = data.stocks.url

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

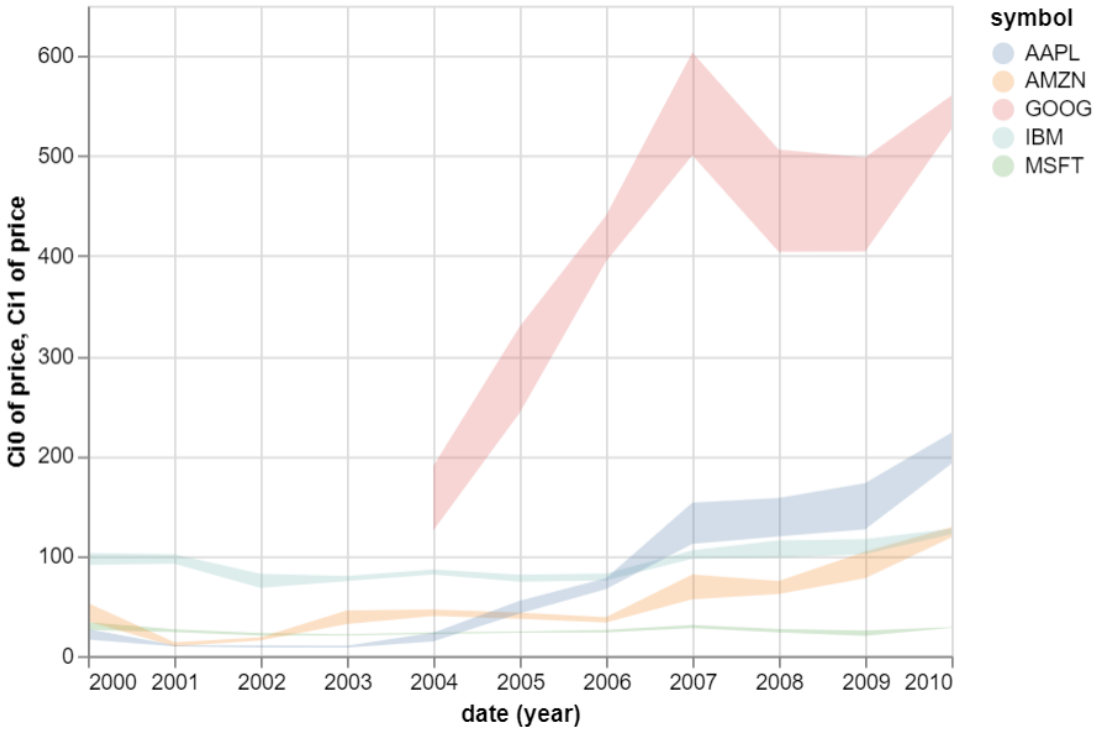
Note that now we use the area mark. And since there are values that overlap, in some regions it is not possible to see exactly where the different values start and end.



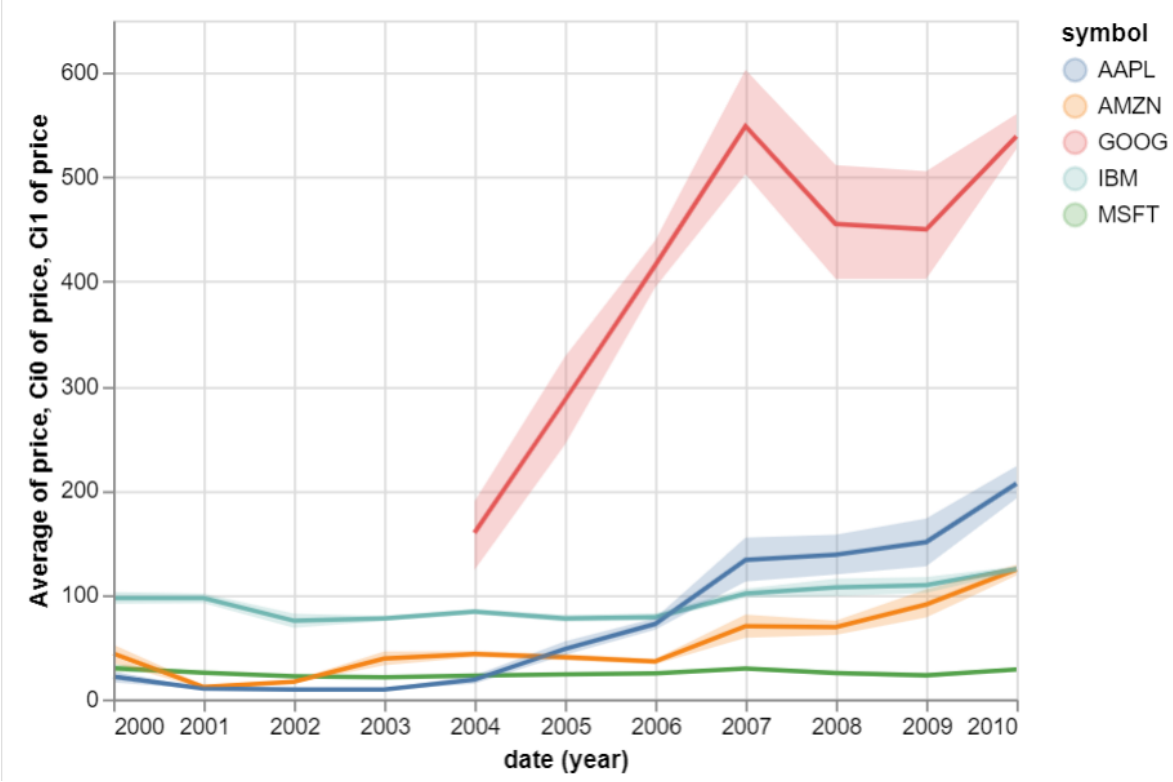

We can improve this by changing the opacity:

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

df = data.stocks.url

alt.Chart(df).mark_area(opacity=0.25
    ).encode(
    x=alt.X('date:T', timeUnit='year'),
    y='ci0(price):Q',
    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:

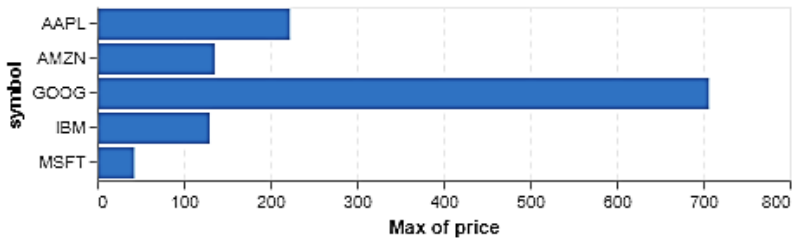



## 6.3 CUSTOMIZATION OPTIONS

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:

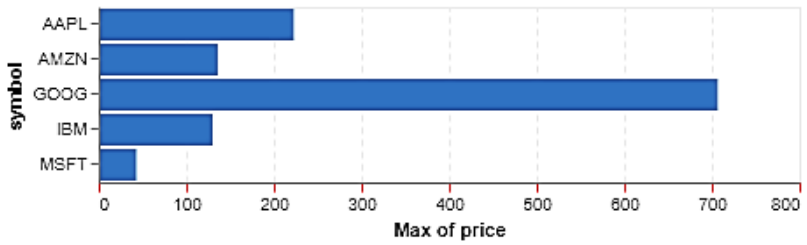
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

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




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

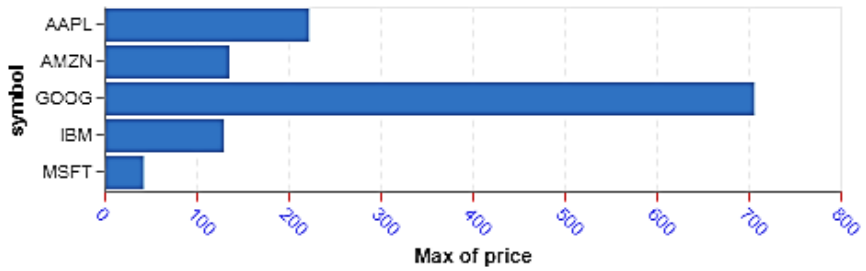
In the same manner, we can also add other features, such as colors to the ticks:

```
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',
)
```




Or we can even modify how the labels are shown:

```
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',
                          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.

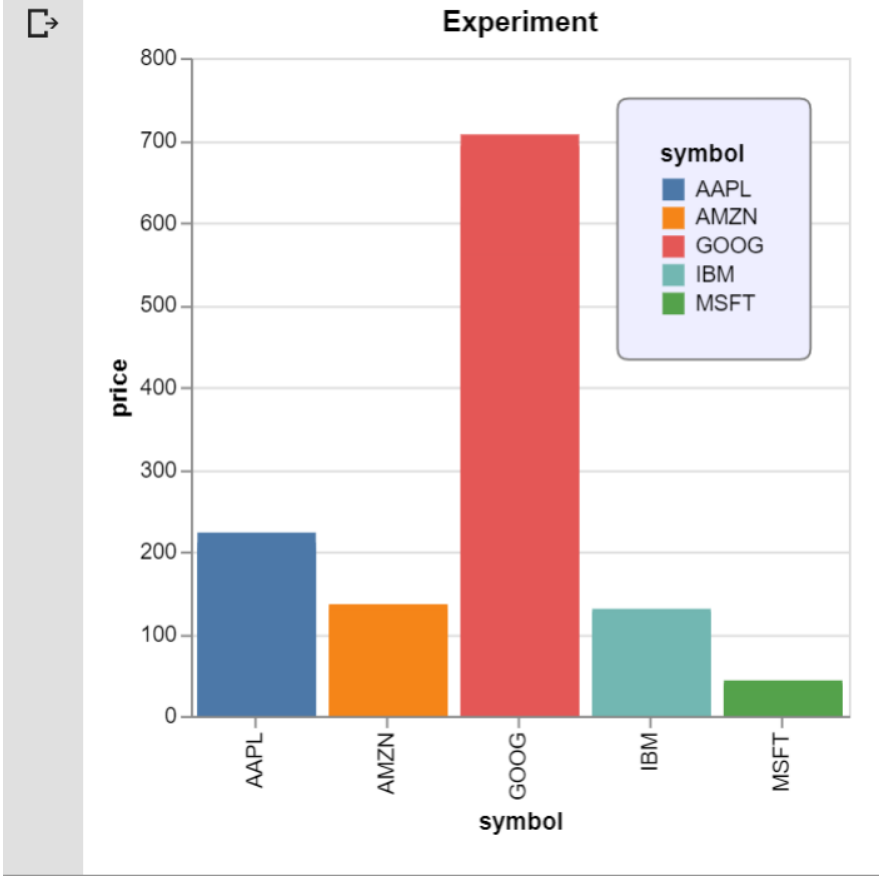
```
import altair as alt
from vega_datasets import data

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:

```
import altair as alt
from vega_datasets import data

source = data.stocks.url

alt.Chart(source).mark_bar().encode(
    x='symbol:N',
    y='price:Q',
    color = 'symbol:N'
).properties(width = 300, height = 300,
             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.

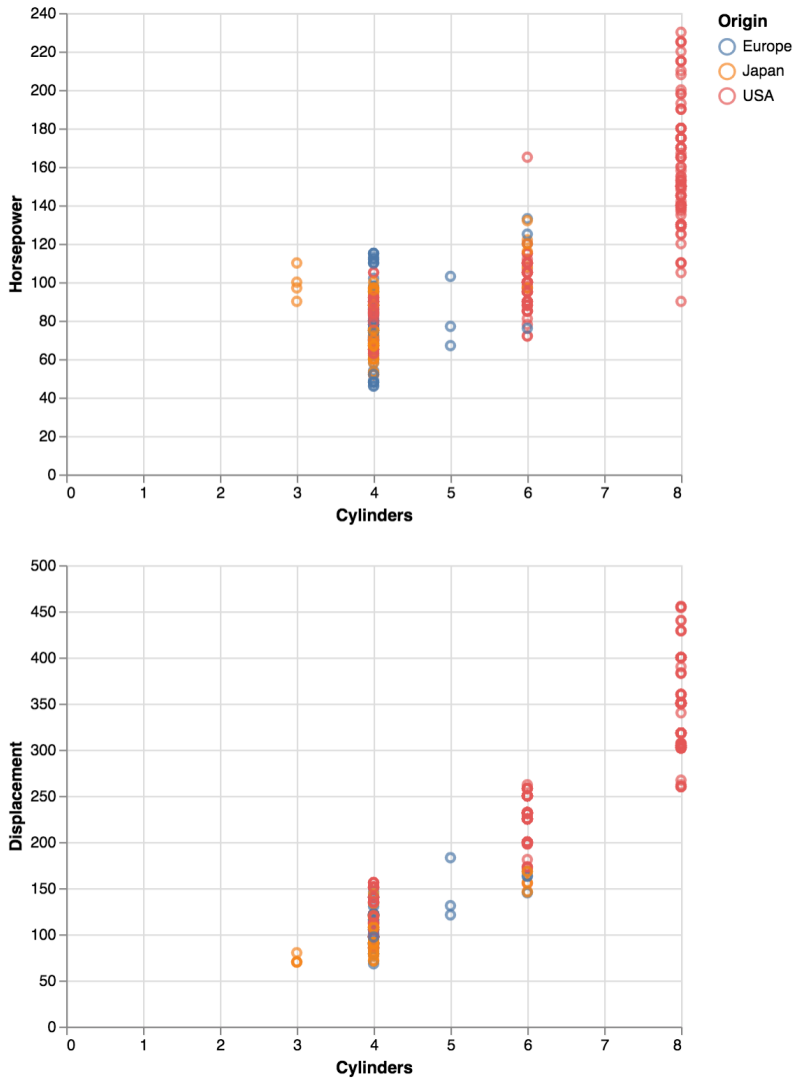
## 6.4 MULTIPLE CHARTS: SIMPLE COMBINATIONS

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:

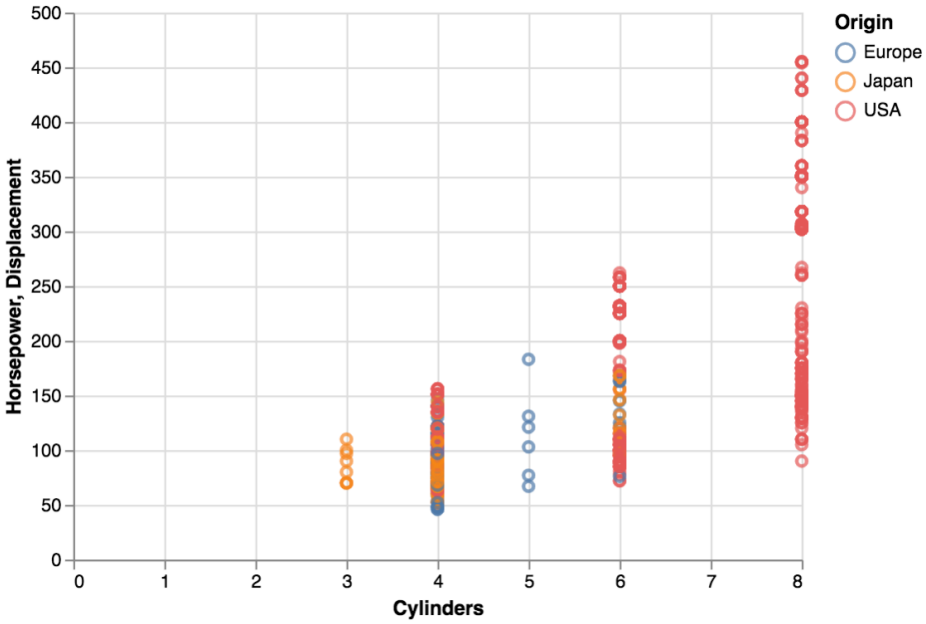
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars()

(alt.Chart(df).mark_point(shape = 'circle'
    ).encode(
    x='Cylinders:Q',
    y='Horsepower:Q',
    color='Origin'
) &
alt.Chart(df).mark_point(shape = 'circle'
    ).encode(
    x='Cylinders:Q',
    y='Displacement:Q',
    color='Origin'
))
```

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:



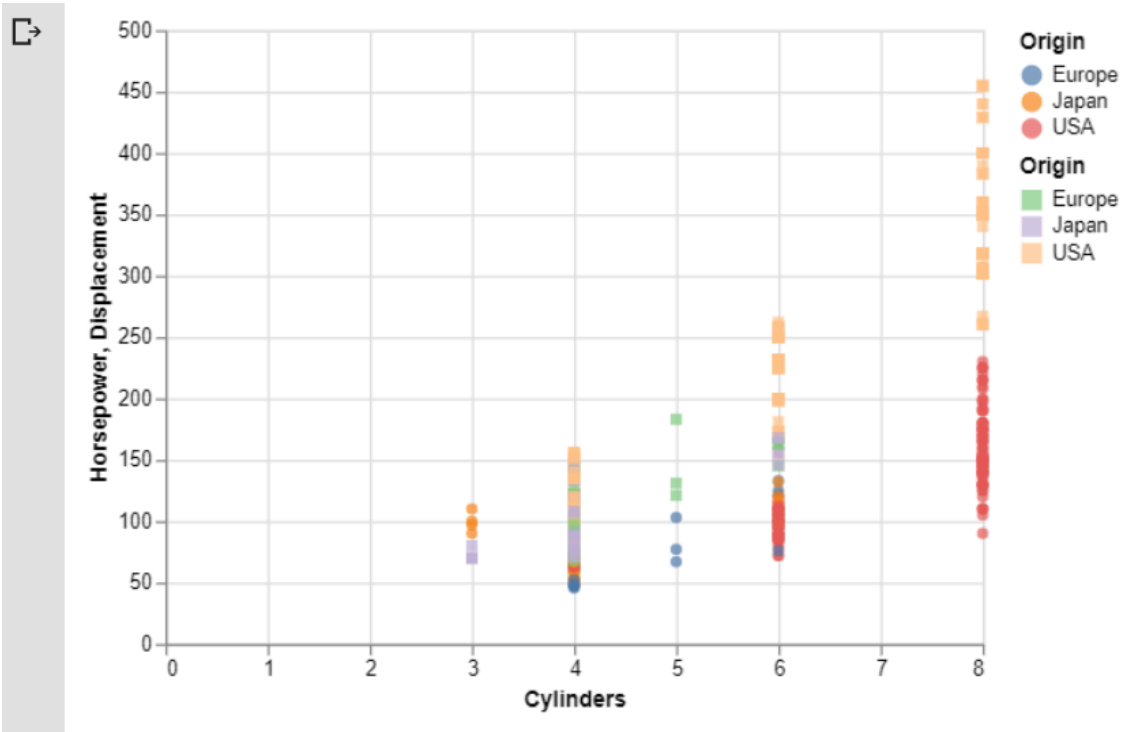

In this case, there is ambiguity since we are using color encodings of the data that are shared among both plots. If they are indicating different data, marks must be different, to avoid confusion.

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:

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

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:

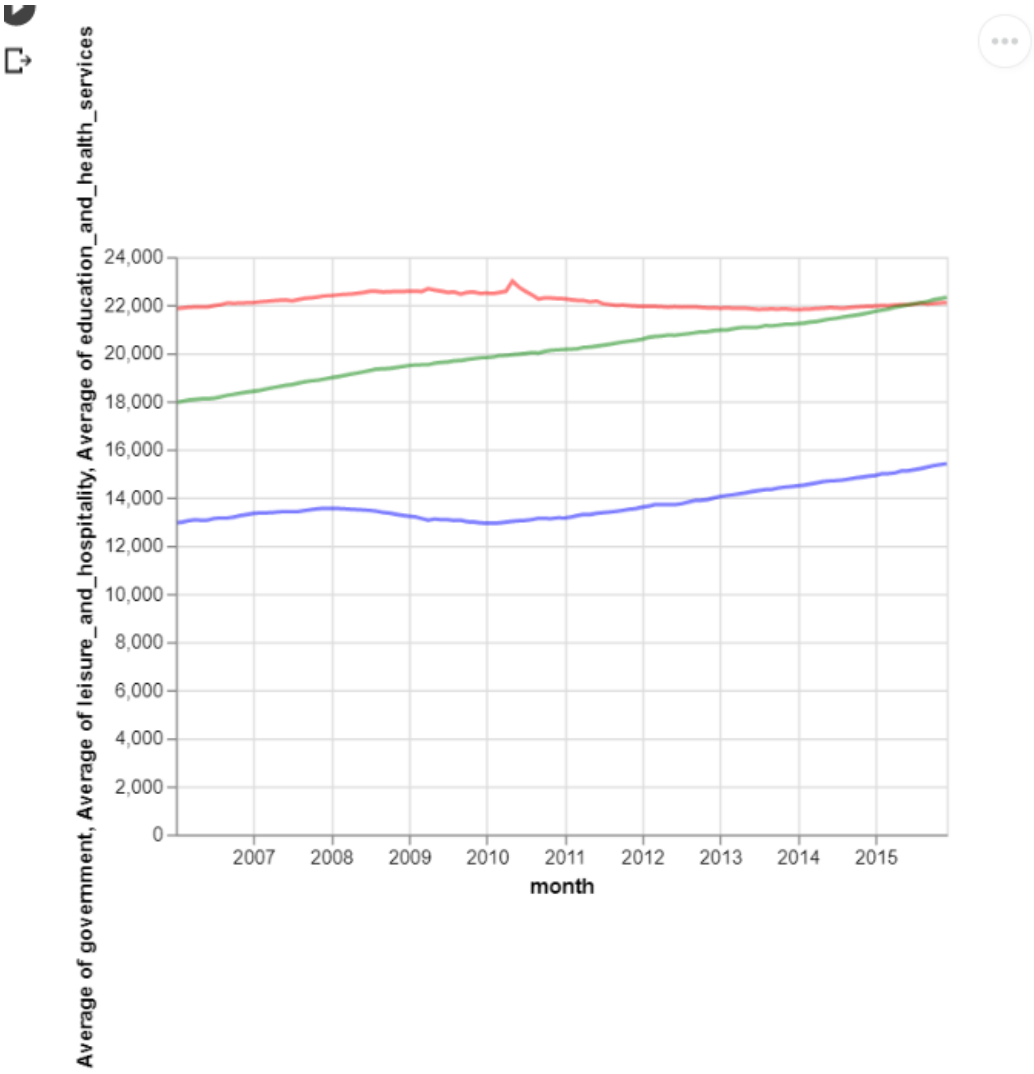



Layers can be used to visualize wide form data, such as the *stocks* dataset:

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

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:

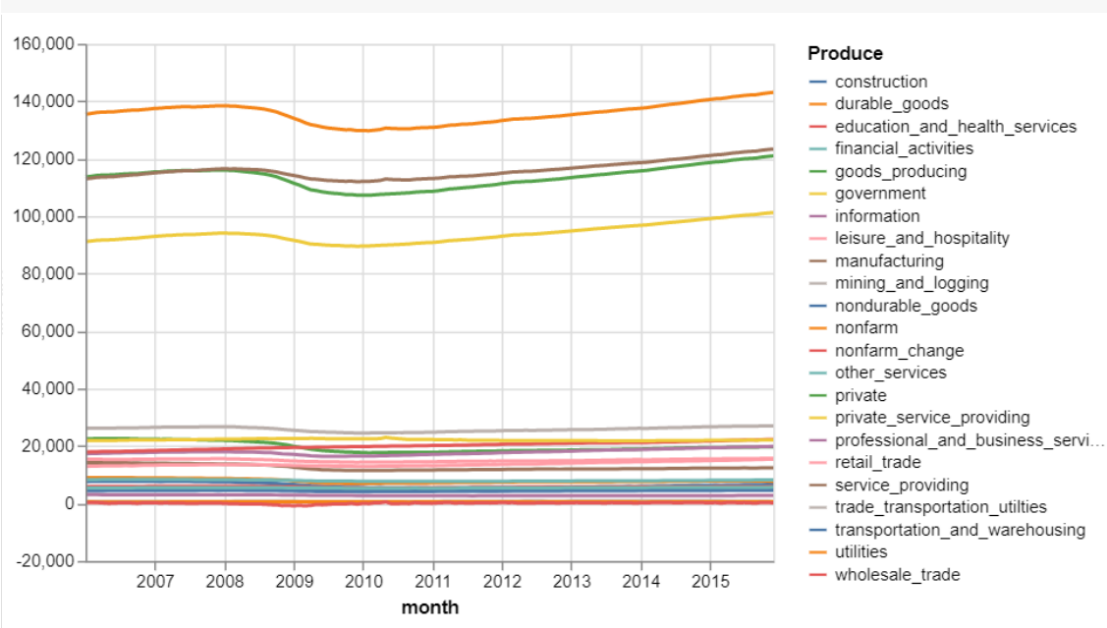



This is far less convenient than when data is in long form.

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:

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

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',
)
```

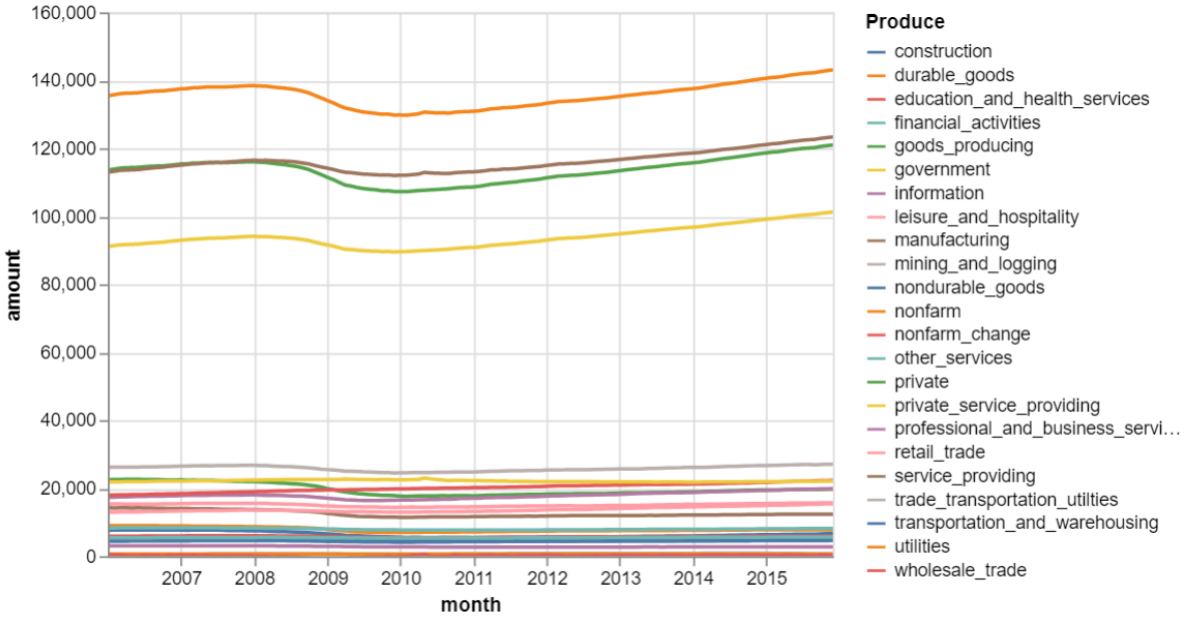
This would result in the following chart:




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:

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

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.



# 7. 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.

## 7.1 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*).

## 7.2 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='%')),
    color='sex:N'
).transform_filter(
    alt.datum.job == 'Janitor'
)
```

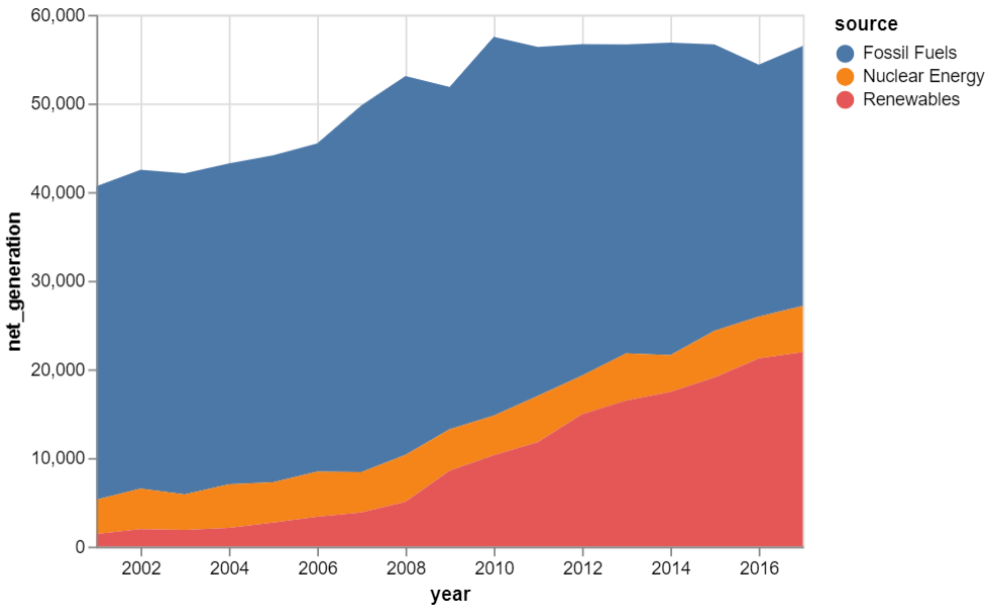
Otherwise, the Y axis would have values such as 0.018, instead of 1.8%.

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:

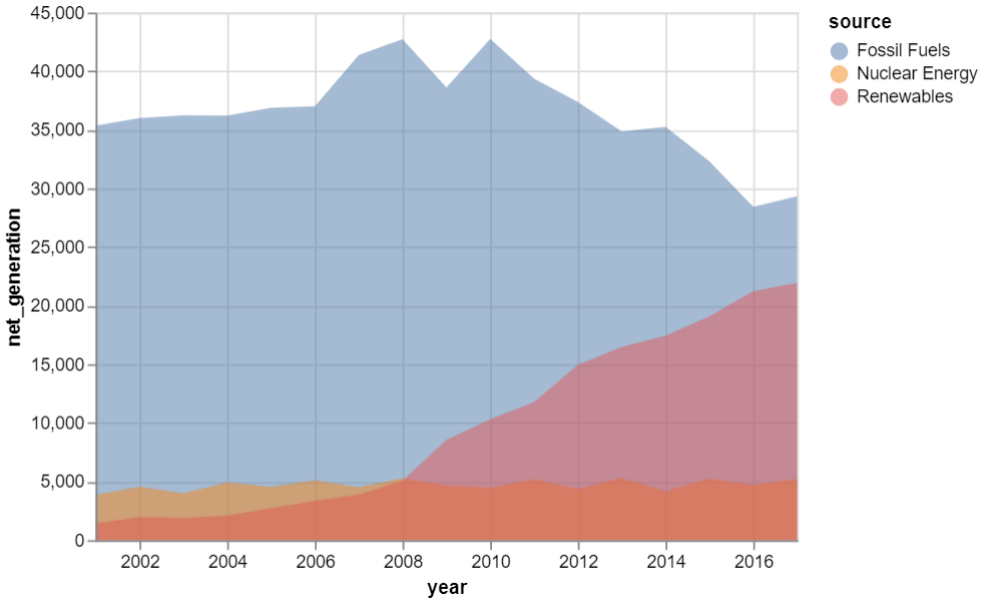
```
source = data.iowa_electricity.url

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

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




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

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

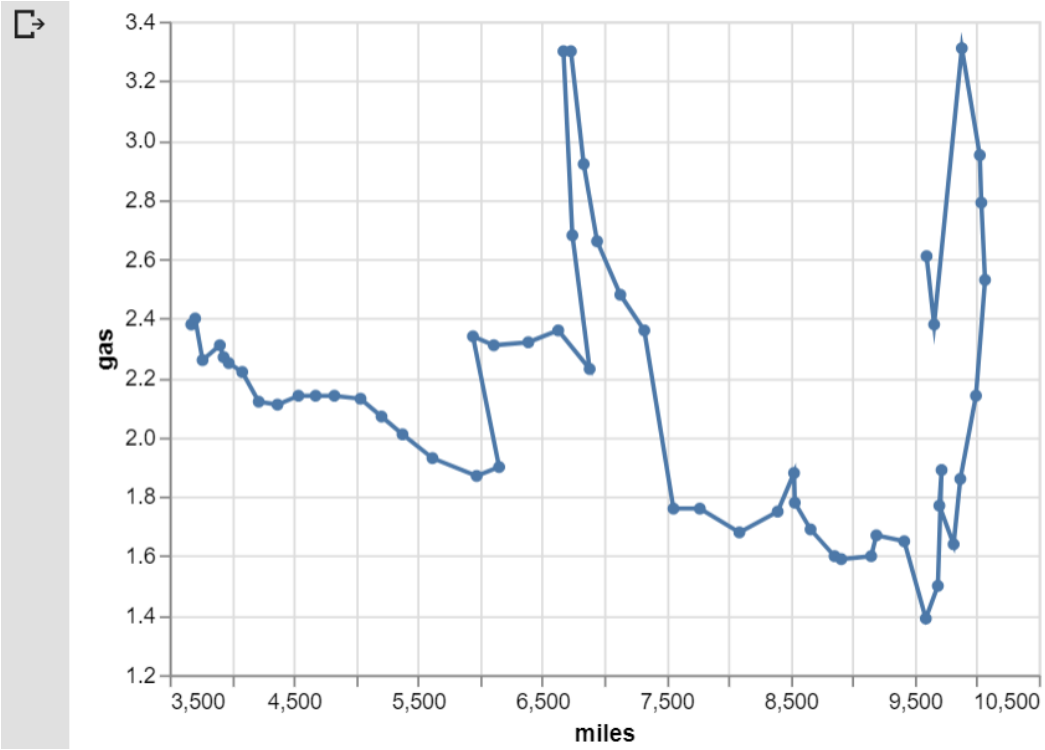
```
alt.Chart(source).mark_area().encode(
    x = 'year:T',
    y = 'net_generation:Q',
    color='source:N',
    row='source:N'
)
```

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: 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:

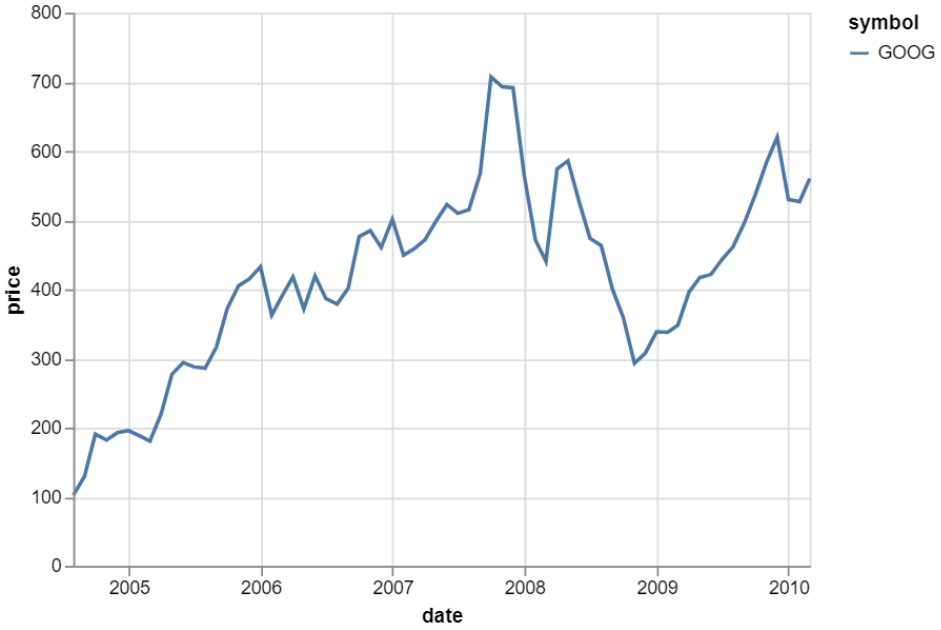
```
import altair as alt
from vega_datasets import data

driving = data.driving()

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

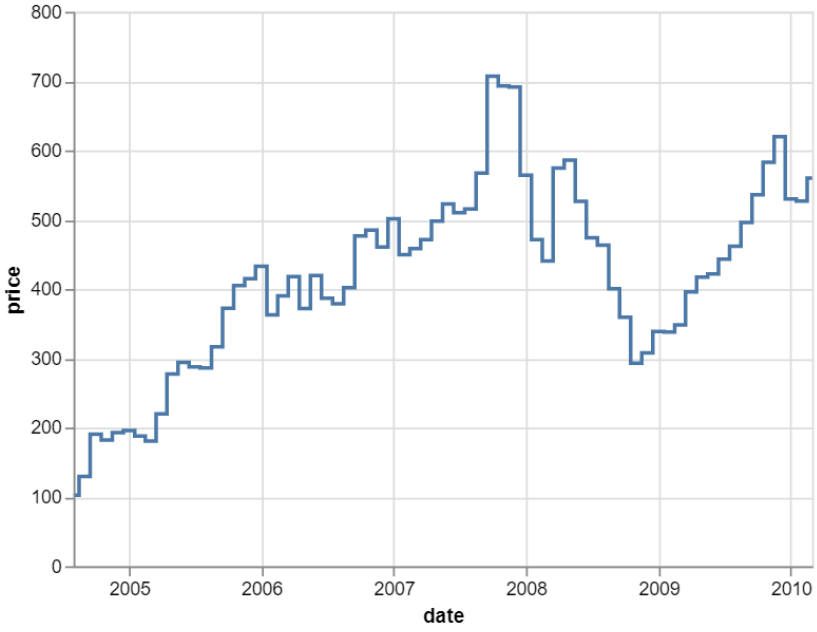
The result would be:




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:



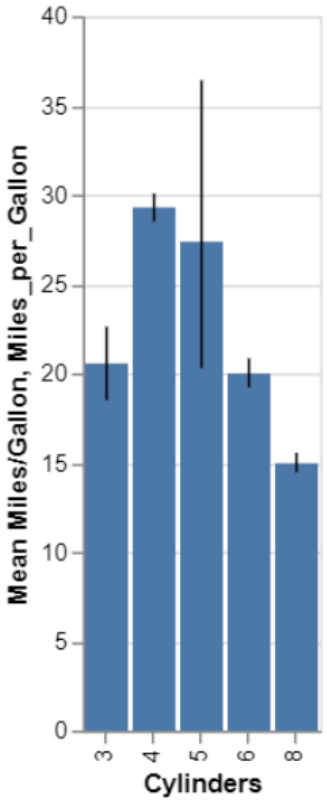




Other interpolation options include *linear*, *linear-closed*, *step*, *step-before*, *step-after*, *basis*, *basis-open*, *basis-closed*, *cardinal*, *cardinal-open*, *cardinal-closed*, *bundle*, and *monotone*.

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:



## 7.3 ADVANCED CHART TYPES

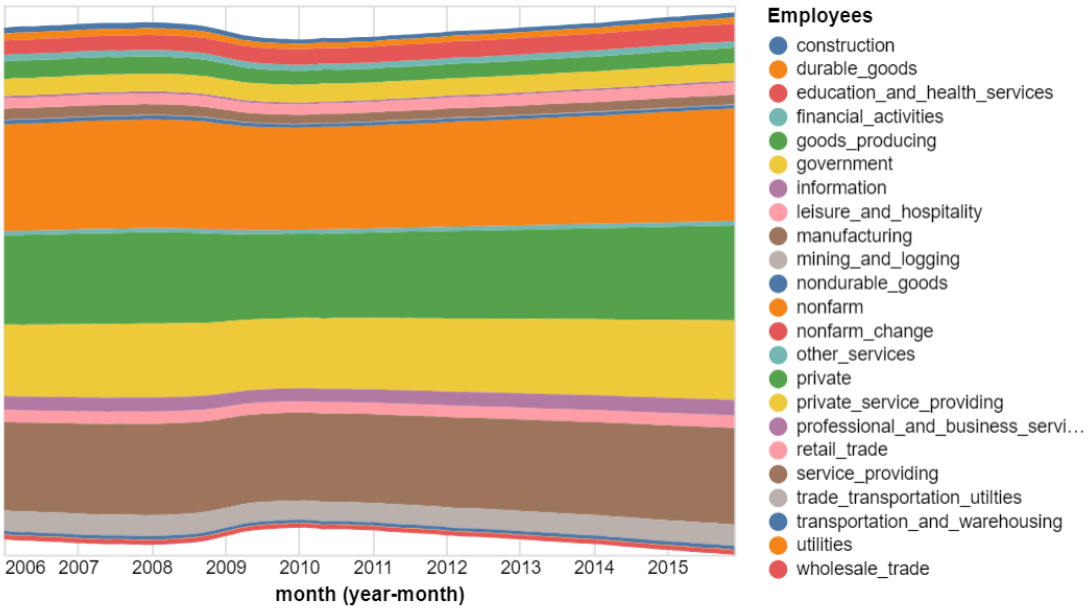
Besides those simple charts and its derived versions, there are other, more sophisticated charts that are focused on showing a certain type of information.

**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:

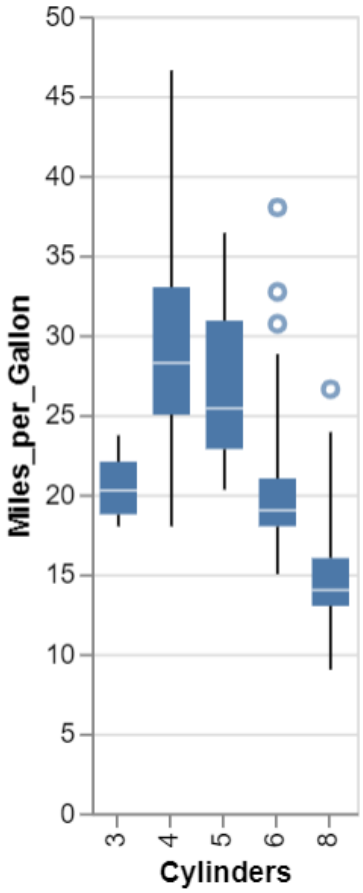
```
import altair as alt
from vega_datasets import data

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 75$^{th}$ and 25$^{th}$ 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:

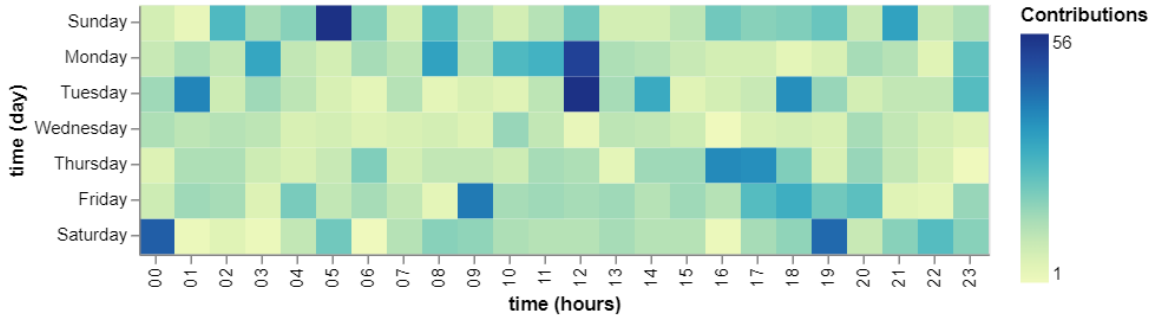



We could even ignore the outliers totally by defining the extend to be equal to the maximum – minimum values: *extent = 'max – min'*.

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:

```
import altair as alt
from vega_datasets import data

source = data.github.url

alt.Chart(source).mark_rect().encode(
    x='hours(time):O',
    y='day(time):O',
    color=alt.Color('sum(count):Q',
                    legend=alt.Legend(title='Contributions'))
)
```

And the result will be the following heatmap:

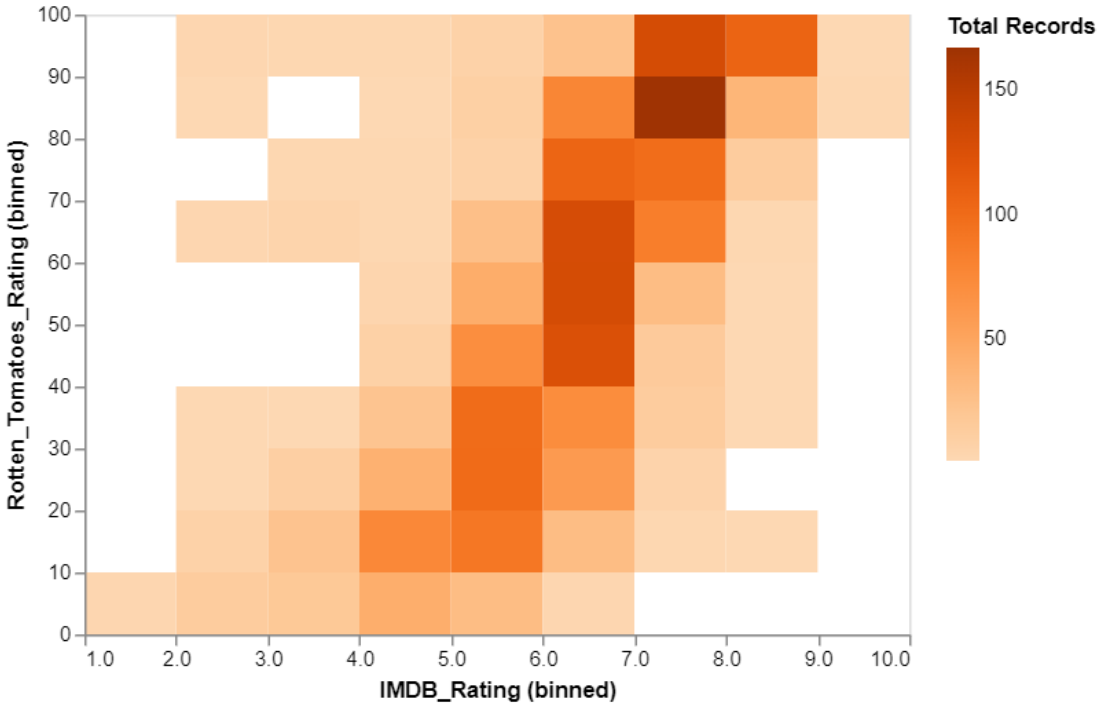



Note that we changed the legend title in the *color* field to ensure that the name makes sense. Otherwise, it would have read "Sum of count".

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

```
source = data.movies.url

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

And the result is:

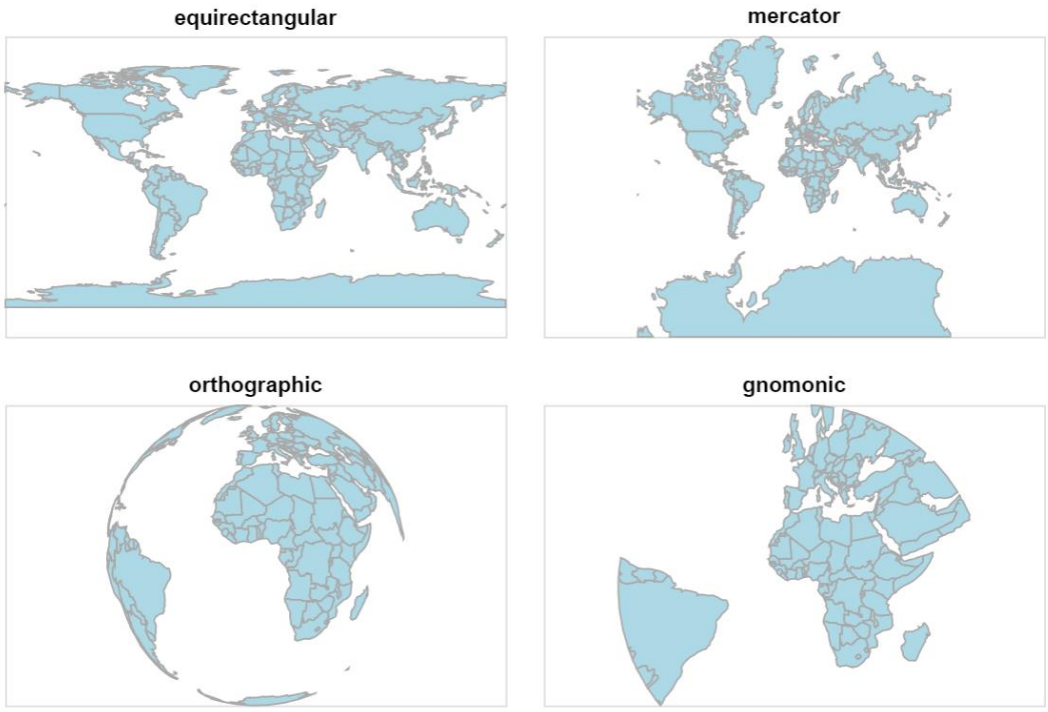



**Maps** are used to visualize **all kinds of information that is linked to geographic positions**, from routes, to flows, or demographic or economic datasets, the use of maps is widespread. With Altair, they can be generated using the *mark_geoshape* type. However, its usage is more cumbersome than 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:

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

background = alt.Chart(states).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

points = alt.Chart(airports).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']
).properties(
    title='Number of airports in US'
)
```

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.

Note that here, 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.


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

# 8. Data transformation

## 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_lookup* performs a one-sided join of two datasets based on a lookup key.

- *transform_timeunit* discretizes a date by a time unit (day, month, year, etc.)
- *transform_regression*: generates a regression line.

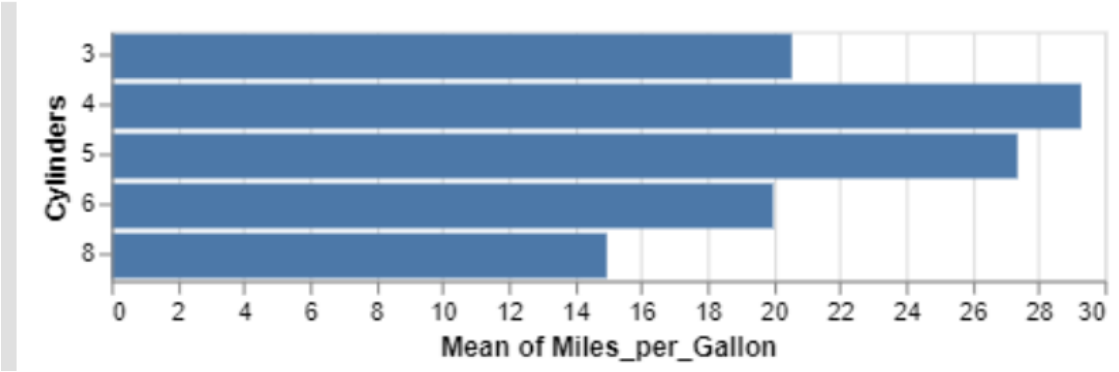
## 8.2 AGGREGATE TRANSFORMS

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:

```
import altair as alt
from vega_datasets import data

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:

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

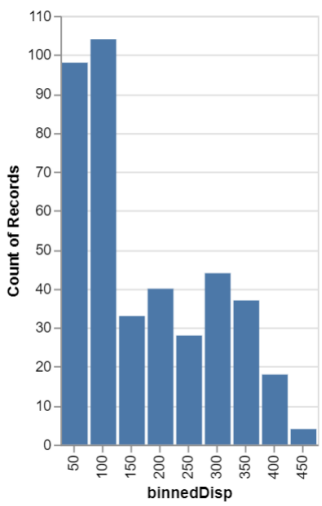
Like in the previous case, we can create bin transforms through the encoding and by explicitly defining it. So, the following code:

```
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:

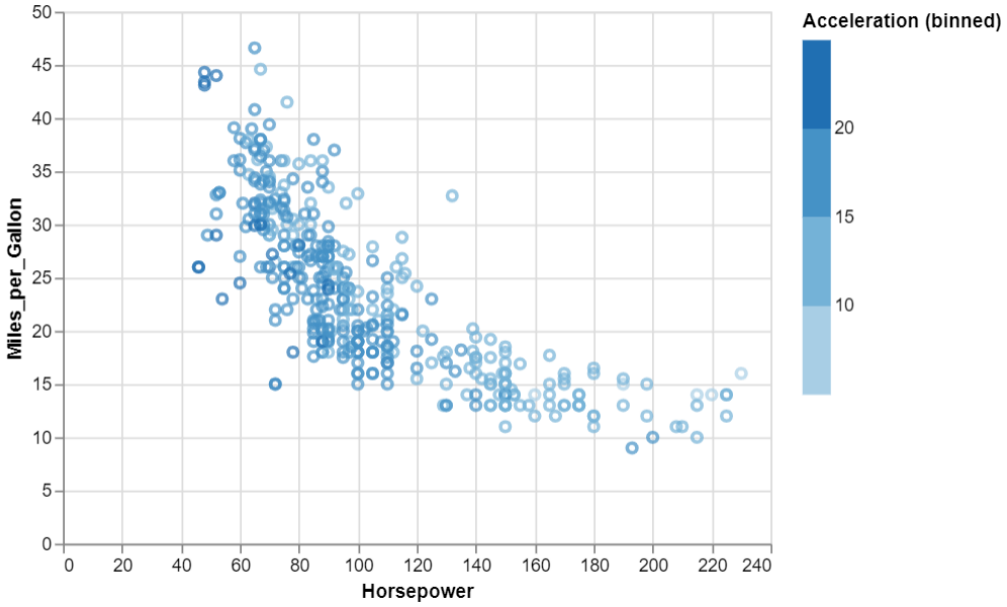



With the only difference being the name of the X axis (that can be changed in any case).

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

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

Which would render:




And here is the transformed color scale using a top-level bin transform:

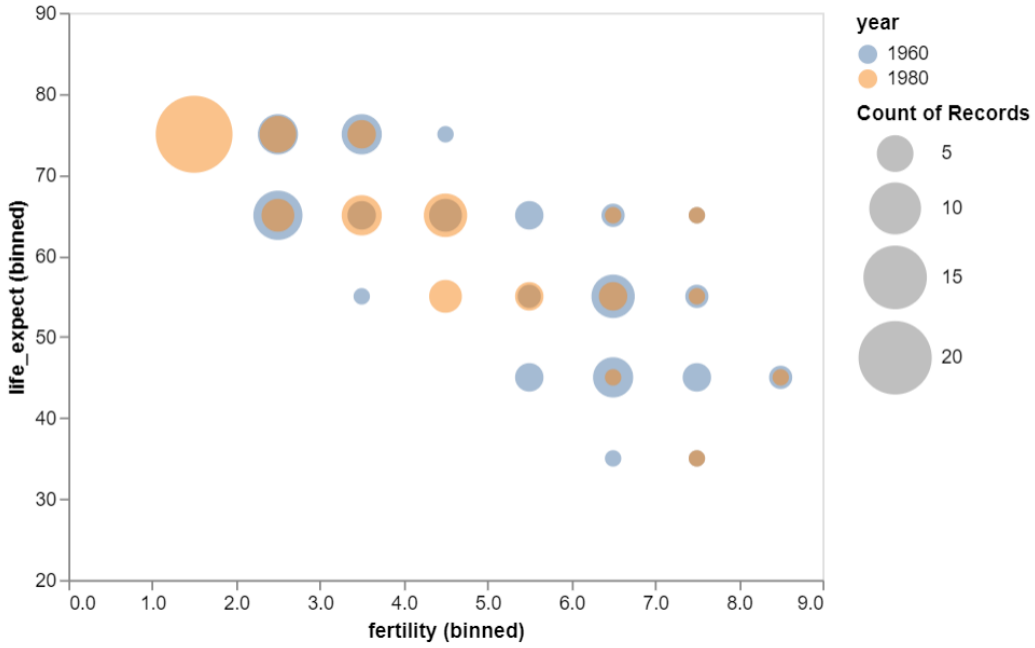
```
alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.Color('Acceleration:Q', bin=alt.Bin(maxbins=5))
)
```

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.

```
le = alt.Chart(df).mark_circle(opacity=0.5).encode(
    alt.X('fertility:Q', bin=True),
    alt.Y('life_expect:Q', bin=True),
    size = 'count()',
    color = 'year:N'
)

alt.layer(le.transform_filter(alt.datum.year == 1960),
          le.transform_filter(alt.datum.year == 1980))
```

The result is this plot:



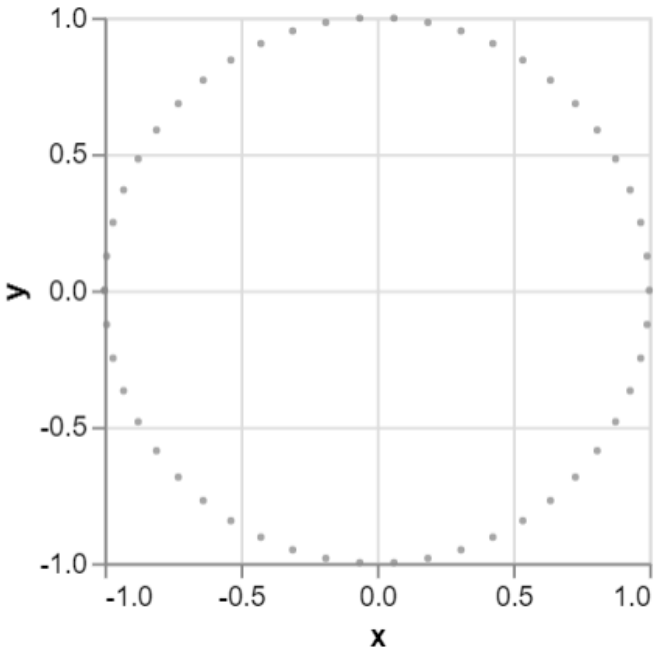

## 8.4 TRANSFORMING DATA THROUGH CALCULATIONS

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:



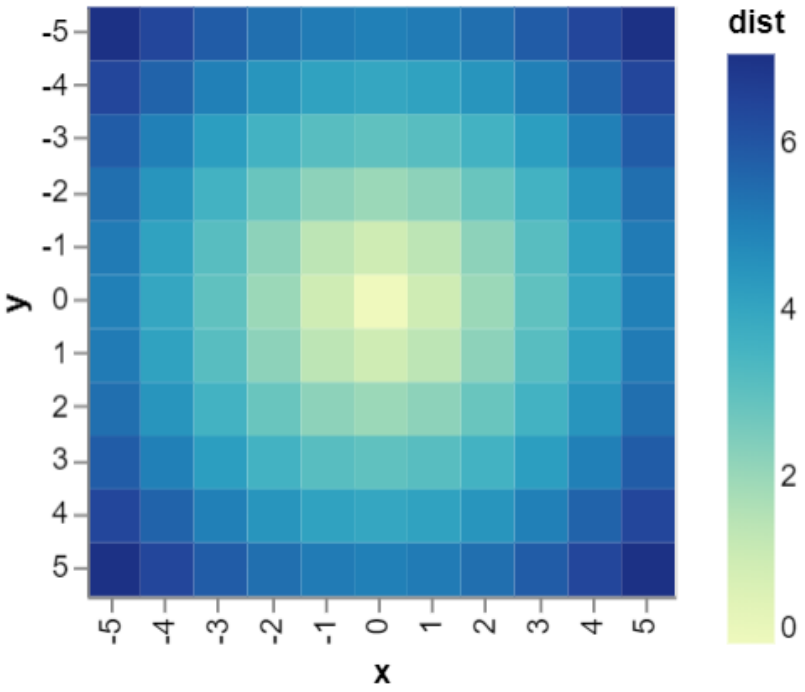

Take into account that the sizes of the plot are decided by Altair, so if we do not impose a square shape, in this case, the chart appears deformed.

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:

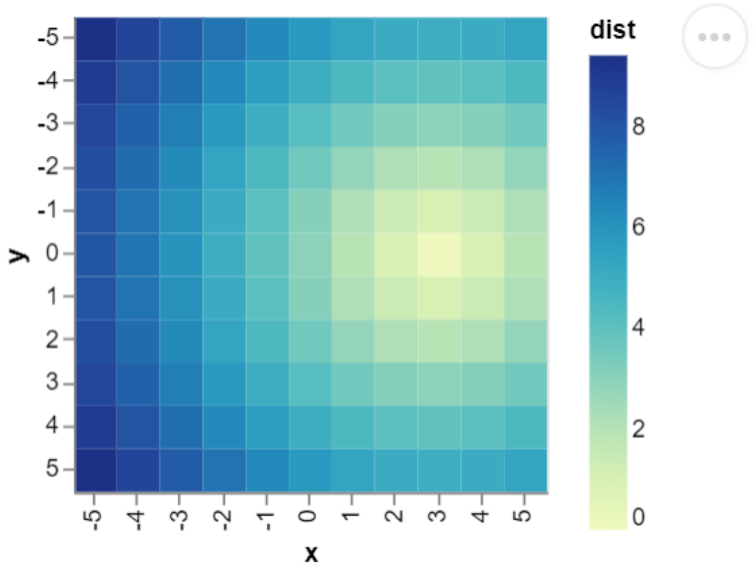
```
import altair as alt
import numpy as np
import pandas as pd

# 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.

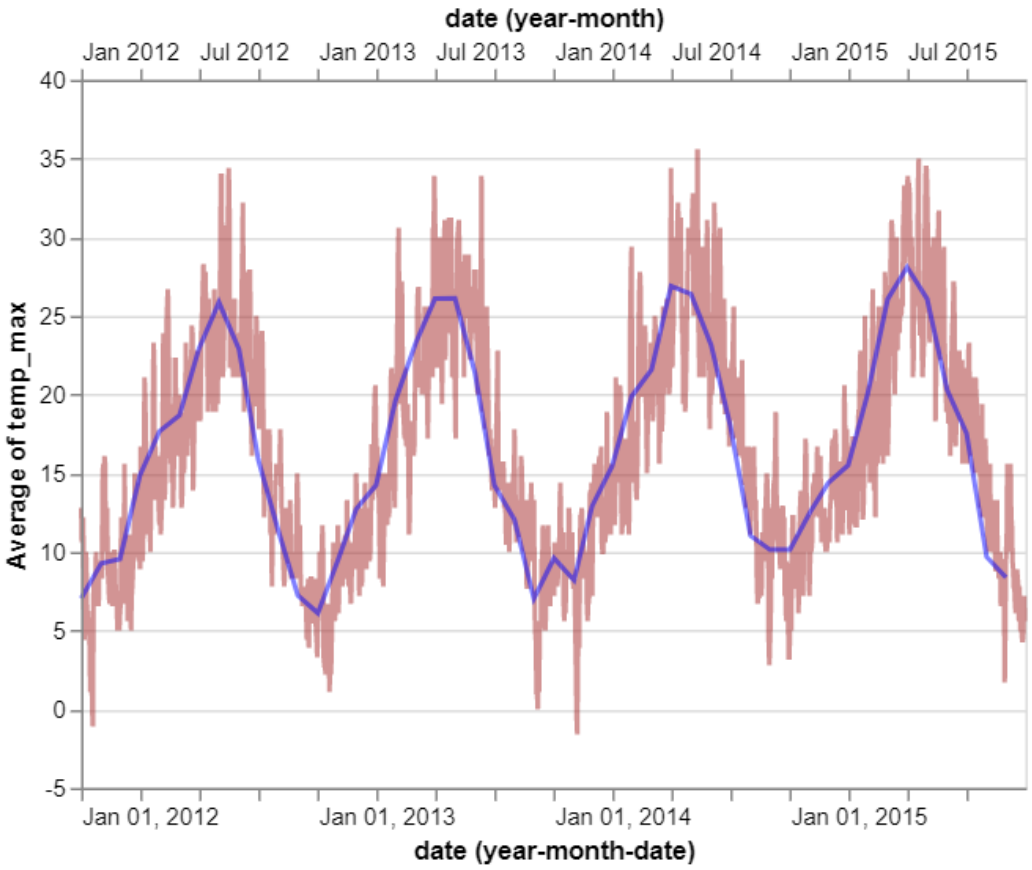
The following example uses the *weather.csv* dataset from the Vega datasets, uploaded manually:

```
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.

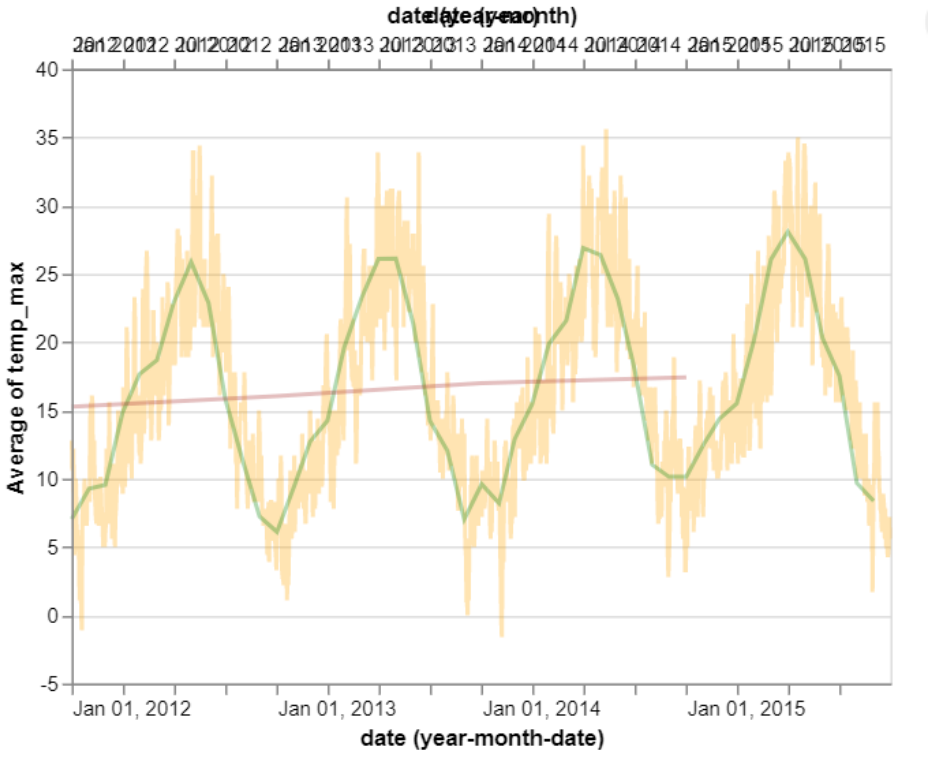
The result is shown next:




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')
```

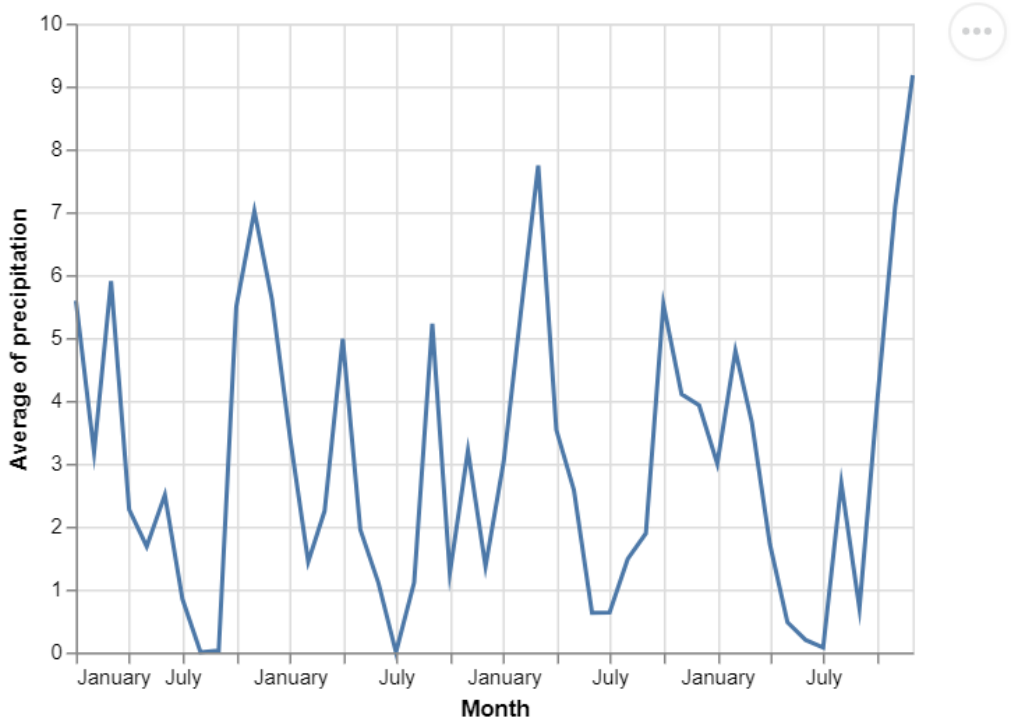
The result, with the two top axis on top of each other is:



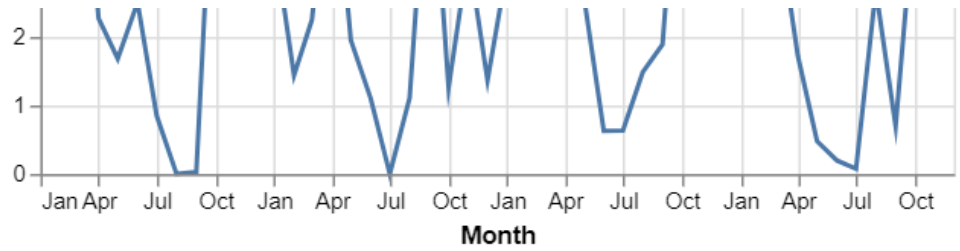
Time unit transformations can also be carried out using the transform_timeunit function. This can be used to assign a new name to the calculated field so that it can be reused.

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

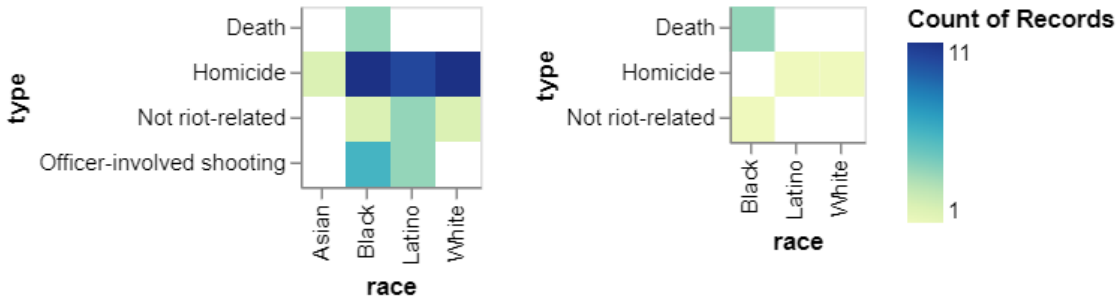
This transformation eliminates part of the data following a specified criterion. In order to specify the criterion, we can use the *datum* object, that refers to the

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')
```

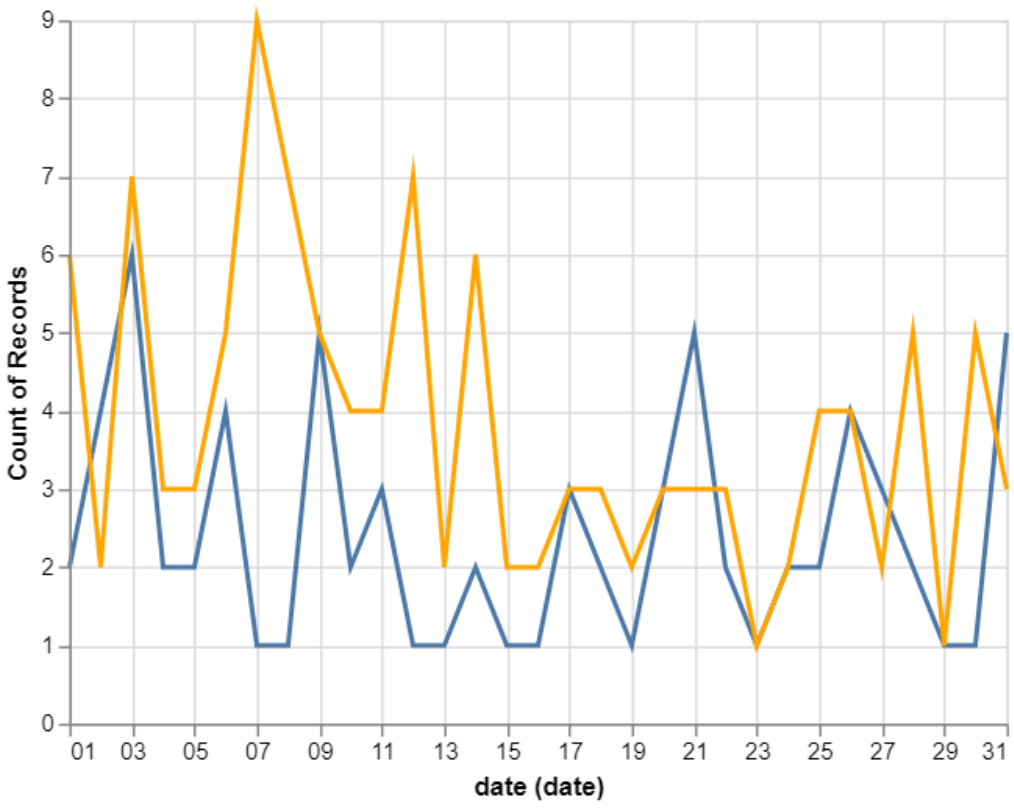
This way, we are filtering the gender and the first plot will show the data corresponding to males, and the second to females.




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.

```
ch1 = alt.Chart(df).mark_line().encode(
    x='date(date):T',
    y='count(destination):O',
).transform_filter(
    alt.datum.origin == 'LAX')

ch2 = alt.Chart(df).mark_line(color = 'orange').encode(
    x='date(date):T',
    y='count(destination):O',
).transform_filter(
    alt.datum.origin == 'HOU')
```

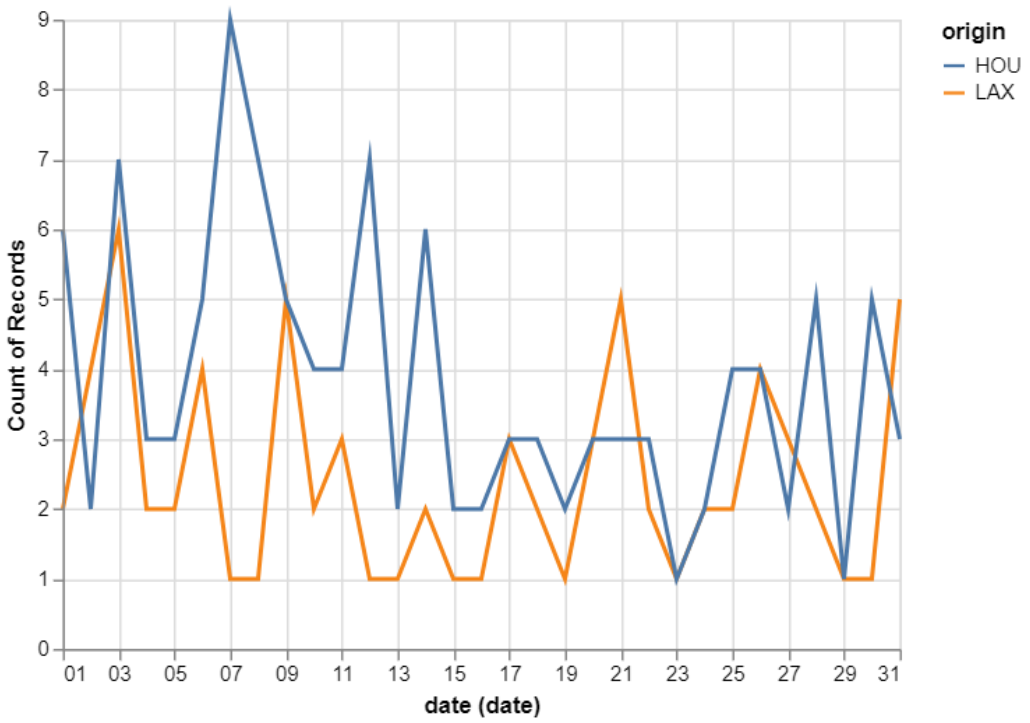
The result would be:




Since each chart plots a single line, there is 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. Note that the origin is fixed for each chart. This would result in the following code and plot:

```
ch1 = alt.Chart(df).mark_line().encode(
    x='date(date):T',
    y='count(destination):O',
    color = 'origin:N'
).transform_filter(
    alt.datum.origin == 'LAX')

ch2 = alt.Chart(df).mark_line(color = 'orange').encode(
    x='date(date):T',
    y='count(destination):O',
    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* = ‘’.

## 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.

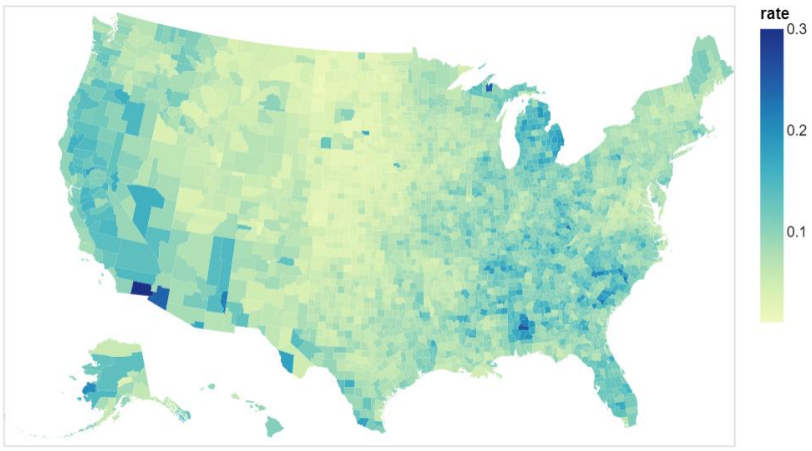
First, we will load the US map representation from the *us_10m* dataset, and then, we will look up for the employment rate in the *unemployment* file by performing a *transform_lookup* operation. This operation has two parameters, the data origin, and the search function, which are stated as two different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

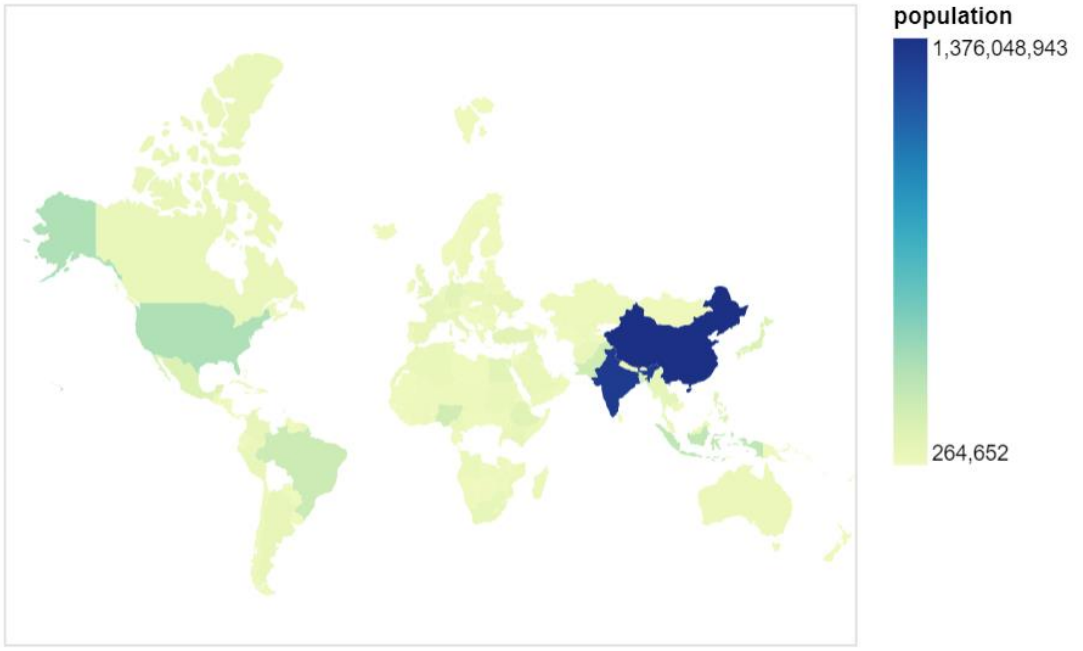
To load the geographic data, we only need to call the proper function:

```
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

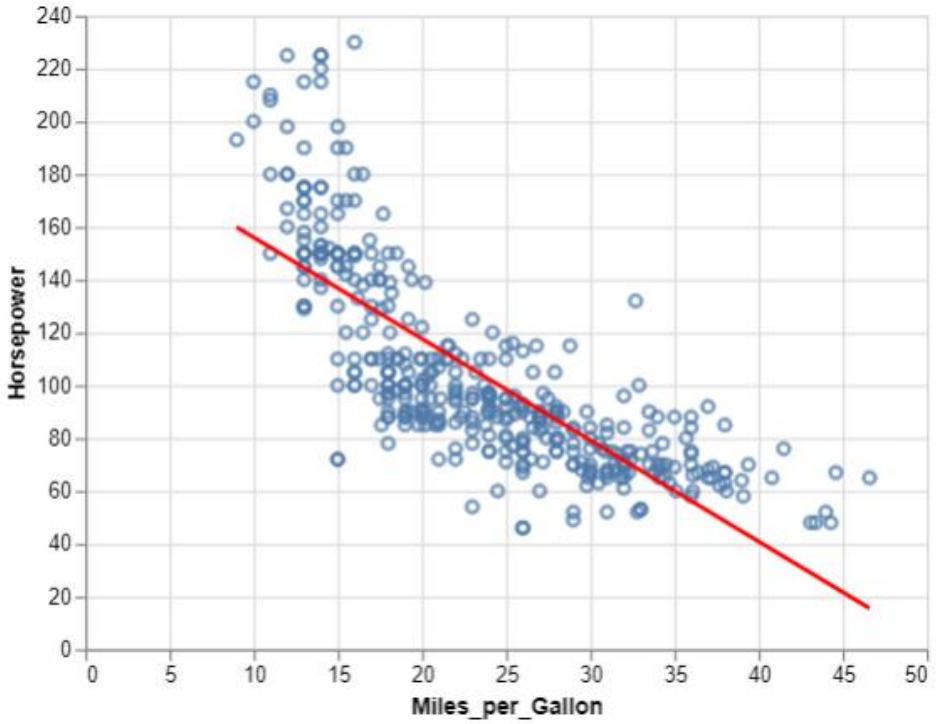
The regression transform may fit a two-dimensional regression model to smooth and predict data. The transform can fit multiple models for input data and generate new data objects that represent points for summary trend lines.

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

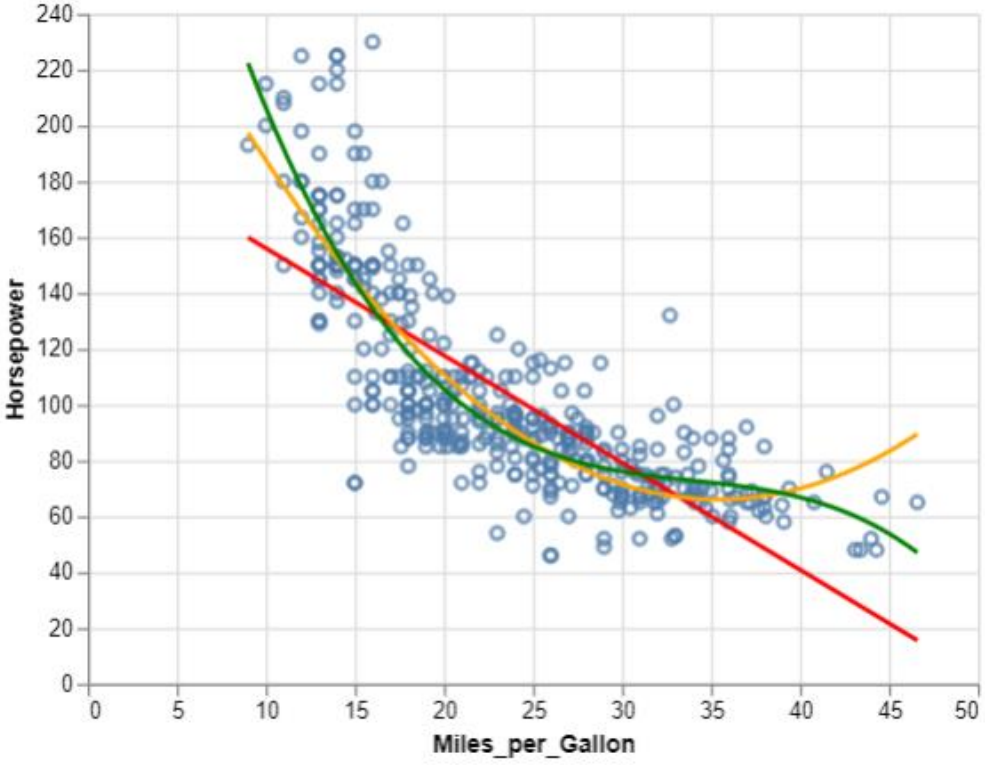
The following example shows a linear regression based on the *cars* dataset:

```
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)
```

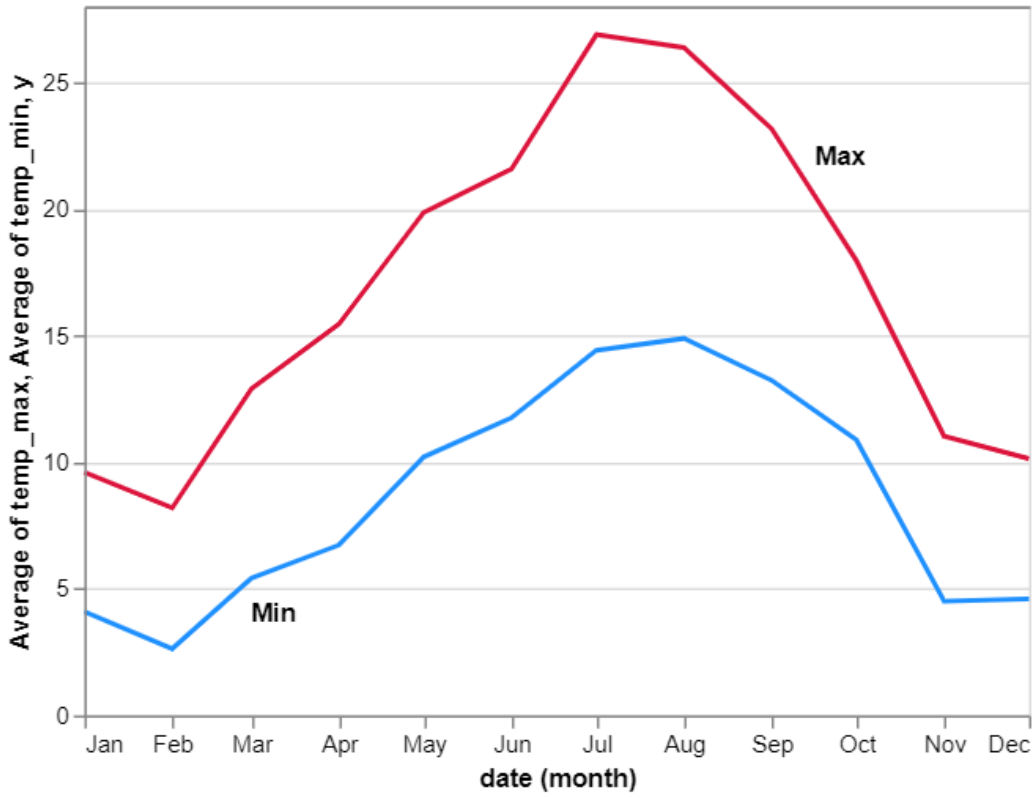
Note the data of the positions is synthetic and takes into account the positions we want to put the labels, as well as the size of the virtual space we will be plotting to.

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.

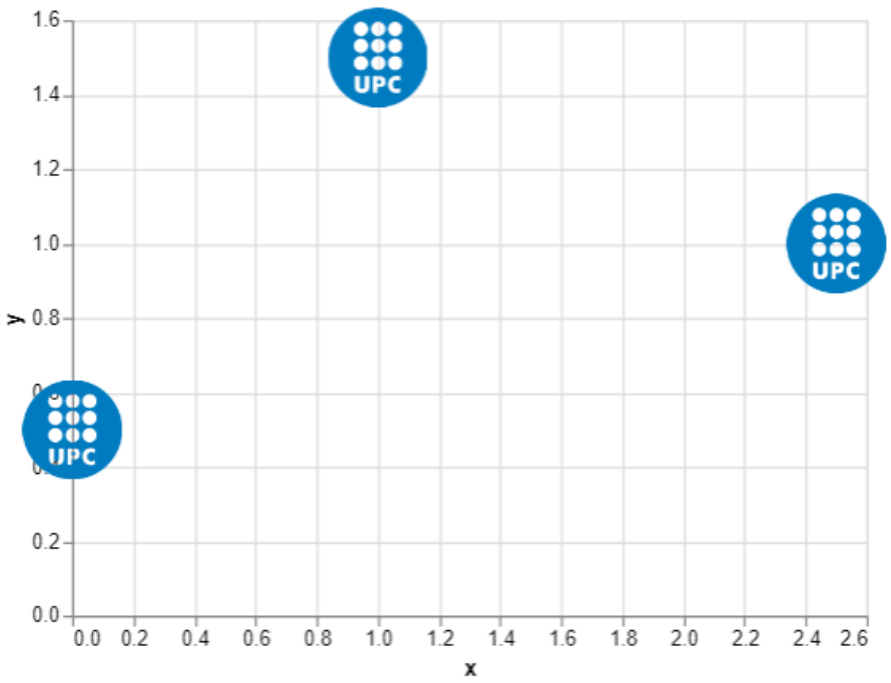
By setting the *labels* parameter as false, labels will not appear, and by setting the parameter *ticks* as false, will remove the ticks from the axis.

## 9.4 PLOTTING 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:



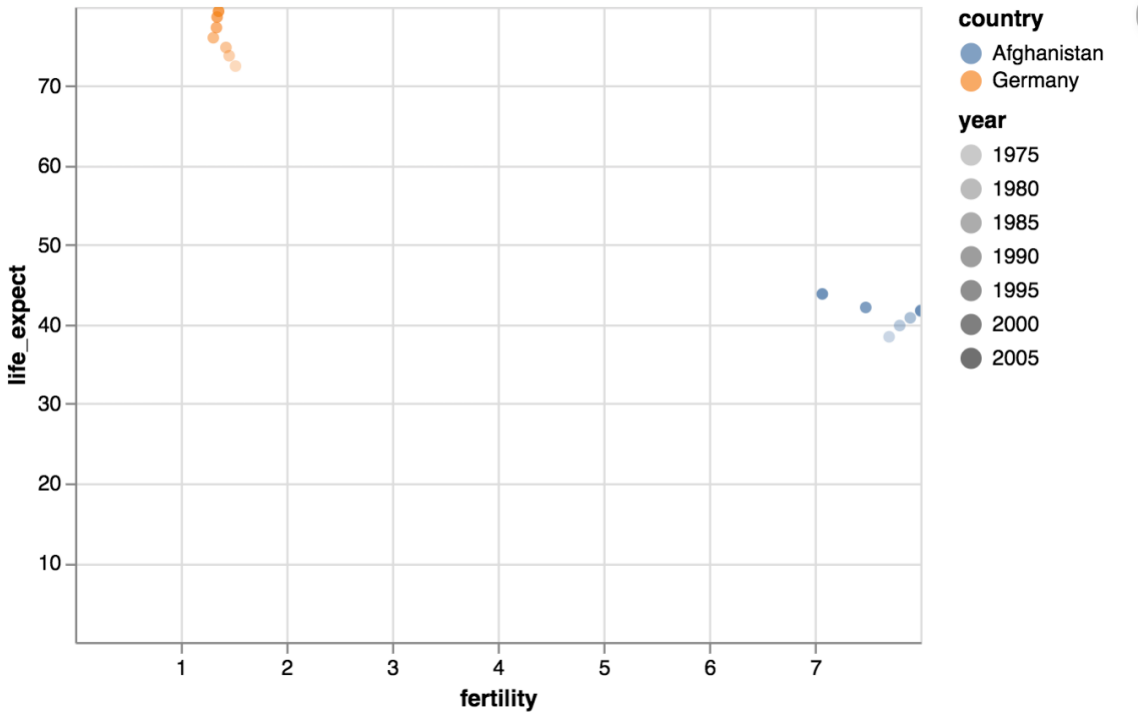
# 10. Simple Interaction

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

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 Germany, for the several years after 1970.

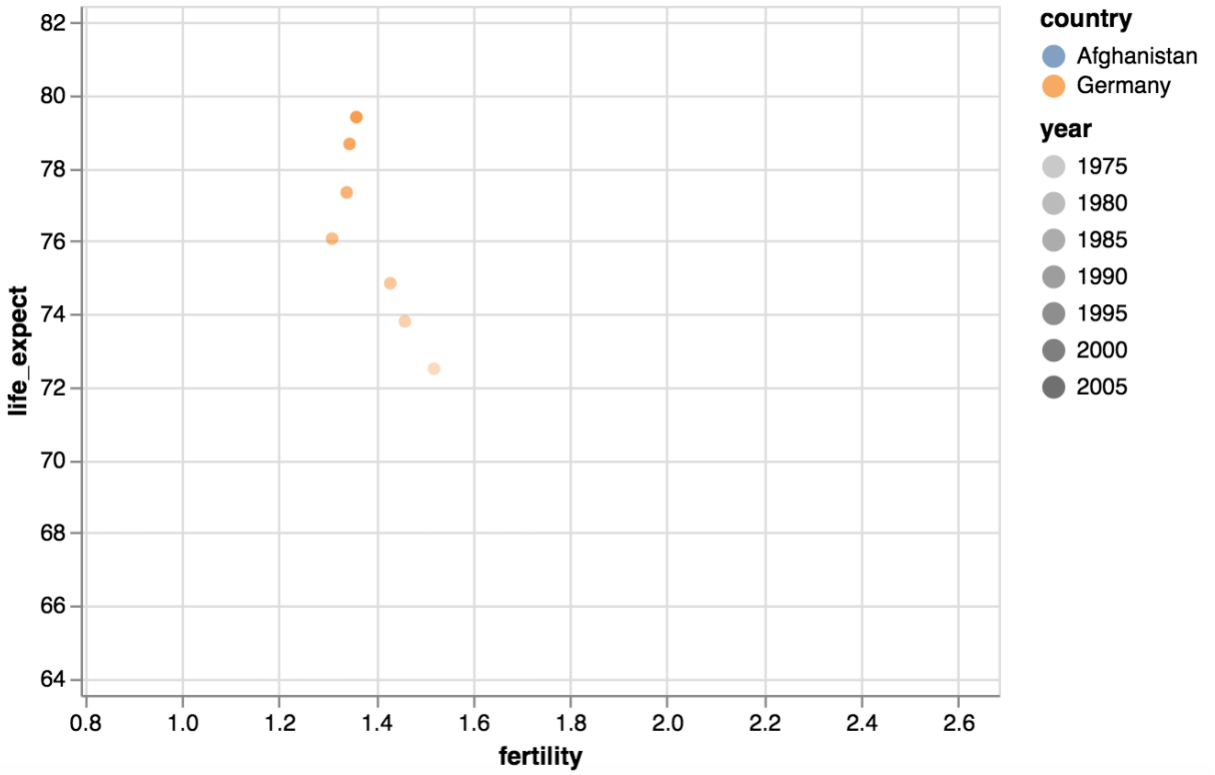
```
import altair as alt
from vega_datasets import data

df = data.gapminder()

alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = 'country:N',
    opacity = 'year:O'
).transform_filter((alt.datum.year > 1970)
  & ((alt.datum.country == 'Afghanistan')
  | (alt.datum.country == 'Germany')))
```

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 Germany 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 select elements and make something happen upon selection. 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.

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

- Individual: Only one item is selected
- Multiple: Multiple items are selected
- Interval: A set of items within a range of values 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.

We can select individual items by setting the selection option as single, with the following code:

```
singleSel = alt.selection_single()

ch = alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = 'country:N',
    opacity = 'year:O'
).transform_filter((alt.datum.year > 1970)
  & ((alt.datum.country == 'Afghanistan')
  | (alt.datum.country == 'Germany'))
  )

ch.interactive().add_selection(singleSel)
```

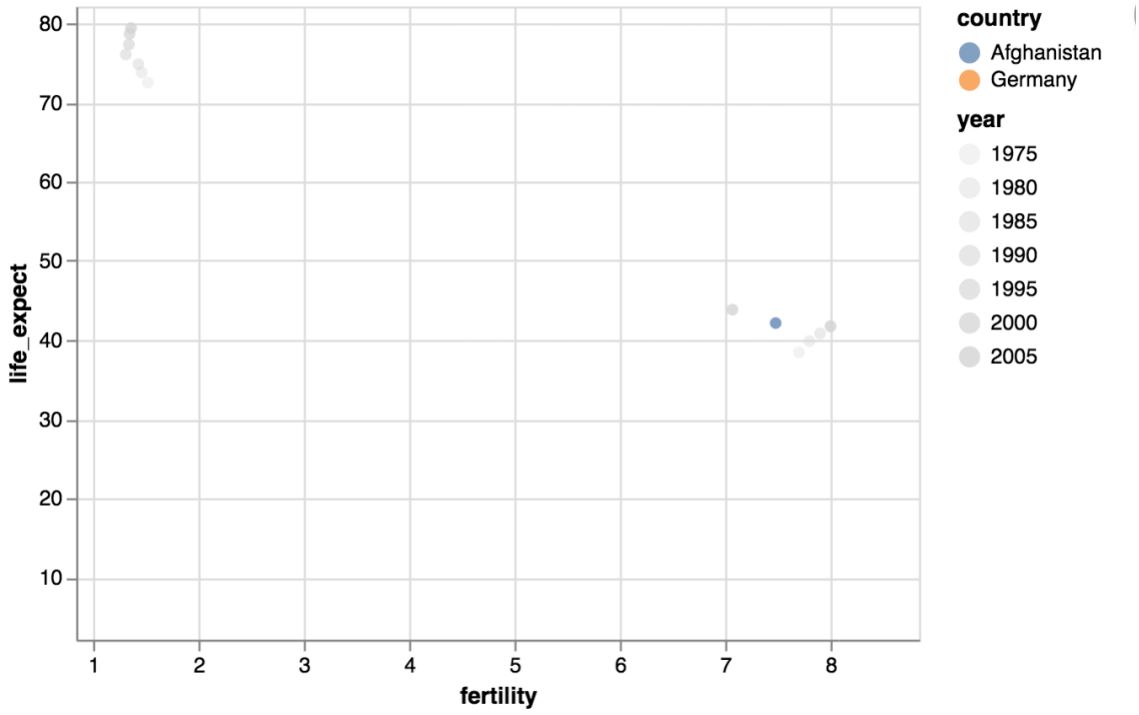
However, just adding a selection does not do anything to the chart. Since we have not implemented any task upon selection, we are not changing the way the items are being drawn, either selected or not.

The next step to interact with our charts will be the creation of conditions that depend on the selection. For example, highlighting the selected element.

This would be done using a condition, as in the following example:

```
ch = alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = alt.condition(singleSel, 'country:N', alt.value('lightgray')),
    opacity = 'year:O'
```

Note that we only modified a line, the one that determines the color, with a condition, that says that, if the element is in the selection, set the color as the country color, otherwise, the elements will appear grey. This condition modifies all the non-selected elements to grey, and keeps the color of the original element:




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* and set it as *True*. The following code shows an example:

```
ch = alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = alt.condition(singleSelNearest, 'country:N', alt.value('lightgray')),
    opacity = 'year:O'
).transform_filter((alt.datum.year > 1970)
  & ((alt.datum.country == 'Belgium')
  | (alt.datum.country == 'Germany')
  | (alt.datum.country == 'France'))
  )

ch.interactive().add_selection(singleSelNearest)
```

## 11.2 MULTIPLE SELECTION

Multiple selections behave like individual selections. The user can select multiple elements by holding the *Shift* key and clicking onto individual items.

The selection object can be defined using the following code:

```
multiSel = alt.selection_multi()
```

And, like in the previous case, adding the object with the *add_selection* function call.

Multiple selection can also be achieved using mouse hover if we appropriately define the selection object. The object can have several parameters, which include, as we saw, the *nearest* option, as well as the *on* option, that determines the selection trigger. We can set the mouse hovering operation as the one that makes the selection:

```
singleSelNearest = alt.selection_single(on='mouseover', nearest=True)
```

If, on top of the *mouseover* we also let the *nearest* option as true, the chart will always show one of the elements as selected, as long as the mouse is inside the chart area.

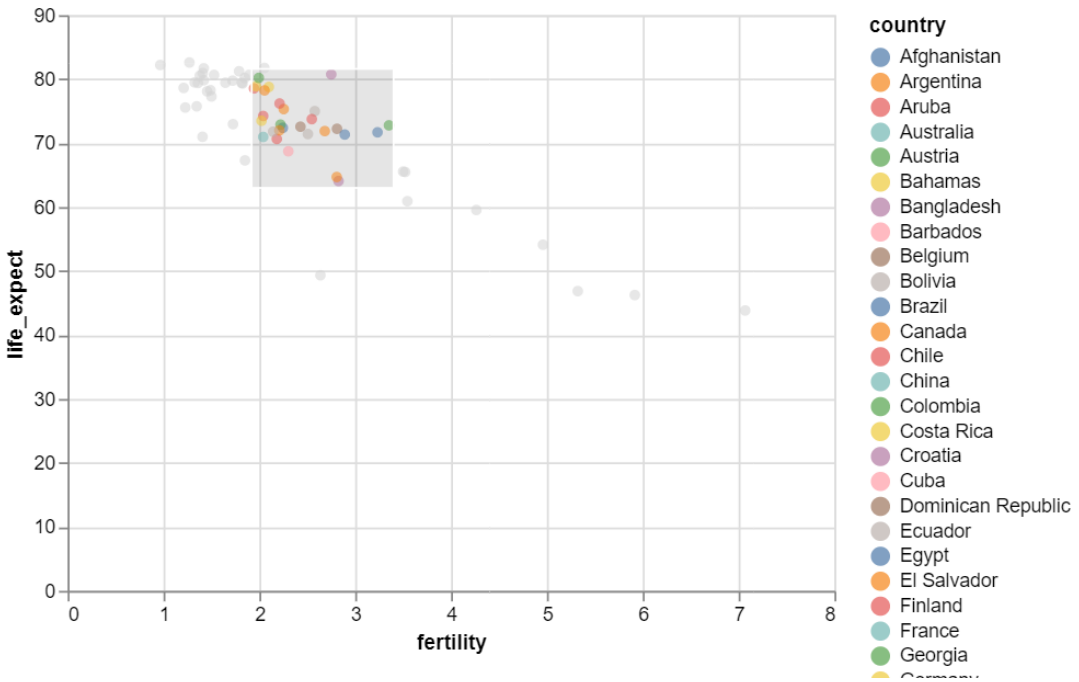
## 11.3 INTERVAL SELECTION

A well-known technique in interaction that lets the user select a set of elements through an initial click and drag 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.

In Altair, we can define a brush by simply creating a selection object of the type *interval*. We modify the previous chart slightly to make more points appear, remove the interactive exploration from the chart (which collides in event types with the brushing), and create a brush:

```
brushSel = alt.selection_interval()

ch = alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = alt.condition(brushSel, 'country:N', alt.value('lightgray')),
    opacity = 'year:O'
).transform_filter(alt.datum.year == 2005)

ch.add_selection(brushSel)
```

Now we can select a data interval and have it rendered with the proper color:



Note that once the brush has been defined, we can move it around by mouse dragging.

We could also make all the elements grey by default, so that we do not have the awkward effect of suddenly getting the colors again for all the points when we start a new brushing. This can be achieved by defining another parameter to the selection constructor. More concretely, we must define the selection as:

```
brushSel = alt.selection_interval(empty='none')
```

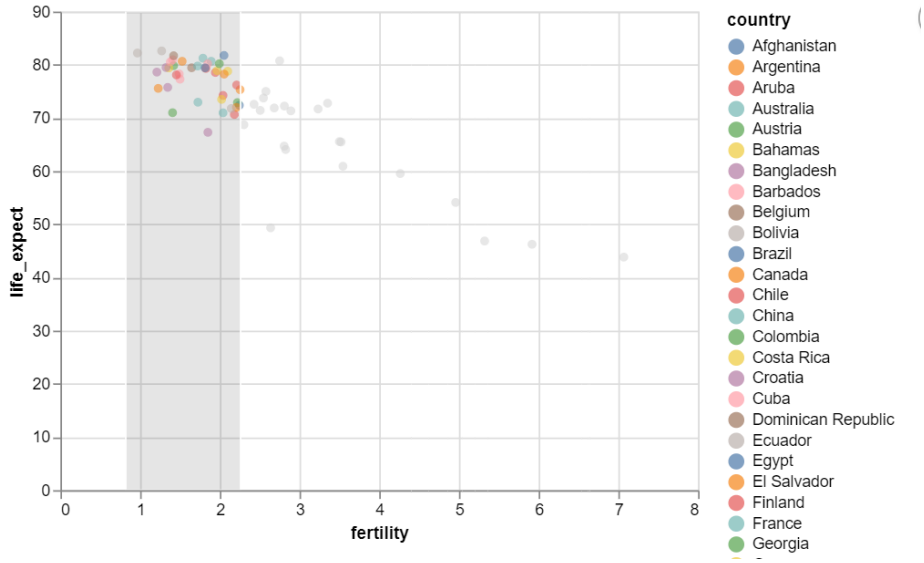
The result is that, before selecting any region, the items in the chart are unselected, and therefore light grey.

This parameter also works for individual and range selections.

We can further customize the selection function so that it only takes into account the brushing in one dimension, for example 'x', and therefore select only in the X range:

```
brushSel = alt.selection_interval(encodings=['x'],empty='none')
```

And the result would be seen as:




Note that, in a previous example, we removed 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:

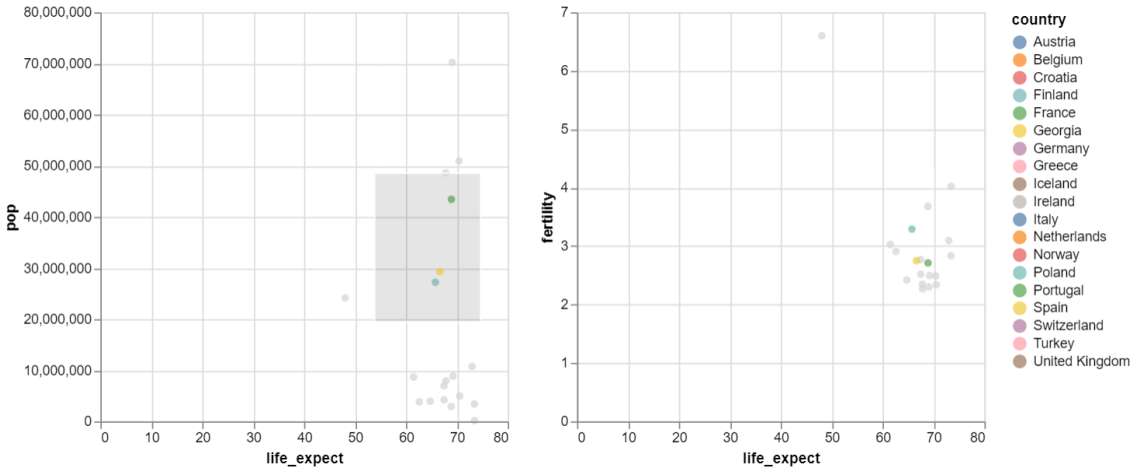
```
scalesSel = alt.selection_interval(bind='scales')
```

This way, the behavior will be the same than with the *interactive* function.

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:

```
source = data.gapminder()

xSel = alt.selection_interval(empty='none')

chPop = alt.Chart(source).mark_circle().encode(
    x=alt.X('life_expect:Q'),
    y=alt.Y('pop:Q'),
    color = alt.condition(xSel, 'country:N', alt.value('lightgray')),
).transform_filter((alt.datum.cluster== 1)
  & (alt.datum.year == 1955)).add_selection(xSel)

chIncomeHealth = alt.Chart(source).mark_circle().encode(
    x=alt.X('life_expect:Q'),
    y=alt.Y('fertility:Q'),
    color = alt.condition(xSel, 'country:N', alt.value('lightgray')),
).transform_filter((alt.datum.cluster== 1)
  & (alt.datum.year == 1955)).add_selection(xSel)

chPop.properties(width=300) | chIncomeHealth.properties(width=300)
```

And the result, with a selected region:

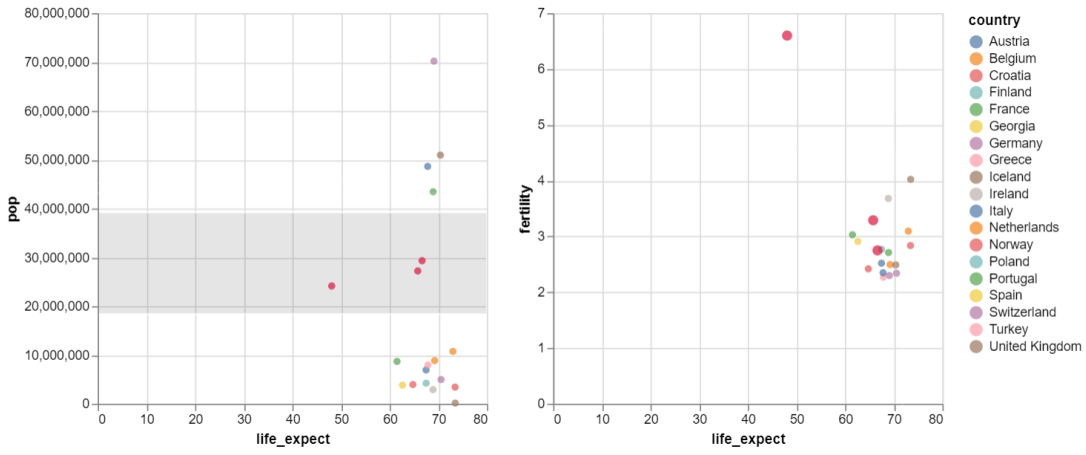



We can further exploit the selection, by providing selection in one axis. For example, we can ask the interval to be in the Y axis. Since the Y axis is different in both charts, the selected elements in the left chart may appear at different positions in the right chart.

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:

```
source = data.gapminder()

xSel = alt.selection_interval(encodings = ['y'], empty='none')

chPop = alt.Chart(source).mark_circle().encode(
    x=alt.X('life_expect:Q'),
    y=alt.Y('pop:Q'),
    color = alt.condition(xSel, alt.value('crimson'), 'country:N'),
).transform_filter((alt.datum.cluster== 1)
  & (alt.datum.year == 1955)).add_selection(xSel)

chIncomeHealth = alt.Chart(source).mark_circle().encode(
    x=alt.X('life_expect:Q'),
    y=alt.Y('fertility:Q'),
    color = alt.condition(xSel, alt.value('crimson'), 'country:N'),
    size = alt.condition(xSel, alt.value(60), alt.value(30))
).transform_filter((alt.datum.cluster== 1)
  & (alt.datum.year == 1955)).add_selection(xSel)

chPop.properties(width=300) | chIncomeHealth.properties(width=300)
```

The result would be like this:



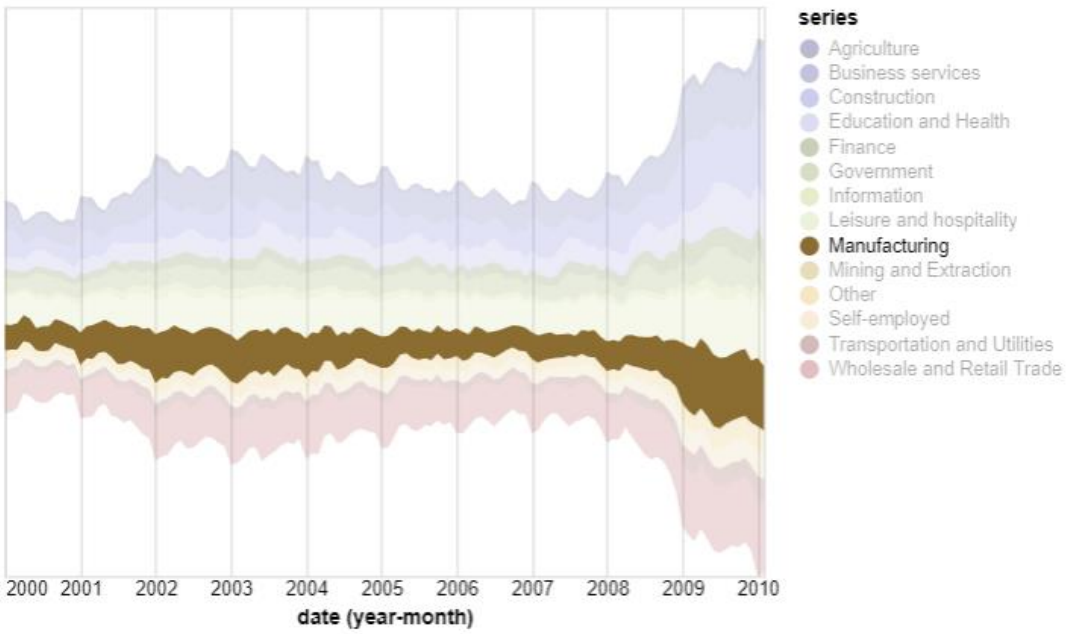

## 11.4 SELECTING BY FIELDS OR ENCODINGS

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.

From version 4 of altair, interactive legends are created very simply, just by adding the binding as the legend in the selection definition, such as in this example:

```
import altair as alt
from vega_datasets import data

source = data.unemployment_across_industries.url

selection = alt.selection_multi(fields=['series'], bind='legend')

alt.Chart(source).mark_area().encode(
    alt.X('yearmonth(date):T', axis=alt.Axis(domain=False, format='%Y',
                                             tickSize=0)),
    alt.Y('sum(count):Q', stack='center', axis=None),
    alt.Color('series:N', scale=alt.Scale(scheme='category20b')),
    opacity=alt.condition(selection, alt.value(1), alt.value(0.2))
).add_selection(
    selection
)
```

And the result would be something like this:



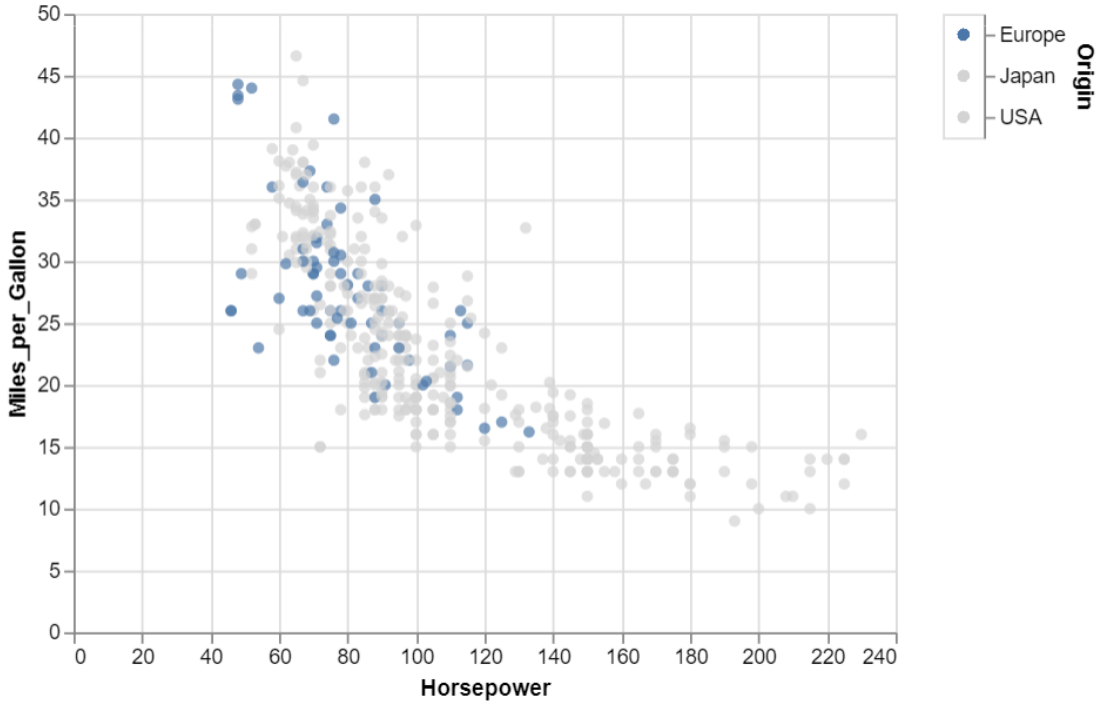

There is, however, a different way to create an interactive legend, which consists on creating another chart, and base the selection on the interaction with this chart. This was a workaround before the interactive legends were implemented

in altair, but also gives an idea of the potential of the tools that are within the platform. Thus, we show an example here. In this case, the 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. Note that you might need to tinker a bit to make sure that the legend appears with the size and position needed, depending on its contents, but the possibilities are great:

```
cars = data.cars()

selection = alt.selection_single(encodings=['color'])
color = alt.condition(selection,
                      alt.Color('Origin:N', legend=None),
                      alt.value('lightgray'))

scatter = alt.Chart(cars).mark_circle().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=color,
    tooltip='Name:N'
)

legend = alt.Chart(cars).mark_circle().encode(
    y=alt.Y('Origin:N', axis=alt.Axis(orient='right')),
    color=color
).add_selection(
    selection
)

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 Europe cars selected would be:

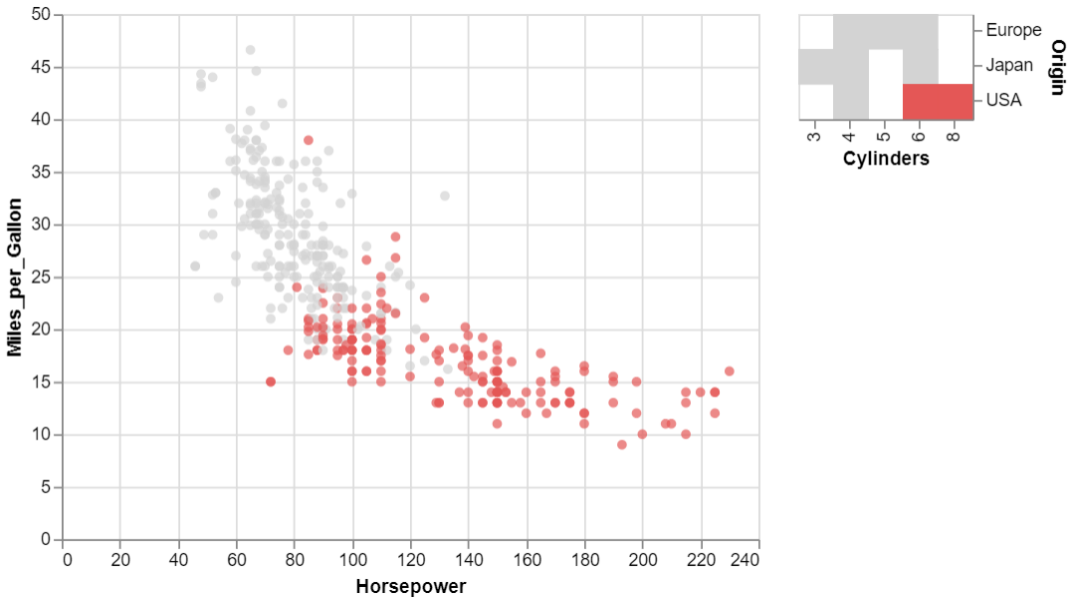



A more complex selection could include more than one field, for instance the origin and the number of cylinders. Since there are different combinations of cylinder numbers and countries, we can create a more complex selector. In this case, a matrix with values for each valid combination. By clicking 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:

```
selection = alt.selection_multi(fields=['Origin', 'Cylinders'])
color = alt.condition(selection,
                      alt.Color('Origin:N', legend=None),
                      alt.value('lightgray'))

scatter = alt.Chart(cars).mark_circle().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=color,
    tooltip='Name:N'
)

legend = alt.Chart(cars).mark_rect().encode(
    y=alt.Y('Origin:N', axis=alt.Axis(orient='right')),
    x='Cylinders:O',
    color=color
).add_selection(
    selection
)

scatter | legend
```

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

The result, if we select all the US cars with 6 or 8 cylinders would be:



# 12. Binding interactions to user input

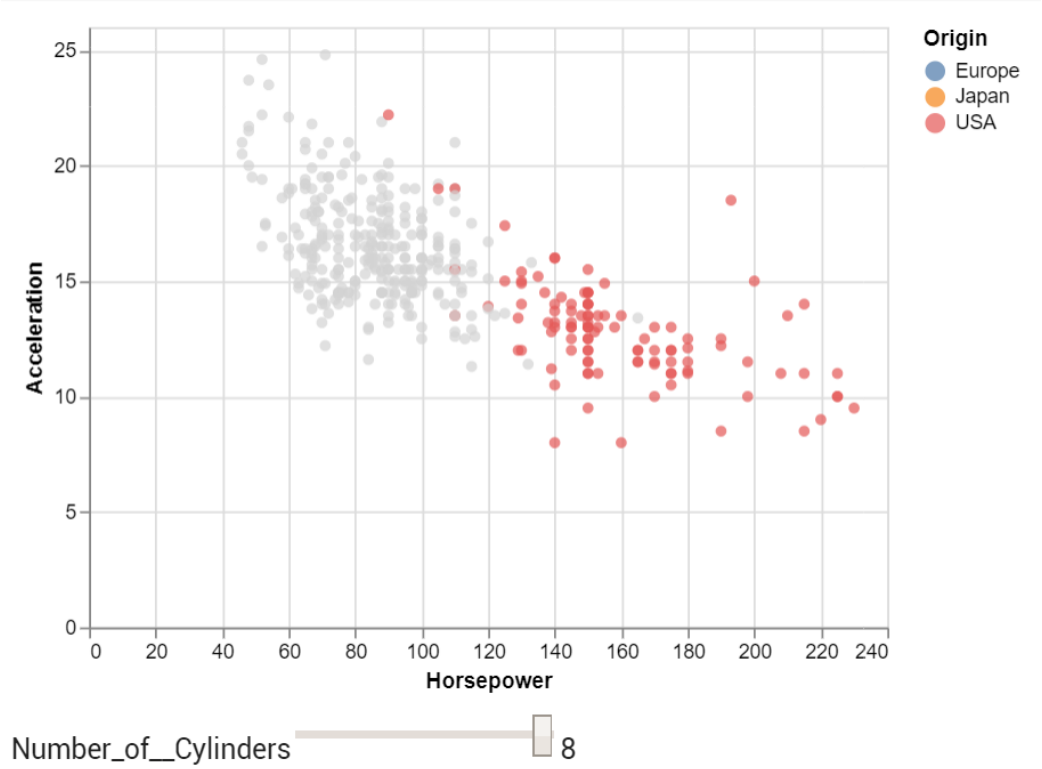
The latter examples show how we can create an extra chart to get the parameters from the selection. However, Vega, and in its turn, Altair, have a method that can achieve this without the necessity of creating a new chart. The method is called binding, and it consists basically in linking a widget, such as a dropdown menu, to the property we want to select.

## 12.1 SLIDERS

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:

```
input_slider = alt.binding_range(max=8, min=3, step=1)

selection = alt.selection_single(fields=['Cylinders'],
                                 bind=input_slider, name='Number of ')
color = alt.condition(selection,
                      alt.Color('Origin:N'),
                      alt.value('lightgray'))

alt.Chart(cars).mark_circle(filled=True).encode(
    x='Horsepower:Q',
    y='Acceleration:Q',
    color=color,
    tooltip='Name:N'
).add_selection(
    selection
)
```

The result shows, when we select all the cars with 8 cylinders, that they are coming from the US.




## 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.

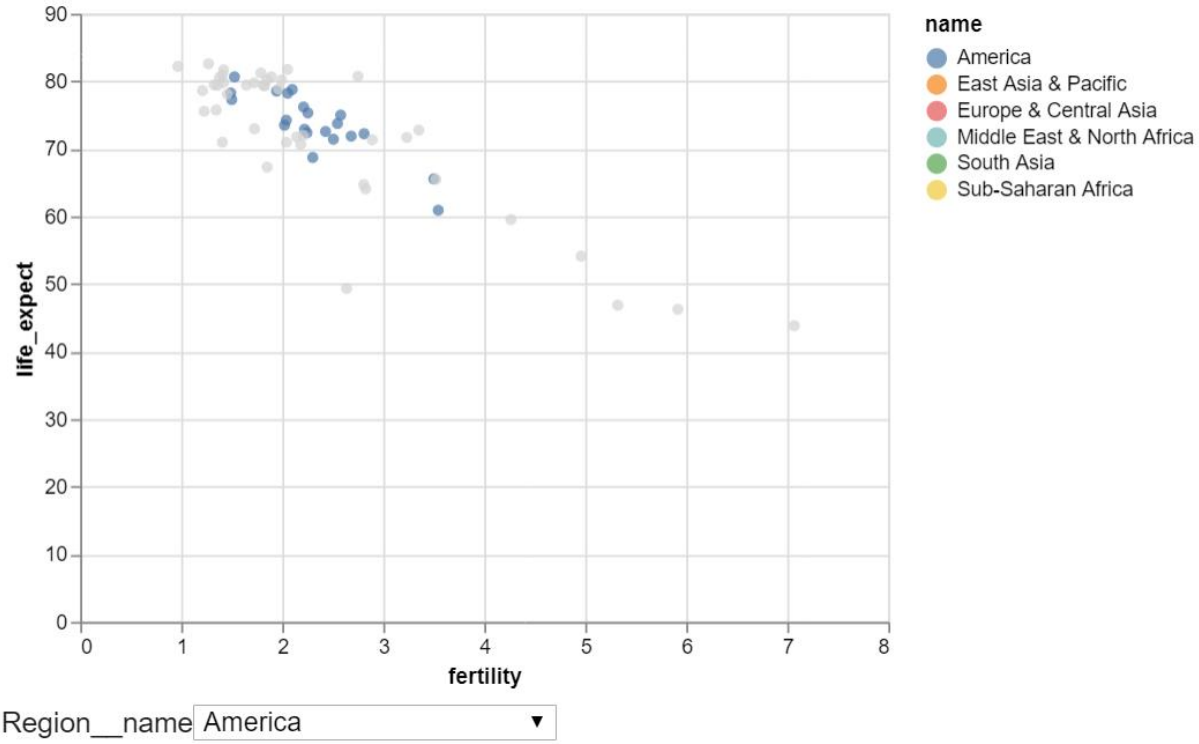
The first part of the code declares the DataFrame that maps the ids of the clusters to their names, and defines the selection object.

```
df = 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"}
      ])

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

dropSelect = alt.selection_single(fields=['name'],
                                  bind=input_dropdown,
                                  name='Region ')
```

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

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

ch1 = ch.transform_filter(alt.datum.year == 2005)

ch1
```

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




Region__name America

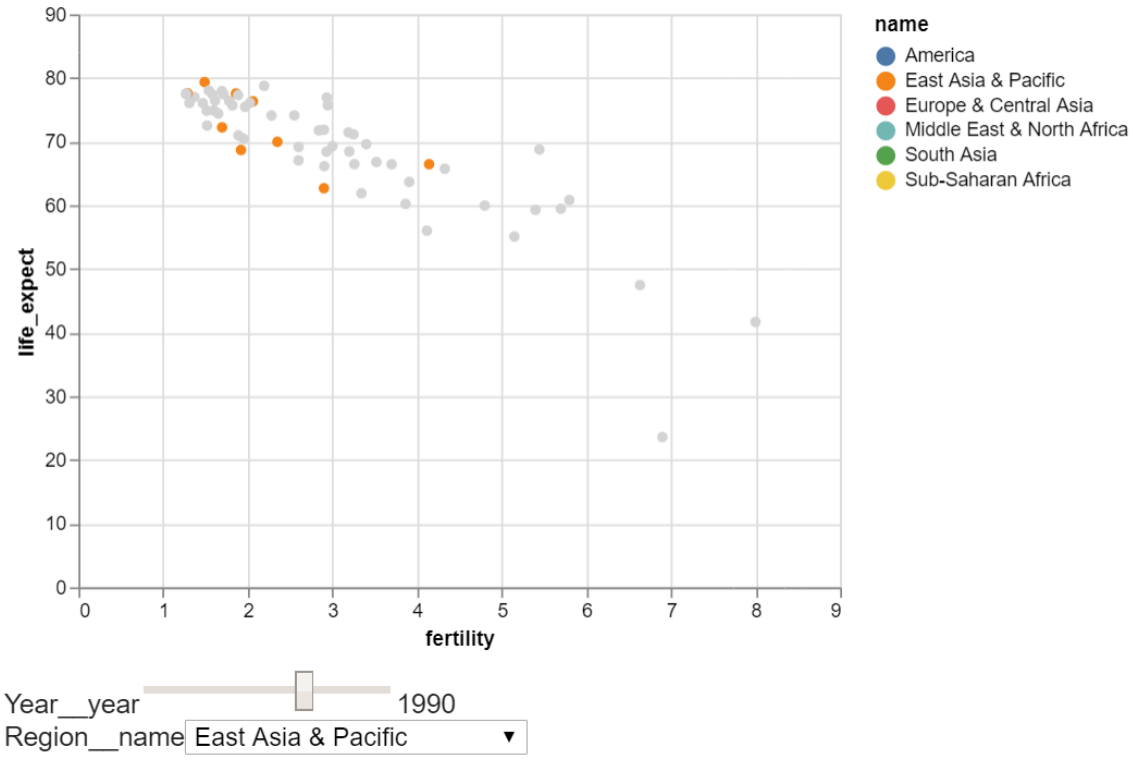
We can further exploit this approach if we want to navigate all the years in the data. We can do so by adding another selection condition that lets the user choose the year she wants to display.

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):

```
regionSelect = alt.selection_single(fields=['name'],
                                    bind=input_dropdown,
                                    name='Region ')

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

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

alt.Chart(df).mark_circle().encode(
    alt.X('fertility:Q'),
    alt.Y('life_expect:Q'),
    color = alt.condition(regionSelect, '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_selection(regionSelect).add_selection(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.

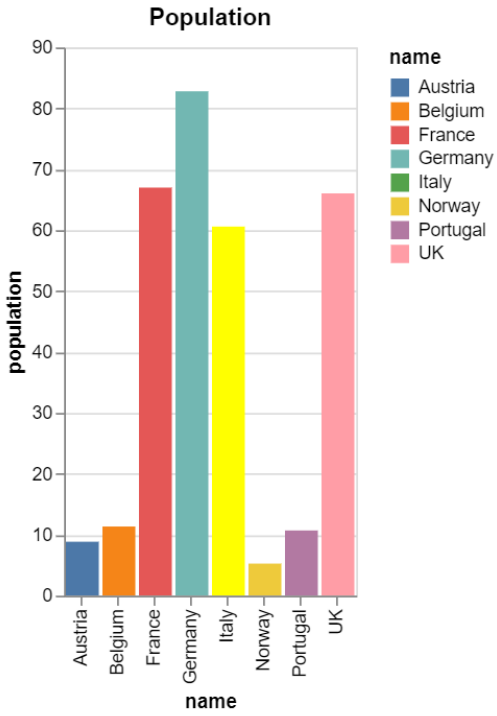
## 12.3 OTHER WIDGETS

Other data bindings are radio buttons and checkboxes.

The following example illustrates the use of a checkbox, that is used to highlight the bar corresponding to the selected country:

```
country_names = ['Austria', 'Belgium', 'France', 'Germany',
                            'UK', 'Italy', 'Norway', 'Portugal']
country_radio = alt.binding_radio(options=country_names)
countrySelect = alt.selection_single(fields=['name'],
                                     bind=country_radio,
                                     name='Country ',
                                     init = {'name':'Italy'})

data = pd.DataFrame({'name': ['Austria', 'Belgium', 'France', 'Germany',
                            'UK', 'Italy', 'Norway', 'Portugal'],
                     'population': [8.85, 11.35, 66.99, 82.8,
                                    66.04, 60.59, 5.26, 10.7]})

alt.Chart(data).mark_bar().encode(
    x='name:O',
    y='population:Q',
    color=alt.condition(countrySelect, alt.value('yellow'), 'name:N'),
).properties(
    title = 'Population'
  ).add_selection(
      countrySelect
      )
```

The result, when the code is executed, will highlight the default country: Italy.




Country__name Austria Belgium France Germany UK Italy Norway Portugal

The checkbox option works in a similar way.

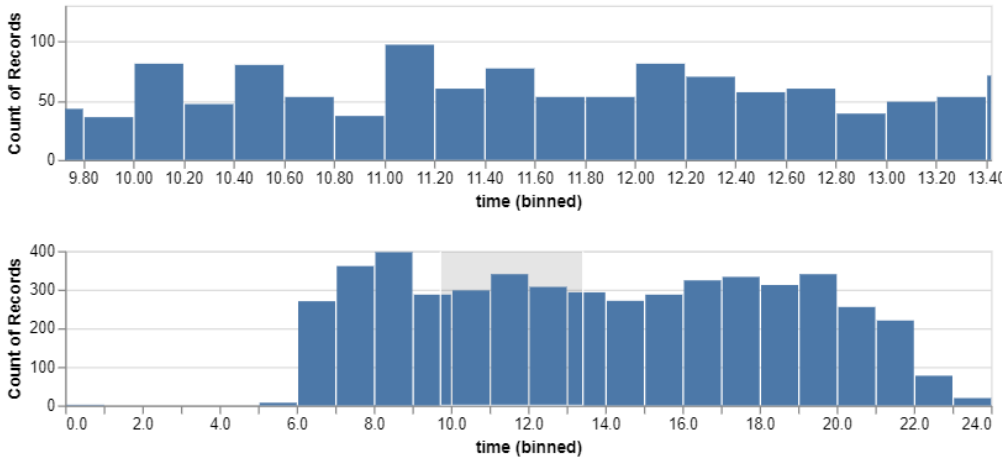
## 12.4 RESPONSIVE CHARTS

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:

```
import altair as alt
from vega_datasets import data

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(
    y='count():Q'
).properties(
    width=600,
    height=100
)

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

The result may look like this:



## 12.5 USING WIDGETS IN CREATIVE WAYS

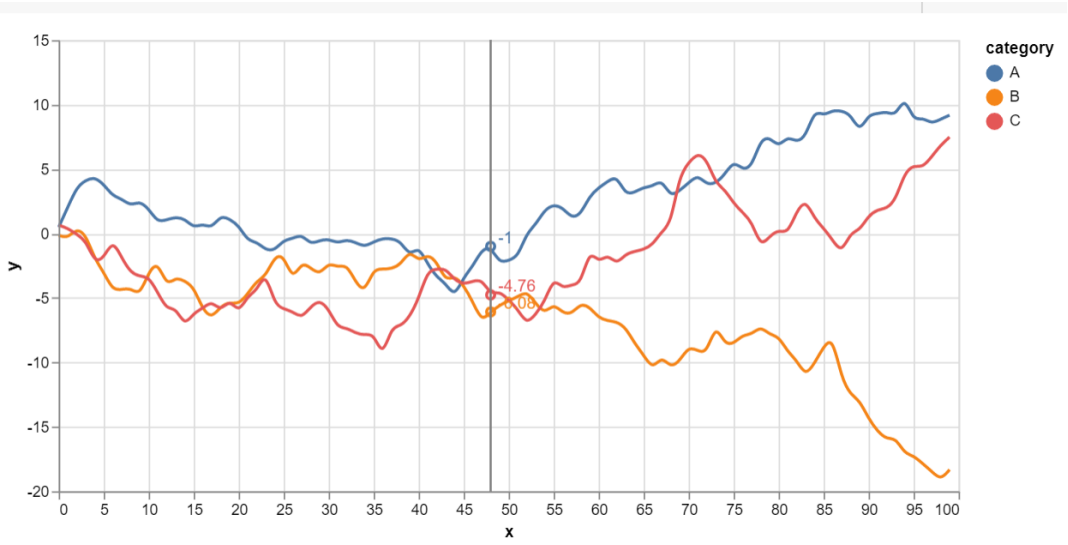
The combination of selection and hover can be used in several ways, such as the example "Multi-line tooltip" in the Altair library, where the plot creates a number of hidden tooltips that are rendered visible only when the mouse is hovering close to them. Note the clever usage of *nearest* and hover.

```
# Create a selection that chooses the nearest point
# & selects based on x-value
nearest = alt.selection(type='single', nearest=True, on='mouseover',
                        fields=['x'], empty='none')

# The basic line chart
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_selection(
    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:

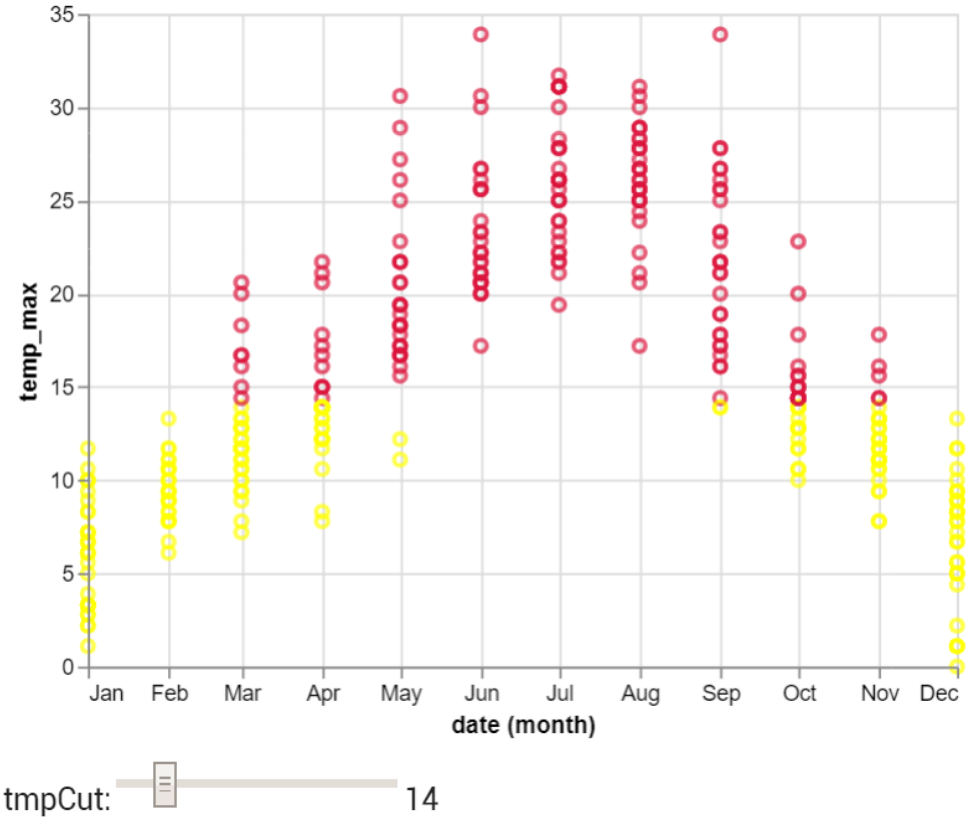



However, this approach, that is completely unaware of the data distribution, may cause some labels to overlap, and then, reading them might be difficult.

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:

```
df = data.seattle_weather()

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


alt.Chart(df).mark_point().encode(
    x=alt.X('month(date):T'),
    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_selection(sel_temp)
```

The result is:




## 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:

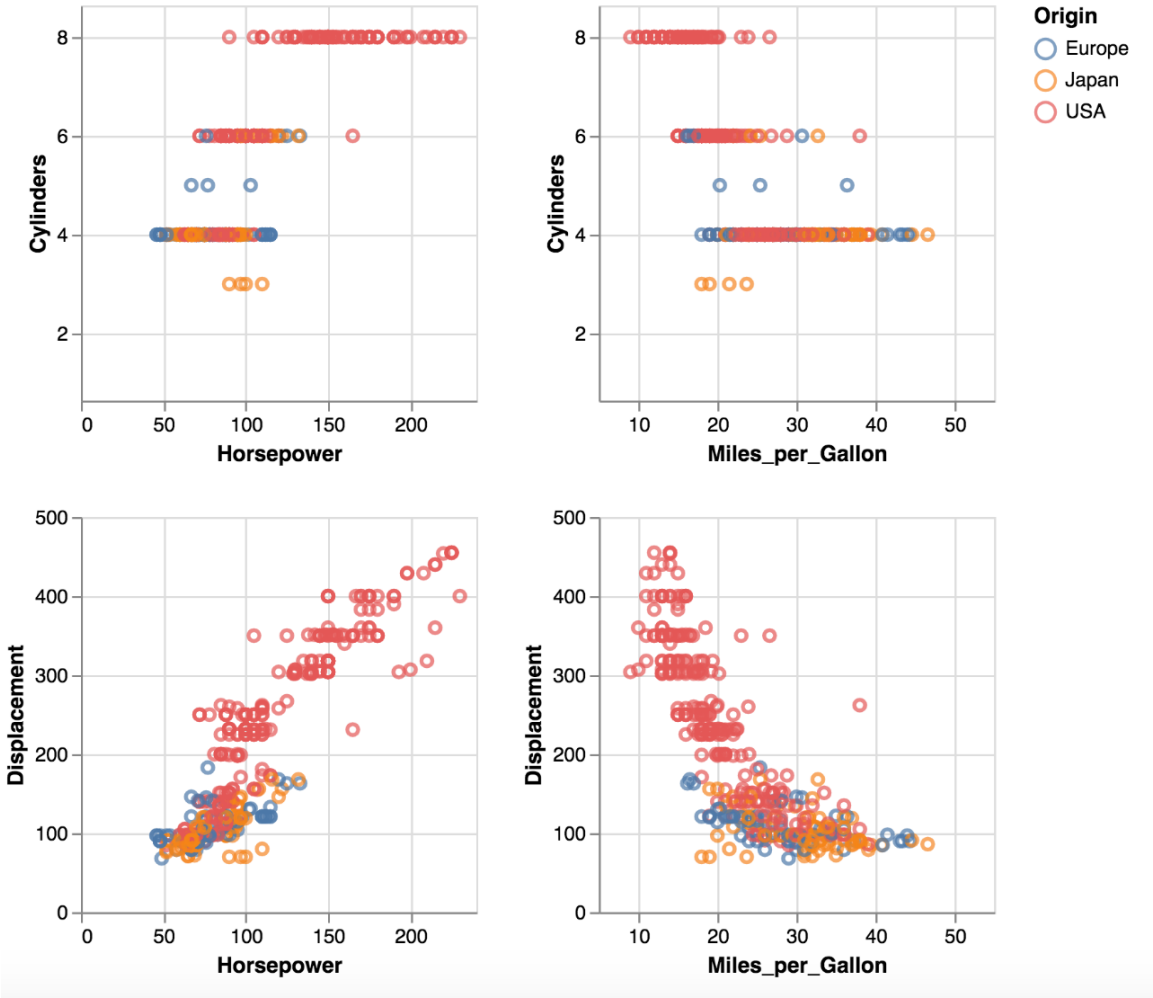
- Repeated charts: Are intended to draw multiple charts in vertical or horizontal layout, where the only change between them is the modification of one or more encodings.
- Faceted charts: Their objective is to produce multiple views of a dataset where for each chart, the represented information is a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

```
import altair as alt
from vega_datasets import data
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:



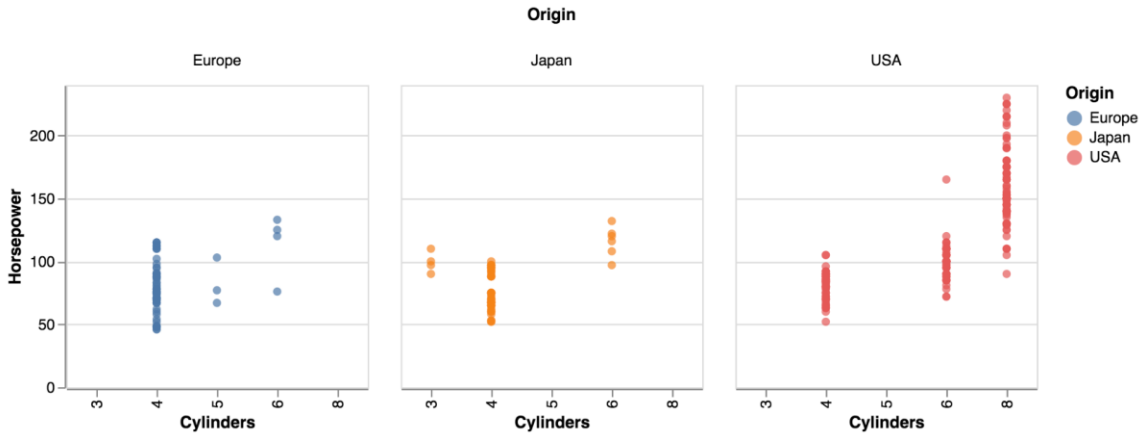

The same result can be obtained by a combination of vertical and horizontal layouts. However, the code is more cumbersome.

## 13.2 FACETED CHARTS

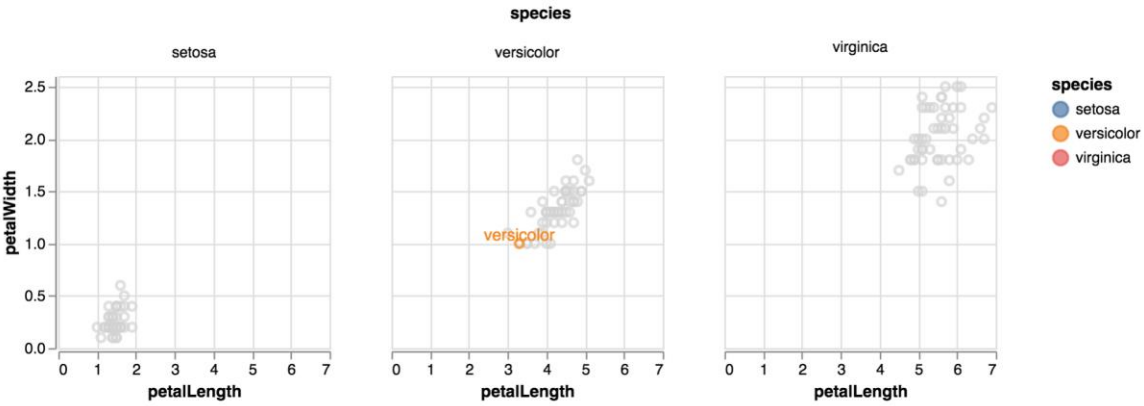
Laying multiple facets of the same chart can be also done with a horizontal or vertical concatenation where different filters are applied to each of the individual charts. However, the *facet* operation makes it slightly easier if the filtering operation can be expressed easily. For example, we can plot 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.

## 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.

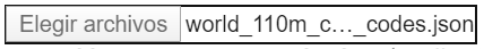
You can find a complex example in the following website:

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:

```
import altair as alt
from vega_datasets import data
import io
from google.colab import files

uploaded = files.upload()
```




- **world_110m_country_codes.json**(application/json) - 9975 bytes, last modified: 22/12/2019 - 100% done

```
Saving world_110m_country_codes.json to world_110m_country_codes.json
```

Then, we read and merge the data and the country codes files.

```
import pandas as pd

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

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

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

#print(merged)

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_single(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_selection(selection)
```

The result is something like this: