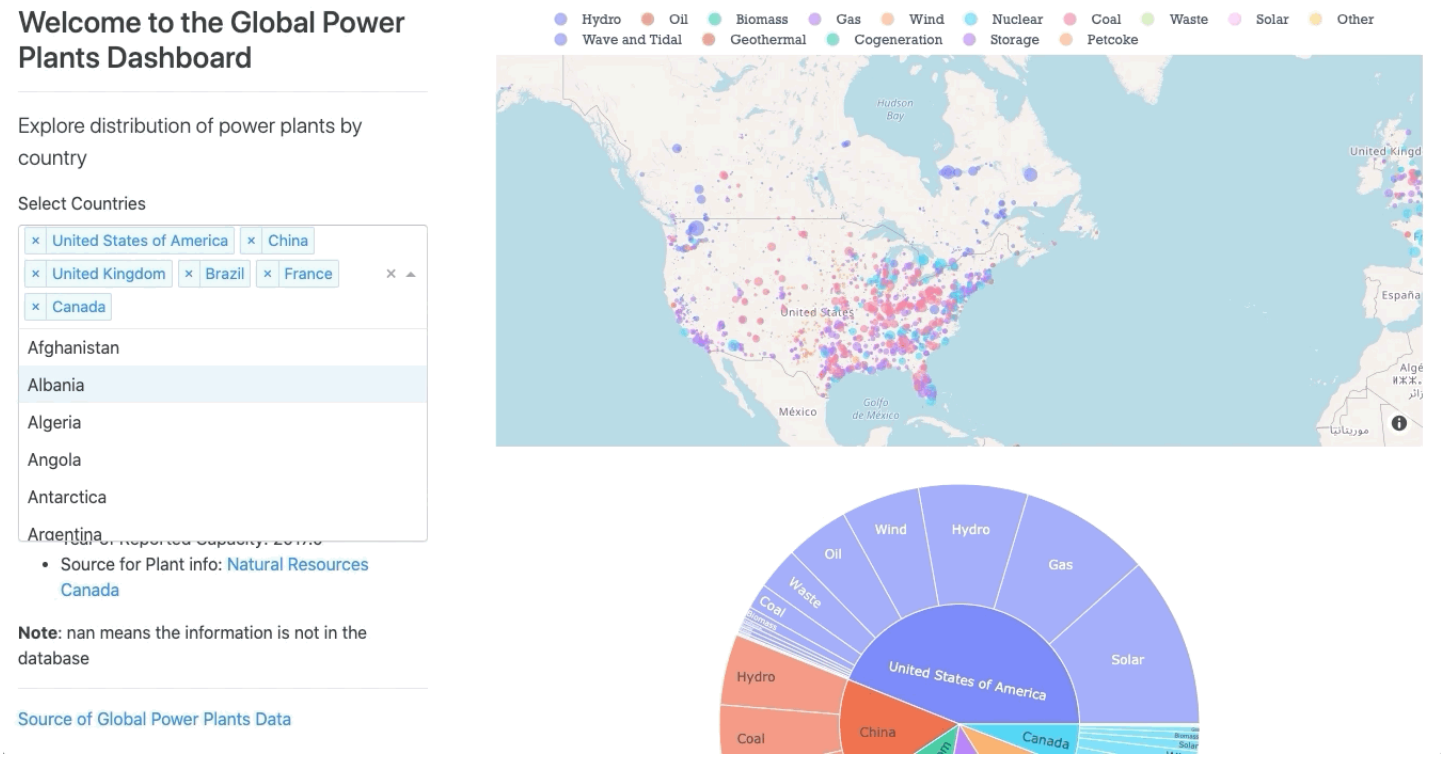
Analytics Vidhya · Follow publication



Global Power Plants dashboard created using Dash

End-to-End Guide: Creating a Web Application using Dash

8 min read · Oct 17, 2020

Hanan Ather Follow

Listen Share ••• More

By the end of this guide you will know how to **create** and **deploy** your own dashboard on the web.

Why Dash?

Why use Dash to build an application over other great tools? I became interested in

:

h other big data frameworks such as Spark for building a datasets

- it can create highly customizable interactive Dashboards for unstructured datasets using R or Python
- it requires no knowledge of HTML or Javascript

Why not just use BI tools?

I want to preface by saying Tableau and Power BI are great tools. If your goal is to make a great-looking interactive dashboard using a structured dataset, they are a great option.

However, Tableau and other BI tools don't offer the same level of flexibility as Python and R when it comes to working with unstructured data. It can be more efficient to use Python or R if the data requires a lot of pre-processing and transformation before analytics.

What I find most appealing about Dash is its integrability with big data and parallel computing frameworks. Dash apps can serve as the front-end to Spark clusters. If your file is too large to fit on your memory, and you don't want to work with cloud computing frameworks, Dash is also compatible with Vaex. Vaex is a python library for lazy "Out-of-Core" DataFrames (similar to Pandas) which can be as large as the size of your hard drive! The founders of Vaex wrote an excellent blog post, which I highly recommend if you're interested in working with large datasets.

The Global Power Plant Dataset

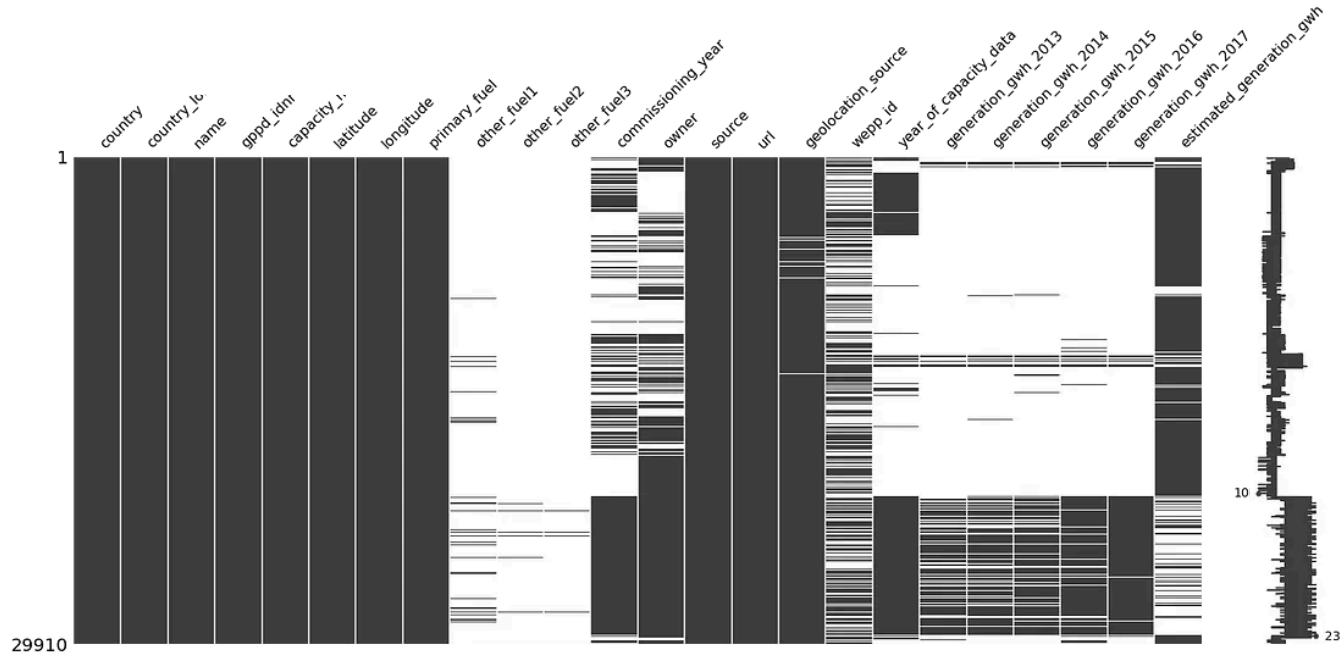
The Global Power Plant data displayed on the dashboard comes from the World Resources Institute. The Global Power Plant dataset is comprehensive and contains information about approximately 30,000 power plants across 164 different countries. The dataset contains detailed features about power plants such as geolocation, plant owner, primary fuel and secondary fuel types, etc.

Getting to know the data

The first step towards building the dashboard was gaining familiarity with the dataset and figuring out what relevant information I wanted to convey. After

narrowing down a list of features, the quality of the data was accessed.

I used the missingno library to visualize the distribution of missing values in the



This is a crucial step because it helped refine the list of import features. I decided it was not worth including values from fields such as `other_fuel2` & `other_fuel3` since most of the values are missing.

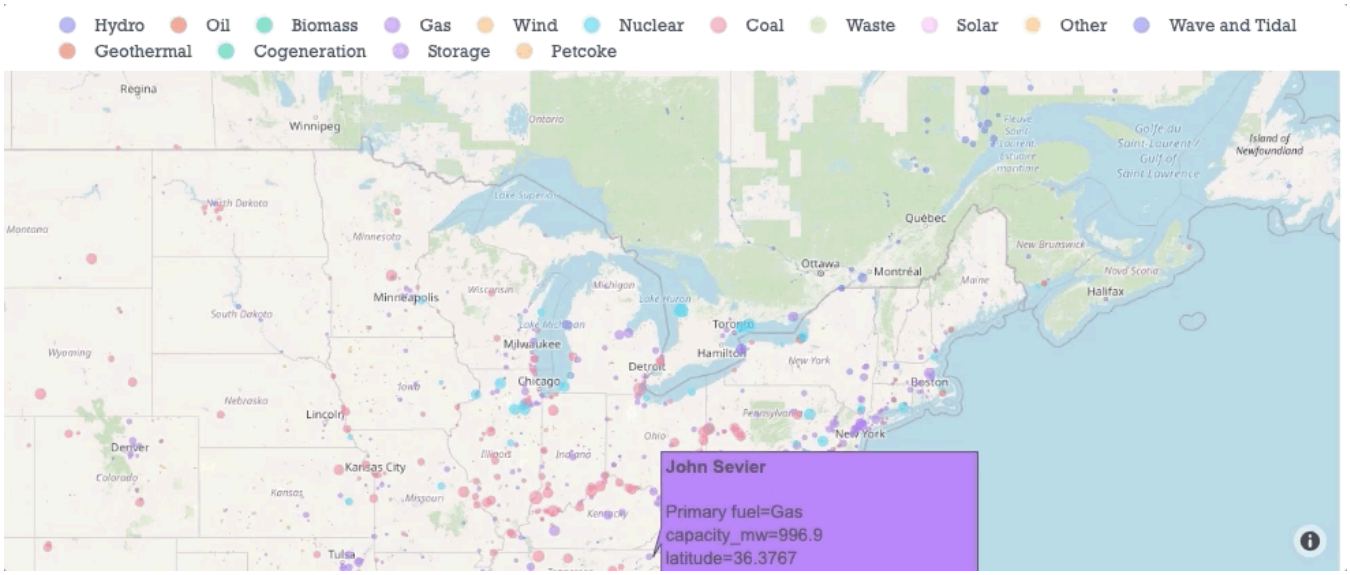
Designing the Dashboard

I decided that an interactive map would be the best way to capture the dataset. I started by plotting all power plant's latitude and longitude with Plotly's `px.scattermap`.

```
# filtering data
dff = df[df.country_long.isin(countries)]

pbox(dff, lat="latitude", lon="longitude",
 hover_name="name", hover_data=["country_long",
 "primary_fuel"],
 color="primary_fuel",zoom=2,
 center = {'lat': 45.4215, 'lon' :-75.6972},opacity = 0.6,
 mapbox_style='open-street-map',
 labels = {'primary_fuel': 'Primary fuel'},
 size = 'capacity_mw')
# customize font and legend orientation & position
fig.update_layout(
    font_family="Rockwell",
    autosize=True,
    margin=go.layout.Margin(l=0, r=0, t=0, b=35),
    legend=dict(title= None, orientation="h", y=1, yanchor="bottom", x=0.5,
                xanchor="center")
        )
```


scatter-map.py hosted with ❤ by GitHub view raw



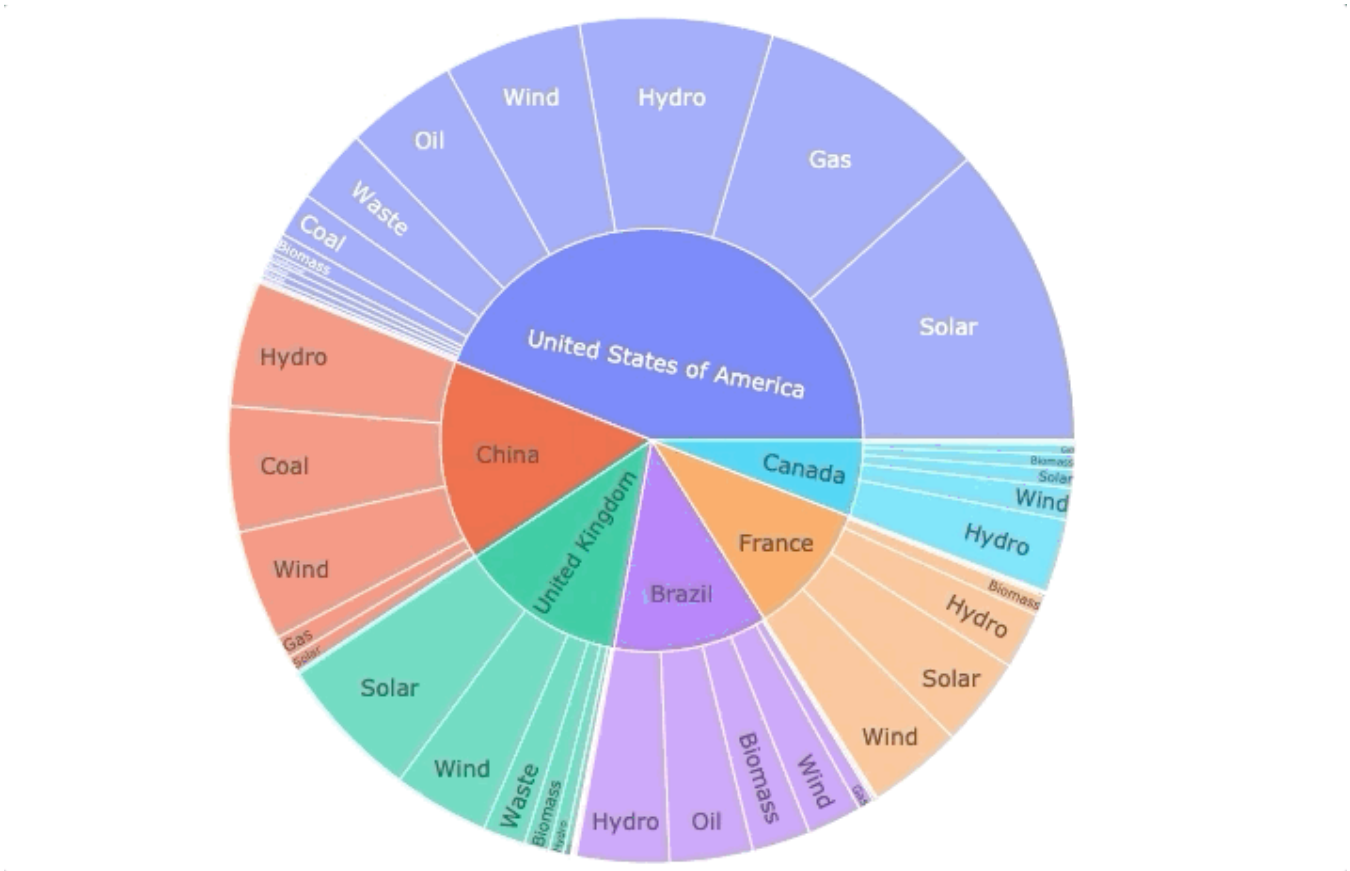
It also has additional functionality that allows the user to filter power plants by fuel types by clicking on the legend.

One of the principal quantities I was interested in comparing was the distribution of different categories of power plants worldwide. I used the Sunburst plots because they can be an effective way to visualize hierarchical data distributions.

```
Grouped = dff.groupby(['country_long','primary_fuel'])['primary_fuel'].count().rename_ax
                            .reset_index(name='Count')

rst(Grouped,path=['Country', 'Fuel'], values='Count')
out(margin=go.layout.Margin(l=0, r=0, t=0, b=0))
```

by GitHub view raw



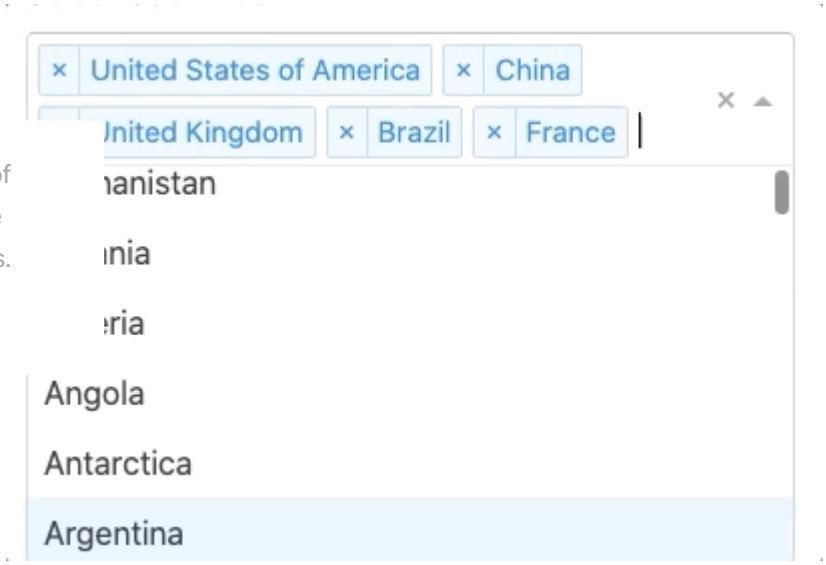
This plot displays the distribution of power plants top five countries with the most power plants.

I used Dash's Multi-Value Dropdown to filter the countries displayed on the sunburst plot and the map.

```
df = pd.read_csv("global_power_plant_database.csv")

countries = df['country_long'].unique()
dcc.Dropdown(id='country',
                        placeholder='Select a Country',
                        options=[{'label': i, 'value': i} for i in countries],
                        value=['United States of America', 'China',
                                'United Kingdom', 'Brazil', 'France','Canada'],
                        multi=True
                    )
```

multi-dropdown.py hosted with ❤ by GitHub view raw



After importing the dataset as a pandas DataFrame, the names of all the countries in the dataset are extracted into a pandas series. **The multi-value dropdown takes the Pandas series containing the countries as an iterable to create a dictionary that the dash-component requires.**

Connecting the components

Up until now, we have successfully created individual components, but we still haven't connected them.

To connect the different components, we have to use Dash Callbacks functions.

What are Dash callback functions?

> Dash callback functions are python functions that are automatically called by Dash whenever an input component's property changes.

Whenever the user selects a different set of countries in the multi-value dropdown, a python function, referred to as a **Dash Callback**, will be called. This function's input parameter will be the new countries that have been selected by the user, and it will return an updated map and a sunburst plot.

The code looks something approximate to this:

```
@app.callback([
    Output('map', 'figure'),
    Output('graph', 'figure')],
                ', 'value')])
                tries):
                a
                ntry_long.isin(countries)]

    # code for map
    fig = px.scatter_mapbox(dff, lat="latitude", lon="longitude",
                        hover_name="name", hover_data=["country_long",
                        "primary_fuel"],
                        color="primary_fuel",zoom=2,
                        center = {'lat': 45.4215, 'lon' :-75.6972},opacity = 0.6,
                        mapbox_style='open-street-map',
                        labels = {'primary_fuel': 'Primary fuel'},
                        size = 'capacity_mw')
    # customize font and legend orientation & position
    fig.update_layout(
        font_family="Rockwell",
        autosize=True,
        margin=go.layout.Margin(l=0, r=0, t=0, b=35),
        legend=dict(title= None, orientation="h", y=1, yanchor="bottom", x=0.5,
                    xanchor="center")
        )
    # code for sunburst chart
    Grouped = dff.groupby(['country_long','primary_fuel'])['primary_fuel'] \
                            .count().rename_axis(['Country','Fuel']) \
                            .reset_index(name='Count')

    graph = px.sunburst(Grouped,path=['Country', 'Fuel'], values='Count')
    graph.update_layout(margin=go.layout.Margin(l=0, r=0, t=0, b=0))

    return fig, graph
```


callback.py hosted with ❤ by GitHub view raw

NOTE: If you try to run the code, it won't compile. This is just an intermediate step. You can scroll below to find a version that will compile and run for you.

The "Inputs" (a set of countries) and the "Outputs" (updated map and sunburst plot) are described **declaratively** as the arguments of the `@app.callback` decorator.

If you don't know what decorator is, essentially, declaring the `@app.callback` decorator tells Dash to call the function below it whenever the "Input" value changes.

Therefore, whenever the selected countries are changed by the user, the `def update_figures` function is called. The input parameter of this function takes the

ı the `@app.callback` decorator. For instance, if we had two ɔr, the `def update_figures` function would have to have two ator has two “Outputs”: the map and the graph. Hence, `def` two objects: scatter_map and graph.

Similarly, I used another `@app.callback` decorator for updating the summary when the user clicks on a particular power plant.

```
@app.callback(dash.dependencies.Output('plant_summary', 'children'),
              [dash.dependencies.Input('map', 'clickData')])
def update_summary(click_Data):
    plant_name  = click_Data['points'][0]['hovertext']
    owner = df[df['name'] == plant_name]['owner'].iloc[0]
    gwh = round(df[df['name'] == plant_name]['estimated_generation_gwh'].iloc[0],2)
    commissioning_year = df[df['name'] == plant_name]['commissioning_year'].iloc[0]
    capacity_mw = df[df['name'] == plant_name]['capacity_mw'].iloc[0]
    estimated_generation_gwh = df[df['name'] == plant_name]['estimated_generation_gwh']
    year_capacity = df[df['name'] == plant_name]['year_of_capacity_data'].iloc[0]
    source = df[df['name'] == plant_name]['source'].iloc[0]
    url = df[df['name'] == plant_name]['url'].iloc[0]
    update = f'''
                    **Summary of *{plant_name}*:**
                    - Plant owner: {owner}
                    - Estimated Yearly Generation (GWH):
                      {gwh}
                    - First Year the Plant Generated Energy:
                      {commissioning_year}
                    - installed electrical capacity (MW):
                      {capacity_mw}
                    - Year of Reported Capacity: {year_capacity}
                    - Source for Plant info: [{source}]({url})
                    - Annual generation in gigawatt hours (GWhs):
                      {estimated_generation_gwh}

                    '''


    return update
```

Connecting-components-2.py hosted with ❤ by GitHub view raw

Note that in the decorator, the “Input” is now the `dcc.Graph` component (the scatter map), and the "Output" is a Dash markdown-component.

The function `update_summary` takes a parameter called `click_Data`. This comes from the `clickData` attribute of `dcc.Graph` components. `clickData` is one of four user-

of `dcc.Graph` components. The other attributes are `hoverData`, `ayData`. More information about them can be found here.

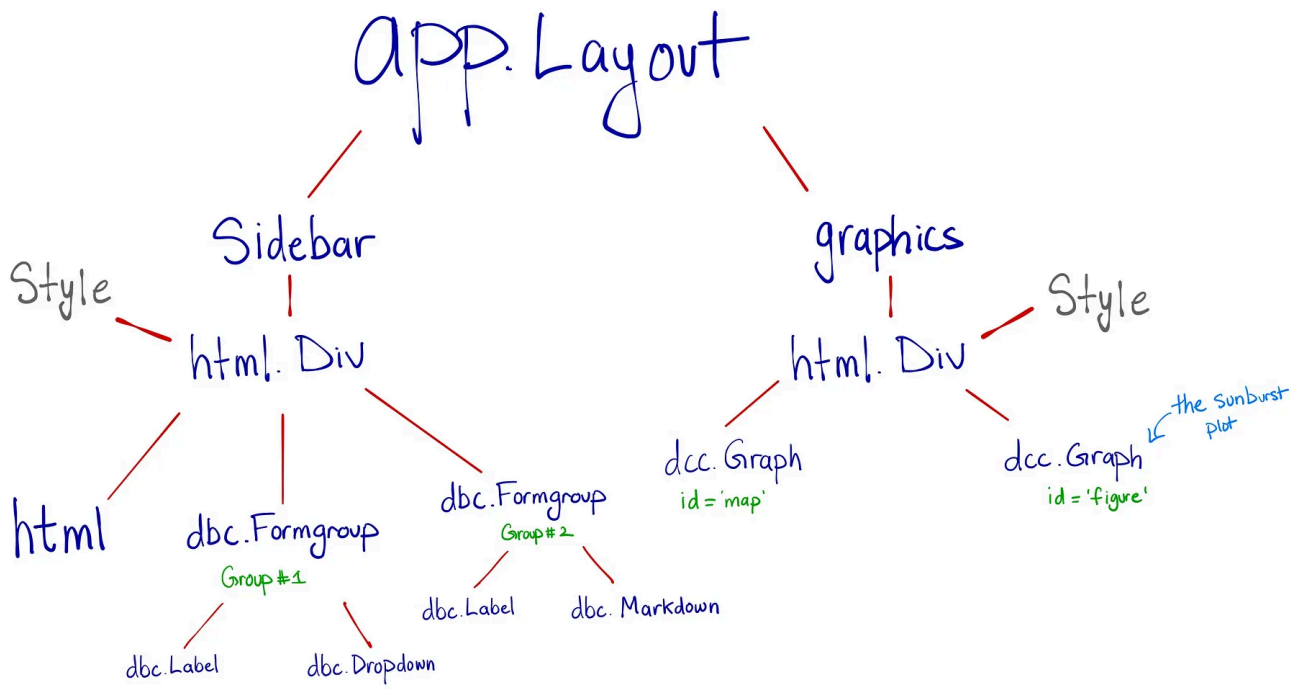
be the best choice for the Global Power Plants dashboard. I em out to see which one works best for your Dashboard.

Structure of App Layout

There are a few different ways to configure the layout of a Dash application. I used the dash-bootstrap-components library for styling my app. If you're just getting started, I highly recommend this video by Charming Data.

Using Bootstrap I separated the app into two main components:

- the Sidebar
- graphics



- The `dash_html_components` (HTML)library contains components for all HTML tags.
- The `dash_core_componets` (dcc) describes higher-level interactive components such as the map and the sunburst plot

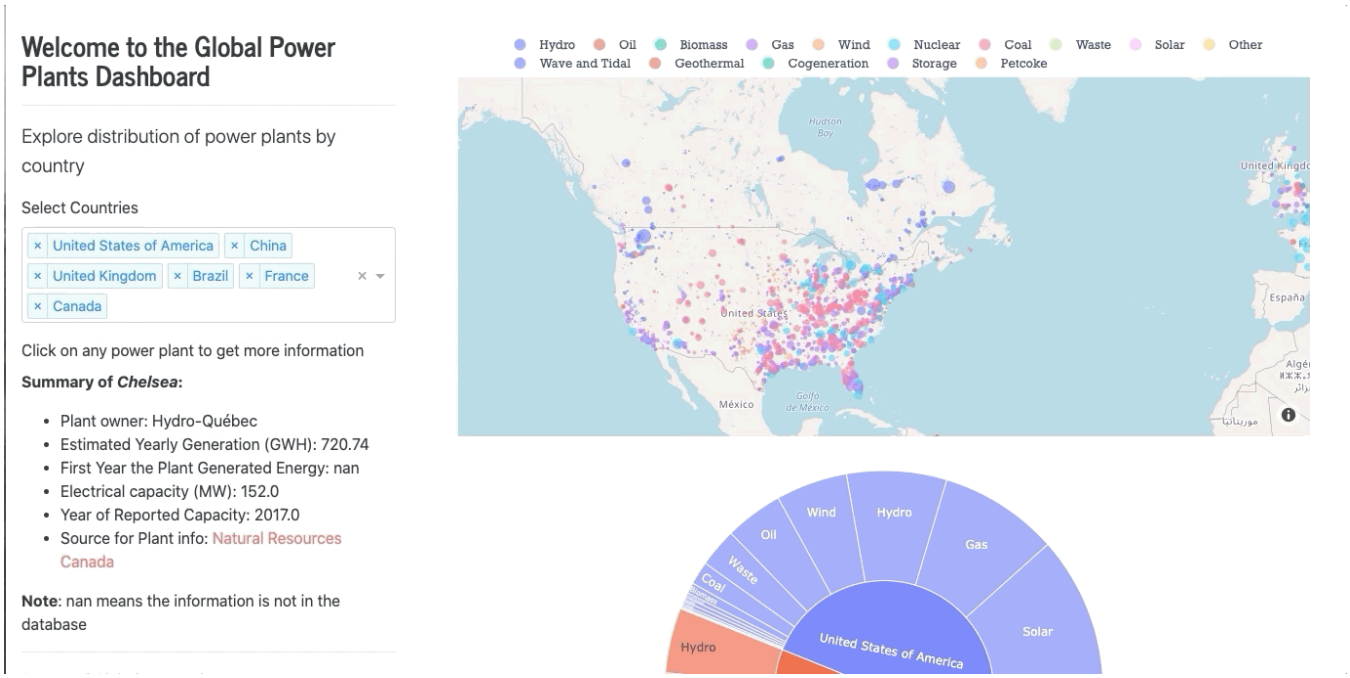
- the `dash_bootstrap_components` (dbc) library contains Bootstrap components for Dash

view, the layout of Dash apps is structured like a tree. This ucture provides a lot of flexibility in terms of adding or . It isn't necessary to use Bootstrap for styling your application, however, it does provide a lot more flexibility. There are many free Bootstrap stylesheets available and the theme of the application can be changed with a single line:

```
df = pd.read_csv("data/globalpowerplantdata/global_power_plant_database.csv")

countries = df['country_long'].unique()

app = dash.Dash(external_stylesheets=[dbc.themes.BOOTSTRAP])
```



The above example demonstrates the ease by which the theme of an app can be modified.

The `dash_bootstrap_components.themes` module contains the Content Delivery Network (CDN) links for Bootstrap and Bootswatch themes. It is possible to modify or compile your own theme. It can be served locally by replacing `dbc.themes.BOOTSTRAP` with the URL of the stylesheet. For more information, you can read the documentation.

After tweaking and styling everything, our final python script for creating the dashboard looks like this:

```
@author: hananather
"""
import dash
                    omponents as dcc
                    omponents as html
                    cies import Input, Output
                    d
                    ess as px
import dash_bootstrap_components as dbc
from plotly import graph_objs as go


app = dash.Dash(__name__,external_stylesheets=[dbc.themes.BOOTSTRAP])
server = app.server

df = pd.read_csv("data/global_power_plant_database.csv")
countries = df['country_long'].unique()

# the style arguments for the sidebar. We use position:fixed and a fixed width
SIDEBAR_STYLE = {
    "position": "fixed",
    "top": 0,
    "left": 0,
    "bottom": 0,
    "width": "26rem",
    "padding": "2rem 1rem",
    "background-color": "#f8f9fa",
}

# the styles for the main content position it to the right of the sidebar and
# add some padding.
CONTENT_STYLE = {
    "margin-left": "28rem",
    "margin-right": "2rem",
    "padding": "2rem 1rem",
}

sidebar = html.Div(
    [
        html.H3("Welcome to the Global Power Plants Dashboard"),
        html.Hr(),
        html.P(
            "Explore distribution of power plants by country", className="lead"
        ),

          dbc.FormGroup( # GROUP 1
            [
                dbc.Label("Select Countries")
```

```
                dbc.Label("Select Countries"),
                dcc.Dropdown(id='country',
                             placeholder='Select a Country',
                             options=[{'label': i, 'value': i} for i in countries],
                             value=['United States of America', 'China',
                                    'United Kingdom', 'Brazil', 'France','Canada'],
                             multi=True
                            ),
```



```
            ),
        dbc.FormGroup( # GROUP 2
            [
                dbc.Label("Click on any power plant to get more information"),
                dcc.Markdown(id='plant_summary')
            ]
        ),

        html.Hr(),
            html.A("Source of Global Power Plants Data", href='https://datasets.wri.or

    ],
    style=SIDEBAR_STYLE,
)

content =html.Div(children= [dcc.Graph(id='map',clickData={'points': [{'hovertext': 'C
                  dcc.Graph(id='graph')],style=CONTENT_STYLE)



app.layout = html.Div([sidebar, content])

@app.callback([
    Output('map', 'figure'),
    Output('graph', 'figure')],
    [Input('country', 'value')])
def update_map(countries):
    dff = df[df.country_long.isin(countries)]

    fig = px.scatter_mapbox(dff, lat="latitude", lon="longitude",
                        hover_name="name", hover_data=["country_long", "primary_fuel"]
                        color="primary_fuel",zoom=2,
                        center = {'lat': 45.4215, 'lon' :-75.6972},opacity = 0.6,
                        mapbox_style='open-street-map',
                        labels = {'primary_fuel': 'Primary fuel'},
                        size = 'capacity_mw')
    fig.update_layout( # customize font and legend orientation & position

```

```
        # font and legend
        font_family="Rockwell",
        autosize=True,
                        ut.Margin(l=0, r=0, t=0, b=35),
                        t(
                        tle= None, orientation="h", y=1, yanchor="bottom", x=0.5, xanchor="c
```



```
        Grouped = dff.groupby(['country_long','primary_fuel'])['primary_fuel'].count().ren
        graph = px.sunburst(Grouped,path=['Country', 'Fuel'], values='Count')

        graph.update_layout(margin=go.layout.Margin(l=0, r=0, t=0, b=0)
                )

        return fig, graph

@app.callback(dash.dependencies.Output('plant_summary', 'children'),
```



```
        gwh = round(df[df['name'] == plant_name]['estimated_generation_gwh'].iloc[0],2)
        commissioning_year = df[df['name'] == plant_name]['commissioning_year'].iloc[0]
        capacity_mw = df[df['name'] == plant_name]['capacity_mw'].iloc[0]
        year_capacity = df[df['name'] == plant_name]['year_of_capacity_data'].iloc[0]
        source = df[df['name'] == plant_name]['source'].iloc[0]
        url = df[df['name'] == plant_name]['url'].iloc[0]


        update = f'''
                        **Summary of *{plant_name}*:**
                        - Plant owner: {owner}
                        - Estimated Yearly Generation (GWH): {gwh}
                        - First Year the Plant Generated Energy: {commissioning_year}
                        - Electrical capacity (MW): {capacity_mw}
                        - Year of Reported Capacity: {year_capacity}
                        - Source for Plant info: [{source}]({url})



                        **Note**: nan means the information is not in the database
                        '''


        return update

if __name__ == '__main__':
```

```
144          app.run_server(debug=True)
```

Deploying the application on the Web

hare the dashboard, we can start the deployment process.

o deploy your application on the Web. The method shown

thod that I've come across for quickly deploying your web

nvolve creating a virtual environment or using the command

Step 1: Initialize and create a GitHub repository. To deploy your application on the Web, we need the following five files in our repository:

- `app.py`
- CSV file containing your dataset
- `README.md`
- `requirement.txt`
- `Procfile`

The first three files are self-explanatory. The `app.py` is what we have been working on thus far, and the `README.md` is a markdown file that contains information regarding the project.

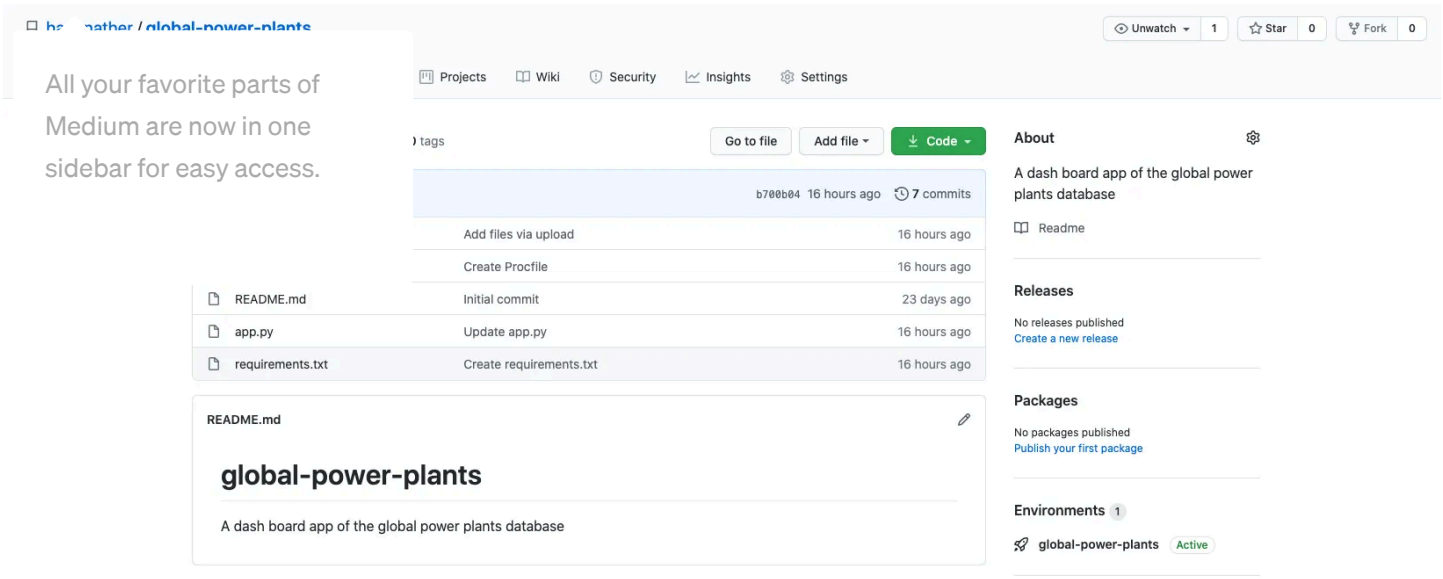
The `requirement.txt` looks like this:

This file contains all the packages and their respective versions that our application uses.

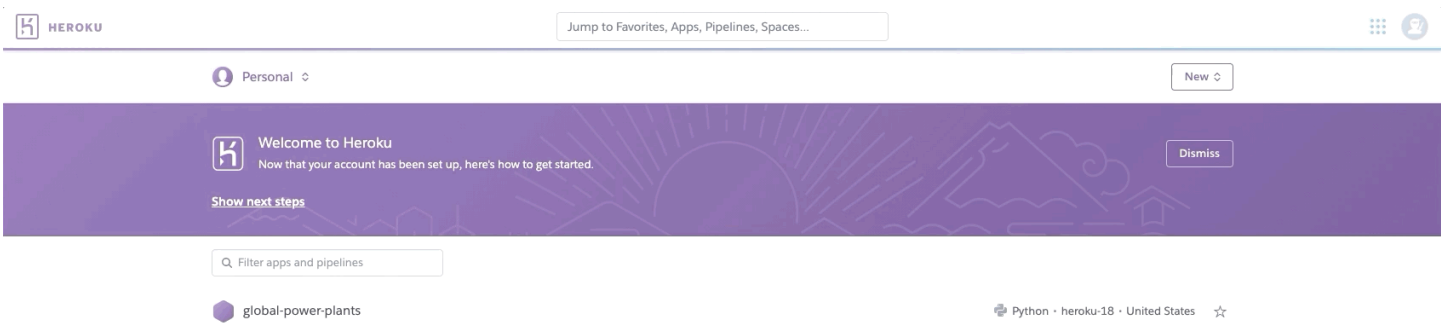
It is important to add the correct version of the packages you are using in your environment. If you don't know the versions of your packages don't worry! The `print(name_of_package.__version__)` command will print the version of the package, and you can directly copy-paste them in the `requirement.txt` file using the format I showed above. **The `gunicorn` is an essential package for deployment!** It is a Python WSGI HTTP Server for UNIX. If you don't have this package simply use the `$ pip install gunicorn` command in the terminal.

The `Procfile` is an essential file for deploying our application, and it looks like this :

You can copy-paste the line above into your repository. When finished, our repository for the application looks like this.

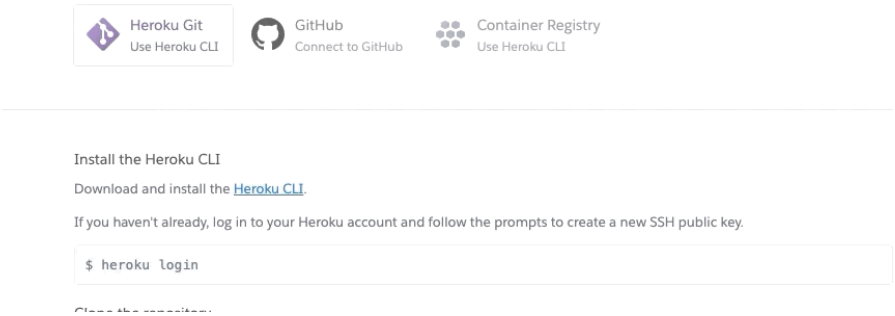


Step 2: Create a new App on Heroku. After registering for Heroku, we create a new app and choose a name for our application.



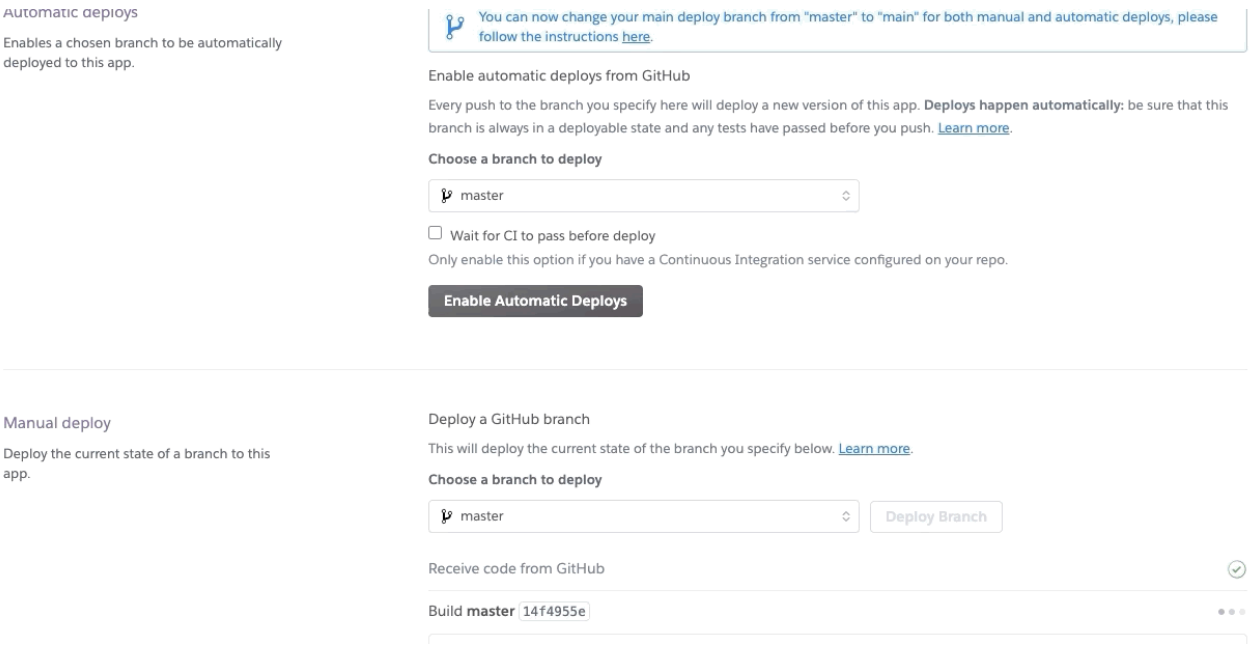
Step 3: Connecting our GitHub repository to Heroku app. Once we have created our Heroku app, we can see several deployment methods. In my opinion, the simplest option is connecting with GitHub since it requires no knowledge of Command Line Interface (CLI) or Virtual environments. After selecting GitHub as our deployment method, we enter the name of the GitHub repository we created in **Step 1.**

All your favorite parts of Medium are now in one sidebar for easy access.



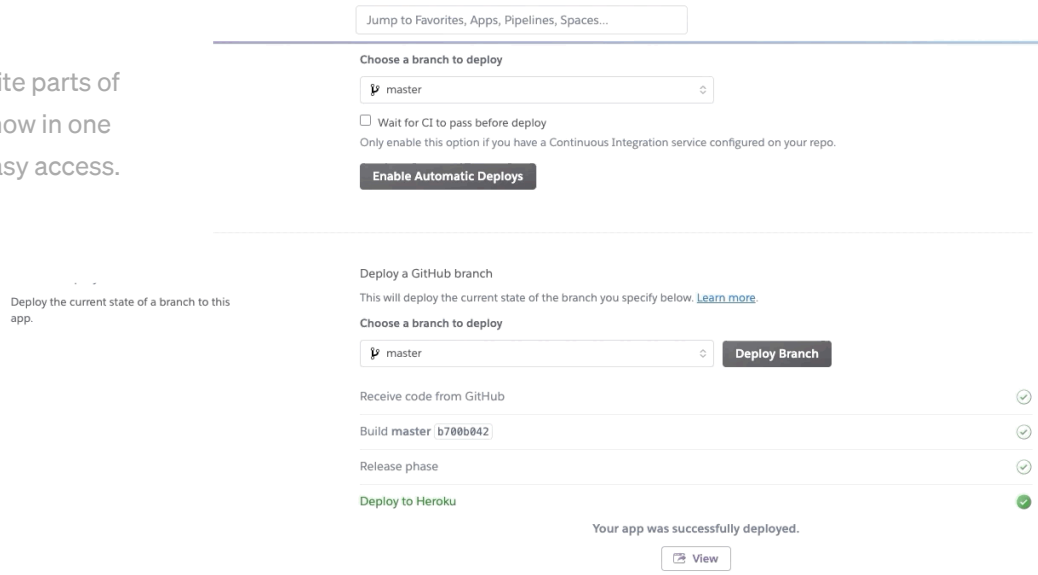
This is how it looks when the Heroku app successfully connects to our GitHub repository.

Step 4: Manual Deployment. Finally, we deploy our app by the "Deploy Branch" command. This step takes a few minutes.



It takes a few minutes to install all the required packages and deploy the GitHub repository to a website.

Step 5: View App! Once the deployment step is complete we can view the webpage of our application in the browser! The Global Power Plants Dashboard can be viewed here!



Data Science | Data Visualization | Dashboard | Dash | Plotly

Follow

Published in Analytics Vidhya

75K followers · Last published 2 days ago

Analytics Vidhya is a community of Generative AI and Data Science professionals. We are building the next-gen data science ecosystem https://www.analyticsvidhya.com

Follow

Written by Hanan Ather

13 followers · 15 following

http://hananather.github.io/

Responses (1)

All your favorite parts of Medium are now in one sidebar for easy access.

Matt Emrac
Apr 24

Fantastic work 👍 However do you mind sharing a link to your source code in case someone wanted to try it out please?

Reply

More from Hanan Ather and Analytics Vidhya

In Analytics Vidhya by Harikrishnan N B

Confusion Matrix, Accuracy, Precision, Recall, F1 Score

Binary Classification Metric

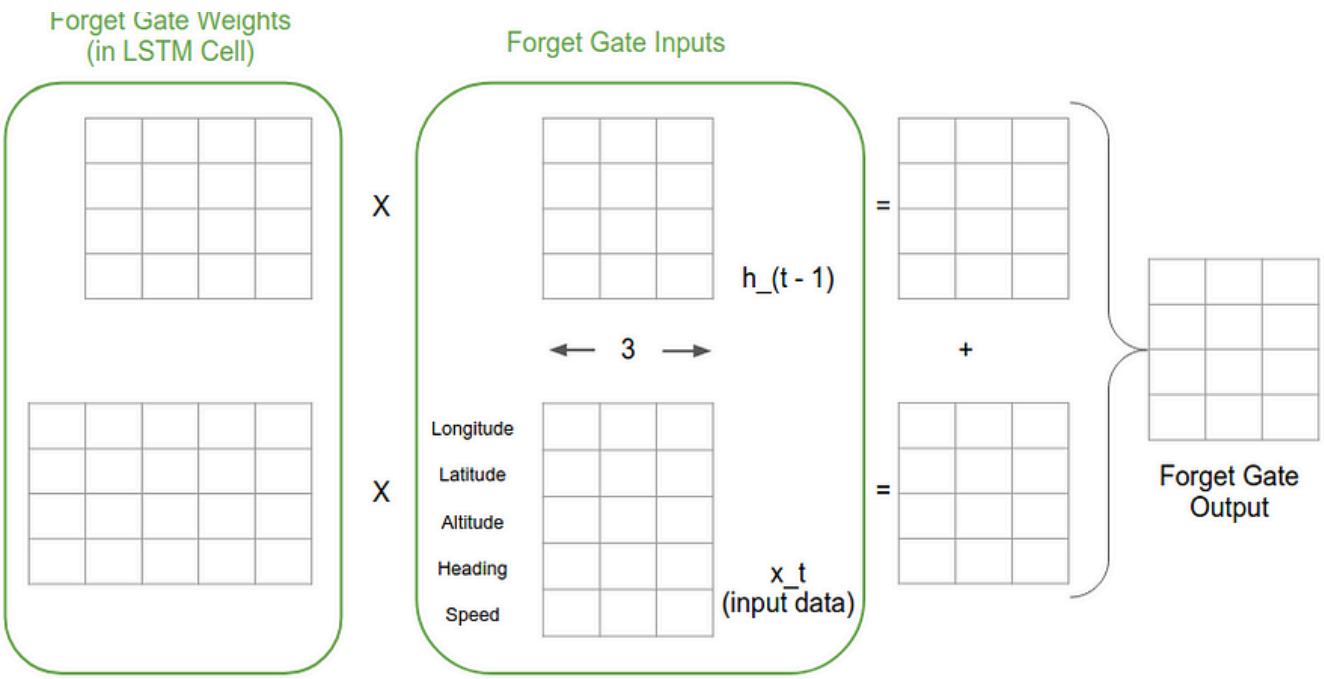
All your favorite parts of Medium are now in one sidebar for easy access.

In Analytics Vidhya by Leland Roberts

Understanding the Mel Spectrogram

(and Other Topics in Signal Processing)

Mar 6, 2020 2.3K 27



In Analytics Vidhya by Ryan T. J. J.

LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras

All your favorite parts of Medium are now in one sidebar for easy access.

er guide on LSTMs / RNNs / Keras / whatever. There are SO many them full of false...

In Analytics Vidhya by Kia Eisinga

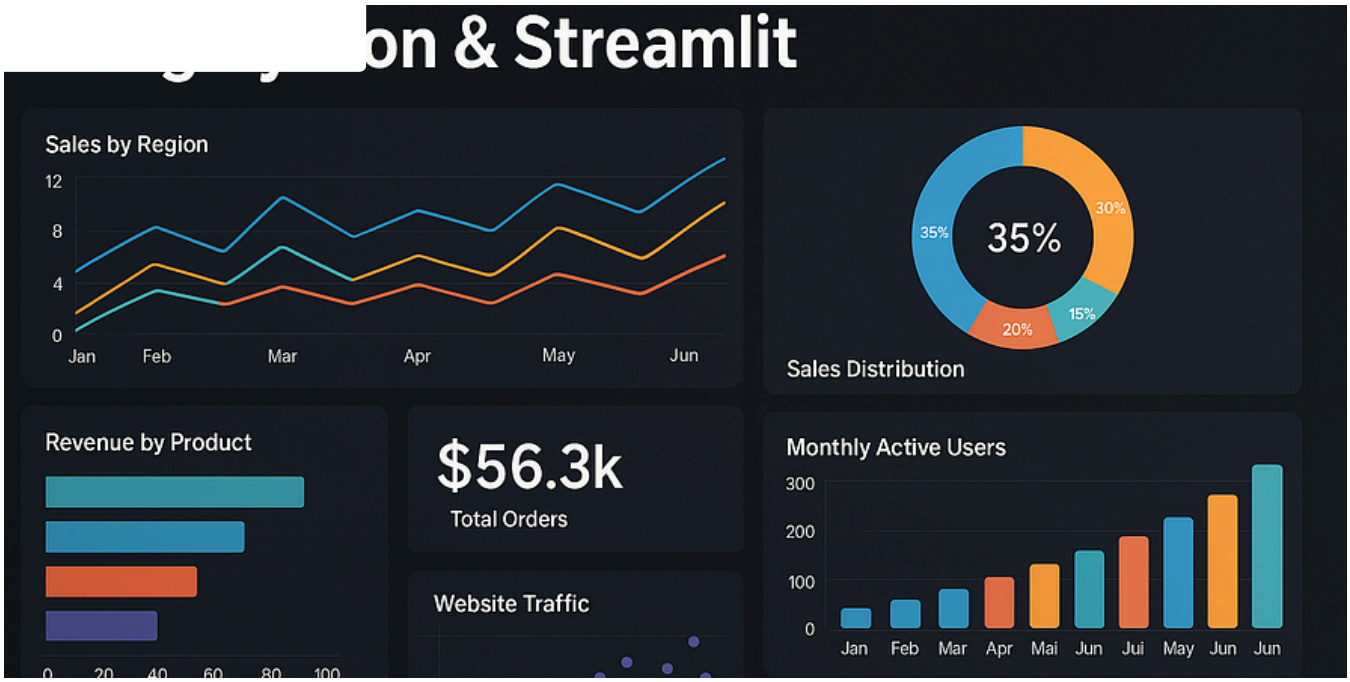
How to create a Python library

Ever wanted to create a Python library, albeit for your team at work or for some open source project online? In this blog you will learn...

Jan 26, 2020 2.9K 33

See all from Hanan Ather

See all from Analytics Vidhya

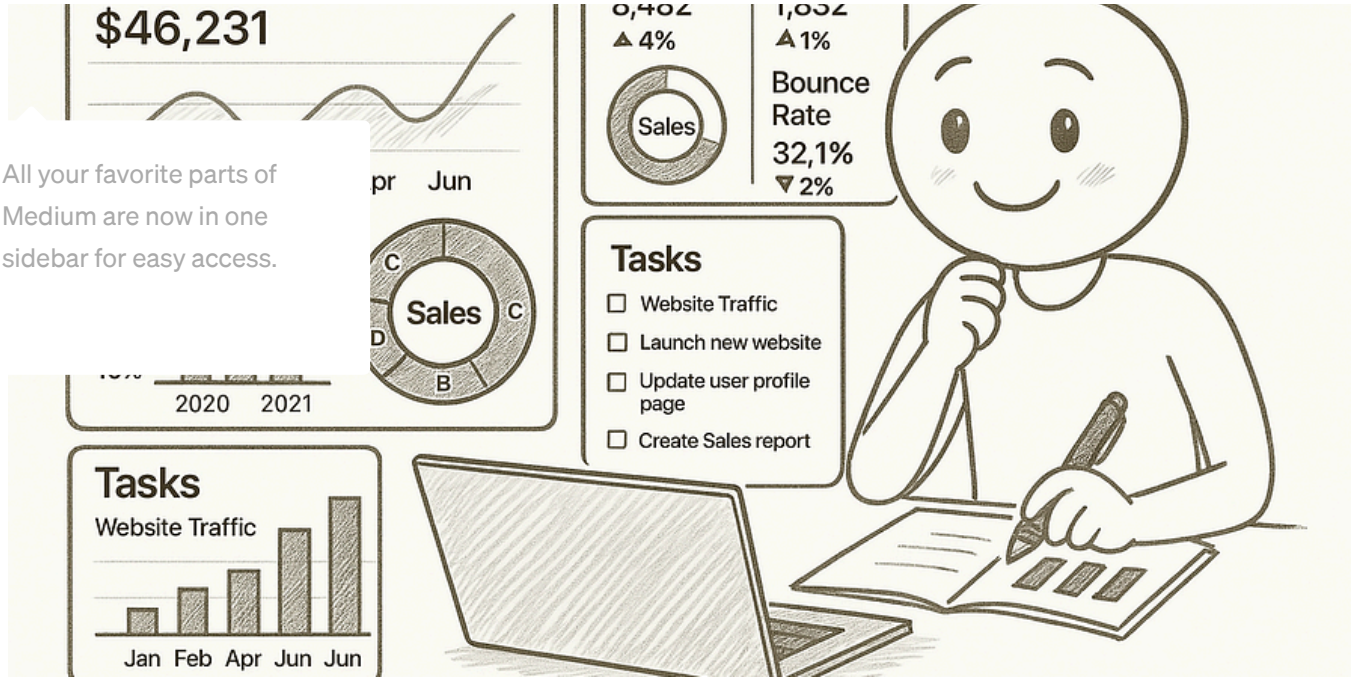


Nikulsinh Rajput

The Dashboard I Built in 20 Minutes Using Python & Streamlit

How Streamlit turned my messy CSV files into a polished, interactive data app—with almost no web dev.

Jun 18 · 46 · 3



In Code Like A Girl by Hang Nguyen

Dashboard Design Lessons I Gained from Exploring 100+ Impressive Dashboard Examples

Patterns, principles, and practical tips for better dashboard design

4d ago 481 3

Linking

Streamlit + Matplotlib: Visualization Dashboards

Hello,

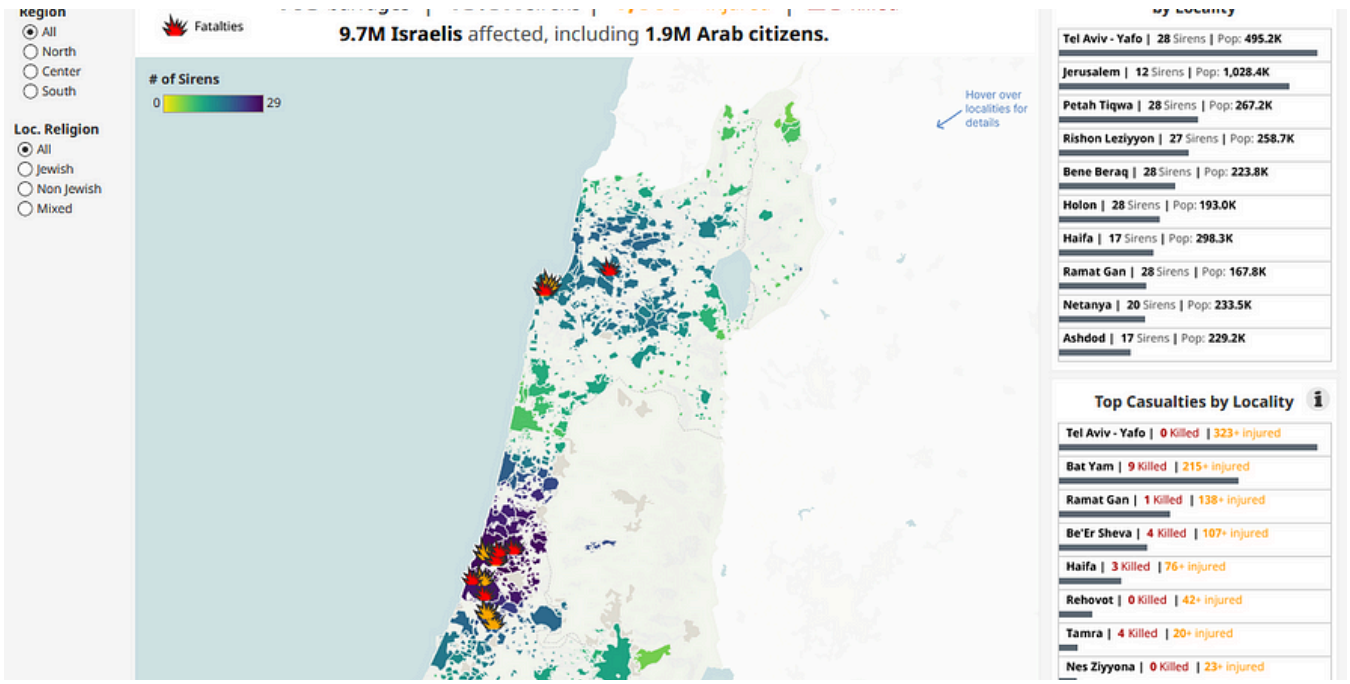
Apr 21

In Level Up Coding by Thomas Reid

Building a modern data dashboard

Using Python and the Gradio library

Jul 7 88 2



Nir Smilga

The Data behind “Iranian Projectiles” viz

In the days following the Iranian missile attacks, I wanted to create a clear, data-driven view of their impact on Israel's home front. The...

In Python in Plain English by Abdur Rahman

8 Python Libraries So Good, I Stopped Writing My Own Scripts

These will save you time, bugs, and brainpower

Jun 29 2.1K 26