Exploratory Data Analysis

What is Exploratory Data Analysis?

Exploratory data analysis is an approach/ philosophy for data analysis that employs a variety of techniques to:

1. maximize insight into a data set;
2. uncover underlying structure;
3. extract important variables;
4. detect outliers and anomalies;
5. test underlying assumptions;
6. develop parsimonious models; and
7. determine optimal factor settings.

Exploratory data analysis is an attitude about how a data analysis should be carried out. It is not statistical graphics but statically graphics is part of EDA. We use techniques like

1. Plotting the raw data
2. Plotting simple statistic
3. Positioning such plots to maximize our natural pattern – recognition abilities.

EDA is not only about Graphical representation of data, but it is the most used technique of EDA. We use different tools like/ Techniques like Graphical analysis, quantitate analysis and probability distribution.

Python has few best libraries that makes Python Visualization easy for any dataset. These libraries make Python Visualization affordable for large and small datasets.

Some of the most popular Libraries for Python Data Visualizations are:

Matplotlib, Seaborn, Pandas, Plotly and many more

GOOGLE CHARTS

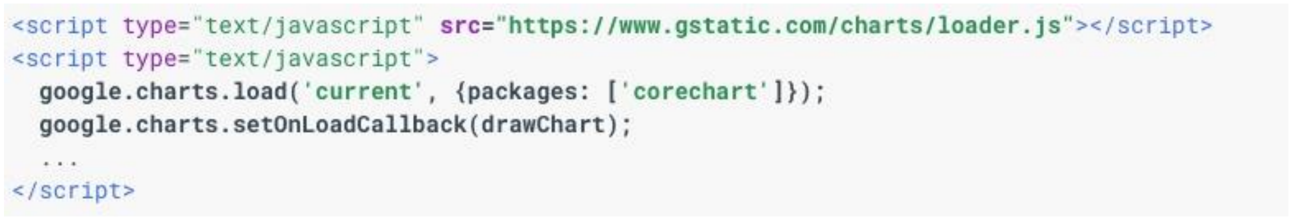
Google Charts is an interactive Web service that creates graphical charts from user-supplied information. The user supplies data and a formatting specification expressed in JavaScript embedded in a Web page; in response the service sends an image of the chart.
Google Charts provides a perfect way to visualise data on our website. From simple line charts to complex hierarchical tree maps, the chart gallery provides a large number of ready-to-use chart types.The most common way to use Google Charts is with simple JavaScript that we embed in our web page.

STEPS TO MAKE GOOGLE CHARTS:

1. We load some Google Chart libraries
2. List the data to be charted
3. Select options to customise the chart
4. Create a chart object with an id that we choose

Charts are exposed as JavaScript classes, and Google Charts provides many chart types for we to use.

1. LOADING GOOGLE CHARTS LIBRARY



```
<script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
<script type="text/javascript">
  google.charts.load('current', {packages: ['corechart']});
  google.charts.setOnLoadCallback(drawChart);
  ...
</script>
```

The first line of this example loads the loader itself.
We can only load the loader one time no matter how many charts we plan to draw. After loading the loader, we can call the google.charts.load function one or more times to load packages for particular chart types.

After loading the loader, we can call the google.charts.load function one or more times to load packages for particular chart types.
The first argument to google.charts.load is the version name or number, as a string. If we specify 'current', this causes the latest official release of Google Charts to be loaded. If we want to try the candidate for the next release, use 'upcoming' instead.

The example above assumes we want to display a corechart (bar, column, line, area, stepped area, bubble, pie, donut, combo, candlestick, histogram, scatter). If we want a different or additional chart type, substitute or add the appropriate package name for corechart above
(e.g., {packages: ['corechart','geochart', 'table', 'sankey']}.

Before we can use any of the packages loaded by google.charts.load we have to wait for the loading to finish. It is not enough to just wait for the document to finish loading. Since it can take some time before this loading is finished, **we need to register a callback function.**

This example also assumes that we have a JavaScript function named drawChart defined somewhere in our web page. We can name that function whatever we like, but be sure it is globally unique and that it is defined before we reference it in our call to google.charts.setOnLoadCallback.

LIMITATIONS**:**

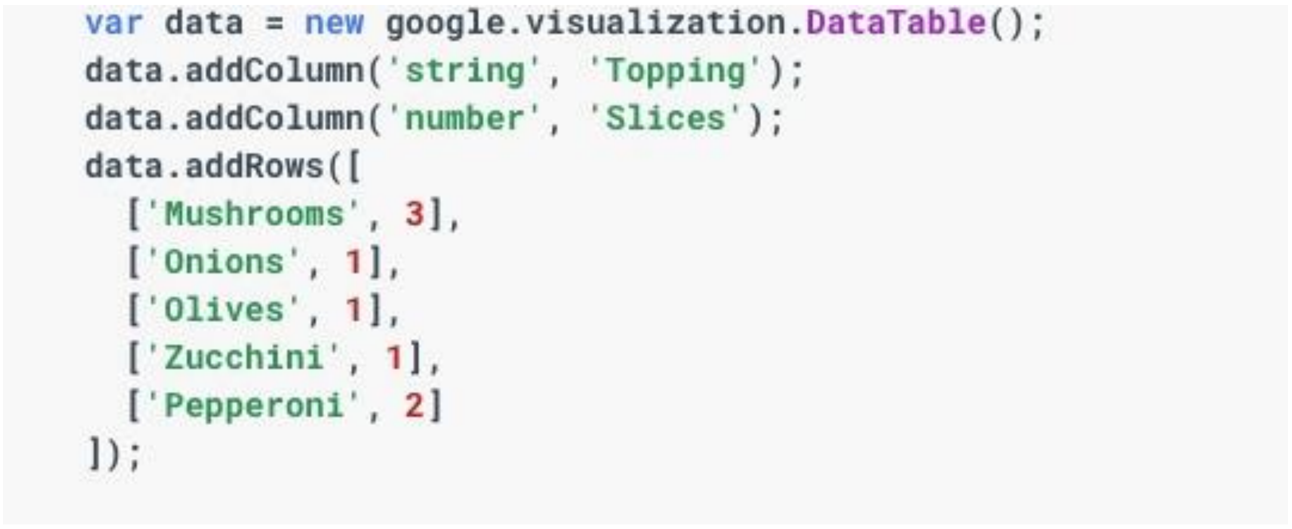
There are a couple minor but important limitations with how we load Google Charts in versions prior to 45:

1. We can only call google.charts.load once. But we can list all the packages that we'll need in one call, so there's no need to make separate calls.
2. We can't autoload the library.
3. If we're using a ChartWrapper, we must explicitly load all the packages we'll need, rather than relying on the ChartWrapper to automatically load them for we.
4. For Geochart and Map Chart, we must load both the old library loader and the new library loader.

2. PREPARING THE DATA TO BE CHARTED

Now we prepare the data. It can be done using JavaScript as well as Python.

In Google Charts, we create a DataTable. Google Chart Tools charts require data to be wrapped in a JavaScript class called google.visualization.DataTable.



```
var data = new google.visualization.DataTable();
data.addColumn('string', 'Topping');
data.addColumn('number', 'Slices');
data.addRows([
  ['Mushrooms', 3],
  ['Onions', 1],
  ['Olives', 1],
  ['Zucchini', 1],
  ['Pepperoni', 2]
]);
```

All charts require data. Google Chart Tools charts require data to be wrapped in a JavaScript class called google.visualization.DataTable. This class is defined in the Google Visualisation library that we loaded previously.

Using python to create the datable:

Google has open-sourced a Python library that creates DataTable objects for consumption by visualisations.
The output of it is in any of three formats -
JSON String - If we are hosting the page that hosts the visualisation that uses the data, we can generate a JSON string to pass into a DataTable constructor to populate it.
JSON response - If we do not host the page that hosts the visualisation, and just want to act as a data source for external visualisations, we can create a complete JSON response string that can be returned in response to a data request.
Javascript string -We can output the data table as a string that consists of several lines of JavaScript code that will create and populate a google.visualization.DataTable object with the data from the Python table. We can then run this JavaScript in an engine to generate and populate the google.visualization.DataTableobject. This is typically used for debugging only.

LIBRARY TO IMPORT AND USE IN PYTHON:

1. Import the gviz_api.py library
2. instantiate the gviz_api.DataTable class - The class takes two parameters: a table schema, which will describe the format of the data in the table, and optional data to populate the table with.
3. Describe the table schema - The table schema is specified by the table_description parameter passed into the constructor.
4. Each column is described by a tuple: **(ID [,data_type [,label [,custom_properties]]]).**

- ID - A string ID used to identify the column. Can include spaces. The ID for each column must be unique.

- data_type - [optional] A string descriptor of the Python data type of the data in that column. We can find a list of supported data types in the SingleValueToJS() method. Examples include "string" and "boolean". If not specified, the default is "string."
- label - A user-friendly name for the column, which might be displayed as part of the visualisation. If not specified, the ID value is used.
- custom_properties - A {String:String} dictionary of custom column properties.

The table schema is a collection of column descriptor tuples.

Examples of descriptor tuple:

- List of columns: [('a', 'number'), ('b', 'string')]
- Dictionary of lists: {('a', 'number'): [('b', 'number'), ('c', 'string')]}
- Dictionary of dictionaries: {('a', 'number'): {'b': 'number', 'c': 'string'}}

The table schema is a collection of column descriptor tuples. Every list member, dictionary key or dictionary value must be either another collection or a descriptor tuple. We can use any combination of dictionaries or lists, but every key, value, or member must eventually evaluate to a descriptor tuple.

Populate the data - To add data to the table, build a structure of data elements in the exact same structure as the table schema.

Example:
schema is a list

- schema: [("color", "string"), ("shape", "string")]
- data: [["blue", "square"], ["red", "circle"]]

schema is a dictionary:

- schema: {("rowname", "string"): [("color", "string"), ("shape", "string")] }
- data: {"row1": ["blue", "square"], "row2": ["red", “circle"]}
-

Output the data —> The most common output format is JSON. For this we use, ToJsonResponse() function

3. Customise the Chart

Every chart has many customisable options, including title, colors, line thickness, background fill, and so on. Although the Chart Tools team has worked hard on the default chart appearance, we might want to customise the chart, for example to add titling or axis labels.

Specify custom options for the chart by defining a JavaScript object with *option_name*/*option_value* properties. Use the option names listed in the chart's documentation. Every chart's documentation lists a set of customisable options.

Specify Chart Size

One very common option to set is the chart height and width. We can specify the chart size in two places: in the HTML of the container <div> element, or in the chart options. If we specify the size in both locations, the chart will generally defer to the size specified in the HTML. If we don't specify a chart size either in the HTML or as an option, the chart might not be rendered properly.

There are advantages to specifying the size in one or the other place:

- **Specifying the size in HTML** - A chart can take a few seconds to load and render. If we have the chart container already sized in HTML, the page layout won't jump around when the chart is loaded.
- **Specifying the size as a chart option** - If the chart size is in the JavaScript, we can copy and paste, or serialise, save, and restore the JavaScript and have the chart resized consistently.

4. Draw the Chart

The last step is to draw the chart. First we must instantiate an instance of the chart class that we want to use, and then we must call draw() on the it.

```
    // Instantiate and draw our chart, passing in some options.
    var chart = new google.visualization.PieChart(document.getElementById('chart_div'));
    chart.draw(data, options);
  }
```

Each chart type is based on a different class, listed in the chart's documentation. For instance, the pie chart is based on the google.visualization.PieChart class, the bar chart is based on the google.visualization.BarChart class, and so on. Both pie and bar charts are included in the corechart package. However, if we want a treemap or geo chart on our page, we must additionally load the 'treemap' or 'geomap' packages.

Our page must have an HTML element (typically a <div>) to hold our chart. We must pass our chart a reference to this element, so assign it an ID that we can use to retrieve a reference using document.getElementById(). Anything inside this element will be replaced by the chart when it is drawn.

Every chart supports a draw() method that takes two values: a DataTable (or a DataView) object that holds our data, and an optional chart options object. The options object is not required, and we can ignore it or pass in null to use the chart's default options.
The draw() method is asynchronous: that is, it returns immediately, but the instance that it returns might not be immediately available. In many cases this is fine; the chart will be drawn eventually. However, if we want to set or retrieve values on a chart after we've called draw(), we must wait for it to throw the ready event, which indicates that it is populated.

RUNNING DEMO:

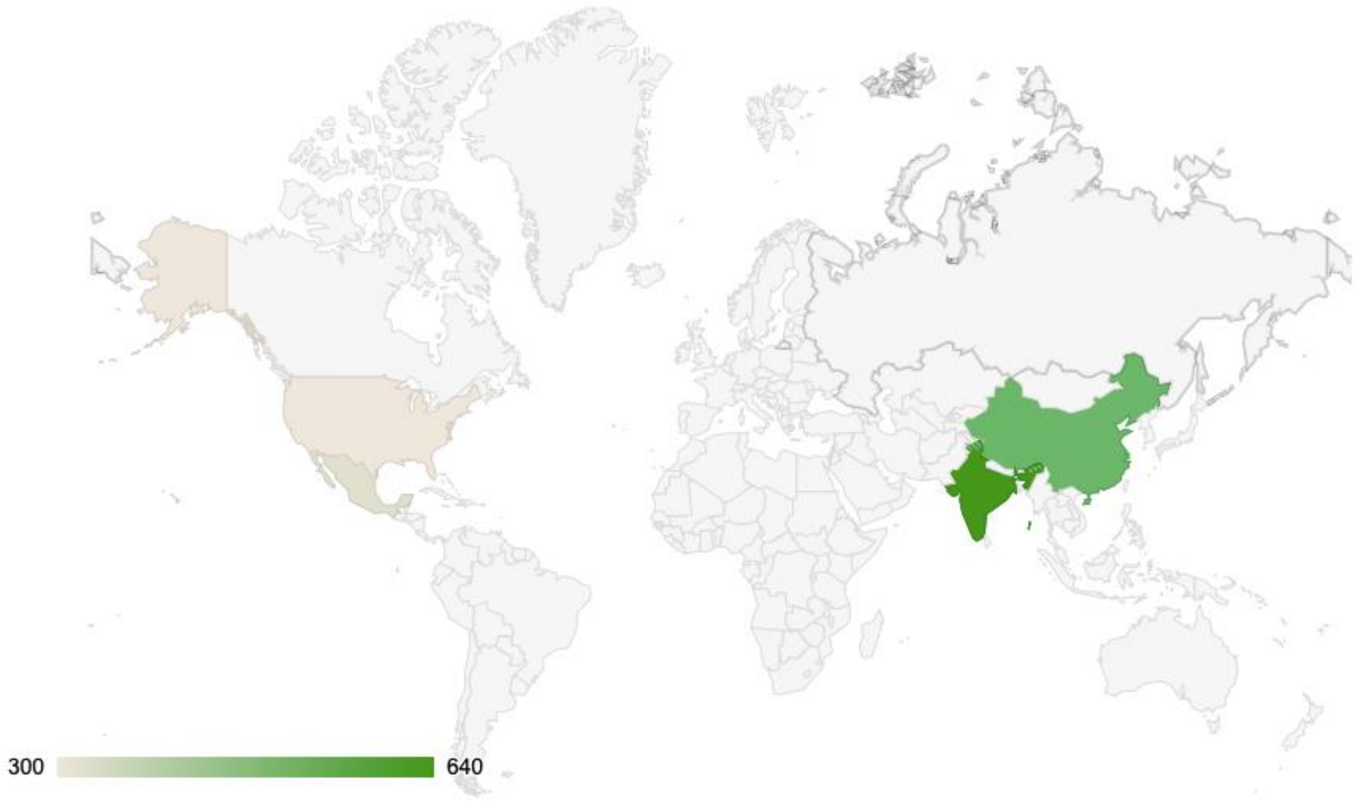
JavaScript code:

```
<html>
  <head>
    <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
    <script type="text/javascript">
      google.charts.load('current', {
        'packages':['geochart'],
        // Note: you will need to get a mapsApiKey for your project.
        // See: https://developers.google.com/chart/interactive/docs/basic_load_libs#load-settings
        'mapsApiKey': 'AIzaSyD-9tSrke72PouQMnMX-a7eZSW0jkFMBWY'
      });
      google.charts.setOnLoadCallback(drawRegionsMap);

      function drawRegionsMap() {
        var data = google.visualization.arrayToDataTable([
          ['Country', 'Popularity'],
          ['India', 640],
          ['Mexico', 320],
          ['United States', 300],
          ['China', 500],
        ]);

        var options = {};

        var chart = new google.visualization.GeoChart(document.getElementById('regions_div'));

        chart.draw(data, options);
      }
    </script>
  </head>
  <body>
    <div id="regions_div" style="width: 900px; height: 500px;"></div>
  </body>
</html>
```

<u>Calling it from Python:</u>

```
import webbrowser
url = 'file:///FilePath'
webbrowser.open(url, new=2)
```

<u>OUTPUT</u>:



Seaborn

Introduction:

- Seaborn is a graphic library for creating informative and attractive statistical data visualization in python.
- It can support some more complicated, complex and 3D visualizations approaches simpler to create.
- Seaborn offers various features such as built in themes, colour palettes, functions and tools to visualize univariate, bivariate, linear regression, matrices of data, statistical time series etc. which let us to build complex visualizations and also it integrates well with the functionality provided by Pandas DataFrames.

Seaborn vs Matplotlib:

- Seaborn is built on top of Matplotlib and introduces additional plot types. It is meant to serve as a complement, and not a replacement.
- It seeks to make default data visualizations much more visually appealing and it also has the goal of making more complicated plots simpler to create.
- Seaborn helps resolve the two major problems faced by Matplotlib
  1. Default Matplotlib parameters
  2. Working with DataFrames
- Matplotlib takes too much work to get reasonable looking graphs. While with seaborn, it is easier to get nice looking visualization without a lot of code.

How to import data to construct plots:

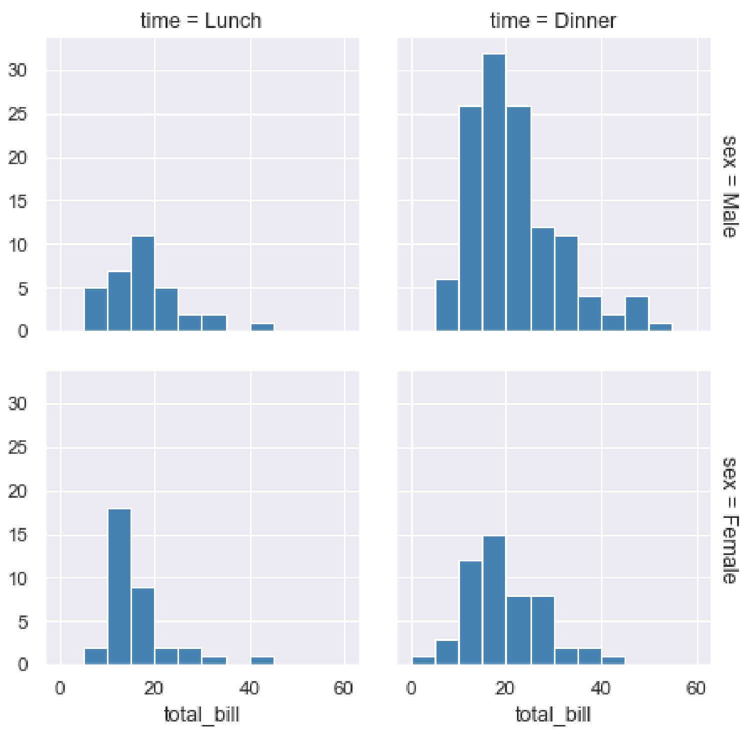
- There are two ways to load the data :
  1. Use one of the built-in data sets that the library itself has to offer
     e.x : iris = sns.load_dataset(“iris”)
  2. Load a Pandas DataFrame
     e.x : x = pd.read_csv(“file1.csv”)

- Of course, most of the fun in visualizing data lies in the fact that you would be working with your own data and seaborn works best with Pandas DataFrames and arrays that contain a whole data set.

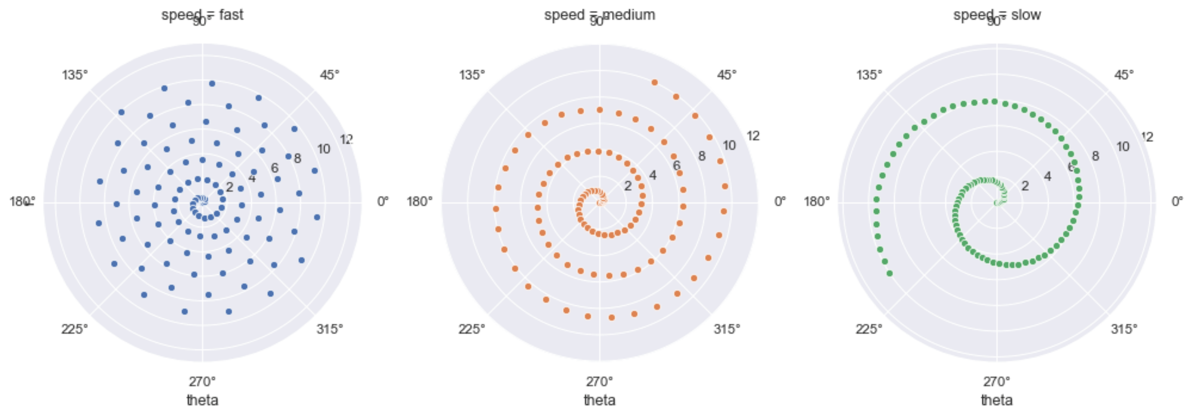
Exploring Seaborn plots:

Let's take a look at a few of the plot types available in Seaborn

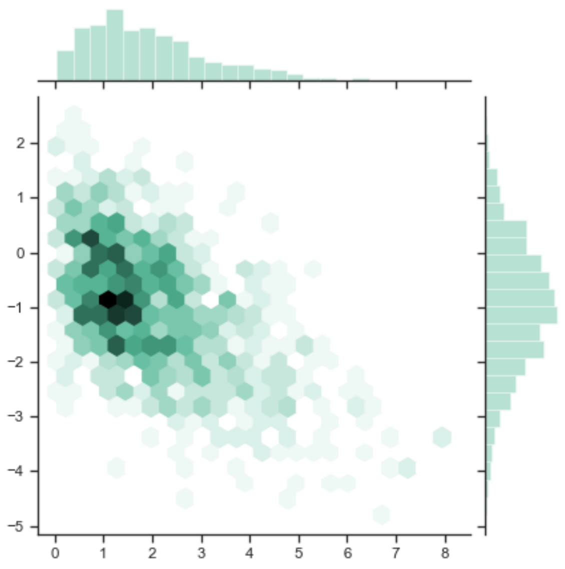
- *Facetting histograms by subsets of data*



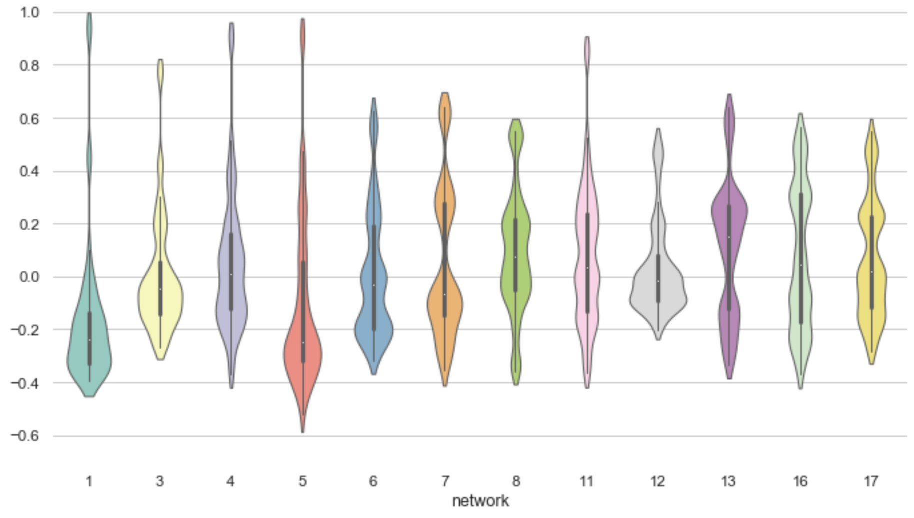
- *FacetGrid with custom projection*



- *Hexbin plot with marginal distributions*



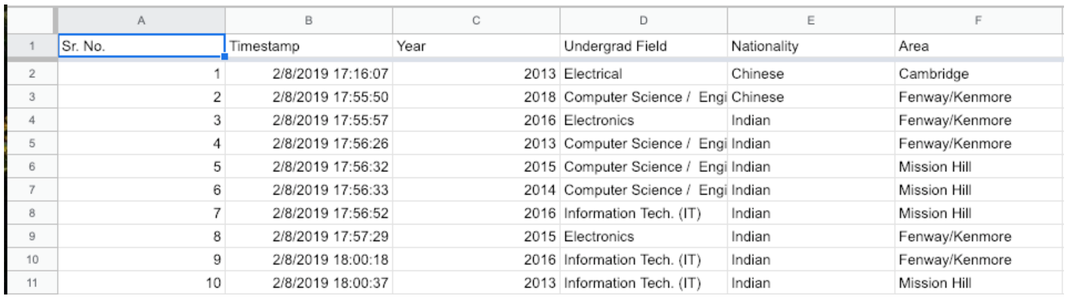
- *Violinplot from a wide-form dataset*



Example:

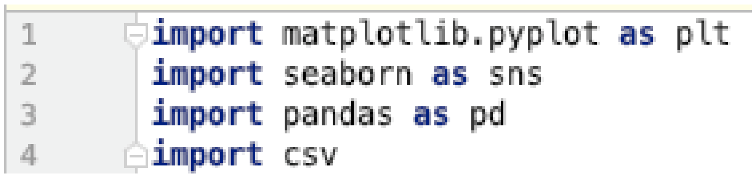
Here is an example of creating a graphical visualization in python:

We have following DataFrame which has some information of students studying in our class and we want to analyse that most of us were in which field during our undergrad.



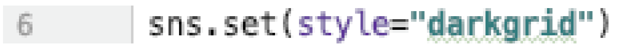
	A	B	C	D	E	F
1	Sr. No.	Timestamp	Year	Undergrad Field	Nationality	Area
2	1	2/8/2019 17:16:07	2013	Electrical	Chinese	Cambridge
3	2	2/8/2019 17:55:50	2018	Computer Science / Engi	Chinese	Fenway/Kenmore
4	3	2/8/2019 17:55:57	2016	Electronics	Indian	Fenway/Kenmore
5	4	2/8/2019 17:56:26	2013	Computer Science / Engi	Indian	Fenway/Kenmore
6	5	2/8/2019 17:56:32	2015	Computer Science / Engi	Indian	Mission Hill
7	6	2/8/2019 17:56:33	2014	Computer Science / Engi	Indian	Mission Hill
8	7	2/8/2019 17:56:52	2016	Information Tech. (IT)	Indian	Mission Hill
9	8	2/8/2019 17:57:29	2015	Electronics	Indian	Fenway/Kenmore
10	9	2/8/2019 18:00:18	2016	Information Tech. (IT)	Indian	Fenway/Kenmore
11	10	2/8/2019 18:00:37	2013	Information Tech. (IT)	Indian	Mission Hill

First, import the necessary libraries



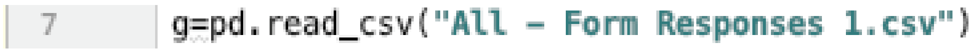
```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import csv
```

Set the style by calling Seaborn's set()



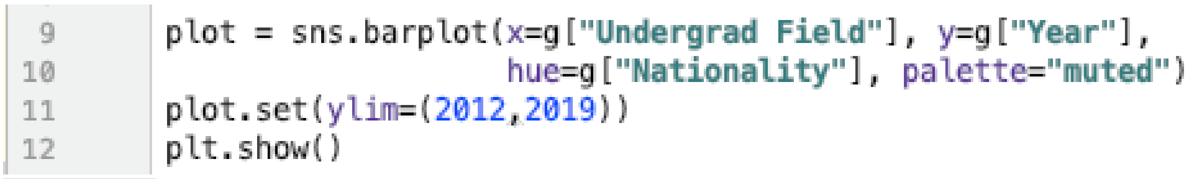
```python
sns.set(style="darkgrid")
```

Load CSV file



```python
g=pd.read_csv("All - Form Responses 1.csv")
```

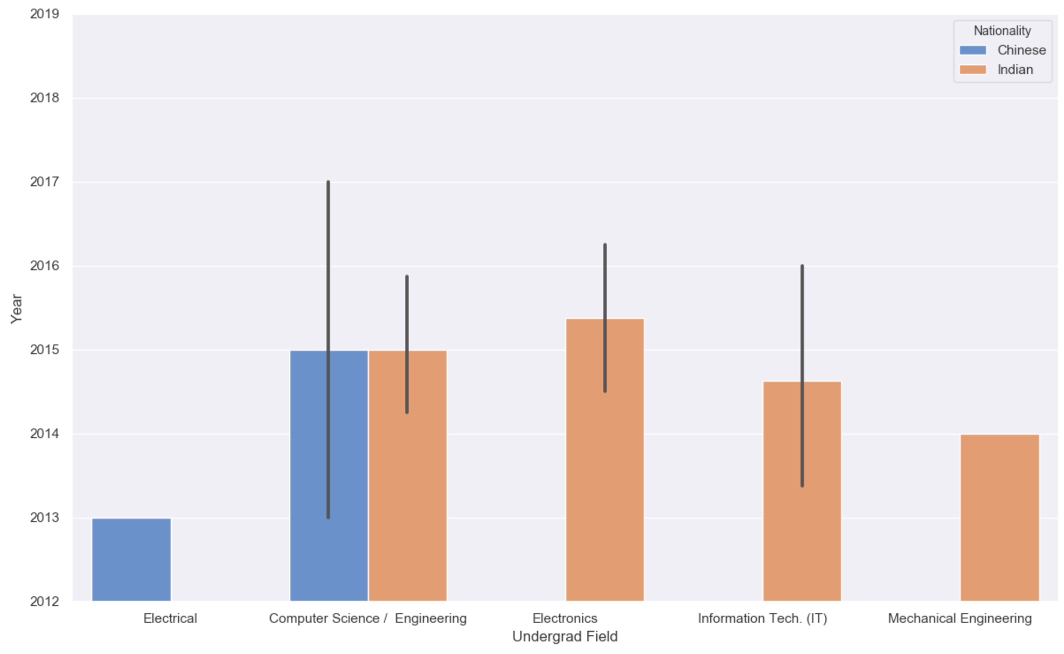
Some plotting cade to plot a barplot



```python
plot = sns.barplot(x=g["Undergrad Field"], y=g["Year"],
                   hue=g["Nationality"], palette="muted")
plot.set(ylim=(2012,2019))
plt.show()
```

plt.show() to make the image appear to you.

Plotly Report



Plotly Report

1. What is Plotly?

Plotly, also known by its URL, is a technical computing company that develops online data analytics and visualization tools.

Plotly provides online graphing, analytics, and statistics tools for individuals and collaboration, as well as scientific graphing libraries for Python, R, MATLAB, Perl, Julia, Arduino, and REST.

2. The Advantages of Plotly?

It is interfacing with multiple mainstream graphing softwares and can do interactive drawing like excel. It has many kinds of charts and can be shared open source for free.

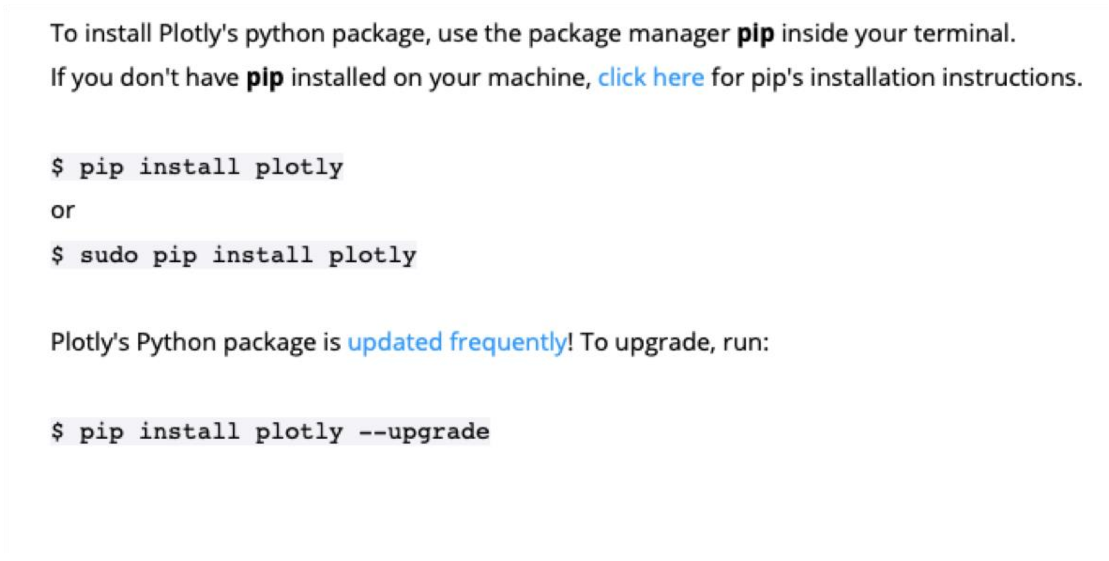
The chart type: basic chart: 20 kinds; statistics and shipping mode: 12 kinds; scientific chart: 21 kinds; financial chart: 2 kinds; map: 8 kinds; 3D chart: 19 kinds; fitting tools: 3 kinds; Flow chart: 4 kinds

Interactivity: can be connected to R, python, Matlab and other software, and is open source free

Aesthetics: based on modern color combination, chart form, more modern and beautiful than Matlab, R language chart

3. How to use Plotly with Python?

- Install


To install Plotly's python package, use the package manager **pip** inside your terminal.
If you don't have **pip** installed on your machine, click here for pip's installation instructions.

```
$ pip install plotly
```
or
```
$ sudo pip install plotly
```

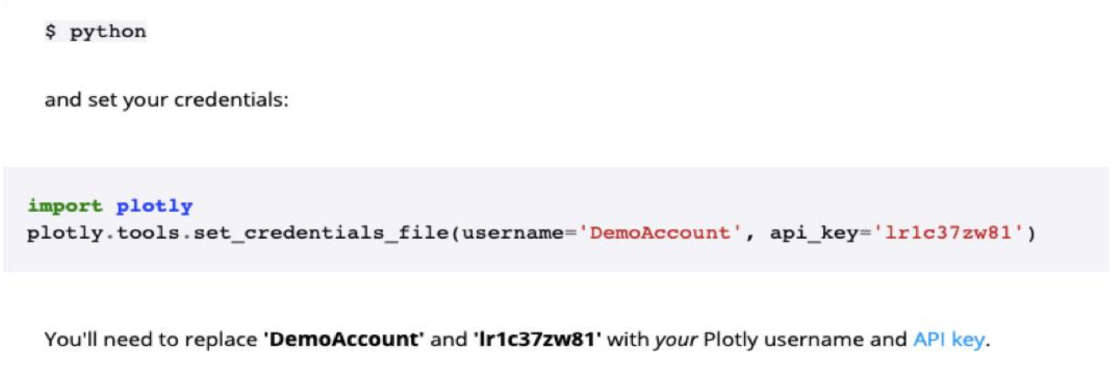
Plotly's Python package is updated frequently! To upgrade, run:

```
$ pip install plotly --upgrade
```

- Use
  Online-Using

Initialization of online using:

tools.set_credentials_file(username='yours', api_key='yours’)


```
$ python
```

and set your credentials:

```
import plotly
plotly.tools.set_credentials_file(username='DemoAccount', api_key='lr1c37zw81')
```

You'll need to replace **'DemoAccount'** and **'lr1c37zw81'** with *your* Plotly username and API key.

Offline-Using

Initialize the drawing mode in the Jupyter Notebook first.

Also, offline drawing has two methods: plotly.offline.plot() and plotly.offline.iplot(). Heatmap (Jupyter Notebook), the former one comes out an html page automatically, the latter one shows the result directly in Jupyter Notebook.

Google Data Studio

Google Data Studio is a communication tool. It brings together data you store in several places so you can visualize it on one screen. The goal of using Data Studio is to become a data communicator, not a data plumber.

Key Features & Benefits

● Connect to all your data sources in one place

● Clean and transform your data

● Build reports using the drag-and-drop visual editor

● Create interactive reports and dashboards

● Tell a story using a combination of words, numbers, and visuals

● Easily share reports and dashboards

Reasons to Use Data Studio

1. Create as many reports as you need
2. Customize the appearance of reports and dashboards
3. Combine data from different views and sources
4. Create calculated dimensions and metrics
5. Share reports easily

Connecting the data in Google visual data studio

You can create report in Data Studio using data source from

● Google Analytics

● AdWords

● Search Console

● Google Sheets

● BigQuery

● CSV (Comma Separated Values) file

Metrics & Dimensions

- City, Campaign, and Device Category are **Dimensions** (attributes)
- Sessions, Pages/Session, Avg. Session Duration are **Metrics** (Measures)

Dimensions (attributes)

City

Device Category

Page Title

Campaign Name

Browser

Event Category

Metrics (measures)

Users

Sessions

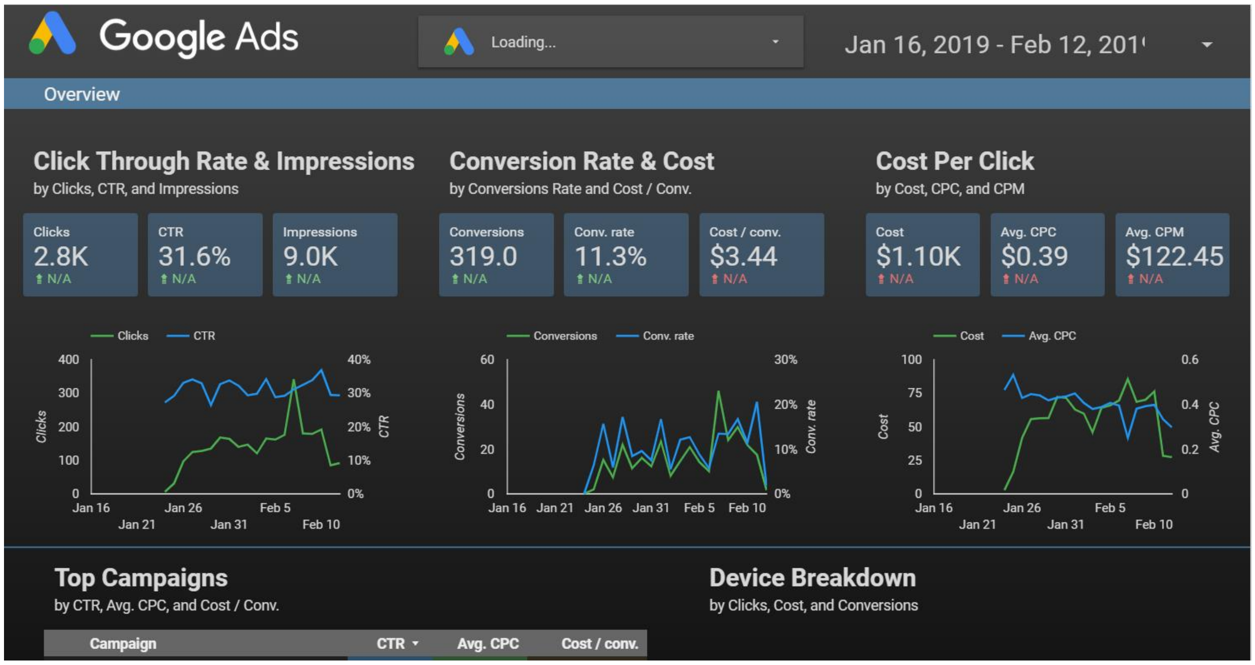
Pageviews

Avg. Session Duration

Ecommerce Conversion Rate

Bounce Rate

Reports:



Click Through Rate & Impressions
by Clicks, CTR, and Impressions

Clicks	CTR	Impressions
2.8K	31.6%	9.0K
N/A	N/A	N/A

Conversion Rate & Cost
by Conversions Rate and Cost / Conv.

Conversions	Conv. rate	Cost / conv.
319.0	11.3%	$3.44
N/A	N/A	N/A

Cost Per Click
by Cost, CPC, and CPM

Cost	Avg. CPC	Avg. CPM
$1.10K	$0.39	$122.45
N/A	N/A	N/A

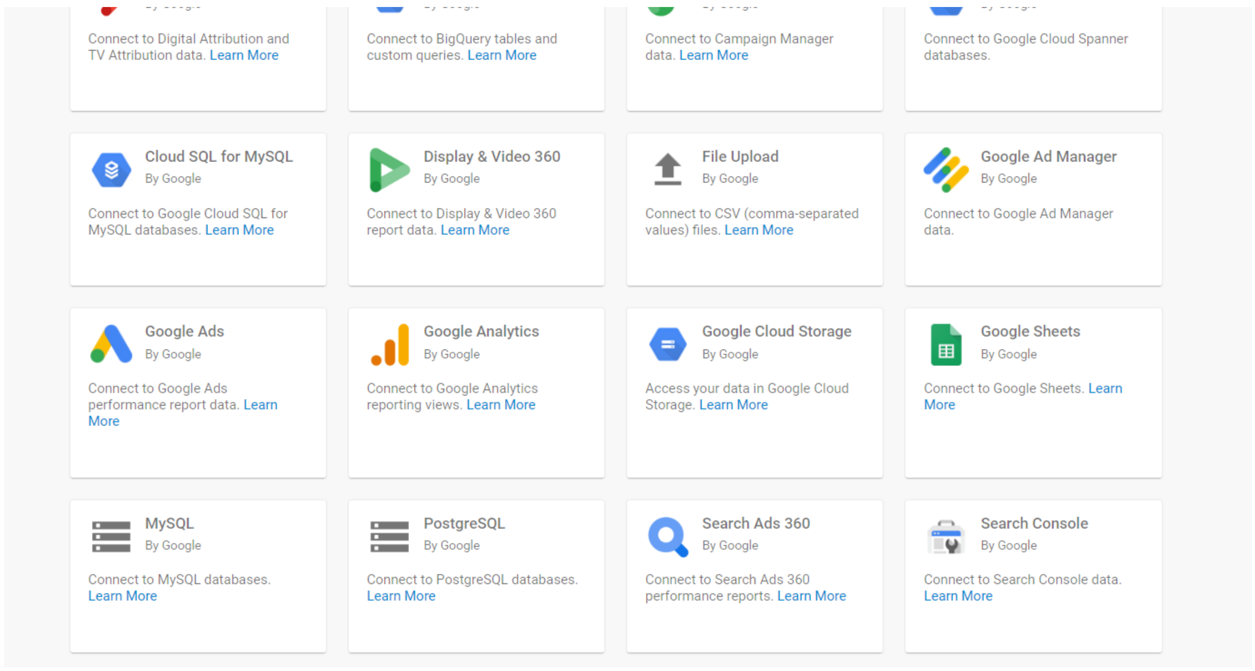
Top Campaigns
by CTR, Avg. CPC, and Cost / Conv.

Campaign	CTR	Avg. CPC	Cost / conv.

Device Breakdown
by Clicks, Cost, and Conversions

You can use different graphs and charts to create visual report shown above.

You can connect the data from the following source:



Connect to Digital Attribution and TV Attribution data. Learn More

Connect to BigQuery tables and custom queries. Learn More

Connect to Campaign Manager data. Learn More

Connect to Google Cloud Spanner databases.

Cloud SQL for MySQL
By Google
Connect to Google Cloud SQL for MySQL databases. Learn More

Display & Video 360
By Google
Connect to Display & Video 360 report data. Learn More

File Upload
By Google
Connect to CSV (comma-separated values) files. Learn More

Google Ad Manager
By Google
Connect to Google Ad Manager data.

Google Ads
By Google
Connect to Google Ads performance report data. Learn More

Google Analytics
By Google
Connect to Google Analytics reporting views. Learn More

Google Cloud Storage
By Google
Access your data in Google Cloud Storage. Learn More

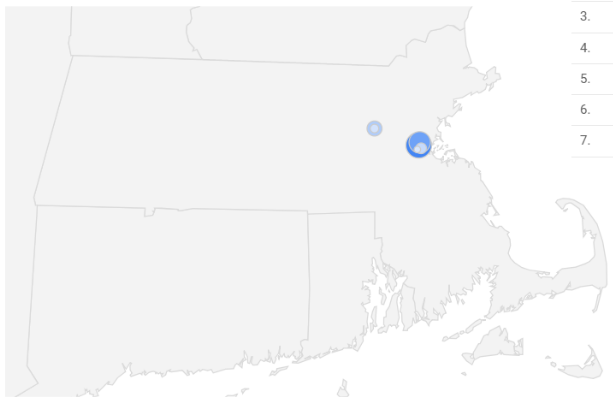
Google Sheets
By Google
Connect to Google Sheets. Learn More

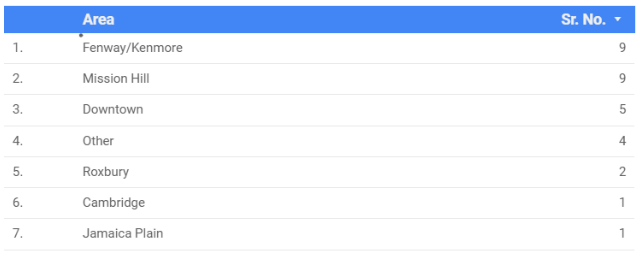
MySQL
By Google
Connect to MySQL databases. Learn More

PostgreSQL
By Google
Connect to PostgreSQL databases. Learn More

Search Ads 360
By Google
Connect to Search Ads 360 performance reports. Learn More

Search Console
By Google
Connect to Search Console data. Learn More





	Area	Sr. No.
1.	Fenway/Kenmore	9
2.	Mission Hill	9
3.	Downtown	5
4.	Other	4
5.	Roxbury	2
6.	Cambridge	1
7.	Jamaica Plain	1

1 - 7 / 7

Demographic of class students based on data collected from the class

Prepared By:

Smit Chandarana

Zoe

Nidhi Sharma

Krishna Modi

Vaibhavi Khamar