UNIT-V

Introduction to Seaborn

Seaborn is a statistical plotting library and is built on top of matplotlib. It has beautiful default styles and is compatible with pandas dataframe objects. In order to install it use the following commands:

Anaconda users: conda install seaborn

Python users: pip install seaborn

Seaborn code is opensource so one can read it at

https://github.com/mwaskom/seaborn. This page has information along with the link to the official documentation page - https://seaborn.pydata.org/. The subsection of this page (https://seaborn.pydata.org/examples/index.html) shows the example visualizations Seaborn is able to work with. The other important section to visit is the one which has API information - https://seaborn.pydata.org/api.html.

Seaborn functionalities and usage

Let us now delve into some of the functionalities Seaborn provides. We will run some snippet of codes in Jupyter notebook (although you can use any other IDE) to exhibit the key features.

Distribution Plots

We will first try to do distribution plots. To get started, we first import one of the standard datasets that comes with Seaborn. The one we choose for our exercise is diamonds.csv. You can pick other datasets from https://github.com/mwaskom/seaborn-data.

```
###### By convention we import Seaborn as sns
In [1]: import seaborn as sns
```

The below command will allow us to see the visualizations inline

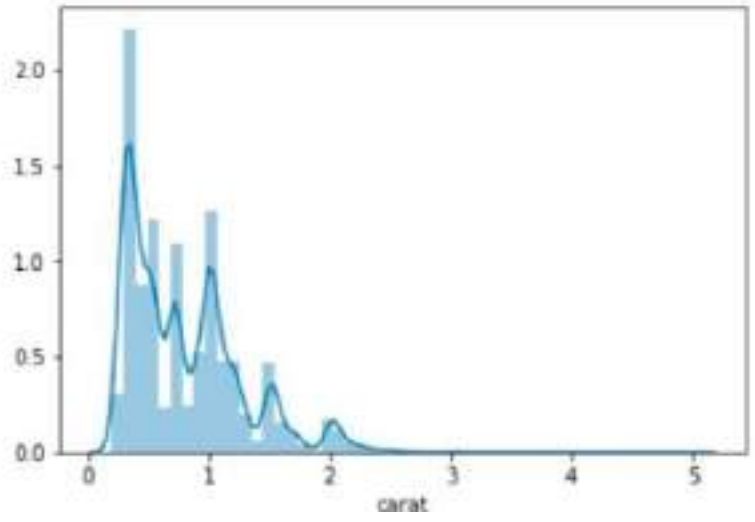
```
In [2]: %matplotlib inline

In [3]: dmd = sns.load_dataset('diamonds')

In [4]: dmd.head()
```

Out[4]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

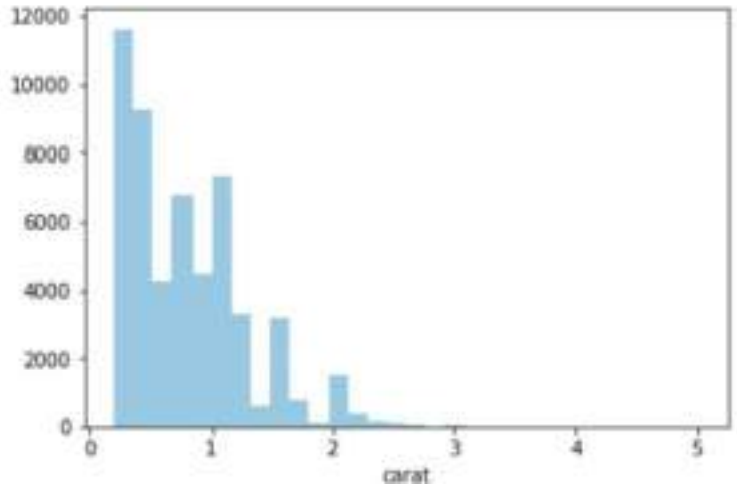
We then use the “displot” function to plot the distribution of a single variable

```
In [6]: sns.distplot(dmd['carat'])
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0xb0112e8>
```



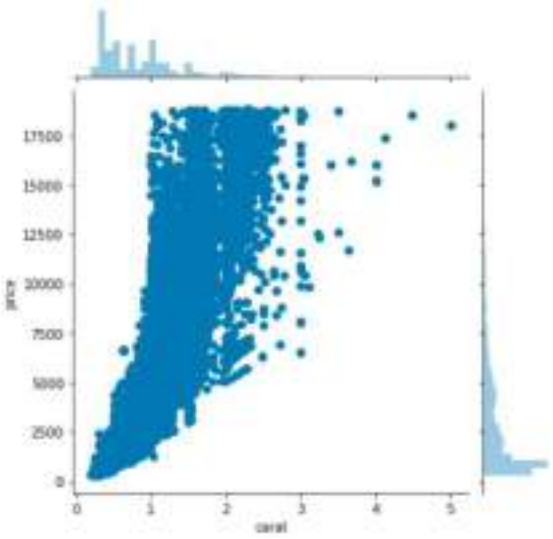
As seen above, we get a histogram and a Kernel Density Estimate (KDE) plot. We can customize it further by removing KDE and specifying the number of bins.

```
In [13]: sns.distplot(dmd['carat'],kde=False,bins=30)
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0xcb67128>
```



Joint plot allows us to plot the relationship represented by bivariate data.

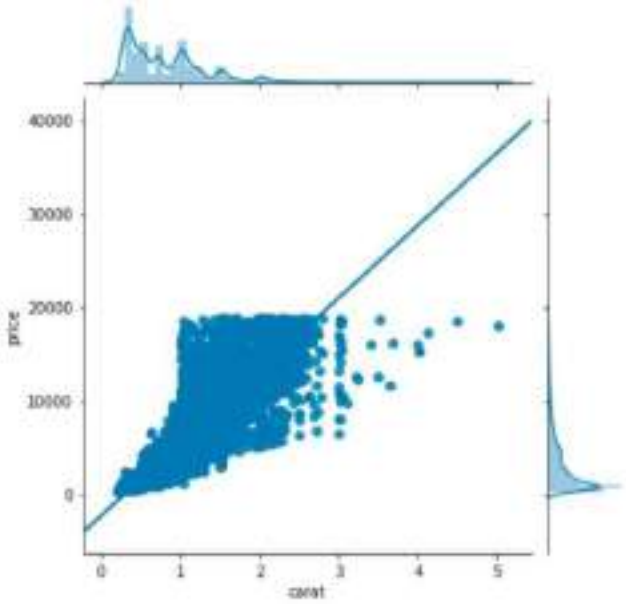
```
In [16]: sns.jointplot(x='carat',y='price',data=dmd,kind='scatter')
Out[16]: <seaborn.axisgrid.JointGrid at 0xcd530b8>
```



The above diagram shows that as the carat approaches the value '5', the value of the diamond also increases which is a phenomenon we also observe. The 'jointplot' function can take multiple values for the parameter 'kind' – scatter, reg, resid, kde, hex.
Let us see the plot using 'reg' which will provide regression and kernel density fits.

```
sns.jointplot(x='carat',y='price',data=dmd,kind='reg')
```

```
<seaborn.axisgrid.JointGrid at 0xd1f2438>
```



Next Steps for students to try :

1. Pairplot function: will plot pairwise relationships across an entire dataframe such that each numerical variable will be depicted in the y-axis across a single row and in the x-axis across a single column. Try the command: sns.pairplot(dmd)
2. Rugplot function: It plots datapoints in an array as sticks on an axis. Try the command: sns.rugplot(dmd['price'])
3. Once you are comfortable with these then you can try out KDE plotting. KDE plotting is used for visualizing the probability density of a continuous variable or a single graph for multiple samples.
   Reference: https://seaborn.pydata.org/generated/seaborn.kdeplot.html

Categorical Plots :

Now let us discuss how to use seaborn to plot categorical data. But let us first understand what categorical variable is. A categorical variable is one that has multiple categories but has no intrinsic ordering specified for categories. For example: Blood type of a person can be any one of A, B, AB or O.

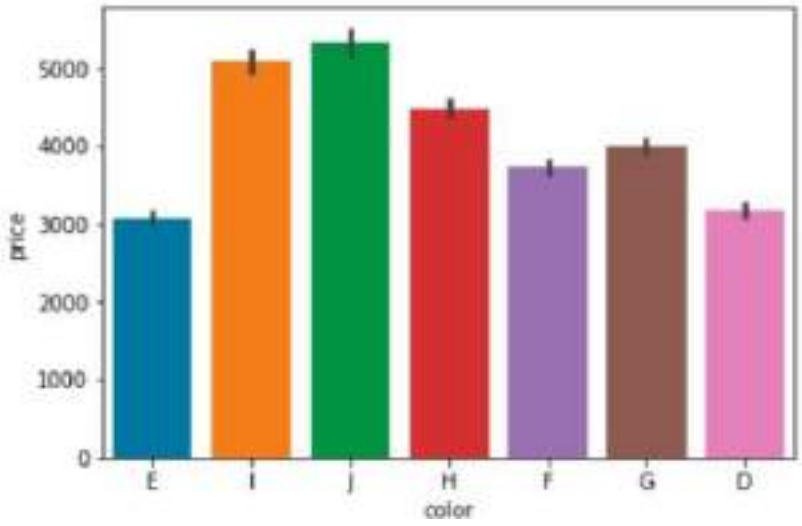
Now let us see examples of the plots:

Barplot and countplot allow you to aggregate data with respective to each category. Barplot

allows to you aggregate around some function but the default is mean.

```
sns.barplot(x='color',y='price',data=dmd)
```

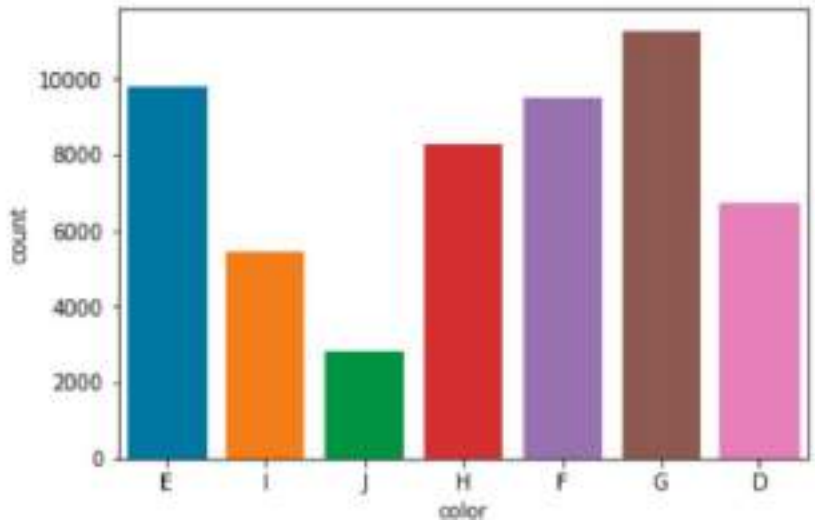
```
<matplotlib.axes._subplots.AxesSubplot at 0x1655af98>
```



The difference between countplot and barplot is that countplot explicitly counts the number of occurrences.

```
sns.countplot(x='color',data=dmd)
```

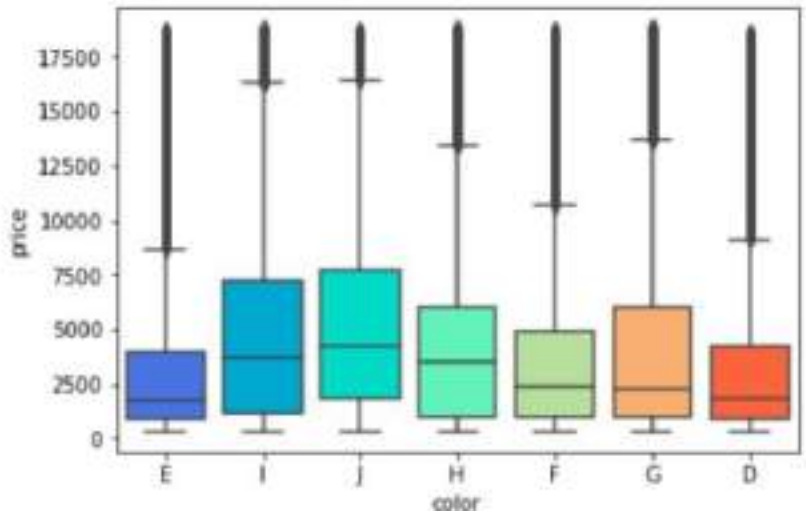
```
<matplotlib.axes._subplots.AxesSubplot at 0x16551860>
```



Boxplot shows the quartiles of the dataset while the whiskers extend encompass the rest of the distribution but leave out the points that are the outliers.

```
sns.boxplot(x="color", y="price", data=dmd,palette='rainbow')
```

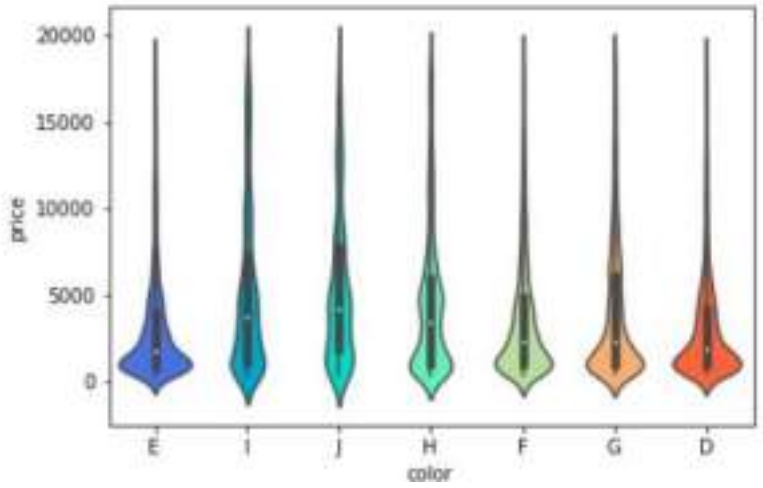
```
<matplotlib.axes._subplots.AxesSubplot at 0x16688358>
```



Violinplot shows the distribution of data across several levels of categorical variable(s) thus helping in comparison of the distribution. Wherever actual datapoints are not present, KDE is used to estimate the remaining points.

```
sns.violinplot(x="color", y="price", data=dmd,palette='rainbow')
```

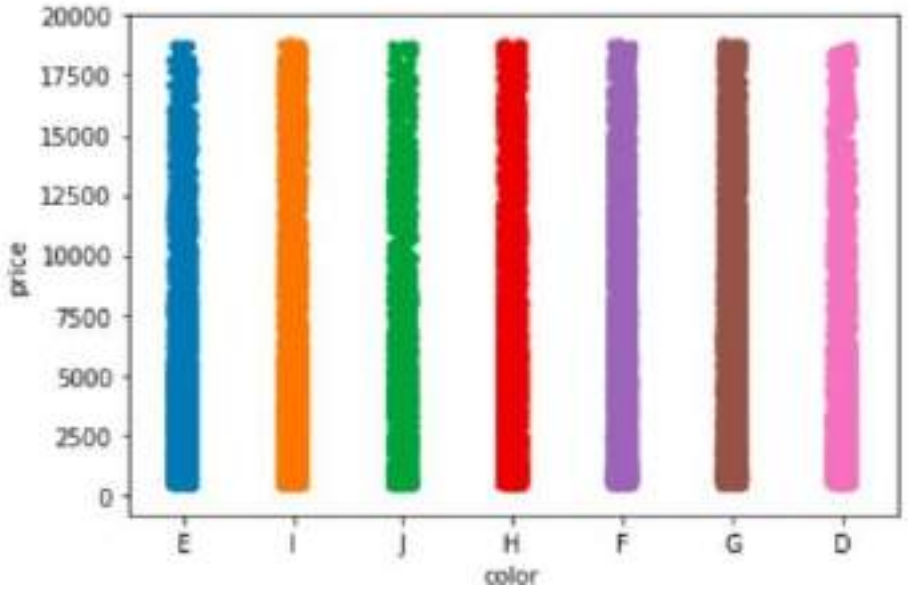
```
<matplotlib.axes._subplots.AxesSubplot at 0x16688470>
```



The stripplot draws a scatterplot where one variable is categorical.

```
sns.stripplot(x="color", y="price", data=dmd)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x177bc828>
```



Matrix Plots :

Now let us delve into matrix plots. It helps to segregate data into color-encoded matrices which can further help in unsupervised learning methods like clustering.

```
import seaborn as sns
```

```
%matplotlib inline
```

```
dmd = sns.load_dataset('diamonds')
```

```
dmd.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

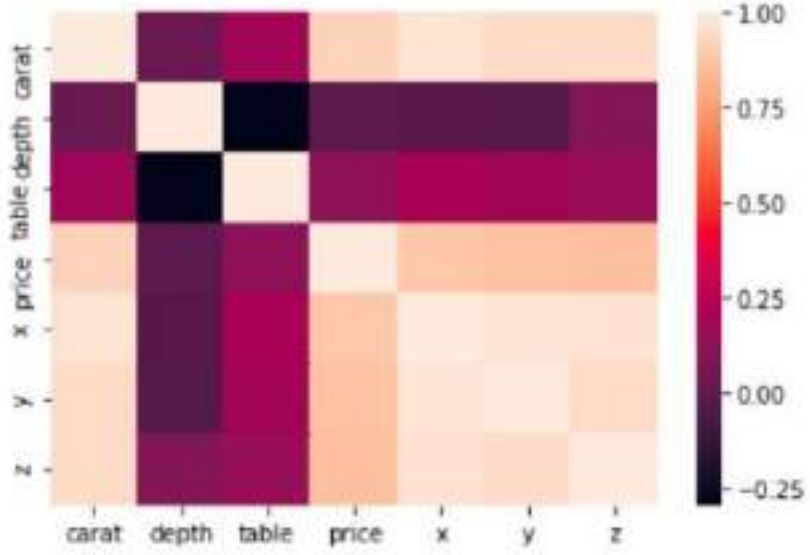
```
dmd.corr()
```

	carat	depth	table	price	x	y	z
carat	1.000000	0.028224	0.181618	0.921591	0.975094	0.951722	0.953387
depth	0.028224	1.000000	-0.295779	-0.010647	-0.025289	-0.029341	0.094924
table	0.181618	-0.295779	1.000000	0.127134	0.195344	0.183760	0.150929
price	0.921591	-0.010647	0.127134	1.000000	0.884435	0.865421	0.861249
x	0.975094	-0.025289	0.195344	0.884435	1.000000	0.974701	0.970772
y	0.951722	-0.029341	0.183760	0.865421	0.974701	1.000000	0.952006
z	0.953387	0.094924	0.150929	0.861249	0.970772	0.952006	1.000000

The corr() function gives the matrix form to correlation data.
Below command generates the heatmap.

```
sns.heatmap(dmd.corr())
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xc6bdef0>
```




```
flightdata = sns.load_dataset('flights')
flightdata.head()
```

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Let us now try the pivot_table formation. Now we need to select the appropriate data for that. Among the available datasets in seaborn, flights data is most suitable to depict this. Let us try to depict the total number of passengers for each month of the year.

```
flightdata = sns.load_dataset('flights')
flightdata.head()
```

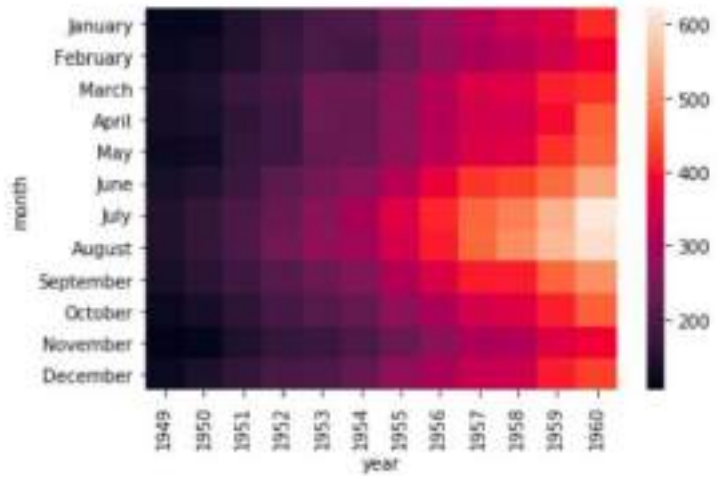
	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

```
flightdata.pivot_table(values='passengers',index='month',columns='year')
```

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
January	112	115	145	171	196	204	242	284	315	340	360	417
February	118	126	150	180	196	188	233	277	301	318	342	391
March	132	141	178	193	236	235	267	317	356	362	406	419
April	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
June	135	149	178	218	243	264	315	374	422	435	472	535
July	148	170	199	230	264	302	364	413	465	491	548	622
August	148	170	199	242	272	293	347	405	467	505	559	606
September	136	158	184	209	237	259	312	355	404	404	463	508
October	119	133	162	191	211	229	274	306	347	359	407	461
November	104	114	146	172	180	203	237	271	305	310	362	390
December	118	140	166	194	201	229	278	306	336	337	405	432

```
sns.heatmap(flightdata.pivot_table(values='passengers',index='month',columns='year'))
```

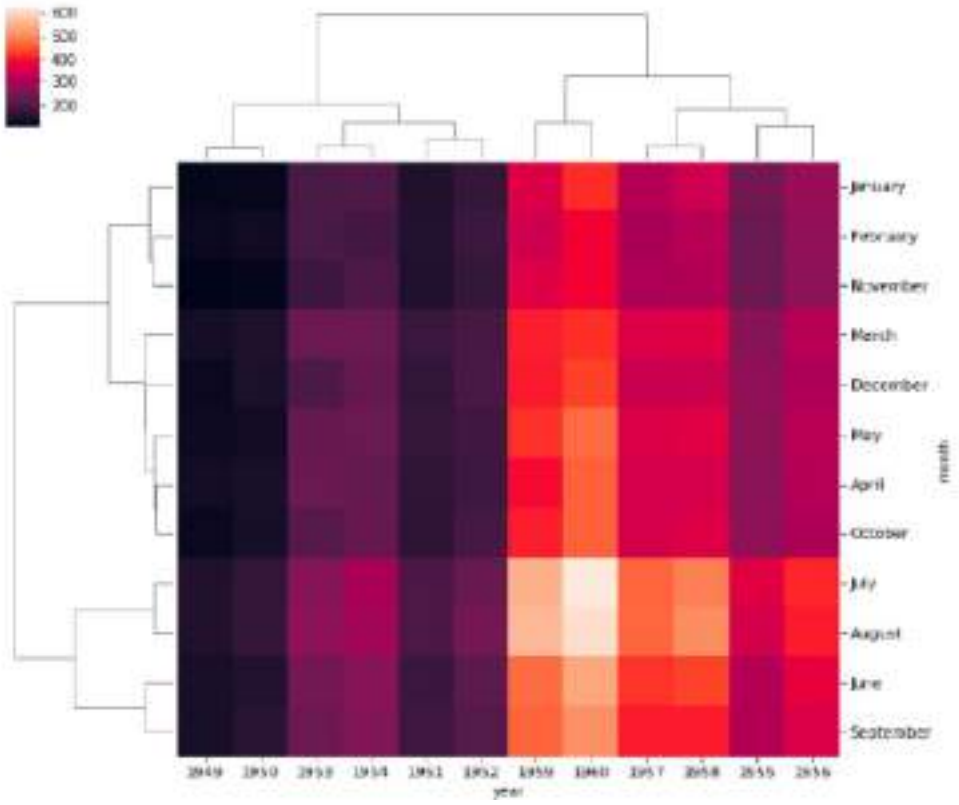
```
<matplotlib.axes._subplots.AxesSubplot at 0x83a84e0>
```



The cluster map uses hierarchal clustering. It no more depicts months and years in order but groups them similarity in the passenger count. So, it can be inferred that April and May are similar in passenger volume.

```
sns.clustermap(flightdata.pivot_table(values='passengers',index='month',columns='year'))
```

```
<seaborn.matrix.ClusterGrid at 0x30dccc0>
```



Regression plot :

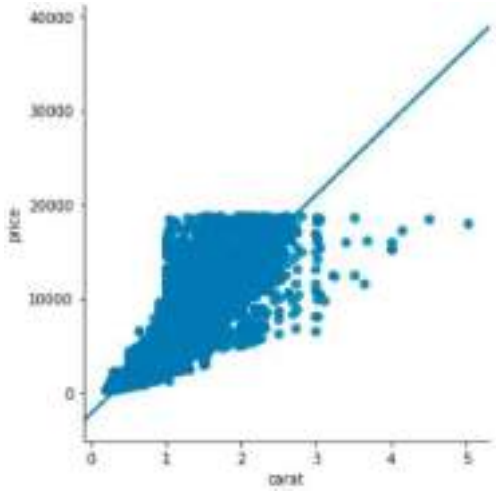
In this final section, we will explore seaborn and see how efficient it is to create regression lines and fits using this library. Implot function allows you to display linear models.

```
dmd.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

```
sns.lmplot(x='carat',y='price',data=dmd)
```

```
<seaborn.axisgrid.FacetGrid at 0xc5354a8>
```



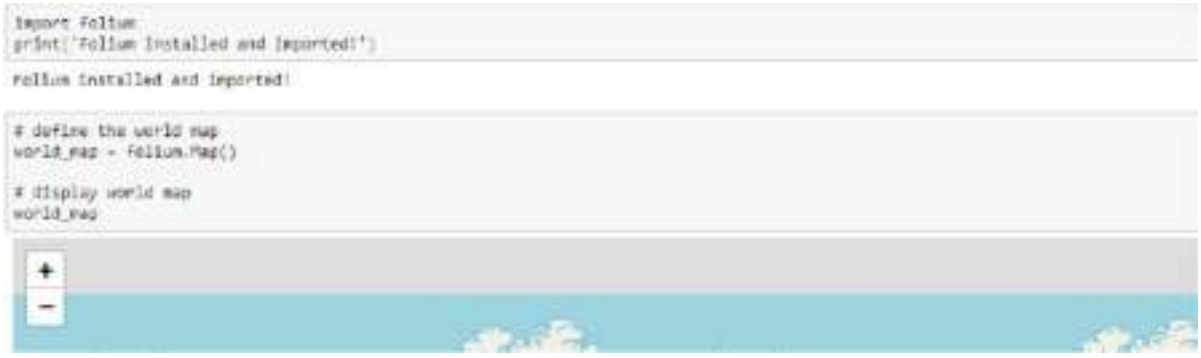
Spatial Visualizations and Analysis in Python with Folium

Folium is a powerful data visualization library in Python that was built primarily to help people visualize geospatial data. With Folium, you can create a map of any location in the world if you know its latitude and longitude values. You can also create a map and superimpose markers as well as clusters of markers on top of the map for cool and very interesting visualizations. You can also create maps of different styles such as street level map, stamen map.

Folium is not available by default. So, we first need to install it before we can import it. We can use the command : conda install -c conda-forge folium=0.5.0 --yes

It is not available via default conda channel. Try using conda-forge channel to install folium as shown: conda install -c conda-forge folium

Generating the world map is straightforward in Folium. You simply create a Folium Map object and then you display it. What is attractive about Folium maps is that they are interactive, so you can zoom into any region of interest despite the initial zoom level.



Go ahead. Try zooming in and out of the rendered map above. You can customize this default definition of the world map by specifying the center of your map and the initial zoom level.

All locations on a map are defined by their respective Latitude and Longitude values. So, you can create a map and pass in a center of Latitude and Longitude values of [0, 0]. For a defined center, you can also define the initial zoom level into that location when the map is rendered. The higher the zoom level the more the map is zoomed into the center. Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map.

Let's create the map again with a higher zoom level

As you can see, the higher the zoom level the more the map is zoomed into the given center.

Stamen Toner Maps

These are high-contrast B+W (black and white) maps. They are perfect for data mashups and exploring river meanders and coastal zones. Let's create a Stamen Toner map of Canada with a zoom level of 4.

```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Toner')

# display map
world_map
```

Stamen Terrain Maps

These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads. Let's create a Stamen Terrain map of Canada with zoom level 4.

```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Terrain')

# display map
world_map
```

Mapbox Bright Maps

These are maps that are quite like the default style, except that the borders are not visible with a low zoom level. Furthermore, unlike the default style where country names are displayed in each country's native language, Mapbox Bright style displays all country names in English. Let's create a world map with this style.

Case Study

Now that you are familiar with folium, let us use it for our next case study which is as mentioned below:

Case Study: An e-commerce company ‘ wants to get into logistics “Deliver4U” . It wants to know the pattern for maximum pickup calls from different areas of the city throughout the day. This will result in:

i) Build optimum number of stations where its pickup delivery personnel will be located.
ii) Ensure pickup personnel reaches the pickup location at the earliest possible time.

For this the company uses its existing customer data in Delhi to find the highest density of probable pickup locations in the future.

Solution:

Pre-requisites : Python, Jupyter Notebooks, Pandas

Data set : Please download the following from the location specified by the trainer.

The dataset contains two separate data files – train_del.csv and test_del.csv. The difference is that train_del.csv contains additional column which is trip_duration which we will not be needed for our present analysis.

Importing and pre-processing data:

a) Import libraries – Pandas and Folium. Drop the trip_duration column and combine the 2 different files as one dataframe.

```
import pandas as pd
import folium
```

```
df_train = pd.read_csv('train_del.csv').drop(columns=['trip_duration', 'dropoff_datetime'])
```

```
df_train.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id2875421	2	2016-03-14 17:24:55	1	77.207046	28.637937	77.225370	28.636502	N
1	id2377394	1	2016-06-12 00:43:35	1	77.209587	28.680564	77.180519	28.601132	N
2	id3858529	2	2016-01-19 11:35:24	1	77.210373	28.633938	77.184607	28.539087	N
3	id3504673	2	2016-04-06 19:32:31	1	77.179040	28.589971	77.177712	28.576718	N
4	id2181028	2	2016-03-26 13:30:55	1	77.216042	28.563200	77.217377	28.652520	N

```
df_test=pd.read_csv('test_del.csv')
```

```
df_test.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id3004672	1	2016-06-30 23:59:58	1	77.201077	28.592323	77.189827	28.626680	N
1	id3505355	1	2016-06-30 23:59:53	1	77.225797	28.549963	77.230192	28.525403	N
2	id1217141	1	2016-06-30 23:59:47	1	77.192563	28.597583	77.203842	28.539525	N
3	id2150126	2	2016-06-30 23:59:41	1	77.233932	28.641980	77.203573	28.600459	N
4	id1598245	1	2016-06-30 23:59:33	1	77.219788	28.631475	77.229492	28.625690	N

```
df=pd.concat([df_train,df_test],sort=False,ignore_index=True)
```

```
df.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id2875421	2	2016-03-14 17:24:55	1	-73.982155	40.767937	-73.964630	40.765602	N
1	id2377394	1	2016-06-12 00:43:35	1	-73.980415	40.738564	-73.999481	40.731152	N
2	id3858529	2	2016-01-19 11:35:24	1	-73.979027	40.763939	-74.005333	40.710087	N
3	id3504673	2	2016-04-06 19:32:31	1	-74.010040	40.719971	-74.012268	40.706718	N
4	id2181028	2	2016-03-26 13:30:55	1	-73.973053	40.793209	-73.972923	40.782520	N

preview of the data we will be working with

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2083778 entries, 0 to 2083777
Data columns (total 9 columns):
id                  object
vendor_id           int64
pickup_datetime     object
passenger_count     int64
pickup_longitude    float64
pickup_latitude     float64
dropoff_longitude   float64
dropoff_latitude    float64
store_and_fwd_flag  object
dtypes: float64(4), int64(2), object(3)
memory usage: 143.1+ MB
```

```
df.pickup_datetime = pd.to_datetime(df.pickup_datetime, format='%Y-%m-%d %H:%M:%S')
```

```
pd.to_datetime(df.pickup_datetime)
```

```
0         2016-03-14 17:24:55
1         2016-06-12 00:43:35
2         2016-01-19 11:35:24
3         2016-04-06 19:32:31
4         2016-03-26 13:30:55
5         2016-01-30 22:01:40
6         2016-06-17 22:34:59
7         2016-05-21 07:54:58
8         2016-05-27 23:12:23
9         2016-03-10 21:45:01
10        2016-05-10 22:08:41
11        2016-05-15 11:16:11
12        2016-02-19 09:52:46
13        2016-06-01 20:58:29
14        2016-05-27 00:43:36
15        2016-05-16 15:29:02
```

We will need to generate some columns such as month or other time features using Datetime package of python. Let us then use it with Folium

```
df['month'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.month)
df['week'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.week)
df['day'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.day)
df['hour'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.hour)
```

```
df.head()
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2083778 entries, 0 to 2083777
Data columns (total 13 columns):
id                  object
vendor_id           int64
pickup_datetime     datetime64[ns]
passenger_count     int64
pickup_longitude    float64
pickup_latitude     float64
dropoff_longitude   float64
dropoff_latitude    float64
store_and_fwd_flag  object
month               int64
week                int64
day                 int64
hour                int64
dtypes: datetime64[ns](1), float64(4), int64(6), object(2)
memory usage: 206.7+ MB
```

Please note that month, week, day, hour columns will be used next for our analysis
Note the following regarding visualizing spatial data with Folium:
• Maps are defined as folium.Map object. We will need to add other objects on top of this before rendering

• Different map tiles for map rendered by Folium can be seen at

: https://github.com/pythonvisualization/folium/tree/master/folium/templates/tiles

• Folium.Map() : First thing to be executed when you work with Folium.

Let us define the default map object:

```
def generateBaseMap(default_location=[28.617937, 77.2078415], default_zoom_start=12):
    base_map = folium.Map(location=default_location, control_scale=True, zoom_start=default_zoom_start)
    return base_map
```

```
base_map=generateBaseMap()
base_map
```

Let us now visualize the rides data using a class method called Heatmap()

Using data from May to June 2016 to generate the heat map

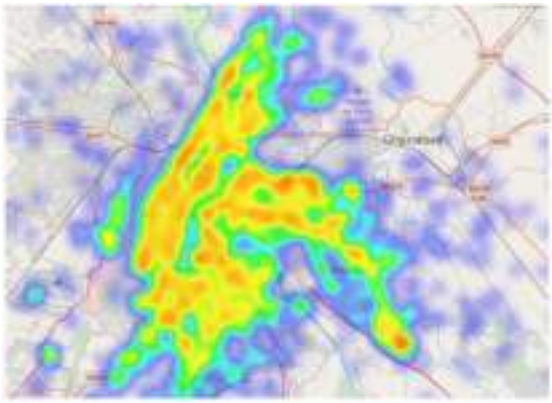
```
from folium.plugins import HeatMap
```

```
df_copy = df[df.month<6].copy()
df_copy['count'] = 1
base_map = generateBaseMap()
```

```
###HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_latitude', 'pickup_longitude']).su
HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_latitude', 'pickup_longitude']).sum())
```

```
<folium.plugins.heat_map.HeatMap at 0x25bcb2b0>
```

```
base_map
```



Code for reference:

```
from folium.plugins import HeatMap
df_copy = df[df.month>4].copy()
df_copy['count'] = 1
base_map = generateBaseMap()
HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude',
'count']].groupby(['pickup_latitude',
'pickup_longitude']).sum().reset_index().values.tolist(), radius=8,
max_zoom=13).add_to(base_map)
```

Interpretation of the output:

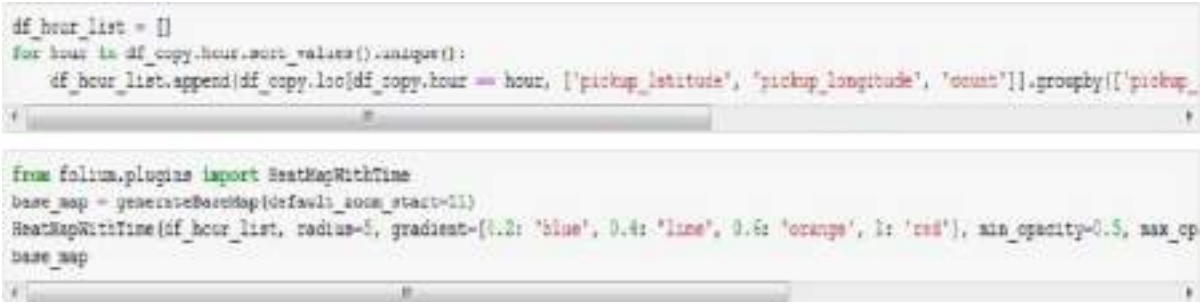
There is high demand for cabs in areas marked by the heat map which is central Delhi most probably and other surrounding areas.

Now let us add functionality to add markers to the map by using the folium.ClickForMarker() object.

After adding the below line of code, we can add markers on the map to recommends points where logistic pickup stops can be built

We can also animate our heat maps to dynamically change the data on timely basis based on a certain dimension of time. This can be done using HeatMapWithTime(). Use the following code :

```
df_hour_list = []
for hour in df_copy.hour.sort_values().unique():
df_hour_list.append(df_copy.loc[df_copy.hour == hour, ['pickup_latitude',
'pickup_longitude', 'count']].groupby(['pickup_latitude',
'pickup_longitude']).sum().reset_index().values.tolist())
from folium.plugins import HeatMapWithTime
base_map = generateBaseMap(default_zoom_start=11)
HeatMapWithTime(df_hour_list, radius=5, gradient={0.2: 'blue', 0.4: 'lime', 0.6:
'orange', 1: 'red'}, min_opacity=0.5, max_opacity=0.8,
use_local_extrema=True).add_to(base_map)
base_map
```



Conclusion

Throughout the city, pickups are more probable from central area so better to set lot of pickup stops at these locations

Therefore, by using maps we can highlight trends and uncover patterns and derive insights from the data.