Dashboard

December 7, 2021

DIY Covid-19 Dashboard Kit (C) Yusra Saib 180295106

1 DIY Covid-19 Dashboard

This is a template for your DIY Covid Dashboard, to which you can add the code you developed in the previous notebooks. The dashboard will be displayed using voila, a Python dashboarding tool that converts notebooks to standalone dashboards. Contrary to the other libraries we have seen, the `voila` package must be installed using *pip* or *conda* but it does not need to be imported - it rather acts at the level of the notebook server. Package `voila` is already installed on the EECS JupyterHub as well as in the binder - to install it locally, follow the instructions online.

Broadly speaking, Voila acts by **running all the cells in your notebook** when the dashboard is first loaded; it then hides all code cells and displays all markdown cells and any outputs, including widgets. However, the code is still there in the background and handles any interaction with the widgets. To view this dashboard template rendered in Voila click here.

[ ]:
```
!pip install pandas numpy matplotlib plotly ipywidgets
```

[ ]:
```
import json
import pandas as pd
import numpy as np
import plotly.express as px
import ipywidgets as widgets
from pathlib import Path
from urllib.request import urlretrieve
```

1.1 Load initial data from disk

You should include "canned" data in `.json` files along with your dashboard. When the dashboard starts, it should load that data (the code below will be hidden when the dashboard is rendered by Voila).

[ ]:
```
json_filename = 'covid_data.json'
```

1.2 Wrangle the data

The dashboard should contain the logic to wrangle the raw data into a `DataFrame` (or more than one, as required) that will be used for plotting. The wrangling code should be put into a function

and called on the data from the JSON file (we'll need to call it again on any data downloaded from the API). In this template, we just pretend we are wrangling `rawdata` and generate a dataframe with some random data

[ ]:
```
def refresh_data():
    # See from https://coronavirus.data.gov.uk/details/download
    metrics = ['hospitalCases']
    areatype = 'nation'

    if (Path(json_filename).exists()):
        Path(json_filename).unlink()
    metrics_request_query = map(lambda x: f'metric={x}', metrics)
    url = f'https://api.coronavirus.data.gov.uk/v2/data?
 ↪areaType={areatype}&{"&".join(metrics_request_query)}&format=json'
    urlretrieve(url, filename=json_filename) # Downloads json file

df = pd.DataFrame()

def store_data():
    global df
    # Reading Json File
    with open(json_filename) as json_file:
        df = pd.DataFrame(json.load(json_file)['body'])

    df.drop(columns=['areaType'], inplace=True)

def draw_scatter(x, y):
    display(px.scatter(df, x=x, y=y, title=f'{x} Vs {y}'))

def draw_line(x, y):
    display(px.line(df, x=x, y=y, title=f'{x} Vs {y}'))

def draw_box(y):
    display(px.box(df, y=y))
```

[ ]:
```
refresh_data()
store_data()

x = 'date'
y = 'hospitalCases'

draw_scatter(x, y)
draw_line(x, y)
draw_box(y)
```

[ ]:
```
print ("The graphs shows the increase and decrease of hospital cases in the␣
 ↪nation. Run your mouse on the graph to see interactivity. :)")
```

1.3 Download current data

Give your users an option to refresh the dataset - a "refresh" button will do. The button callback should * call the code that accesses the API and download some fresh raw data; * wrangle that data into a dataframe and update the corresponding (global) variable for plotting; * optionally: force a redraw of the graph and give the user some fredback.

Once you get it to work, you may want to wrap your API call inside an exception handler, so that the user is informed, the "canned" data are not overwritten and nothing crashes if for any reason the server cannot be reached or data are not available.

After you refresh the data, graphs will not update until the user interacts with a widget. You can trick `iPywidgets` into redrawing the graph by simulating interaction, as in the `refresh_graph` function we define in the Graph and Analysis section below.

Clicking on the button below just generates some more random data and refreshes the graph. The button should read *Fetch Data*. If you see anything else, take a deep breath :)

[ ]:
```
# Printout from this function will be lost in Voila unless captured in an
# output widget - therefore, we give feedback to the user by changing the
# appearance of the button
def api_button_callback(button):
    """ Button callback - it must take the button as its parameter (unused in␣
 ↪this case).
    Accesses API, wrangles data, updates global variable df used for plotting.␣
 ↪"""
    # Get fresh data from the API. If you have time, include some error handling
    # around this call.
    apidata=access_api()
    # wrangle the data and overwrite the dataframe for plotting
    global df
    df=wrangle_data(apidata)
    # the graph won't refresh until the user interacts with the widget.
    # this function simulates the interaction, see Graph and Analysis below.
    # you can omit this step in the first instance
    refresh_graph()
    # after all is done, you can switch the icon on the button to a "check" sign
    # and optionally disable the button - it won't be needed again. You can use␣
 ↪icons
    # "unlink" or "times" and change the button text to "Unavailable" in case␣
 ↪the
    # api call fails.
    apibutton.icon="check"
    # apibutton.disabled=True


apibutton=wdg.Button(
    description='refresh', # you may want to change this...
    disabled=False,
```

```
    button_style='danger', # 'success', 'info', 'warning', 'danger' or ''
    tooltip="Keep calm and carry on",
    # FontAwesome names without the `fa-` prefix - try "download"
    icon='exclamation-triangle'
)

# remember to register your button callback function with the button
apibutton.on_click(api_button_callback) # the name of your function inside␣
 ↪these brackets

display(apibutton)

# run all cells before clicking on this button
```

1.4 Graphs and Analysis

1.5 Deploying the dashboard

Once your code is ready and you are satisfied with the appearance of the graphs, replace all the text boxes above with the explanations you would like a dashboard user to see. The next step is deploying the dashboard online - there are several options for this, we suggest deploying as a Binder. This is basically the same technique that has been used to package this tutorial and to deploy this template dashboard. The instructions may seem a bit involved, but the actual steps are surprisingly easy - we will be going through them together during a live session. You will need an account on GitHub for this - if you don't have one already, now it's the time to create it.

Author and Copyright Notice Remember if you deploy this dashboard as a Binder it will be publicly accessible. Take credit for your work! Also acknowledge the data source: *Based on UK Government data published by Public Health England.*