Data Visualisation With Bokeh

Last week we looked at how statistics could inform our understanding of data. In this week, we look at how data can be visualised, in particular using the Bokeh (http://bokeh.pydata.org/en/latest/) library.

We used some of the basic `Chart` (http://bokeh.pydata.org/en/0.12.0/docs/reference/charts.html#bokeh-charts) functions last week to visualise distributions. This week will focus on geographical data and how this can be represented on a map.

In [1]:

```
# You don't need to write anything here
# Add the initial imports
import pandas as pd
from bokeh.plotting import Figure
from bokeh.io import show, output_notebook, push_notebook
from bokeh.models import *
#from bokeh.plotting import figure
from bokeh.tile_providers import WMTSTileSource #보케 타일 공급자
from ipywidgets import *
import ipywidgets as widgets


import numpy as np
output_notebook()

# We will need this function to calculate a position on the map when given lattitude
def wgs84_to_web_mercator(df, lon="lon", lat="lat"):
    """
    Converts decimal longitude/latitude to Web Mercator format
    Source https://github.com/bokeh/bokeh-notebooks/blob/master/tutorial/11%20-%20ge
    """
    k = 6378137
    df["x"] = df[lon] * (k * np.pi/180.0)
    df["y"] = np.log(np.tan((90 + df[lat]) * np.pi/360.0)) * k
    return df
```


Widgets

Widgets are functions which convert **Python** code to **HTML** code for output in the notebook. Bokeh has its own widgets, but for this exercise we will focus on the widgets used within the Jupyter Notebook:**`ipywidgets` (http://ipywidgets.readthedocs.io/en/latest/index.html)**. These widgets could range from a simple text description to interactive widgets which may modify the appearance of Bokeh visualisations in real time.

We will start with a 'slider' (http://bokeh.pydata.org/en/latest/docs/gallery/slider.html) widget, which we will use to modify a simple line graph. We will use the `interact` (http://ipywidgets.readthedocs.io/en/latest/examples/Using%20Interact.html) function to do this. It works by passing a function name as a parameter, and every time that the slider is moved, it calls the function. We demonstrate this in the following cell:

In [ ]:

```
# You don't need to write anything here

# This function is called every time we change the value on the slider
# Notice that we are calling the function 'interact' which creates an interactive wi
# 'interact' is a function called from ipywidgets
def f(x):
    print("Move the slider!", x)
    return x
# x=2 means that the first time it calls the function, the `x` parameter of `f` will
interact(f, x=3)
```

In the next cell, we introduce the `Figure` class (http://bokeh.pydata.org/en/latest/docs/reference/plotting.html#bokeh.plotting.figure.Figure) to create our line graph. We briefly demonstrated this in **week 3** when demonstrating the Iris dataset for linear regression. For this week, we will go into a bit more about it, since we will be using it in our visualisation in the assignment.

We use the `line` method (http://bokeh.pydata.org/en/0.10.0/docs/reference/plotting.html#bokeh.plotting.Figure.line) to add our line to the graph. On this occasion, we are presenting a simple graph of $y = x^2$, and use the `show` function to display the graph. Notice that we pass it **`notebook_handle=true` (http://bokeh.pydata.org/en/latest/docs/user_guide/notebook.html#working-in-the-notebook)**. This gives other methods a means of dynamically updating it later.

In [ ]:

```
# You don't need to write anything here
# Here we create an empty figure to which we add a line representing y=x**2

fig = Figure()
li = fig.line(x=pd.Series([1,2,3,4,5,6,7]), y=pd.Series([1,4,9,16,25,36,49]))

show(fig, notebook_handle=True)
```

If we want to update the figure, we can make use of the **`handle` (http://bokeh.pydata.org/en/latest/docs/user_guide/notebook.html#notebook-handles)** we passed to the `Figure`. Bokeh has a function for this called **`push_notebook` (http://bokeh.pydata.org/en/latest/docs/reference/io.html)**, which will push any changes made to the notebook since the last time it was called.

To **update** the data, we are going to update the data source of the line, by changing the values of the y axis, so we have an equation of $y = x^3$.

So, if we call as follows, it will update the figure above:

In [ ]:

```
# You don't need to write anything here
# Now, we can adjust the value in the line figure above, and see the data change
li.data_source.data['y'] = li.data_source.data['x'] **3
push_notebook()
```

That shows the two elements required to make an interactive visualisation. Now, try and put it all together:

- Create a slider using `interact`, which calls function `g(x)`
- `g(x)` should update the line of a plot to be x to the power `g(x)`
- If you can make a slider work with integers, try and update using a `FloatSlider` (http://ipywidgets.readthedocs.io/en/latest/examples/Using%20Interact.html) which increases in value 0.1 for every change instead of an `IntSlider` as in the example above.

In [ ]:

```
# YOUR CODE HERE
# Place your figure code in this cell
```

In [ ]:

```
# YOUR CODE HERE
# Place your slider code in this cell
```

Tools

Notice on the side of the `Figure` object, there are a series of buttons. These allow interactive exploration of the figure. By default, Bokeh adds some to our figure, but we can add them ourselves in two ways.

Firstly, we can use the `add_tool` (http://bokeh.pydata.org/en/latest/docs/user_guide/tools.html#specifying-tools) function for each tool manually. For example, we might want to add the `LassoSelectTool`, we could simply pass an instance of this class to the `add_tool` function as follows:

In [ ]:

```
fig.add_tools(LassoSelectTool())
show(fig)
```

The easier way of doing this is to pass to the tools parameter when the `Figure` is created, which can be done in one of two ways:

- By passing a list of `Tool` instances
- A comma separated string of different tools

See the two examples of code doing the same thing on empty `Figure` instances as follows:

In [ ]:

```
tools_fig_1 = Figure(tools=[WheelZoomTool(), BoxSelectTool()], height=100)
show(tools_fig_1)
```

In [ ]:

```
tools_fig_2 = Figure(tools='wheel_zoom,box_select', height=100)
show(tools_fig_2)
```

The **full list** of **available tools (http://bokeh.pydata.org/en/latest/docs/user_guide/tools.html#specifying-tools)** and their usage can be seen in the **Bokeh tools user guide (http://bokeh.pydata.org/en/latest/docs/user_guide/tools.html)**. If you don't remember the name of the tool you want and enter the wrong value, Bokeh will warn you and give you some suggestions for tools you might like to add.

Data Sources

This (updating the graph/data) was possible, because when we created the line to display on our graph, we gave it the variable `li`, and we were still able to access the `li` variable. This was a `Glyph` (http://bokeh.pydata.org/en/latest/docs/user_guide/plotting.html#plotting-with-basic-glyphs), which has associated with it a `ColumnDataSource` (http://bokeh.pydata.org/en/latest/docs/reference/models/sources.html#bokeh.models.sources.ColumnDataSou type created from the values inserted.

We have already seen a Pandas `DataFrame`, which is a generic data structure for holding data. The `ColumnDataSource` is part of `Bokeh` rather than `Pandas`, and is used specifically as a means of storing data for a graph.

This object can be accessed as the `data_source` of the `Glyph`, and the `data` attribute is a series of key/value pairs derived from the source data.

In [ ]:

```
# You don't need to write anything here
# Show the variables associated with the data source of the `ColumnDataSource`
print(vars(li.data_source))

# Show a column of the source data
print('\n`y` data from the graph:\n', li.data_source.data['y'])
```

Having introduced the concept of widgets, figures, and the ColumnDataSource, we are now going to make use of the Bokeh map tiling (http://geo.holoviews.org/Working_with_Bokeh.html) feature. We will use sample data from Bokeh, based on states in the USA, and we will match each of these states to the winner in the US presidential election of 2016.

We will create a map of the USA, which will add the correct colour to an individual state when we select from a checkbox.

To begin, we will import the data, and add colour for a single state (California):

In [ ]:

```
# You don't need to write anything here
# Here we import our data and make a copy
# These data includes the co-ordinates of the US state borders
from bokeh.sampledata.us_states import data
us_states = data
# These data is the winners from the 2016 election by state
election_winners = pd.read_csv('election.csv')
```

In [ ]:

```
# You don't need to write anything here
# Here we are preparing a map as a `Tile` which we will use as a background on the i
from bokeh.plotting import figure
from bokeh.tile_providers import WMTSTileSource

# Create a figure which has co-ordinates centred on the USA
x_range,y_range = ((-13884029,-7453304), (2698291,6455972))

# Create the figure
fig = figure(tools='pan, wheel_zoom', x_range=x_range, y_range=y_range)
fig.axis.visible = False
```

In [ ]:

```
# You don't need to write anything here
# In this cell we add a map tile to the figure, adding a URL in a standard format to
url = 'http://a.basemaps.cartocdn.com/dark_all/{Z}/{X}/{Y}.png'
attribution = "Map tiles by Carto, under CC BY 3.0. Data by OpenStreetMap, under ODb
fig.add_tile(WMTSTileSource(url=url, attribution=attribution))
show(fig)
```

In [ ]:

```
# You don't need to write anything here
states = []
lons = []
lats = []
# The sample data is in a slightly difficult format, so we will change it to be in a
# We don't mind about the State being repeated, as long as we have all the latitudes
#     Lat       Lon         State
# 0 -82.88318   -82.88318   FL
# 1 -82.87484   -82.87484   FL
# 2 -82.86562   -82.86562   FL

for s in us_states:
    # The amount of longitudes is the same as the latitudes, so this is safe
    # Iterate through each lat/lon pair

    for data in range(len(us_states[s]['lons'])):
        states.append(s)
        lons.append(us_states[s]['lons'][data])
        lats.append(us_states[s]['lats'][data])


# We created 3 lists of equal length, now we create a
df = pd.DataFrame({'state': states, 'lat': lats, 'lon': lons})#dict(state=states, lo
df.head()
```

Now we have the data set up into two data frames: One with the lat/lng co-ordinates of the borders of states in the USA, the other with the winner of that state in the 2016 US Presidental election.

We have seen what the geographical data look like, now display the first few rows of data about the winners using the variable `election_winners`:

In [ ]:

```
# YOUR CODE HERE
```

To work with the `Tile` we used, we need to include `x` and `y` columns in Web Mercator format. In the cell below, modify the dataset as follows:

- Call the `wgs84_to_web_mercator` function to add extra columns `x` and `y` to the `DataFrame`

In [ ]:

```
# YOUR CODE HERE
```

Using the election dataset, we now want to add an extra column to the `DataFrame` to give the colour of the state depending on the victor. We are going to set the state to blue if Clinton won it, or red if Trump won it.

To do this, we are going to use **`loc` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html)**. This specifies criteria for which rows we are to select, and then provides the name of a column to include the output:

In [ ]:

```
# You don't need to write anything here
for e in range(len(election_winners)):
    winner = election_winners['winner'][e]
    colour = ''
    if winner == 'clinton':
        colour = 'blue'
    else:
        colour = 'red'

election_winners.loc[election_winners['winner'] == 'clinton', 'colour'] = 'blue'
election_winners.loc[election_winners['winner'] == 'trump', 'colour'] = 'red'
election_winners
```

Now we can start using our data to overlay `Glyphs` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html) onto a map. We will start by creating a new map centred over the USA. Create a new map with the same attributes as the previous map you created. Call the `Figure` variable `fig`.

In [ ]:

```
# YOUR CODE HERE
```

Now try and add a `Glpyh` to the `Figure` which gives the outline of California, and colours it in blue. Use the same strategy as you used above for the line graph. The glyph in question is `Patch` (http://bokeh.pydata.org/en/latest/docs/user_guide/plotting.html#patch-glyphs), which uses the function `patch`.

In [ ]:

```
# YOUR CODE HERE
```

Now we can show the map of the USA with the California Glyph:

In [ ]:

```
show(fig)
```

In [ ]:

```
patches_x = []
patches_y = []
colors = []
list_of_states = ['AL', 'AK', 'AZ', 'AR', 'CA', 'CO', 'CT', 'DE', 'DC', 'FL', 'GA',
                  'IA', 'KS', 'KY', 'LA', 'ME', 'MD', 'MA', 'MI', 'MN', 'MS', 'MO',
                  'NJ', 'NM', 'NY', 'NC', 'ND', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC',
                  'VT', 'VA', 'WA', 'WV', 'WI', 'WY']


# Making a copy of a subset of the original data.  This way, if we make a change, it
data = df.loc[df['state'] == 'CA'].copy()
# We're using the patch glyph to colour fill the states
# The .iloc[0] signifies the first row in the dataset, because we only need one colo

ca = fig.patch(data['x'], data['y'], fill_color=colour)

show(fig)
# print(fig)
```

Putting it all Together

Now we have built individual components which we can modify to generate our interactive visualisation. To put them together, we will:

- Create and display a widget which allows text to be entered to select a state
- Create a function for them to call which will modify the selected regions on the map
- Display the map

First, we are going to try and generalise the code we saw above into one which works for all states into the function `callback`, and display a new map.

In [ ]:

```
def callback(state):
    # Filter the data here from `df` using `loc` to get the individual state
    # Then, access the data_source of `ca` to change it to the new data
    # YOUR CODE HERE


    # Get the colour of the new state, and update the fill_color of the glyph
    colour = election_winners.loc[election_winners['state'] == state]['colour'].iloc
    ca.glyph.fill_color = colour
    # Update the map
    # YOUR CODE HERE


fig = Figure(tools='pan, wheel_zoom', x_range=x_range, y_range=y_range)
fig.axis.visible = False
fig.add_tile(WMTSTileSource(url=url, attribution=attribution))
ca = fig.patch(data['x'], data['y'], fill_color=colour)
show(fig, notebook_handle=True)
```

Now we'll add a text box interactive widget, so that by entering a state we can see the map update. This still uses the `interactive` function, except the parameter to the `callback` function is a string rather than a number.

The response to the call for interactive is being assigned to a variable - in this case `i` - to later use.

In [ ]:

```
# You don't need to write anything here
i = interactive(callback, state='')
i
```

Layout

Finally, the widgets need to be set out. Both Jupyter and Bokeh have their own widgets for layout, and they are not yet compatible. To lay them out, we suggest that you keep the plot in one cell, and the widgets in the cell above or below.

The functions `VBox` and `HBox` allow widgets to be laid out in a way which they align either vertically (for `VBox`) or horizontally (for `HBox`). The function takes a list of `ipywidgets` widgets:

In [ ]:

```
HBox(
    # We can put HBox and VBox inside each other as well
    [ HBox([i, i]), VBox([i,i,i]) ]
)
```

Have a look at the list of widgets on the ipywidgets documentation (http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html) and experiment with displaying those in the cell below:

In [ ]:

```
# YOUR CODE HERE
```

In [ ]:

```

```