Unit 7: Reporting and Dashboards

Contents

Lab Questions

For this unit, the notebook itself will not contain lab questions. Instead, we'll create visualizations in Tableau (https://www.tableau.com/) in the second part of the unit.

Getting Started

Source and processed data are often stored in databases. End-users of this data typically don't access a database directly so disseminating information often requires extra steps. In this unit, we'll examine two means of doing this: creating report file containing the information that can be distributed and creating dashboards that can be used to quickly convey information using visual elements.

For reporting, we'll look at extracting data from a database to create an Word document. For dashboard creation, we'll focus primarily on Tableau but discuss how we create dashboards using Python.

We'll use some of the libraries we've used previously including SQLAlchemy and pandas but we'll also use a library the provides the ability to create Word files, python-docx (https://python-docx.readthedocs.io/en/latest/); we can install the library with `pip` .

In [2]:
```
import sys
!{sys.executable} -m pip install python-docx
```

```
Collecting python-docx
  Downloading python-docx-0.8.6.tar.gz (5.3MB)
    100% |████████████████████████████████| 5.3MB 284kB/s ta 0:00:011
Requirement already satisfied: lxml>=2.3.2 in /usr/local/lib/python3.6/si
te-packages (from python-docx)
Building wheels for collected packages: python-docx
  Running setup.py bdist_wheel for python-docx ... done
  Stored in directory: /Users/arthur/Library/Caches/pip/wheels/cc/74/10/4
2b00d7d6a64cf21f194bfef9b94150009ada880f13c5b2ad3
Successfully built python-docx
Installing collected packages: python-docx
Successfully installed python-docx-0.8.6
```

Creating a Report

A typical work flow in report generation is to extract data from a database, summarize it and perform other calculations as required, and create a document with the processed information - this is often a very manual process. Here, we'll look at how we an complete this process using Python. As an example, we'll work with the [Chinook (https://chinookdatabase.codeplex.com/)](https://chinookdatabase.codeplex.com/) database to generate a report for summarizing sales that employees assisted with for each year recorded in the database.

Recall that the database has the following tables and structure.

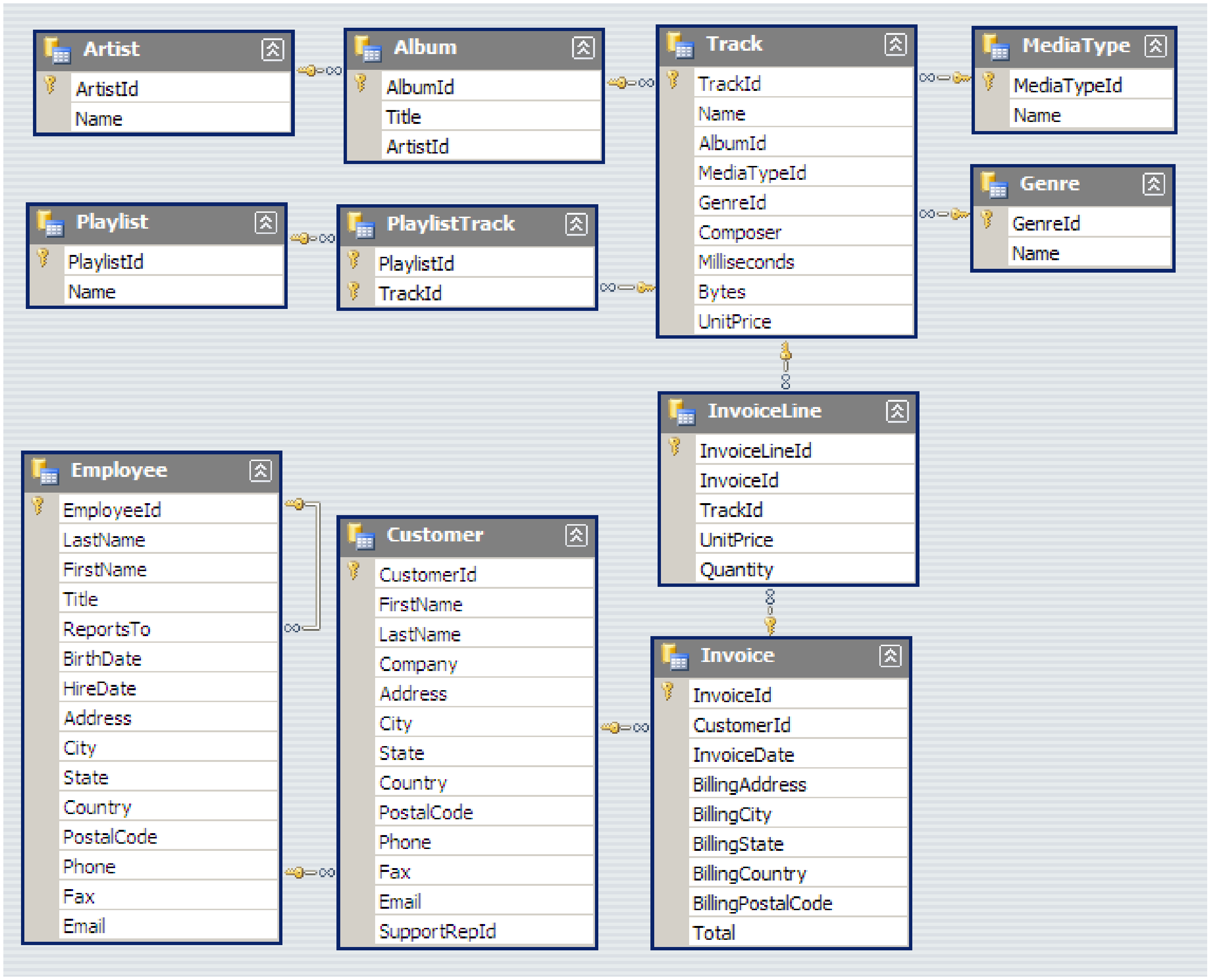


Diagram of the Chinook Database

We need to connect employees with sales. To this, we'll use the `Employee` , `Customer` , and `Invoice` tables. While we could retrieve the records of each row and join the tables in Python, it's generally preferable to do this in the database as it reduces the amount of data that has to be transfered from the database to the computer running the Python code.

Before getting the data for employees and sales, let's first look at the unique values for the `Title` column in the `Employee` table. We can connect to the database using SQLAlchemy. While we're only interested in seeing the distinct values at this point and don't need a DataFrame, working with pandas to execute the query results in simpler code.

In [67]:
```python
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine('sqlite:///data/01-chinook.sqlite')
query = "SELECT DISTINCT Title from Employee"
pd.read_sql(query, engine)
```

Out[67]:

	Title
0	General Manager
1	Sales Manager
2	Sales Support Agent
3	IT Manager
4	IT Staff

Of the position titles, `Sales Support Agent` is likely the one for employees responsible for assisting with sales. We can use the following SQL query to join the `Employee`, `Customer`, and `Invoice` tables while filtering the results based on the value of the `Title` field in the `Employee` table.

```sql
SELECT Employee.EmployeeId, Employee.FirstName,
    Employee.LastName, Invoice.Total, Invoice.InvoiceDate
FROM Employee INNER JOIN (
    Customer INNER JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
) ON Employee.EmployeeId = Customer.SupportRepId
Where Employee.Title = 'Sales Support Agent'
```

We can again use pandas to execute the query.

In [246]:
```
query = """
SELECT Employee.EmployeeId, Employee.FirstName,
    Employee.LastName, Invoice.Total, Invoice.InvoiceDate
FROM Employee INNER JOIN (
    Customer INNER JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
) ON Employee.EmployeeId = Customer.SupportRepId
Where Employee.Title = 'Sales Support Agent'
"""

engine = create_engine('sqlite:///data/01-chinook.sqlite')
sales_data = pd.read_sql_query(query, engine)

display(sales_data.shape)
display(sales_data.head())
```

```
(412, 5)
```

	EmployeeId	FirstName	LastName	Total	InvoiceDate
0	5	Steve	Johnson	1.98	2009-01-01 00:00:00
1	4	Margaret	Park	3.96	2009-01-02 00:00:00
2	4	Margaret	Park	5.94	2009-01-03 00:00:00
3	5	Steve	Johnson	8.91	2009-01-06 00:00:00
4	4	Margaret	Park	13.86	2009-01-11 00:00:00

Each record in the result corresponds to an invoice. For our sales report, we'll calculate the aggregate sales by year for each employee using a pivot table. To simplify the pivot table and report, we'll create a new column, `Name` , that combines the first and last name for each record.

We'll also extract the year from each `InvoiceDate` . The date data returned from the database is stored as an *Object* so we'll need to use pandas' [*to_datetime()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to_datetime.html)](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to_datetime.html) function to convert the type before extracting the year. Alternatively, we could extract the year substring from the original string itself.

In [247]:
```
sales_data['Name'] = sales_data.LastName + ", " + sales_data.FirstName
sales_data['InvoiceYear'] = pd.to_datetime(sales_data.InvoiceDate).dt.year.a
sales_data.head()
```

Out[247]:

	EmployeeId	FirstName	LastName	Total	InvoiceDate	Name	InvoiceYear
0	5	Steve	Johnson	1.98	2009-01-01 00:00:00	Johnson, Steve	2009
1	4	Margaret	Park	3.96	2009-01-02 00:00:00	Park, Margaret	2009
2	4	Margaret	Park	5.94	2009-01-03 00:00:00	Park, Margaret	2009
3	5	Steve	Johnson	8.91	2009-01-06 00:00:00	Johnson, Steve	2009
4	4	Margaret	Park	13.86	2009-01-11 00:00:00	Park, Margaret	2009

For the pivot table, we'll use `Name` as the index. We'll aggregate the values in the `Total` column and use the years in `InvoiceYear` as columns.

In [248]:
```
sales_pivot = sales_data.pivot_table(
    index = ['Name'],
    values = ['Total'],
    columns = ['InvoiceYear'],
    aggfunc = pd.np.sum
)

sales_pivot
```

Out[248]:

	Total				
InvoiceYear	**2009**	**2010**	**2011**	**2012**	**2013**
Name					
Johnson, Steve	164.34	136.77	159.47	133.73	125.85
Park, Margaret	161.37	122.76	125.77	197.20	168.30
Peacock, Jane	123.75	221.92	184.34	146.60	156.43

Now that we have the aggregate sales data, we can generate a report document. Having installed *python-docx* earlier, we can import it. We'll start by creating an instance of the *Document* (https://python-docx.readthedocs.io/en/latest/api/document.html) class.

In [266]:
```
import docx

sales_report = docx.Document()
```

As demonstrated in the python-docx documentation (https://python-docx.readthedocs.io/en/latest/user/quickstart.html), there is a wide variety of content we can add to a document. Our report will be somewhat simple, containing an image, some text, a table, and a chart.

The image we'll use is our company's logo in ./images/07-logo.png (./images/07-logo.png).

Company Logo

To add the image, we use the *add_picture()* (https://python-docx.readthedocs.io/en/latest/api/document.html#docx.document.Document.add_picture) method, specifying the path to the image file and, optionally, the desired width or height; the library will automatically scale the other dimension to maintain the original image's aspect ratio. To specify a size, we use one of many [length objects (https://python-docx.readthedocs.io/en/latest/api/shared.html#length-objects)](https://python-docx.readthedocs.io/en/latest/api/shared.html#length-objects) provided by python-docx; we'll use the *Inches* class.

```
In [267]: sales_report.add_picture('./images/07-logo.png',
                                   width=docx.shared.Inches(1.0))

Out[267]: <docx.shape.InlineShape at 0x11f321160>
```

Next, we add a heading and a small paragraph of text using the *add_heading()* (https://python-docx.readthedocs.io/en/latest/api/document.html#docx.document.Document.add_heading) and *add_paragraph()* (https://python-docx.readthedocs.io/en/latest/api/document.html#docx.document.Document.add_paragraph) methods, respectively.

```
In [268]: paragraph = "The following a summary of sales with which " \
                      "a sales support agent assisted. This summary " \
                      "provides aggregate sales amounts for each agent by year."

          sales_report.add_heading("Sales Summary")
          sales_report.add_paragraph(paragraph)

Out[268]: <docx.text.paragraph.Paragraph at 0x11ef914a8>
```

Next, we'll add a table with data from the pandas pivot table. Adding a table is slightly more complicated than adding an image or text. First, we create a table using the *add_table()* (https://python-docx.readthedocs.io/en/latest/api/document.html#docx.document.Document.add_table) method, specifying the number of rows and columns and, optionally, the style. We'll store the object returned by the method so we can iterate through it and add values to the table. Iterating through the table object will allow us to access each row, one at a time. Iterating through a row, is similar to moving through the columns - giving us access to the cells of the row. We can also access an individual cell using the table's *cell()* (https://python-docx.readthedocs.io/en/latest/api/table.html#docx.table.Table.cell) method. The cell method returns an object with a *text* property that can be used to assign a value to the cell. Note that the value must be a string.

We'll need to iterate through two tables - reading from the pivot table and writing to the document table - so we'll have to be careful about our loop variables and the values they represent. It is easier to iterate through the DataFrame and use the table's *cell()* method than iterating through the table and accessing values in the DataFrame.

In [269]:
```python
# rows, columns of data in pivot table
rows, columns = sales_pivot.shape

# add 1 for header row and column
report_table = sales_report.add_table(rows+1, columns+1,
                                      style = "Medium List 2 Accent 1")

# use header_row, header_col for header row, column numbers
header_row = 0
header_col = 0

# add header
report_table.cell(header_row, header_col).text = "Name"
for col_index, col_name in enumerate(sales_pivot['Total'].columns):
    # convert integer data to string
    report_table.cell(header_row, col_index+1).text = str(col_name)

# iterate through pivot_table data
for row_index, (name, row_data) in enumerate(sales_pivot.iterrows()):
    # add column header
    report_table.cell(row_index+1, header_col).text = name

    # iterate through pivot table row data
    for (col_index, col_data) in enumerate(row_data):
        # format the data for the cell
        cell_value = str(round(col_data, 2))
        report_table.cell(row_index+1, col_index+1).text = cell_value
```

We start by getting the shape of the pivot table in terms of the number of rows and columns it contains. These values do not include the DataFrame's index or column name so we add 1 to the values when creating the document's table. When we do create the table, we also specify a style.

We set a value for `header_row` and `header_col` to represent the row index and column index for the headers; using a named variable makes the code easier to read than if we had used `0` when we needed to.

Next, we begin adding data. We start with the header row. The first value in the header row is the label for the column headers - `Name` . We iterate through the column names of the pivot data for the `Total` values - each of the years - and use these as the remaining values in the header row. The first value in each row corresponds to the employee's name which comes from the pivot table's index. The remaining values for the row are from the pivot table's row data.

We used enumerate throughout to keep track of which row and column we were working in. We had to add one to the values because pandas only counts data rows and column - not the headers.

Unfortunately, the python-docx library doesn't support charts. While there are other libraries, such as [DocxFactory (https://pypi.python.org/pypi/DocxFactory/1.2.5)](https://pypi.python.org/pypi/DocxFactory/1.2.5), that support this ,they can requires steps beyond using `pip` to install. We'll instead create a chart in Python and embed it in the document as an image.

We begin by in-ling all plots.

In [270]:
```
%matplotlib inline
```

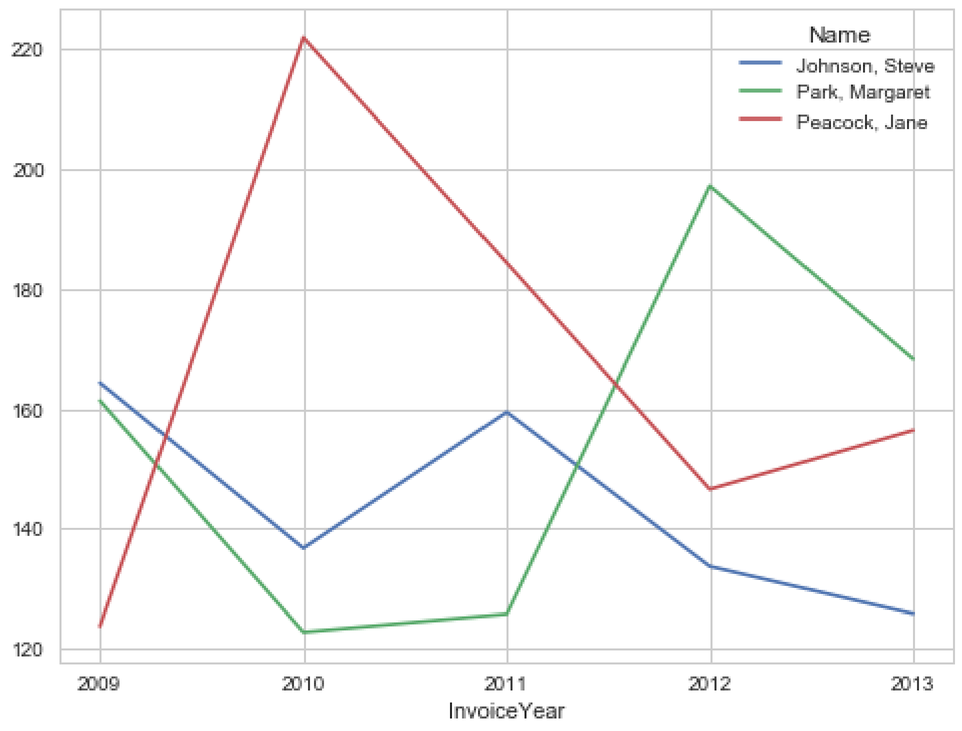
We'll specify some formatting properties. First, we set the figure size. When we embed the image in the document, we'll specify a width a six inches. To avoid excessive pixelation, we'll set plotting figures to have a slightly greater width of eight inches. To maintain a rectangular aspect ratio, we'll set the plotting height to six inches.

We'll also set the Seaborn style (https://seaborn.pydata.org/tutorial/aesthetics.html) to one that will look better in a document that might be printed.

In [271]:
```
import seaborn as sns
sns.set(rc={'figure.figsize':(8,6)})
sns.set_style("whitegrid")
```

Next, we create a line plot from the pivot table. To correctly plot the data, we need to specify that we're interested in the `Total` values as opposed to another set of values (even though there are no others) and we have to transpose the table to swap which variable is used to create different lines and which is used for values of the horizontal axis. We also explicitly specify the x-axis tick labels - relying on automatic generation will create decimal year values like `2010.5`.

In [272]:
```
x_ticks = sales_pivot['Total'].columns.values
sales_plot = sales_pivot['Total'].transpose().plot(xticks=x_ticks)
```



In order to insert the image into the document, we have to save it as a file. Because we only need the file for a short time until we insert it into the document, we'll create a temporary file using the *tempfile* (https://docs.python.org/3.6/library/tempfile.html) module. We import this module with the *os* (https://docs.python.org/3.6/library/os.html) module. To create the temporary file, we get the path to directory used for temporary files using *tempfile.gettempdir()* (https://docs.python.org/3.6/library/tempfile.html#tempfile.gettempdir); depending on the operating system the content of the temporary directory is deleted periodically. To specify the path to a file within in this directory, we use the *os.path.join()* (https://docs.python.org/3.6/library/os.path.html#os.path.join) function to combine the directory path with a desired filename.

In order to save the image of the plot, we need to use the corresponding *Figure* (https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html) object used to displat the plot. The return value of the DataFrame's plotting method is an *Axes* (https://matplotlib.org/api/axes_api.html) object that gives us access to most of the plots properties such as axis and line properties. To get the *Figure* from an *Axes* object, we can use the *Axes* *get_figure()* (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.get_figure.html#matplotlib.axes.Axes.get_figu method. Once we have the *Figure* object, we can use its *savefig()* (https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure.savefig) method to save the plot as an image file.

In [273]:
```
import os
import tempfile

temp_file = os.path.join(tempfile.gettempdir(), "chart.png")
fig = sales_plot.get_figure()
fig.savefig(temp_file)
```

Now that we have an image file for the plot, we can embed it in the document. We'll specify the image's width as six inches.

In [274]:
```
sales_report.add_picture(temp_file,
                         width=docx.shared.Inches(6))
```

Out[274]:
```
<docx.shape.InlineShape at 0x11f1b4d30>
```

Finally, we save the document to a file using the *Document* *save()* (https://python-docx.readthedocs.io/en/latest/api/document.html#docx.document.Document.save) method along with the desired file location.

In [275]:
```
sales_report.save("./data/07-report.docx")
```

We can access the file using this link: ./data/07-report.docx (./data/07-report.docx).

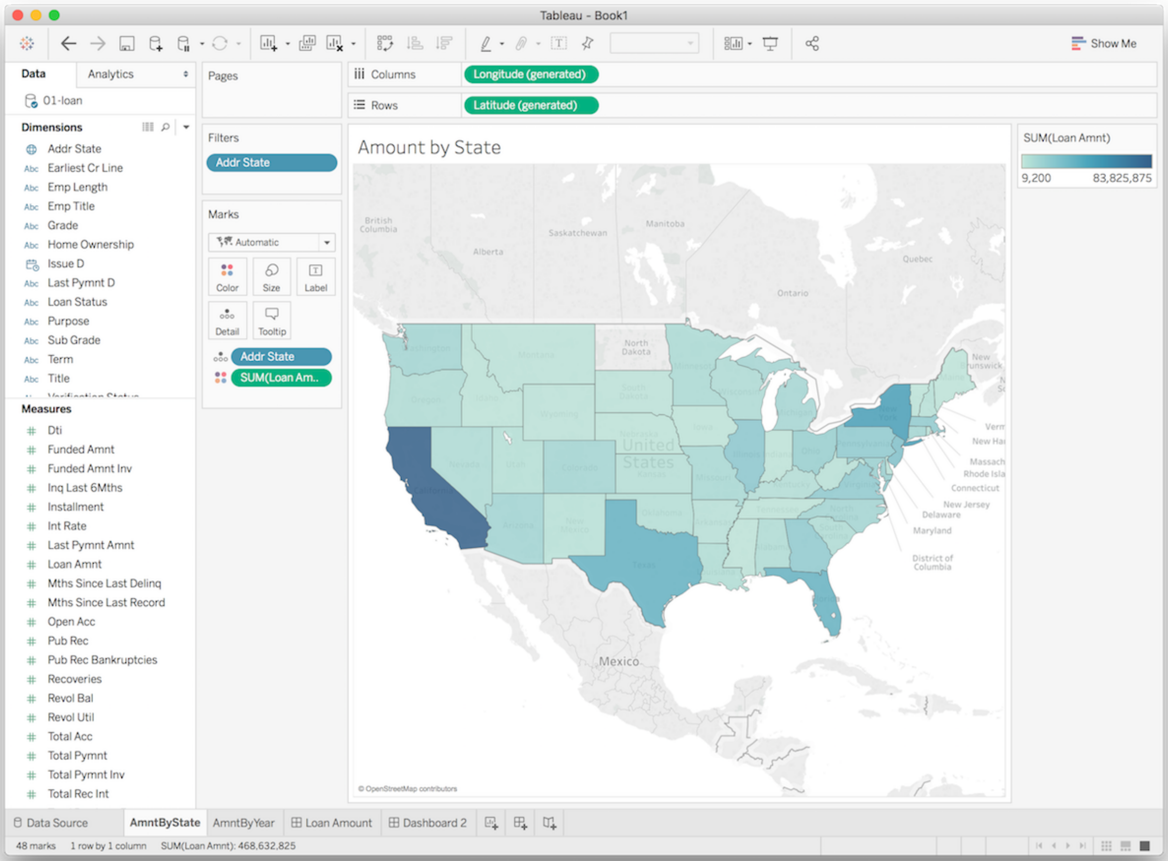
Dashboards

While reports are often used to present summary information with supporting details in the form of a narrative, there is often a need for access to more-frequently-updated summary information. One way of delivering this information is through the use of dashboards. Dashboards are typically presented through a website or some other interactive medium that allows users to quickly and easily determine the state of performance indicators (https://en.wikipedia.org/wiki/Performance_indicator).

Creating Dashboards in Tableau

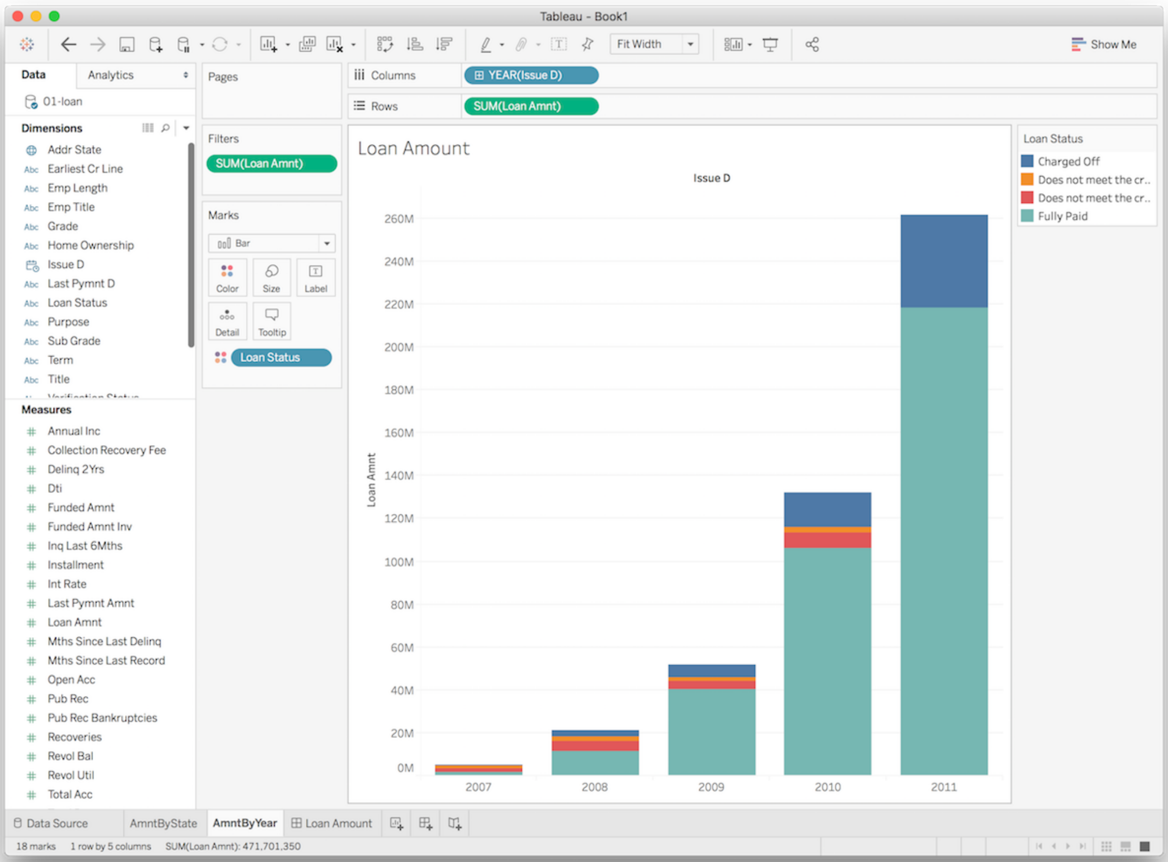
In Tableau, dashboards are generally a collection of visualizations - to create a dashboard, we must first create worksheets containing visualizations of interest. For this example, we'll work with the [Lending Club (https://www.lendingclub.com/info/download-data.action)](https://www.lendingclub.com/info/download-data.action) load data available at [./data/01-loan.csv (./data/01-loan.csv)](./data/01-loan.csv). The first dashboard we create will allow users to view loan amount and status by state.

If necessary, download the data. Open Tableau and load the data file. We'll need to create two worksheets. The first, which we can name *AmntByState* will summarize loan amount by state. To create this visualization, we'll use `Addr State` as data for *Marks* that are used for "detail". We can use `Loan Amnt` as *Marks* data as well but used to indicate "color". Tableau should automatically display a map with each state shaded with a different color depending on the amount of loans issued for that state. To simplify the map, we'll exclude Alaska and Hawaii; to do this, drag `Addr State` to the *Filters* shelf and uncheck `Ak` and `Hi`. Double-click the title and change it to "Amount by State". We should now have a visualization similar to the one in the following image.



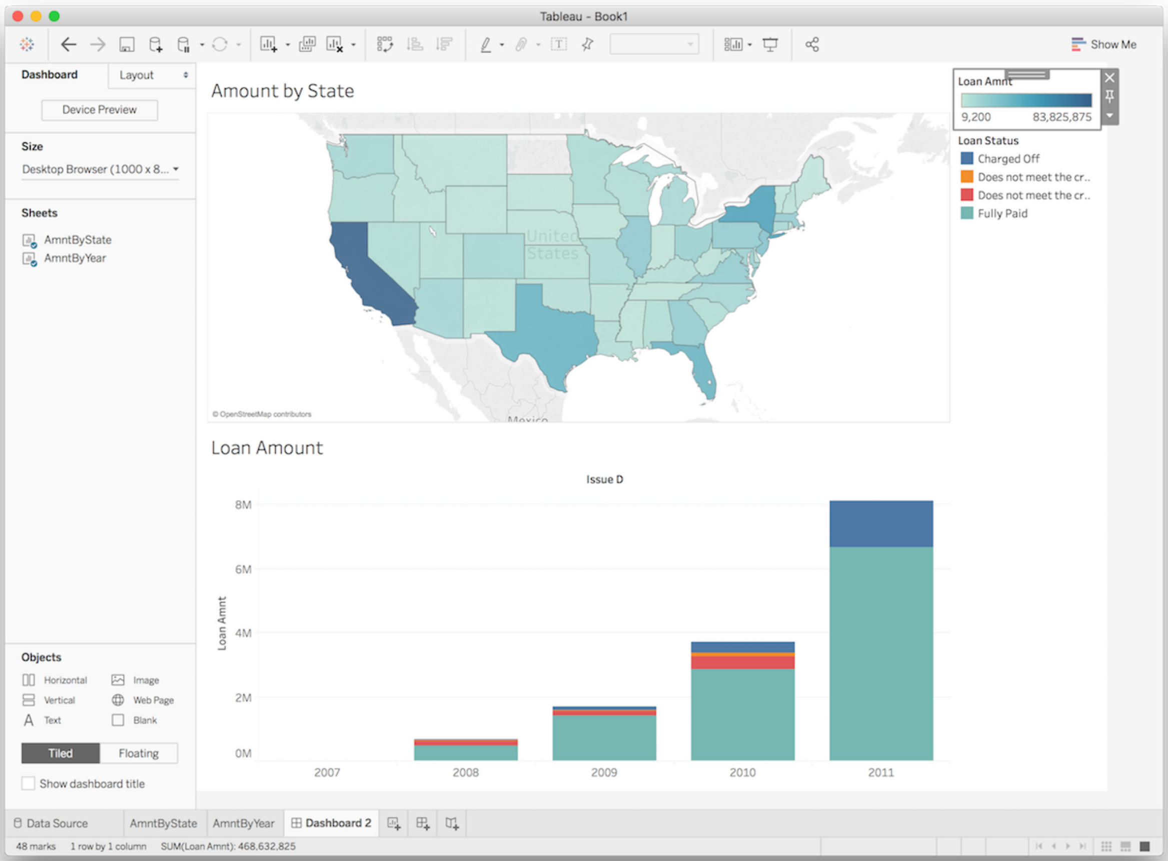
Loan Amount By State

Our next visualization will present a stacked bar chart that presents loan amount by year separated into components based on loan status. To create this chart, we first change the data type of `Issue D` to "Date & Time" and use it for column values. Row values will be based on the sum of `Loan Amnt`; we can create a filter to allow only non-null values of `Loan Amnt`. To create a stacked bar chart, first change the marks to "Bar" and drag `Loan Status` to *Marks*; select the "Color" option for `Loan Status`. Chane the title to "Loan Amount". We should now have a visualization similar to the one in the following image.



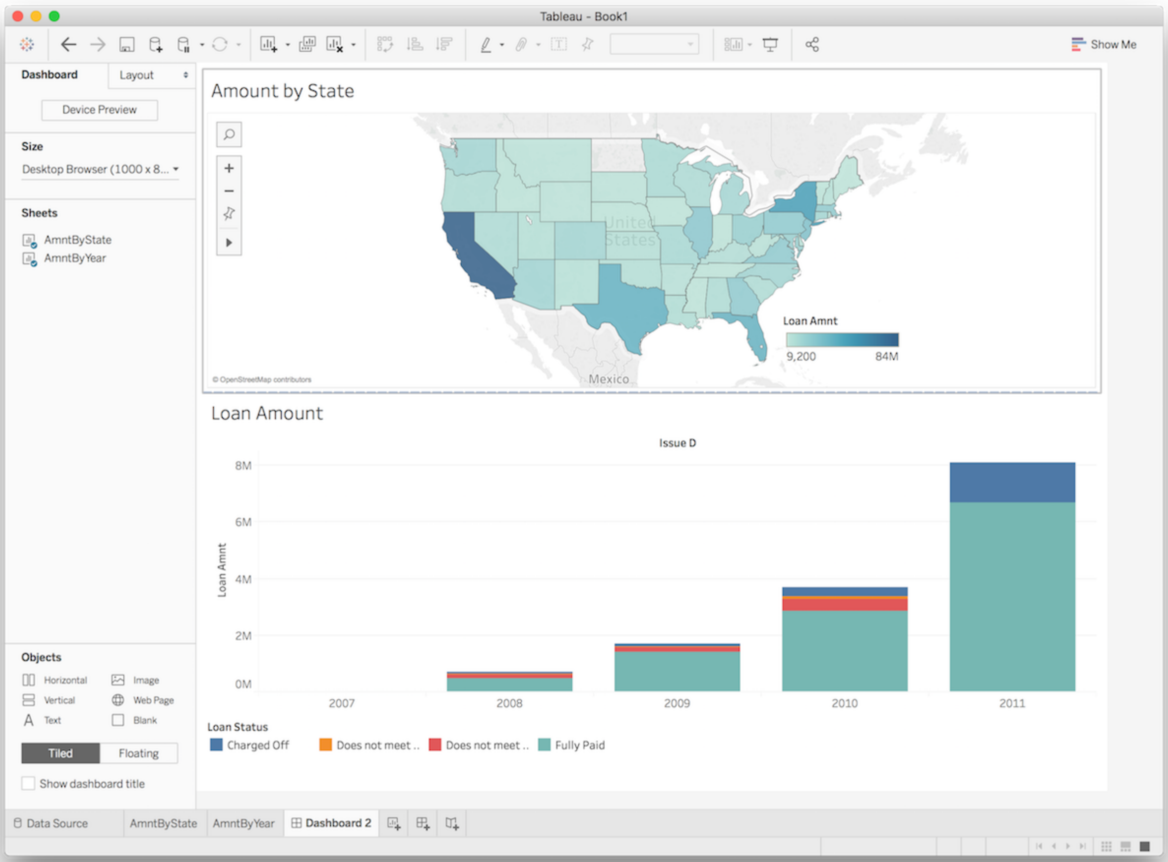
Loan Amount By Year

To create a dashboard using these visualization, click the *New Dashboard* tab; this creates an empty dashboard. We can name the dashboard "Loan Amount". On the left, we can see a list of our existing visualizations. Drag the *AmntByState* visualization to the area labeled `Drop Sheets Here` ; the visualization fills the space. Next, drag the *AmntByYear* visualization to the lower half of the area filled with the other visualization. When you see that the lower half of the area is covered in a transparent gray box, drop the visualization; this splits the area between the two visualizations.



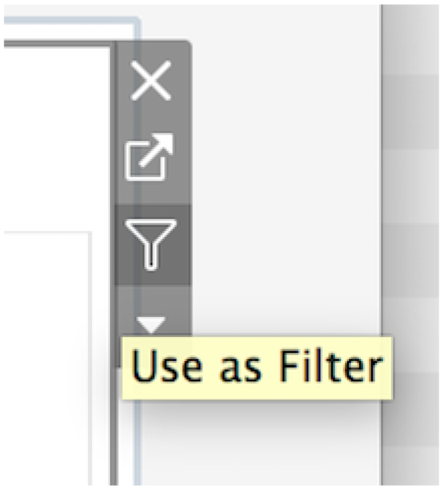
Dashboard with Two Visualizations

Notice that the legends for each visualization appear grouped together on the right. We can move each legend closer to its corresponding visualization. Click on the legend for geographic visualization. On the right, there is a set of buttons including an arrow to access a menu. In this menu is a "Floating" option. Select this option and move the legend to the white space corresponding to the Atlantic Ocean. We can drag the other legend to the area blow the bar chart. After moving it, use its drop-down menu to select "Arrange Items" and "Single Row".



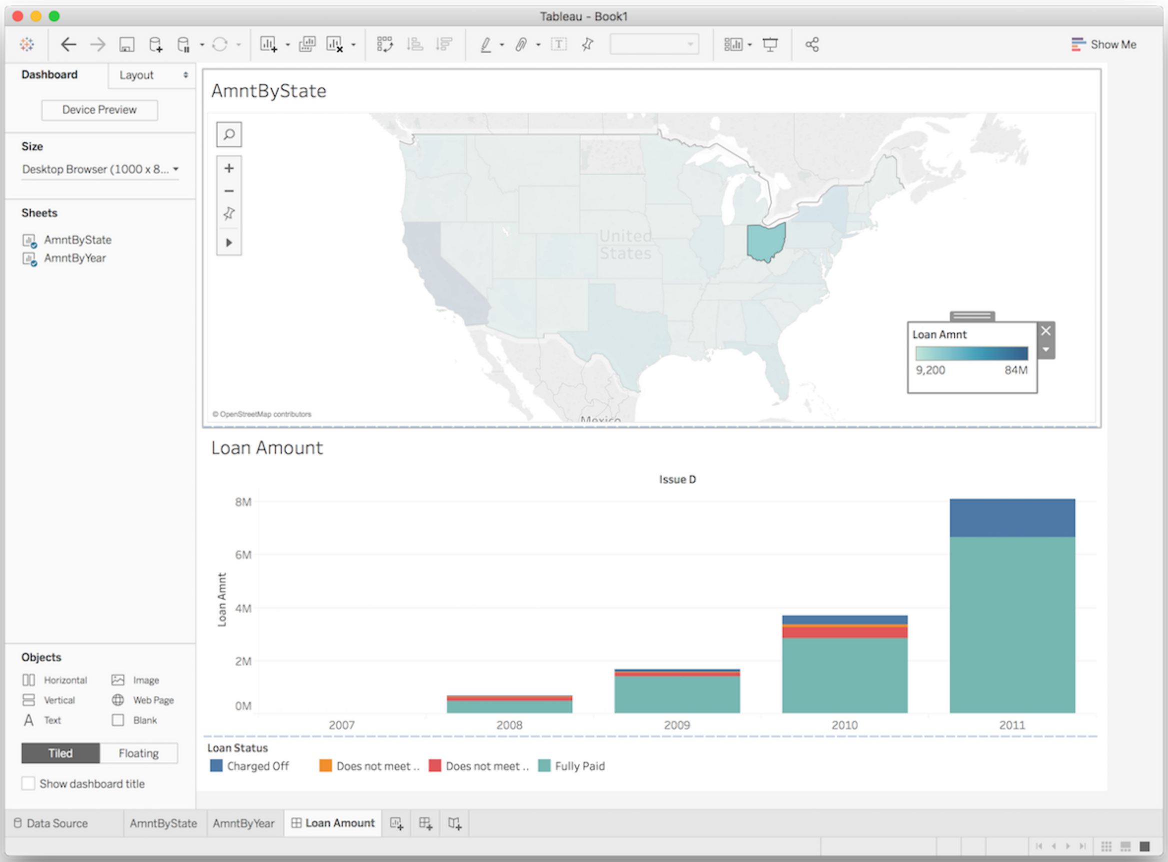
Dashboard with Two Visualizations

Finally, to add some interactivity, click the "Use as filter" button for the first visualization.



Use as Filter

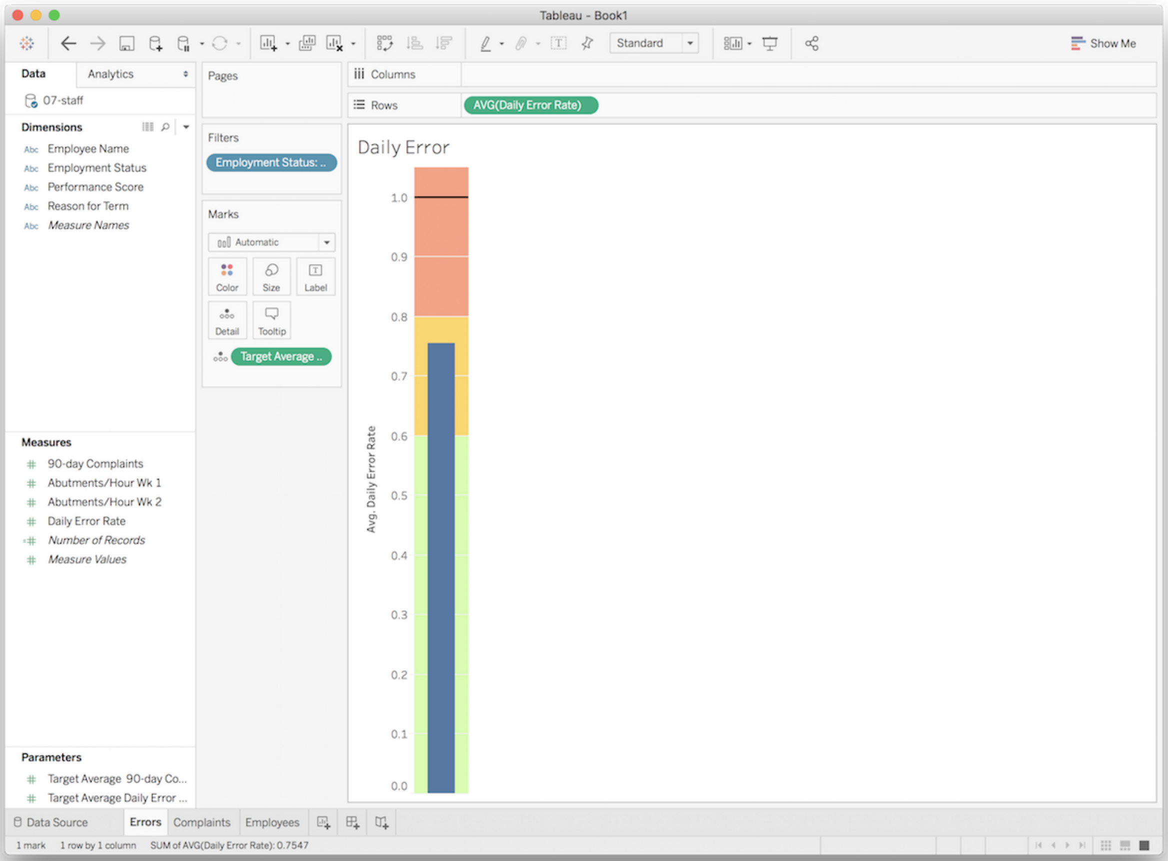
This will cause the other visualizations that depend on the same data to be updated when we select a state in the first visualization - this allows us to see state-specific yearly data.



Interactive Visualization

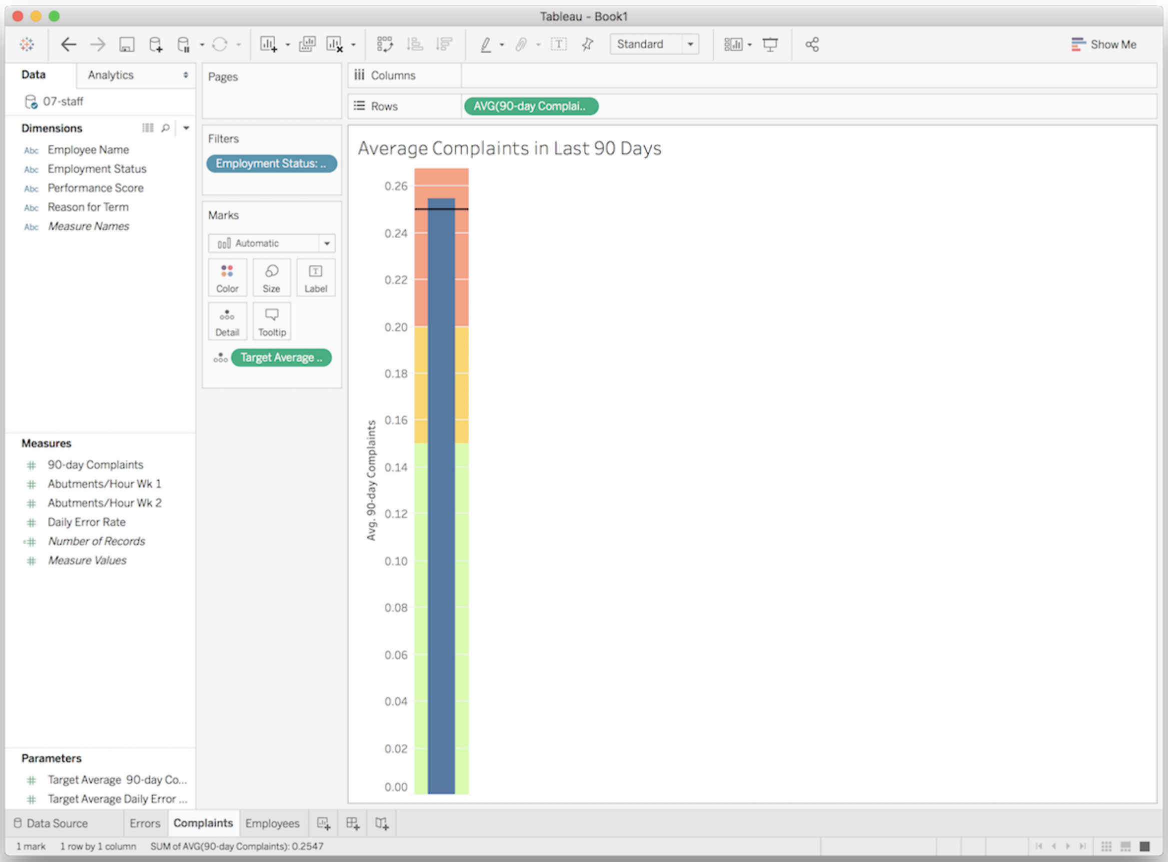
For the next dashboard, we'll work with some data that captures employee performance. The data is located at ./data/07-staff.csv (./data/07-staff.csv). Among the data are daily error rate and number of 90-day complaints per employee. It might be a departmental goal to reduce these numbers or keep aggregates below some threshold. We can create a dashboard that provides this information to management with the ability to filter by employee.

To start, open the dataset in Tableau. On the first worksheet, we'll create a bullet graph (https://en.wikipedia.org/wiki/Bullet_graph) for average daily error rate. A bullet graph is a variation of a bar graph but provides additional such as a desired target and shading or coloring based on percentages of the target. Bullet graphs are alternatives to classic gauges and meters. In addition to a field with data representing the measured value, we need data for target information. If we were working with sales data, we might have actual monthly sales and budgets monthly sales where the budgeted values represent targets. In this data, we don't have an existing field that represents our target; to create one, right-click in the area where fields are displayed and select "Create Parameter". Let's name the new parameter "Target Average Daily Error Rate" and assign it a value of "1". Drag both `Daily Error Rate` and `Target Average Daily Error Rate` to the *Rows* shelf; set the aggregation function for `Daily Error Rate` to average. From the *Show Me* menu, select "bullet graph". If the target field appear in the *Rows* shelf, right click on the values along the vertical axis and click "Swap Reference Line Fields". We might want to only count current employees in the calculation; drag `Employment Status` to the *Filters* shelf and select "Active". Notice that the area from 0 to 60% of the target has a dark gray background, the area between 60% and 80% has a mid-gray background, and the area greater than 80% of the target has a light gray background. To change this coloring, right-click on the values again and select "Edit Reference Line" and "60%, 80% of ...". There are a variety of options here but we'll choose "Stoplight" for the fill color and check "Reverse". Change the title to "Daily Error". The visualization should look like the one in the image below.



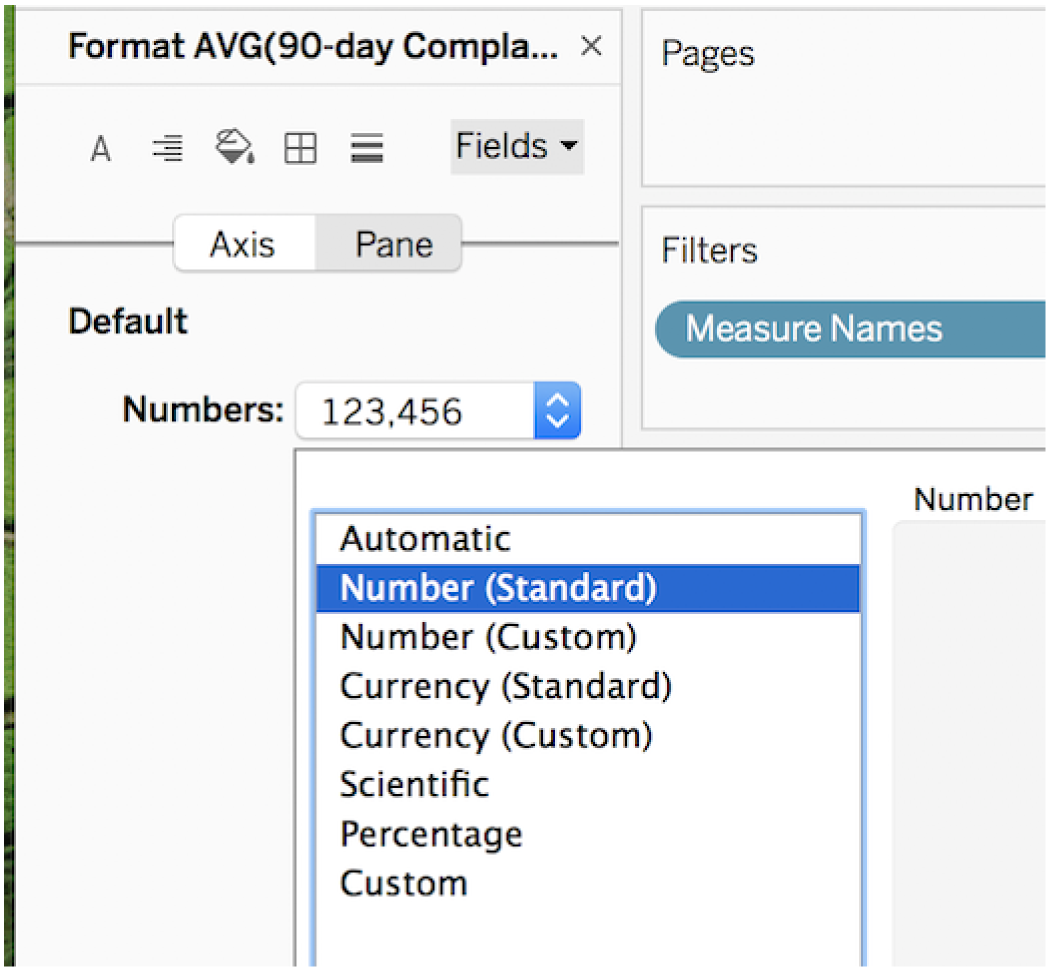
First Bullet Graph

Create a new worksheet with a bullet graph for `90-day Complaints` where the aggregation function is the average and the target is 0.25 (which implies a target of only 1 complaint every year for each employee). Be sure to filter the data so only active employees are used. Use "Average Complaints in Last 90 Days" as the title. The visualization should look like the one in the image below.



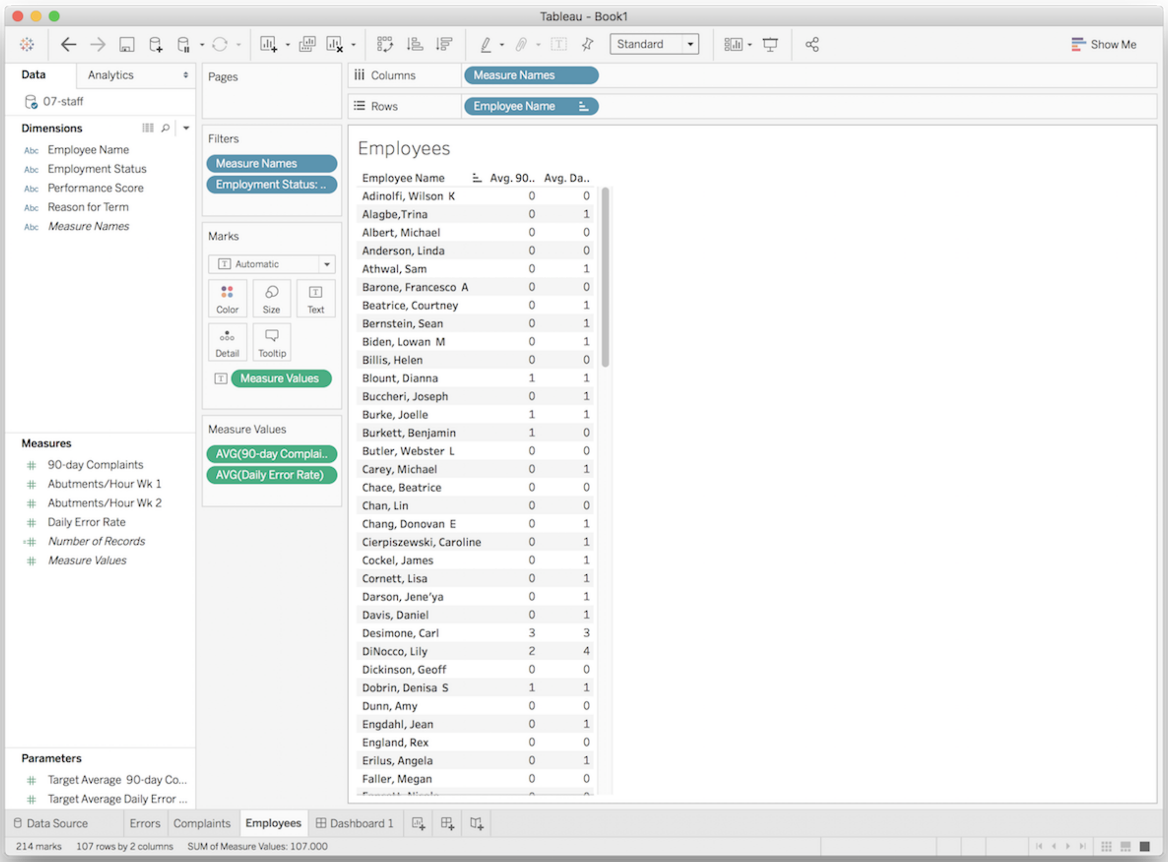
Second Bullet Graph

Our third visualization will be a table of active employees with their complaint and error rate values. Create a new worksheet. Drag `Employee Name` to the *Rows* shelf and `90-day Complaints` and `Daily Error Rate` to *Columns*; be sure to change the aggregation to average if necessary. Select "text table" from the *Show Me* menu. To modify the formatting of the numeric data in the second and third columns, right-click on the values in one column and select "Format". In the "Fields" drop-down menu, select one of the options corresponding to the average calculation. In the "Numbers" drop-down, choose "Numbers (Standard).



Number Format Options

Repeat this for the other column. Be sure to filter the data for active employees only. The table should look similar to the one in the following image.

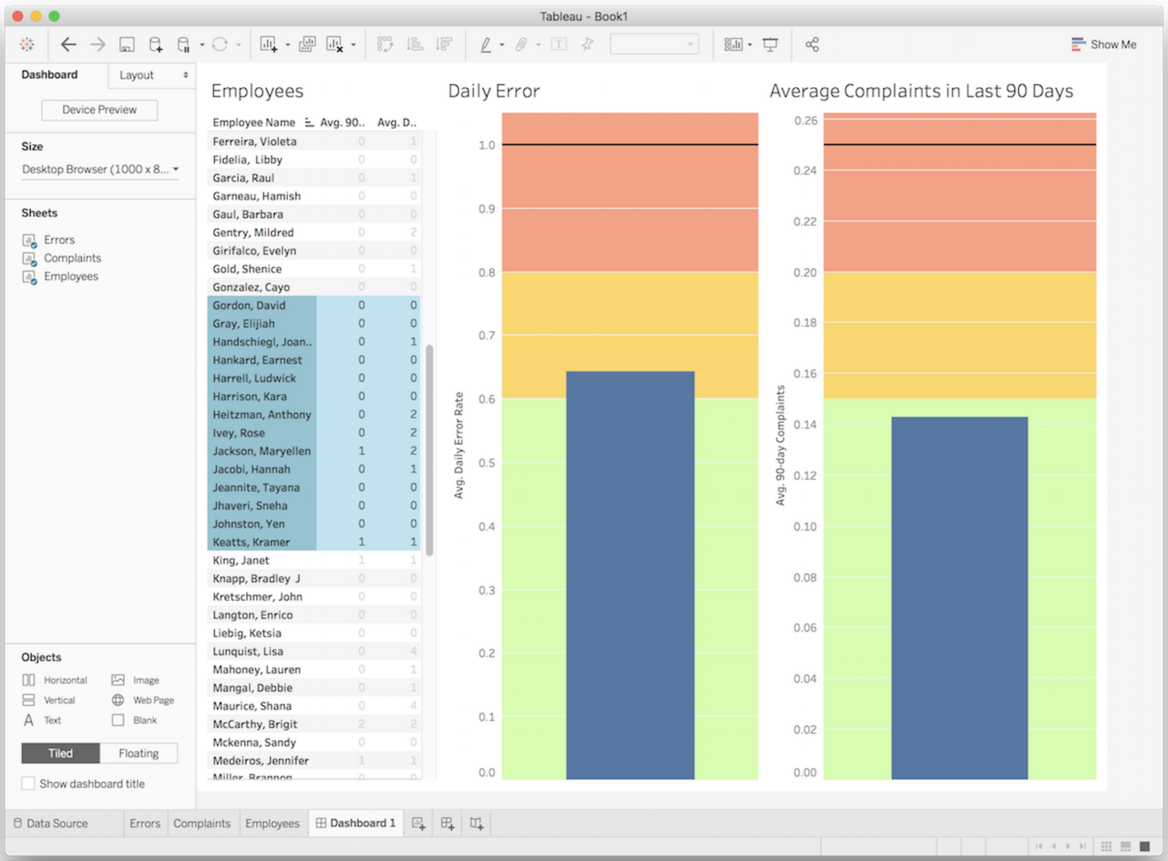


Employees

Employee Name	Avg. 90..	Avg. Da..
Adinolfi, Wilson K	0	0
Alagbe,Trina	0	1
Albert, Michael	0	0
Anderson, Linda	0	0
Athwal, Sam	0	1
Barone, Francesco A	0	0
Beatrice, Courtney	0	1
Bernstein, Sean	0	1
Biden, Lowan M	0	1
Billis, Helen	0	0
Blount, Dianna	1	1
Buccheri, Joseph	0	1
Burke, Joelle	1	1
Burkett, Benjamin	1	0
Butler, Webster L	0	0
Carey, Michael	0	1
Chace, Beatrice	0	0
Chan, Lin	0	0
Chang, Donovan E	0	1
Cierpiszewski, Caroline	0	1
Cockel, James	0	1
Cornett, Lisa	0	1
Darson, Jene'ya	0	1
Davis, Daniel	0	1
Desimone, Carl	3	3
DiNocco, Lily	2	4
Dickinson, Geoff	0	0
Dobrin, Denisa S	1	1
Dunn, Amy	0	0
Engdahl, Jean	0	1
England, Rex	0	0
Erilus, Angela	0	1
Faller, Megan	0	0

Employee Table

We can bring these components together into a dashboard. Create a new dashboard and drag the employees table to the dashboard area. Drag the daily error visualization to the right of the employees table and the complaints visualization to the right of the error visualization. Adjust the width of each section as needed to avoid excess whitespace. With an individual visualization selected, we can change how it fills its pane using the drop-down in the toolbar and selecting "Fit Width". Click the "Use as Filter" button for the employees table. We can now display error and complaint graphs for all employees, individual employees, or a groups of employees - use CTRL/CMD or SHIFT to select multiple employees.



Employees

Employee Name	Avg. 90..	Avg. D..
Ferreira, Violeta	0	1
Fidelia, Libby	0	0
Garcia, Raul	0	1
Garneau, Hamish	0	0
Gaul, Barbara	0	0
Gentry, Mildred	0	2
Girifalco, Evelyn	0	0
Gold, Shenice	0	1
Gonzalez, Cayo	0	0
Gordon, David	0	0
Gray, Elijiah	0	0
Handschiegl, Joan..	0	1
Hankard, Earnest	0	0
Harrell, Ludwick	0	0
Harrison, Kara	0	0
Heitzman, Anthony	0	2
Ivey, Rose	0	2
Jackson, Maryellen	1	2
Jacobi, Hannah	0	1
Jeannite, Tayana	0	0
Jhaveri, Sneha	0	0
Johnston, Yen	0	0
Keatts, Kramer	1	1
King, Janet	1	1
Knapp, Bradley J	0	0
Kretschmer, John	0	0
Langton, Enrico	0	0
Liebig, Ketsia	0	0
Lunquist, Lisa	0	4
Mahoney, Lauren	0	1
Mangal, Debbie	0	1
Maurice, Shana	0	4
McCarthy, Brigit	2	2
Mckenna, Sandy	0	0
Medeiros, Jennifer	1	1
Miller, Brannon	0	0

Performance Dashboard

To get a sense of the dashboard without the rest of the Tableau interface, select "Window" and "Presentation Mode" from the menu; use ESC to exit.

While it's pretty easy to create visually appealing, interactive dashboard in Tableau, in order to make the dashboard easily available to many users, we need to use Tableau Online or Tableau Server. We also have the option of distributing the Tableau workbook containing our worksheets and dashboards along with the free Tableau Reader (https://www.tableau.com/products/reader). An alternative method of creating and distributing dashboards is to use Python.

Creating Dashboards with Python

In the following example, we'll create a dashboard and run a web server from within the notebook. If you are executing this notebook with Azure Notebooks, you will not be able to access the server so there is no need to run the code but you can follow along.

To create web dashboards with Python, we'll use the Dash (https://plot.ly/products/dash/) library; Dash allows us to create web-based user interfaces from Python (rather than having to work directly with HTML, CSS, and Javascript). Dash was created by Plotly (https://plot.ly/), a company that provides tools to create web-based data visualizations.

We'll need to install to install Dash and related libraries using `pip`.

- dash: the core Python library
- dash-renderer: front-end code

- dash-html-components: HTML components
- dash-core-components: core components in Javascript
- plotly: graphing library

In [ ]:
```
import sys
!{sys.executable} -m pip install dash dash-renderer dash-html-components das
```

We'll start with a simple example to get an idea of Dash's syntax and what we need to do to host the dashboard.

We start by importing the necessary modules.

In [276]:
```
import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly.graph_objs as go
```

Executing the code for our dashboard across multiple notebook cells can lead to errors so we'll execute it all at once. Let's examine the code in the following cell before executing it.

```
app = dash.Dash()
```

In the first line, we create an instance of the *Dash()* class which includes the functionality for starting a web server and hosting the dashboard.

```
app.layout = html.Div(children=[
    html.H1(children='Sales'),
    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [2015, 2016, 2017], 'y': [4, 1, 2], 'type': 'b
ar', 'name': 'North'},
                {'x': [2015, 2016, 2017], 'y': [2, 4, 5], 'type': 'b
ar', 'name': 'South'},
            ],
            'layout': {
                'title': 'Sales by Region (in millions)'
            }
        }
    )
])
```

This block of code defines the content of our dashboard as a layout to the *Dash* instance. The layout consists of different components. First, we create an HTML div (https://www.w3schools.com/tags/tag_div.asp) with the *Div()* function to contain our content. When we create a div, we specify the elements contained within it using the *children* argument; here the div has two children: an H1 (https://www.w3schools.com/tags/tag_hn.asp) heading element and an instance of the Dash *Graph()* class.

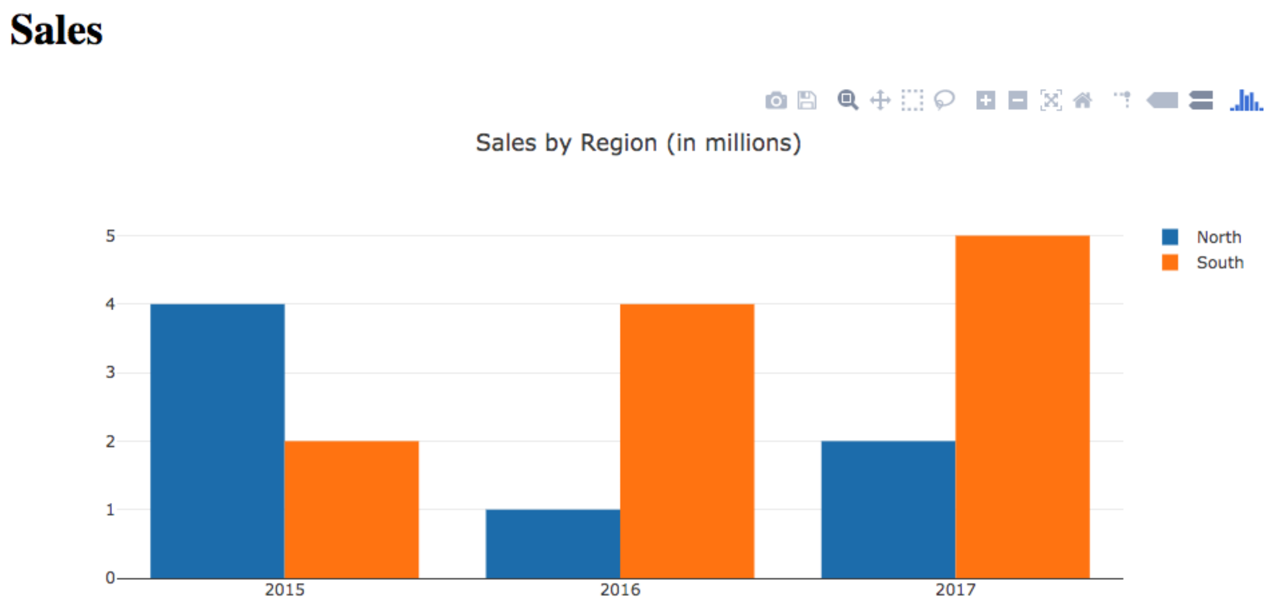
When creating a Dash *Graph()*, we can specify its *id*, which is used to interact with the graph, and the *figure*, which is specified with a dictionary containing the information used to generate the plot. The *figure* dictionary requires two two key-value pairs: the key *data* paired with data used to generate the plot and the key *layout* paired with presentation information.

```
app.run_server()
```

With the page and its content defined, we can serve it using the *run_server()* method. By default, Dash will host the page on the local computer on port (https://en.wikipedia.org/wiki/Port_%28computer_networking%29) 8050 and can be accessed by opening http://localhost:8050 (http://localhost:8050) in a browser. To stop the server, make sure the cell in which the server was started is selected, and press the stop button above or select "Kernel" and "Interrupt" from the menus.

Only one web server can be running at a time on the default port. If you encounter any errors when running the web servers, make sure you've stopped any other server in this notebook. It is also possible for an ad blockers to block content needed to create the pages.

When we run the code below, the dashboard page will look similar to the image below.



Simple Dash Dashboard

In [277]:
```
app = dash.Dash()

app.layout = html.Div(children=[
    html.H1(children='Sales'),
    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [2015, 2016, 2017], 'y': [4, 1, 2], 'type': 'bar', 'na
                {'x': [2015, 2016, 2017], 'y': [2, 4, 5], 'type': 'bar', 'na
            ],
            'layout': {
                'title': 'Sales by Region (in millions)'
            }
        }
    )
])

app.run_server()
```

```
 * Running on http://127.0.0.1:8050/ (http://127.0.0.1:8050/) (Press CTRL
+C to quit)
127.0.0.1 - - [17/Apr/2018 21:09:19] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [17/Apr/2018 21:09:20] "GET /_dash-layout HTTP/1.1" 200 -
127.0.0.1 - - [17/Apr/2018 21:09:20] "GET /_dash-dependencies HTTP/1.1" 2
00 -
127.0.0.1 - - [17/Apr/2018 21:09:20] "GET /favicon.ico HTTP/1.1" 200 -
```

Nearly every time we've worked with data in Python, we've used pandas. Conveniently, we use DataFrames with Dash. For the remaining examples, we'll make use of the EPA fuel economy data. We start by loading the data in a DataFrame. We'll focus on the `co2`, `comb08`, `VClass`, and `year` columns.

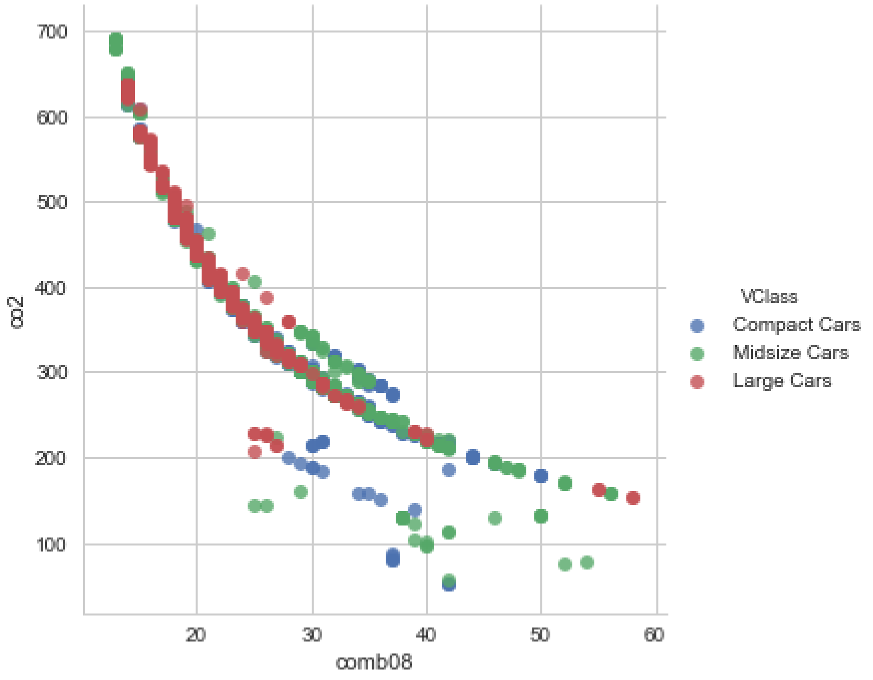
In [278]:
```
epa_data = pd.read_csv("./data/02-vehicles.csv", engine="python")
epa_data = epa_data[['co2', 'comb08', 'VClass', 'year']]
epa_data.dropna(inplace=True)
```

With respect to `VClass`, we'll consider three values: `Compact Cars`, `Midsize Cars`, and `Large Cars`. Recall also that the data contains records where `co2` is negative; we'll exclude those values.

In [283]:
```
epa_data = epa_data[epa_data.VClass.isin(
    ['Compact Cars', 'Midsize Cars', 'Large Cars'])]
epa_data = epa_data[epa_data.co2 > 0]
```

Recall that we previously looked at scatter plots involving some of this data.

```
In [286]: import seaborn as sns
          sns.lmplot(x='comb08', y='co2', hue='VClass', data=epa_data, fit_reg=False)

Out[286]: <seaborn.axisgrid.FacetGrid at 0x11f2b4978>
```



We can create something similar with Dash. Before creating the instance of the *Dash()* class, we can define the figure data. Before defining the figure data, we'll define a set of marker properties that will be used for each data point in the plot.

```
markers = {'size': 15,
           'line': {'width': 0.5, 'color': 'white'}}
```

This sets each marker's size and the outline color and width.

In order to color each set of data points based on the category values of `VClass` , we have add the figure data for each category separately.

```
figure_data = []
for v_class in epa_data.VClass.unique():
    xs = epa_data[epa_data.VClass == v_class].comb08
    ys = epa_data[epa_data.VClass == v_class].co2

    figure_data.append(
        go.Scatter(x=xs, y=ys, mode='markers', opacity=0.7,
                   marker=markers, name=v_class))
```

We create an empty list that will be used to store the data for the figure we create later. We use a for-loop to iterate through the classes in `VClass` . For each class, we extract the the associated `comb08` values and assign them to the variable `xs` . Similarly, `co2` values are assigned to `ys` . With this data, we create an scatter plot using the [*Scatter()* (https://plot.ly/python/line-and-scatter/)](https://plot.ly/python/line-and-scatter/) class and append it to our list of figure data.

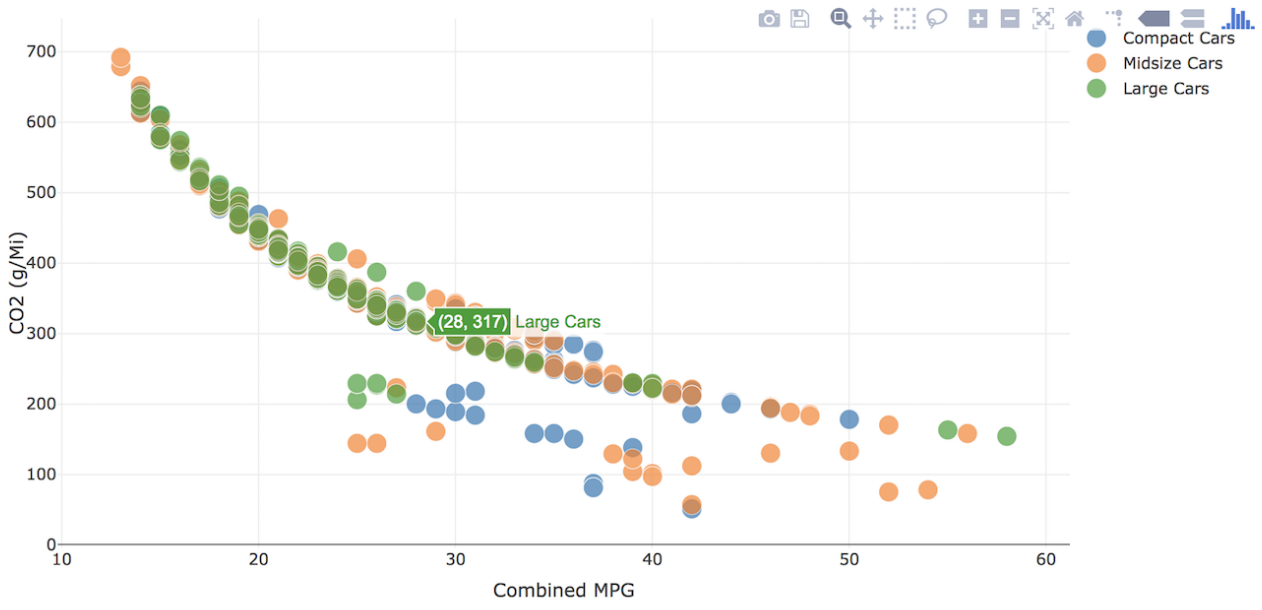
Next, we'll create an instance of *Dash()* as before and specify its layout.

```
app = dash.Dash()

app.layout = html.Div([
    dcc.Graph(
        id='life-exp-vs-gdp',
        figure={
            'data': figure_data,
            'layout': go.Layout(
                xaxis={ 'title': 'Combined MPG'},
                yaxis={'title': 'CO2 (g/Mi)'},
                margin={'l': 40, 'b': 40, 't': 10, 'r': 10},
                legend={'x': 1, 'y': 1},
                hovermode='closest'
            )
        }
    )
])

app.run_server()
```

Here, we use the `figure_data` list we prepared previously to provide data for the graph's figure. We specify additional information about the axes, the margins, and the legend as well.

Once we start the server, it can be accessed by opening http://localhost:8050 (http://localhost:8050) in a browser. To stop the server, press the stop button above or select *Kernel* and *Interrupt* from the menus.

The code below will create a page with a visualization like the one below. Note that we can click on values in the legend to hide/show different vehicle classes.



Dash Dashboard

In [288]:
```
markers = {'size': 15,
           'line': {'width': 0.5, 'color': 'white'}}

figure_data = []
for v_class in epa_data.VClass.unique():
    xs = epa_data[epa_data.VClass == v_class].comb08
    ys = epa_data[epa_data.VClass == v_class].co2

    figure_data.append(
        go.Scatter(x=xs, y=ys, mode='markers', opacity=0.7,
                   marker=markers, name=v_class))

app = dash.Dash()

app.layout = html.Div([
    dcc.Graph(
        id='life-exp-vs-gdp',
        figure={
            'data': figure_data,
            'layout': go.Layout(
                xaxis={ 'title': 'Combined MPG'},
                yaxis={'title': 'CO2 (g/Mi)'},
                margin={'l': 40, 'b': 40, 't': 10, 'r': 10},
                legend={'x': 1, 'y': 1},
                hovermode='closest'
            )
        }
    )
])

app.run_server()
```

```
 * Running on http://127.0.0.1:8050/ (http://127.0.0.1:8050/) (Press CTRL
+C to quit)
127.0.0.1 - - [17/Apr/2018 22:03:34] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [17/Apr/2018 22:03:35] "GET /_dash-layout HTTP/1.1" 200 -
127.0.0.1 - - [17/Apr/2018 22:03:35] "GET /_dash-dependencies HTTP/1.1" 2
00 -
127.0.0.1 - - [17/Apr/2018 22:03:35] "GET /favicon.ico HTTP/1.1" 200 -
```

We can add multiple graphs to a page by creating additional *Graph()* elements. We can also create more interactive graphs that allow users to drill-down or filter data; see the Dash User Guide (https://dash.plot.ly/getting-started) for more details.

Next Steps

In the next unit, we'll begin to look at how we can automate some of the work we've done in previous units. An automated process could be used to generate reports or to refresh the data used to populate a dashboard.

Resources and Further Reading

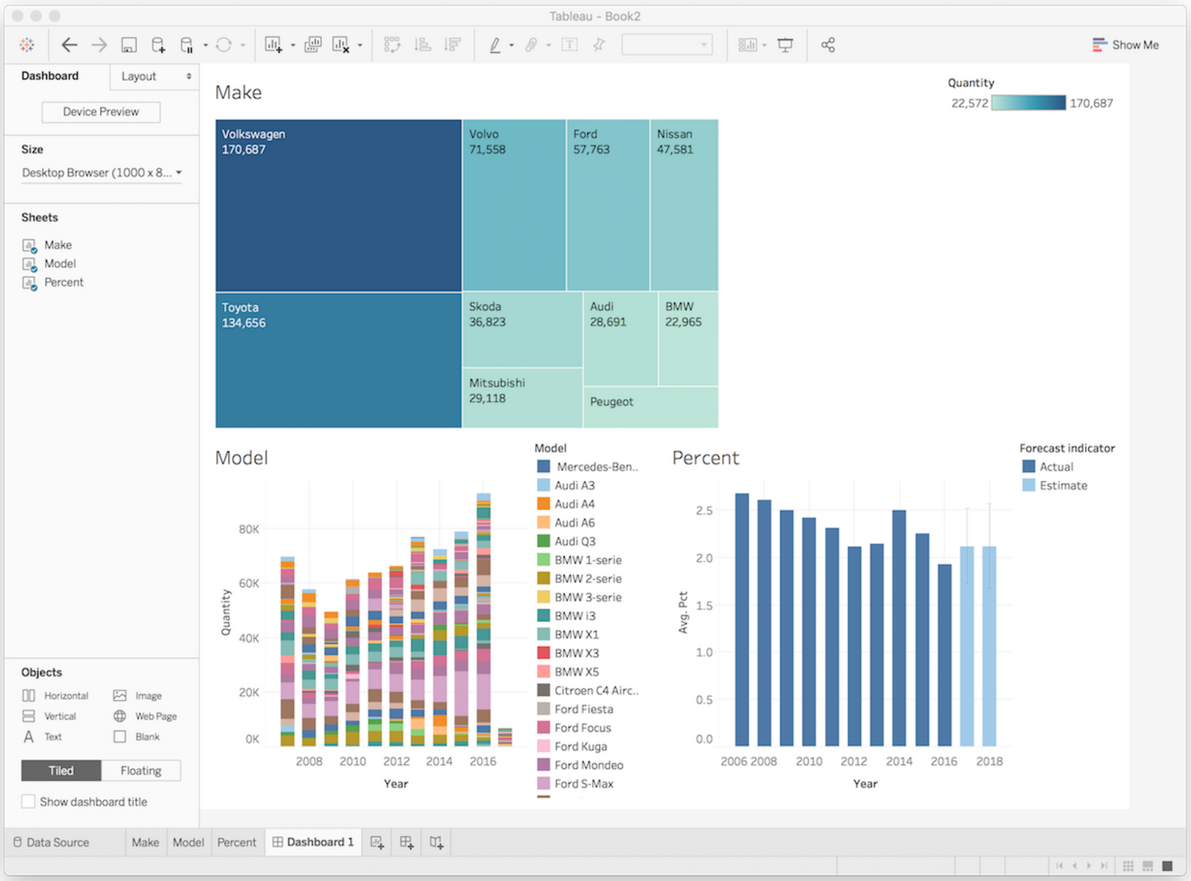
- Automate the Boring Stuff by Sweigart, Chapter 13: Working with PDF and Word Documents (https://automatetheboringstuff.com/chapter13/)
- Openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files (https://openpyxl.readthedocs.io/en/stable/)
- Plotly HTML Reports in Python (https://plot.ly/python/html-reports/)
- python-pptx - A Python library for creating and updating PowerPoint files (https://python-pptx.readthedocs.io/en/latest/)
- Information Dashboard Design by Few (https://archive.org/details/pdfy--fQ3cC8TeDUArgti)
- Tableau - Best Practices for Effective Dashboards (https://onlinehelp.tableau.com/current/pro/desktop/en-us/dashboards_best_practices.html?tocpath=Design%20Views%20and%20Analyze%20Data%7CPresent%20Your%20Work%7CDas
- Tableau Dashboard Cookbook by Stirrup (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/databases/business-intelligence/9781782177906)

Exercise

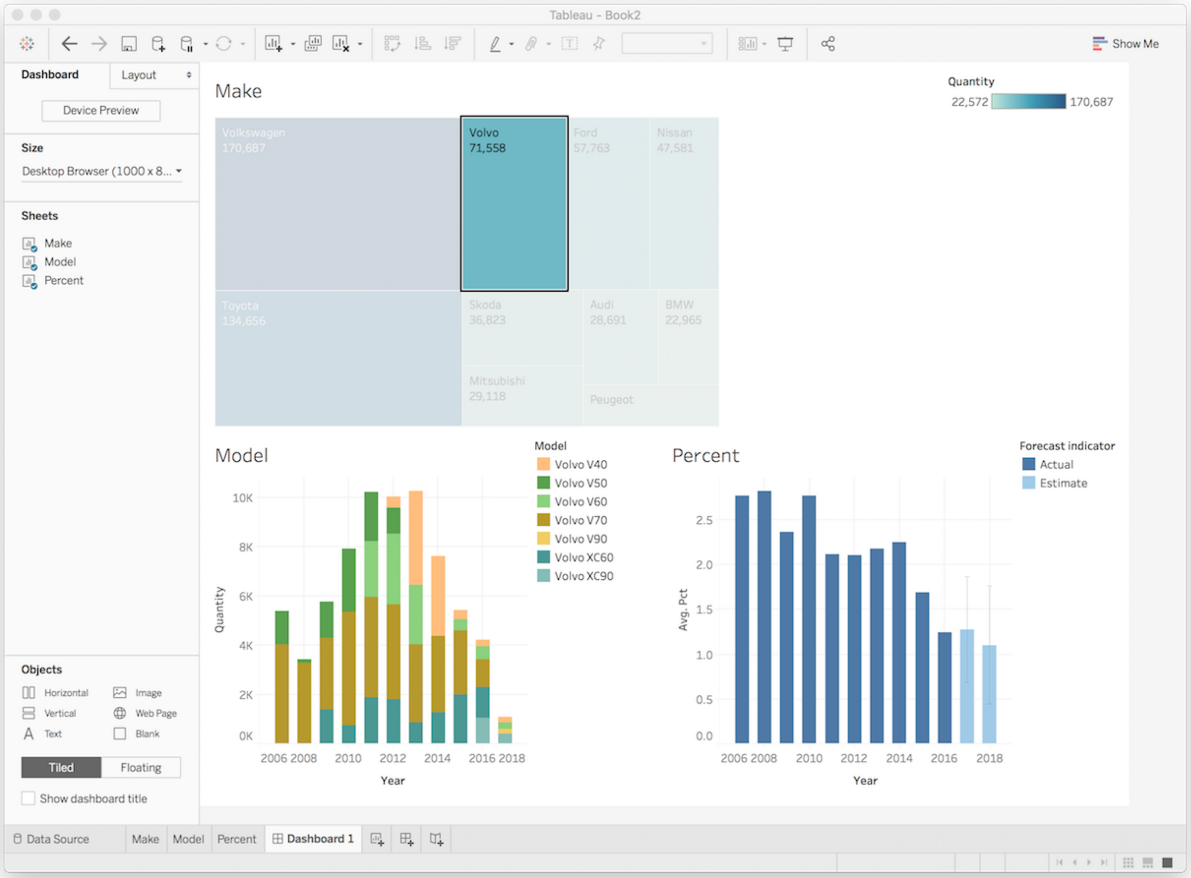
Using the Norwegian vehicle sales data in ./data/02-vehicle-sales-norway.csv (./data/02-vehicle-sales-norway.csv), create a dashboard with the following components:

- A treemap showing the top 10 vehicle manufacturers, `Make` by total quantity sold, `Quantity`. Display both the make and total quantity as labels for the rectangles in the treemap.
- A stacked bar chart showing total quantity by year where each bar is divided into colors based on `Model`; don't filter by make or model - we'll add that functionality in the dashboard.
- A bar chart showing average percent of sales, `Pct`, by year. Show the forecast values as well.

Use the treemap as the top half of the dashboard and split the bottom half between the model bar chart and the percentage bar chart. Use the treemap as a filter for the other two visualizations such that selecting one make will update the model and percentage charts with data specific to that manufacturer. See the images below.



Exercise Dashboard



Exercise Dashboard with Manufacturer Selected

```
In [ ]:
```