Group no. 3



**Data Analysis project on covid-19**

Presented by:

**ANSHU SWATI**

TABLE OF CONTENTS:

|  |  |
| --- | --- |
| 1. | Table of figures |
| 2. | Abstract |
| 3. | Chapters |
| 4. | Complete code |
| 5. | Conclusion |

TABLE OF FIGURES AND GRAPHS

1. STATEWISE COVID DATA IN INDIA
   * Figure 1…………………………………………………………………..18
   * Figure 2…………………………………………………………………..20
   * Figure 3…………………………………………………………………..22
2. OVERALL COVID19 STATUS IN INDIA

* Figure 1…………………………………………………………………..22

1. VISUALIZING THE SPREADS GEOGRAPHICALLY

* Map 1……………………………………………………………………..27
* Map 2……………………………………………………………………..27
* Map 3……………………………………………………………………..28

1. TREND OF VIRUS
   * Graph 1…………………………………………………………………...30
   * Graph 2…………………………………………………………………...31
   * Graph 3…………………………………………………………………...31
   * Graph 4…………………………………………………………………...32
2. FORECAST USING fbprophet
   * Graph 1…………………………………………………………………...37
   * Graph 2…………………………………………………………………...40
   * Graph 3…………………………………………………………………...44

**ABSTRACT**

We are creating a Data Analysis Project on Spread of Novel Corona Virus in India. The project uses raw data in form of .csv files and .xls files and transforms it into a data analysis. This project is an attempt of data analyzing Corona Virus (COVID-19) spread in India with the help of data science and data analytics in python code. This analysis will help us to find the basis behind common notions about the virus spread from purely a data set perspective.

The data used for this project is split across two files:

1. Covid\_19\_India.csv
2. Indian Coordinates.xls

For the analysis we use some libraries such as pandas, numpy, matplotlib, seaborn, plotly, stdlib, folium, warnings and Fbprophet.

In the project, we import files and explore the parameters in both files. By using some user-defined functions we clean the data.

We also filter off the unwanted entities, thus make our data perfect for analysis.

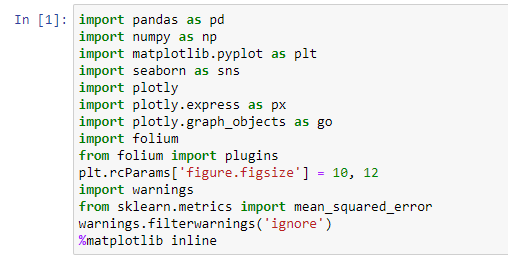
Then we merge the data from datasets into DataFrames for further analysis. For the states of India, we do a state wise or overall data analysis. For this we use pie charts and sub plots.

For visualizing the spread of corona virus geographically, we use folium library and Indian coordinates.xls to map them to world map.

We use Fbprophet and predict the future prospectus by analyzing the existing data. We then compare the results of analysis with the real data.

CHAPTERS

**Importing Data Visualization Libraries in Python**



* **import pandas as pd** : Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data. Usually you would add the second part (‘as pd’) so you can access Pandas with ‘pd.command’ instead of needing to write ‘pandas.command’ every time you need to use it.
* **import numpy as np**: NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed. It also discusses the various array functions, types of indexing, etc
* **import matplotlib.pyplot as plt**: Matplotlib is one of the most popular Python packages used for data visualization. It is a cross-platform library for making 2D plots from data in arrays. It provides an object-oriented API that helps in embedding plots in applications using Python GUI toolkits such as PyQt, WxPythonotTkinter.
* **import seaborn as sns**: Seaborn is a Python data visualization library based on [matplotlib](https://matplotlib.org/). It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn aims to make visualization a central part of exploring and understanding data. Its dataset-oriented plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots.
* **import plotly**: Plotly's Python graphing library makes interactive, publication-quality graphs. Examples of how to make line plots, scatter plots, area charts, bar charts, error bars, box plots, histograms, heatmaps, subplots, multiple-axes, polar charts, and bubble charts.  plotly enables Python users to create beautiful interactive web-based visualizations that can be displayed in Jupyter notebooks, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash.
* **import folium**: it shows how to create a leaflet web map from scratch with python an folium library.that should generate a map.html file. Later you can simply put that html file on a live server and have that map online.
* **plt.rcParams['figure.figsize'] = 10, 12** sets size of the figure 10x12.
* **import warnings**: warnings are displayed using warn() function of warning module.
* **from sklearn.metrics import mean\_squared\_error**:Mean Squared Error. The Mean Squared Error (**MSE**) or Mean Squared Deviation (**MSD**) of an estimator measures the average of error squares i.e. the average squared difference between the estimated values and true value. It is a risk function, corresponding to the expected value of the squared error loss.
* **%matplotlib inline**: to draw graph in the same notebook.
* **df\_India= pd.read\_csv('covid\_19\_India.csv')**: A CSV file is a file with a “.csv” file extension, e.g. “data.csv”, “super\_information.csv”. The “CSV” in this case lets the computer know that the data contained in the file is in “comma separated value” format. **Pandas** is one of those packages and makes importing and analyzing data much easier. **read\_csv** is an important pandas function to read csv files and do operations on it.
* **India\_coord = pd.read\_excel('Indian Coordinates.xlsx')**: We can use the pandas mdodule read\_excel() function to read the excel file into a dataframe object.
* **print(df\_India.info())**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 2450 entries, 0 to 2449

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Sno 2450 non-null int64

1 Date 2450 non-null object

2 Time 2450 non-null object

3 State/UnionTerritory 2450 non-null object

4 ConfirmedIndianNational 2450 non-null object

5 ConfirmedForeignNational 2450 non-null object

6 Cured 2450 non-null int64

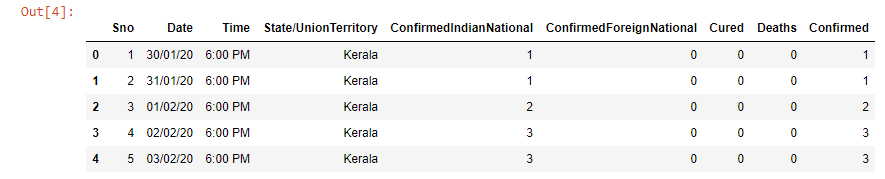
7 Deaths 2450 non-null int64

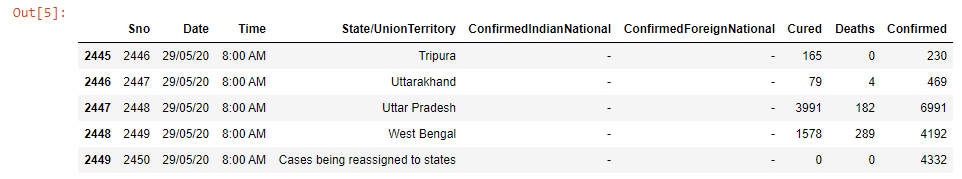
8 Confirmed 2450 non-null int64

dtypes: int64(4), object(5)

memory usage: 172.4+ KB

None

* **df\_India.head():**Pandas **head**() method is used to return top n (5 by default) rows of a data frame or series. 
* **df\_India.tail():**This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.



* **df\_India.dtypes:** This returns a Series with the data type of each column. The result’s index is the original DataFrame’s columns.

Sno int64

Date object

Time object

State/UnionTerritory object

ConfirmedIndianNational object

ConfirmedForeignNational object

Cured int64

Deaths int64

Confirmed int64

dtype: object

* **print(India\_coord.info()):** This message prints information about a DataFrame including the index dtype and column dtypes,non-null values and memory usage.

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 35 entries, 0 to 34

Data columns (total 3 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name of State / UT 35 non-null object

1 Latitude 35 non-null float64

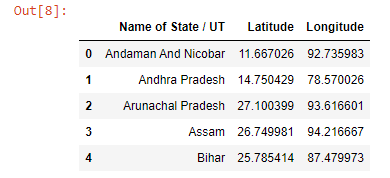
2 Longitude 35 non-null float64

dtypes: float64(2), object(1)

memory usage: 968.0+ bytes

None

* **India\_coord.head():** Pandas **head**() method is used to return top n (5 by default) rows of a data frame or series.



* def replace\_dash\_with\_zeros(inp):

return int(inp.replace("-","0"))

df\_India.drop(['Sno'],axis=1,inplace=True)

df\_India['Date'] = pd.to\_datetime(df\_India['Date'], format = "%d/%m/%y")

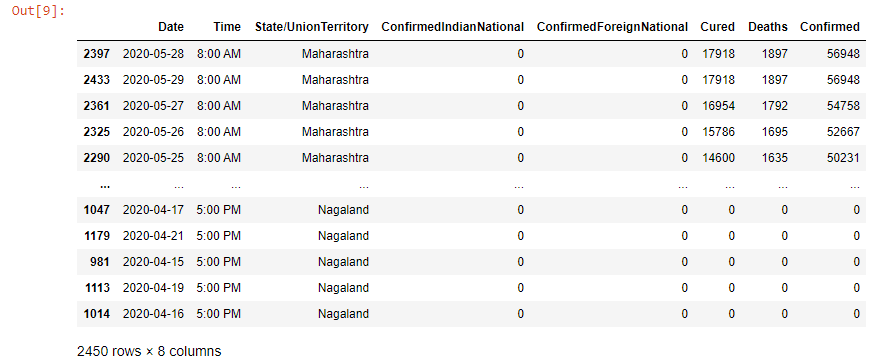
# https://www.stat.berkeley.edu/~s133/dates.html

df\_India['ConfirmedIndianNational']= df\_India['ConfirmedIndianNational'].apply(replace\_dash\_with\_zeros)

df\_India['ConfirmedForeignNational']= df\_India['ConfirmedForeignNational'].apply(replace\_dash\_with\_zeros)

df\_India.sort\_values("Confirmed", ascending = False, inplace = True)

df\_India #prints the DataFrame



* **df\_India.loc[df\_India["ConfirmedForeignNational"] == "-",:]**

checking whether the ConfirmedForeignNational column has value equal to ‘-‘. loc[] is used to access a group of rows and columns by label(s) or a boolean array.



Not a single entry of ’-‘in dataset.

* **list(zip(df\_India.columns,df\_India.dtypes,df\_India.isna().sum())):**The purpose of zip() is to**map the similar index of multiple containers** so that they can be used just using as single entity.

[('Date', dtype('<M8[ns]'), 0),

('Time', dtype('O'), 0),

('State/UnionTerritory', dtype('O'), 0),

('ConfirmedIndianNational', dtype('int64'), 0),

('ConfirmedForeignNational', dtype('int64'), 0),

('Cured', dtype('int64'), 0),

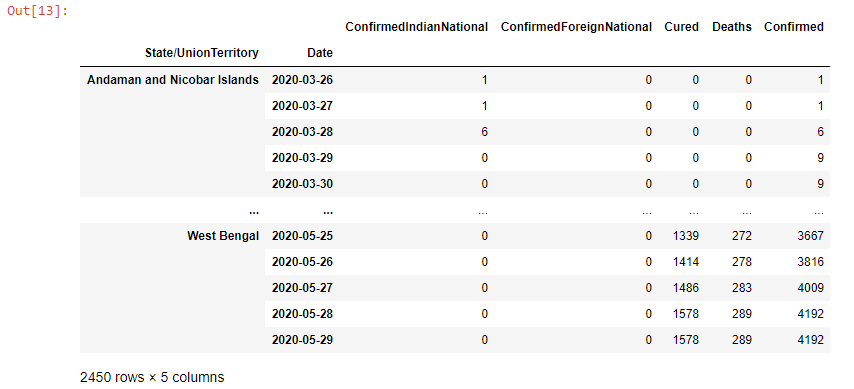
('Deaths', dtype('int64'), 0),

('Confirmed', dtype('int64'), 0)]

* **print(f'We have data available from : {df\_India.Date.min()} to {df\_India.Date.max()}')** To know when the dataset was last updated.

We have data available from : 2020-01-30 00:00:00 to 2020-05-29 00:00:00

* **df\_India.groupby(["State/UnionTerritory", "Date"]).sum()**: groupby() function is used to split the data into groups based on some criteria. Any groupby operation involves one of the following operations on the original object. They are −
  + Splitting the Object
  + Applying a function
  + Combining the results



* **States = df\_India['State/UnionTerritory'].unique().tolist()**

While analyzing the data, many times the user wants to see the unique values in a particular column, which can be done using

Pandas unique() function. Pandas tolist() is used to convert a series to list.

* **States**

Printing all the states.

['Maharashtra',

'Tamil Nadu',

'Delhi',

'Gujarat',

'Rajasthan',

'Madhya Pradesh',

'Uttar Pradesh',

'Cases being reassigned to states',

'West Bengal',

'Andhra Pradesh',

'Bihar',

'Karnataka',

'Punjab',

'Telengana',

'Jammu and Kashmir',

'Odisha',

'Haryana',

'Kerala',

'Assam',

'Uttarakhand',

'Jharkhand',

'Chhattisgarh',

'Chandigarh',

'Himachal Pradesh',

'Tripura',

'Unassigned',

'Goa',

'Ladakh',

'Puducherry',

'Manipur',

'Andaman and Nicobar Islands',

'Meghalaya',

'Nagaland',

'Arunachal Pradesh',

'Dadar Nagar Haveli',

'Sikkim',

'Mizoram']

* **States.remove("Cases being reassigned to states"):** removes "Cases being reassigned to states" from states.
* **States.remove("Unassigned"):** removes “Unassigned” from states.
* **len(States):** len() returns the number of elements in the list.

35

*MERGING DATA FRAMES*

* **df\_final\_India = pd.DataFrame()**

**dates = pd.DataFrame({"Date": pd.date\_range(df\_India.Date.min(),df\_India.Date.max())})**

**for state in States:**

**all\_dates\_df = pd.merge(dates, df\_India.loc[df\_India['State/UnionTerritory'] == state,:], on = "Date", how = "left")**

**all\_dates\_df['State/UnionTerritory'] = state**

**all\_dates\_df = all\_dates\_df.fillna(0)**

**all\_dates\_df['New Cases'] = all\_dates\_df['Confirmed'] -**

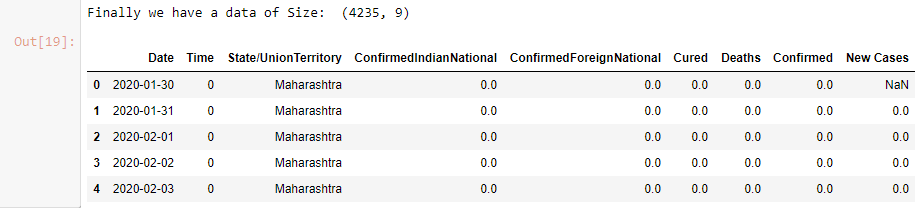
**all\_dates\_df['Confirmed'].shift(1)**

**df\_final\_India = pd.concat([df\_final\_India, all\_dates\_df],axis = 0)**

**print("Finally we have a data of Size: ",df\_final\_India.shape)**

**df\_final\_India.head()**

Here we are creating a new column which will display new cases calculated by subtracting present day cases and previous day cases.



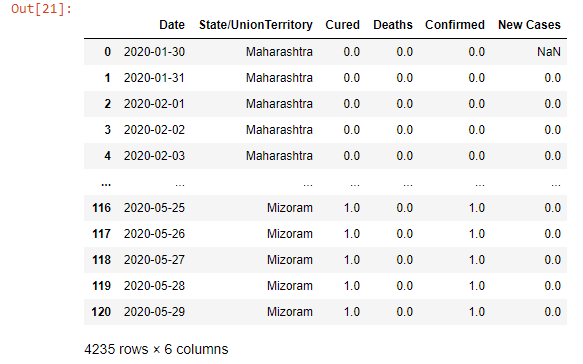
* **del df\_final\_India['Time']**

**del df\_final\_India['ConfirmedIndianNational']**

**del df\_final\_India['ConfirmedForeignNational']**

**df\_final\_India**

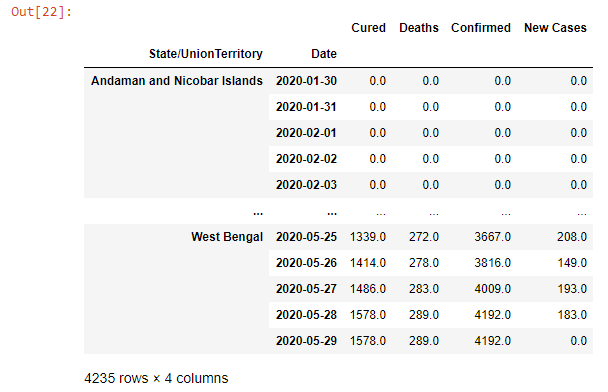
Deleting the columns Time,'ConfirmedIndianNational' and 'ConfirmedForeignNational' and printing the DataFrame df\_final\_India



* **df\_final\_India.groupby(["State/UnionTerritory", "Date"]).sum()**

Pandas dataframe.groupby () function is used to split the data into groups based on some criteria. Pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names.

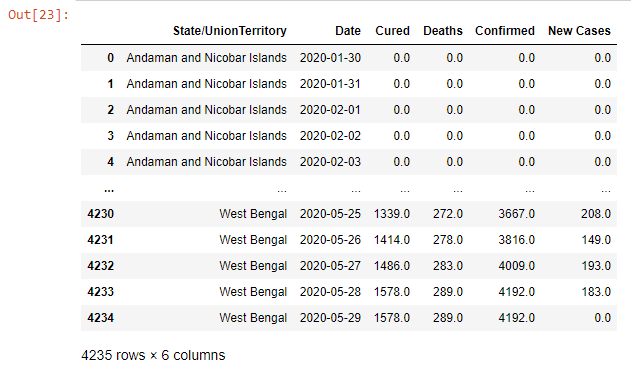
Pandas dataframe.sum () function return the sum of the values for the requested axis. If the input is index axis then it adds all the values in a column and repeats the same for all the columns and returns a series containing the sum of all the values in each column.



* **df\_final\_India=df\_final\_India.groupby(["State/UnionTerritory", "Date"]).sum().reset\_index()**

**df\_final\_India**

Pandas reset\_index() is a method to reset index of a Data Frame.



State Wise Covid19 Status in India

* **def plot\_pie(active,cured,death,title):**

**labels = ['Active','Recovered','Died']**

**sizes = [active,cured,death]**

**color= ['#66b3ff','green','red']**

**explode = []**

**for i in labels:**

**explode.append(0.05)**

**plt.figure(figsize= (15,6))**

**plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=9, explode =explode, colors = color)**

**centre\_circle = plt.Circle((0,0),0.70,fc='white')**

**fig = plt.gcf()**

**fig.gca().add\_artist(centre\_circle)**

**plt.title(title + 'COVID-19 Cases',fontsize = 20)**

**plt.axis('equal')**

**plt.tight\_layout()**

#A function “plot pie” is defined here which takes input argument as no. of active,cured,death cases and title of graph. It plots a pie chart the attributes used are:

|  |  |
| --- | --- |
| x | Array-like. The wedge sixes |
| Labels | List. A sequence of strings providing the labels for each wedge. |
| Colors | A sequence of matplotlibcolorargs through which the piw chart will cycle.  If None, will use the colors in the currently active cycle. |
| Autopct | String, used to label the wedges with their numeric value. The label will be placed inside the wedge. The format string will be fmt%pct. |

* + plt.circle plot with given centerand radius.
  + plt.figure allocates dimensions to a figure.
  + plt.gcf() is primarily used to get the current figure.
  + fig.gca() is used to get the current axes.
* **total\_cases\_india = 0**

**cured\_cases\_india = 0**

**death\_cases\_india = 0**

**active\_cases\_india = 0**

**state\_df = pd.DataFrame()**

**for state in States:**

**one\_state\_df = df\_final\_India.loc[df\_final\_India['State/UnionTerritory'] ==**

**state,:]**

**state\_df = pd.concat([state\_df,pd.DataFrame(one\_state\_df.iloc[-1,:]).T],axis = 0)**

**total\_cases = one\_state\_df['Confirmed'].values[-1]**

**cured = one\_state\_df['Cured'].values[-1]**

**deaths = one\_state\_df['Deaths'].values[-1]**

**active = total\_cases - cured - deaths**

**plot\_pie(active, cured, deaths,state)**

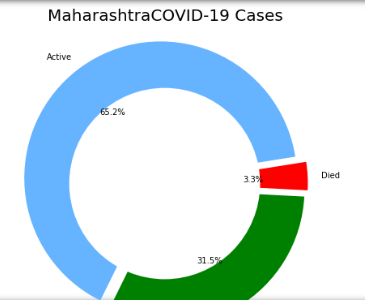
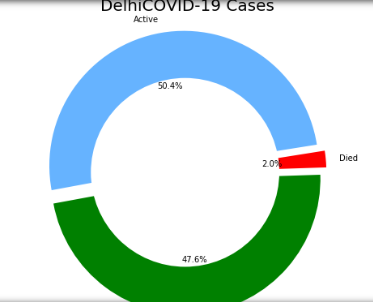
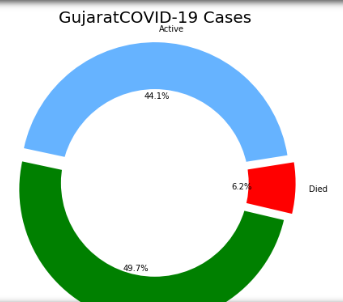
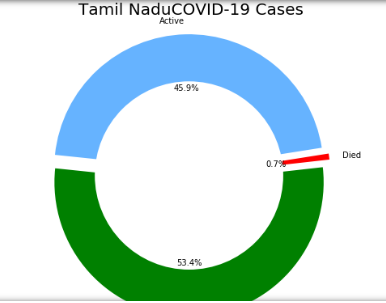
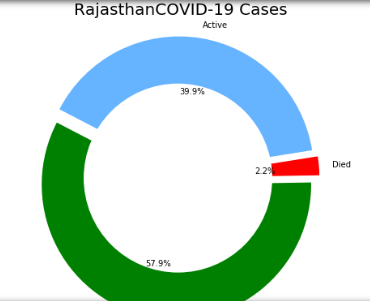
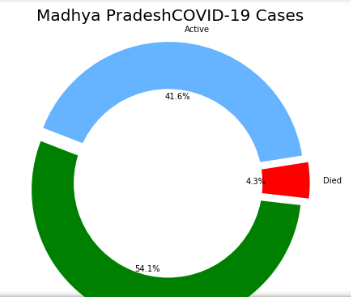
**total\_cases\_india += total\_cases**

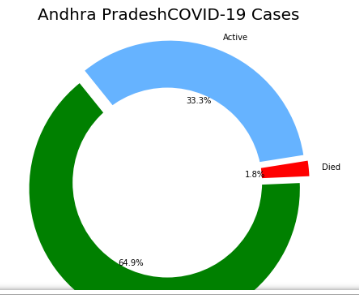
**cured\_cases\_india += cured**

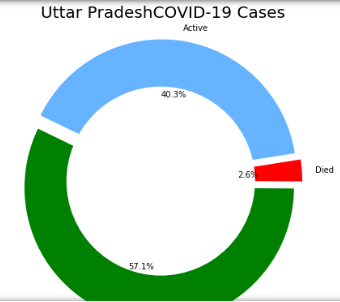
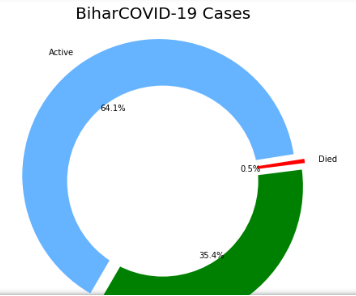
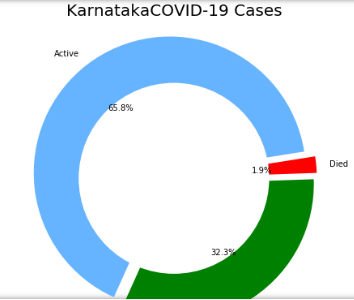
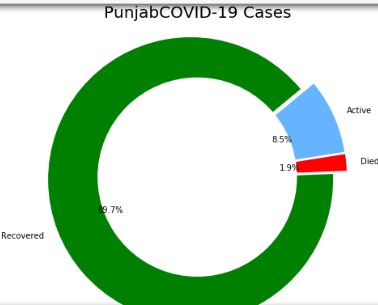
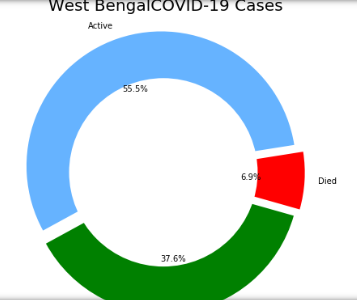
**death\_cases\_india += deaths**

**active\_cases\_india += active**

#Plots pie chart for each state using above function “plot\_pie”

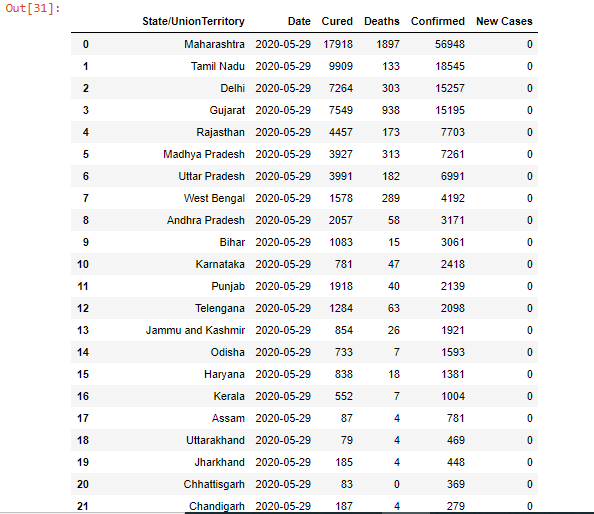
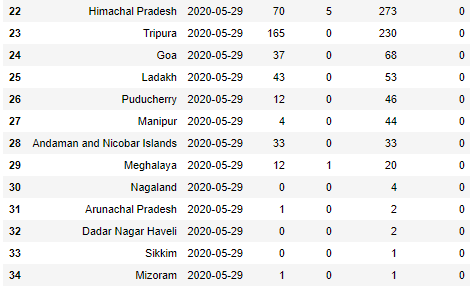




* **state\_df.reset\_index(inplace = True,drop = True)**

**state\_df**



**f, ax = plt.subplots(figsize=(12, 28))**

**data = state\_df[['State/UnionTerritory','Confirmed','Cured','Deaths']]**

**data.sort\_values('Confirmed',ascending=False,inplace=True)**

**sns.set\_color\_codes("pastel")**

**sns.barplot(x="Confirmed", y="State/UnionTerritory", data=data,label="Total", color="red")**

**sns.set\_color\_codes("muted")**

**sns.barplot(x="Cured", y="State/UnionTerritory", data=data, label="Cured", color="green")**

**ax.legend(ncol=5, loc="lower right", frameon=True)**

**ax.set(ylabel="",xlabel="Cases")**

**i = 0**

**for p in ax.patches:**

**x = p.get\_x() + p.get\_width() + 3**

**y = p.get\_y() + p.get\_height()/2**

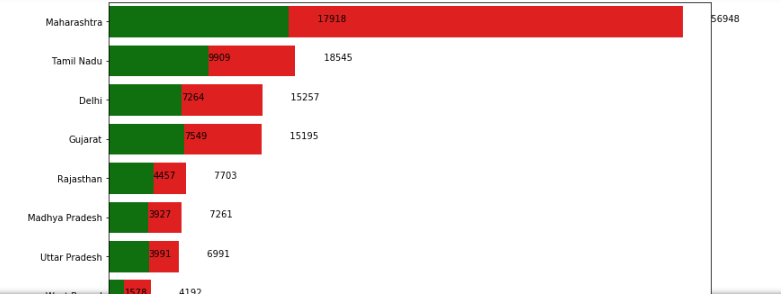
**if i <= len(States):**

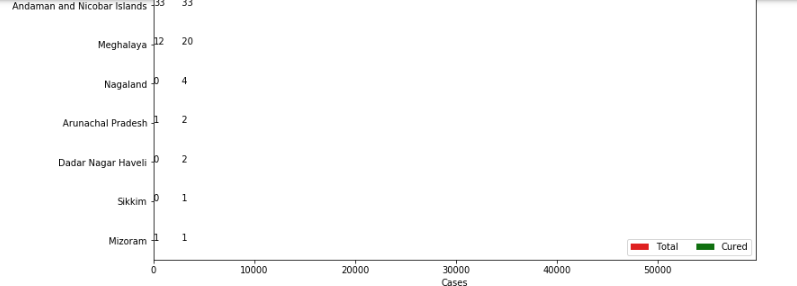
**ax.annotate(" "\*10 + str(int(p.get\_width())), (x, y))**

**else:**

**ax.annotate(int(p.get\_width()), (x, y))**

**i += 1**

#displays no. of people got affected



* **f, ax = plt.subplots(figsize=(12, 28))**

**data = state\_df[['State/UnionTerritory','Confirmed','Cured','Deaths']]**

**data.sort\_values('Confirmed',ascending=False,inplace=True)**

**sns.set\_color\_codes("pastel")**

**sns.barplot(x="Confirmed", y="State/UnionTerritory", data=data,label="Total", color="red")**

**sns.set\_color\_codes("muted")**

**sns.barplot(x="Cured", y="State/UnionTerritory", data=data, label="Cured", color="green")**

**ax.legend(ncol=5, loc="lower right", frameon=True)**

**ax.set(ylabel="",xlabel="Cases")**

**total = total\_cases\_india**

**i = 0**

**for p in ax.patches:**

**percentage = '{:.1f}%'.format(100 \* p.get\_width()/total)**

**x = p.get\_x() + p.get\_width() + 3**

**y = p.get\_y() + p.get\_height()/2**

**if i <= len(States):**

**ax.annotate(" "\*10 + str(percentage), (x, y))**

**else:**

**ax.annotate(percentage, (x, y))**

**i += 1**

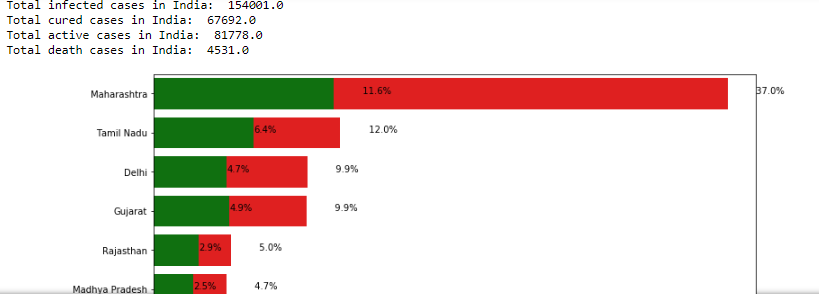
**print("Total infected cases in India: ", total\_cases\_india)**

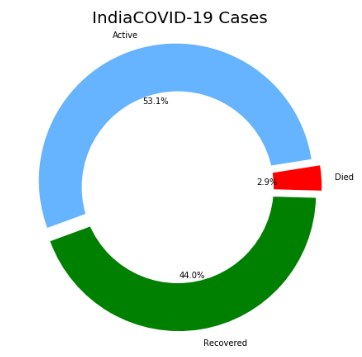
**print("Total cured cases in India: ", cured\_cases\_india)**

**print("Total active cases in India: ", active\_cases\_india)**

**print("Total death cases in India: ", death\_cases\_india)**

#displays no. of people got affected in each state in percentage



* **plot\_pie(active\_cases\_india, cured\_cases\_india, death\_cases\_india, "India")**

VISUALIZING THE SPREADS GEOGRAPHICALLY

* **India\_coord.rename(columns = {"Name of State / UT" : "State/UnionTerritory"},inplace = True)**

#Used to rename the columns

* **set(India\_coord['State/UnionTerritory'].values).symmetric\_difference(set(state\_df['State/UnionTerritory'].values))**

#The Python symmetric\_difference() method returns the symmetric difference of two sets.

{'Andaman And Nicobar ',

'Andaman and Nicobar Islands',

'Arunachal Pradesh',

'Arunachal Pradesh ',

'Assam',

'Assam ',

'Bihar',

'Bihar ',

'Chandigarh',

'Chandigarh ',

'Chhattisgarh',

'Chhattisgarh ',

'Dadar Nagar Haveli',

'Dadra And Nagar Haveli ',

'Goa',

'Goa ',

'Gujarat',

'Himachal Pradesh',

'Himachal Pradesh ',

'Jammu and Kashmir',

'Jharkhand',

'Jharkhand ',

'Ladakh',

'Lakshadweep ',

'Madhya Pradesh',

'Madhya Pradesh ',

'Manipur',

'Manipur ',

'Meghalaya',

'Meghalaya ',

'Mizoram',

'Mizoram ',

'Nagaland',

'Nagaland ',

'Odisha',

'Orissa ',

'Puducherry',

'Puducherry ',

'Sikkim',

'Sikkim ',

'Tripura',

'Tripura ',

'Union Territory of Jammu and Kashmir',

'Union Territory of Ladakh',

'West Bengal',

'West Bengal '}

* **India\_coord['State/UnionTerritory']=India\_coord['State/UnionTerritory'].str.strip)**

**state\_df['State/UnionTerritory'] = state\_df['State/UnionTerritory'].str.strip()**

# .strip function removes the trailing spaces from the text.

* **set(India\_coord['State/UnionTerritory'].values).symmetric\_difference(set(state\_df['State/UnionTerritory'].values))**

#after performing the above operation the following output is obtained.

{'Andaman And Nicobar',

'Andaman and Nicobar Islands',

'Dadar Nagar Haveli',

'Dadra And Nagar Haveli',

'Gujarat',

'Jammu and Kashmir',

'Ladakh',

'Lakshadweep',

'Odisha',

'Orissa',

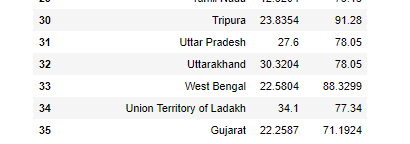
'Union Territory of Jammu and Kashmir',

'Union Territory of Ladakh'}

* **India\_coord.loc[India\_coord.shape[0]] = ['Gujarat','22.2587','71.1924']**

**India\_coord**

**#** Gujrat is not present in coordinate table. So we explicitly adding the coordinates of Gujrat in India\_coord.



* **set(India\_coord['State/UnionTerritory'].values).symmetric\_difference(set(state\_df['State/UnionTerritory'].values))**

{'Andaman And Nicobar',

'Andaman and Nicobar Islands',

'Dadar Nagar Haveli',

'Dadra And Nagar Haveli',

'Jammu and Kashmir',

'Ladakh',

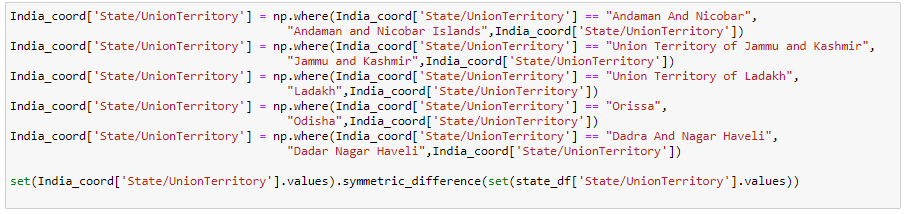
'Lakshadweep',

'Odisha',

'Orissa',

'Union Territory of Jammu and Kashmir',

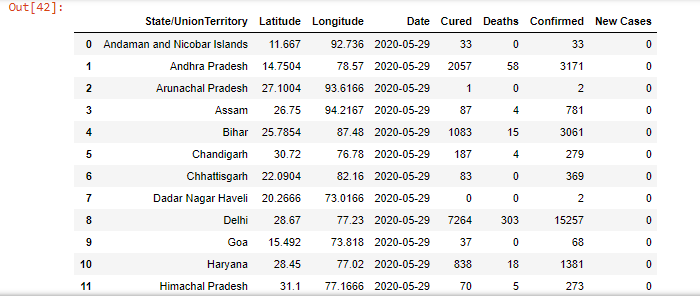
'Union Territory of Ladakh'}

After margining the columns from two different files, and calculating their symmetric difference we find certain differences between the two columns. We removed extra spaces we were left with some differences such as name is available only once or name written differently.

We solve this problem one by one.

* We replace Andaman and Nicobar islands with Andaman and Nicobar. It means that wherever there is Andaman and Nicobar Islands written in state/union territory column replace it with Andaman and Nicobar in the same column.
* Now, similarly we replace Jammu and Kashmir with Union Territory of Jammu and Kashmir.
* We replace Ladakh with Union Territory of Ladakh
* We replace Odisha with Orrisa.
* **df\_full = pd.merge(India\_coord,state\_df,on = 'State/UnionTerritory').reset\_index(drop = True)**

**df\_full**



* **map = folium.Map(location=[20, 70], zoom\_start=4,tiles='Stamenterrain')**

**for lat, lon, value, name in zip(df\_full['Latitude'], df\_full['Longitude'], df\_full['Confirmed'], df\_full['State/UnionTerritory']):**

**folium.CircleMarker([lat, lon], radius=value\*0.0015, popup = ('<strong>State</strong>: ' + str(name).capitalize() + '<br>''<strong>Total Cases</strong>: ' + str(value) + '<br>'),color='red',fill\_color='red',fill\_opacity=0.3 ).add\_to(map)**

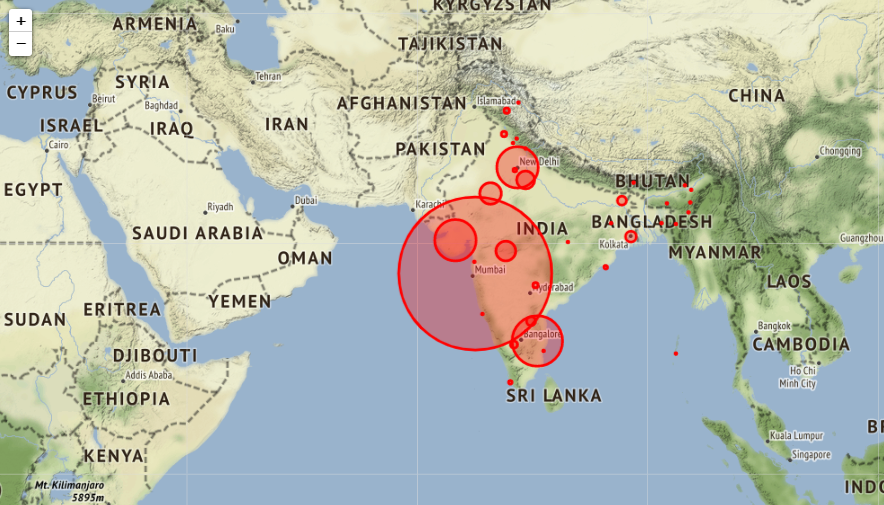
**map**

Folium. makes it easy to visualize data that’s been manipulated in Python on an interactive leaflet map. It enables both the binding of data to a map for choropleth. Visualizations as well as passing rich vector/raster/HTML visualizations as markers on the map.

Map is a variable where we store the code for displaying map. Location means we have to locate this coordinate on the map. Zoom\_start will tell us till what level can we zoom in the map and tiles are different types of maps. Here we use ’stamenterrain’.

The code here says that we want a circlemarker at these these coordinates. The radius of this circle should be 0.015. Popup means (str or folium. Popup, default None) - Input text or visualization for object displayed when clicking, tooltip (str or folium. Tooltip, default None)-Display a text when hovering over the object. So popup will help us to display the text when we point on anything on map. The text will be in capital letters and red in colour. The opacity of the circles is 0.3.

add\_to() :add\_to method is used to add the function to the map or other functions.

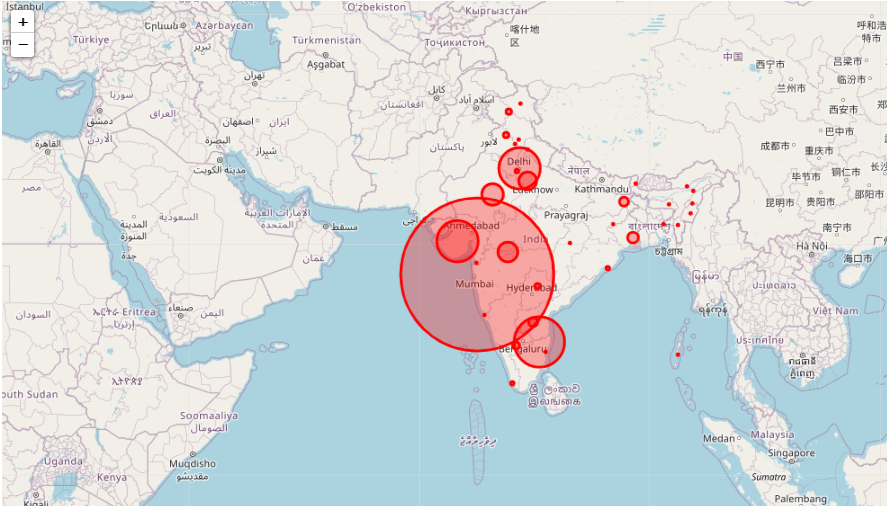


* **map = folium.Map(location=[20, 70], zoom\_start=4,tiles='OpenStreetMap')**

**for lat, lon, value, name in zip(df\_full['Latitude'], df\_full['Longitude'], df\_full['Confirmed'], df\_full['State/UnionTerritory']):**

**folium.CircleMarker([lat, lon], radius=value\*0.0015, popup = ('<strong>State</strong>: ' + str(name).capitalize() + '<br>''<strong>Total Cases</strong>: ' + str(value) + '<br>'),color='red',fill\_color='red',fill\_opacity=0.3 ).add\_to(map)**

**map**

Here we have change the tiles. We have used 'OpenStreetMap', all other attributes remain same the output we obtained is:

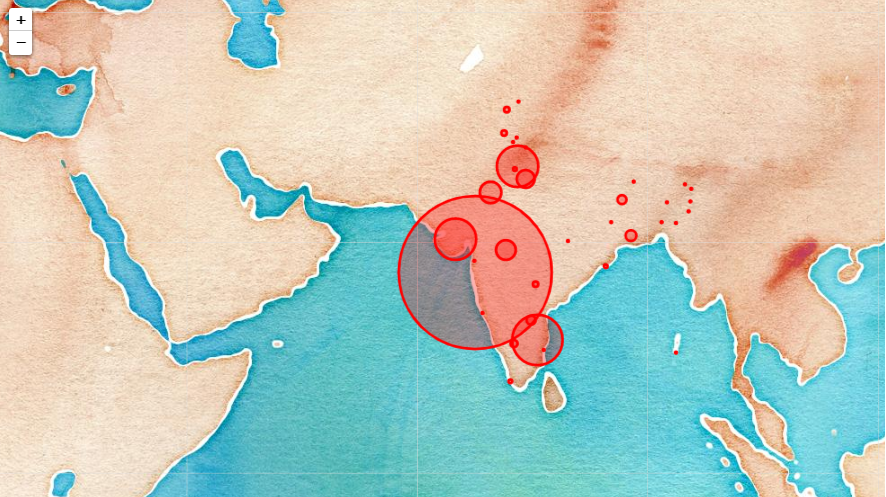
* **map = folium.Map(location=[20, 70], zoom\_start=4,tiles='Stamenwatercolor')**

**for lat, lon, value, name in zip(df\_full['Latitude'], df\_full['Longitude'], df\_full['Confirmed'], df\_full['State/UnionTerritory']):**

**folium.CircleMarker([lat, lon], radius=value\*0.0015, popup = ('<strong>State</strong>: ' + str(name).capitalize() + '<br>''<strong>Total Cases</strong>: ' + str(value) + '<br>'),color='red',fill\_color='red',fill\_opacity=0.3 ).add\_to(map)**

**map**

Here we have change the tiles. We have used 'Stamenwatercolor', all other attributes remain same the output we obtained is:

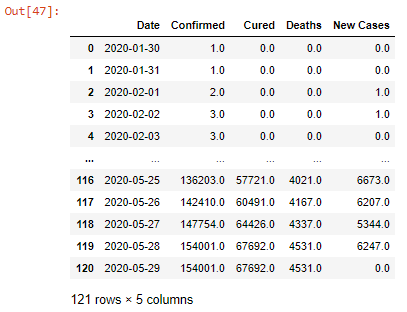


Let’s check the trend of the virus

* **df\_daywise\_India = df\_final\_India.groupby("Date")['Confirmed','Cured','Deaths',"New Cases"].sum().reset\_index()**

**df\_daywise\_India**

#here we are grouping the confirmed, cured, deaths and new cases based on Date.



The plotly python library(plotly.py) is an interactive, open-source, and browser-based graphing library for Python :sparkles: Built on top of plotly.js, plotly.py is a high-level, declarative charting library. plotly.js ships with over 30 chart types, including scientific charts, 3D graphs, statistical charts, SVG maps, financial charts, and more.

The plotly.express module (usually imported as px) contains functions that can create entire figures at once, and is referred to as Plotly Express or PX. Plotly Express is a built-in part of the plotly library, and is the recommended starting point for creating most common figures. Every Plotly Express function uses [graph objects](https://plotly.com/python/graph-objects/) internally and returns a plotly.graph\_objects.Figure instance. Any figure created in a single function call with Plotly Express could be created using graph objects alone, but with between 5 and 100 times more code.

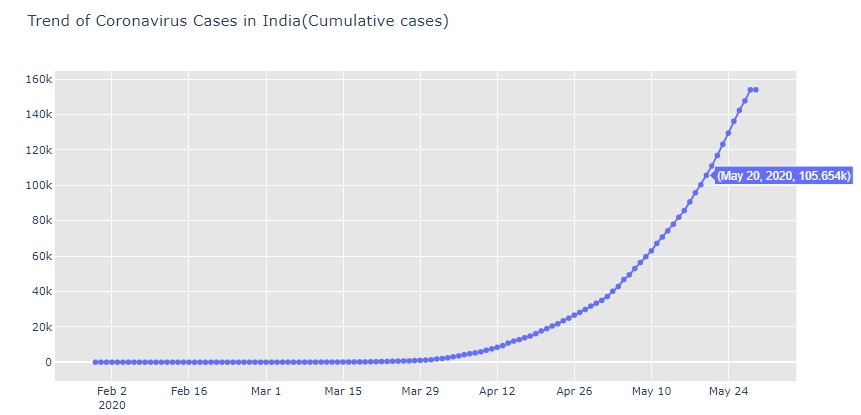
* **fig =go.Figure()**

**fig.add\_trace(go.Scatter(x=df\_daywise\_India['Date'], y=df\_daywise\_India['Confirmed'], mode='lines+markers',name='Total Cases'))**

**fig.update\_layout(title\_text='Trend of Coronavirus Cases in India(Cumulative cases)',plot\_bgcolor='rgb(230, 230, 230)')**

**fig.show()**

#go.Scatter can be used both for plotting points (makers) or lines.

Mode –‘lines+markers’ is used because we want markers plus lines.

* **fig = px.bar(df\_daywise\_India, x="Date", y="New Cases", barmode='group', height=400)**

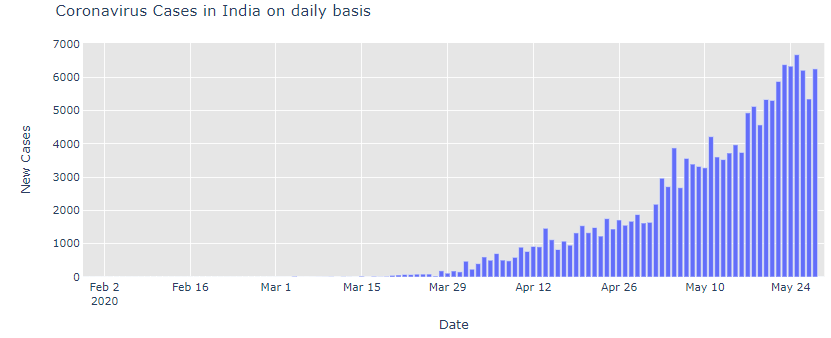
**fig.update\_layout(title\_text='Coronavirus Cases in India on daily basis',plot\_bgcolor='rgb(230, 230, 230)')**

**fig.show()**

#Plotly Express is the easy-to-use, high-level interface to Plotly, which operates on a variety of types of data and produces easy-to-style figures. With px.bar, each row of the DataFrame is represented as a rectangular mark.

Graph objects support higher-level convenience functions for making updates to already constructed figures ((title\_text='Coronavirus Cases in India on daily basis',plot\_bgcolor='rgb(230, 230, 230)'))

To display a figure using the renderers framework, you call the . show() method on a graph object figure, or pass the figure to the plotly.

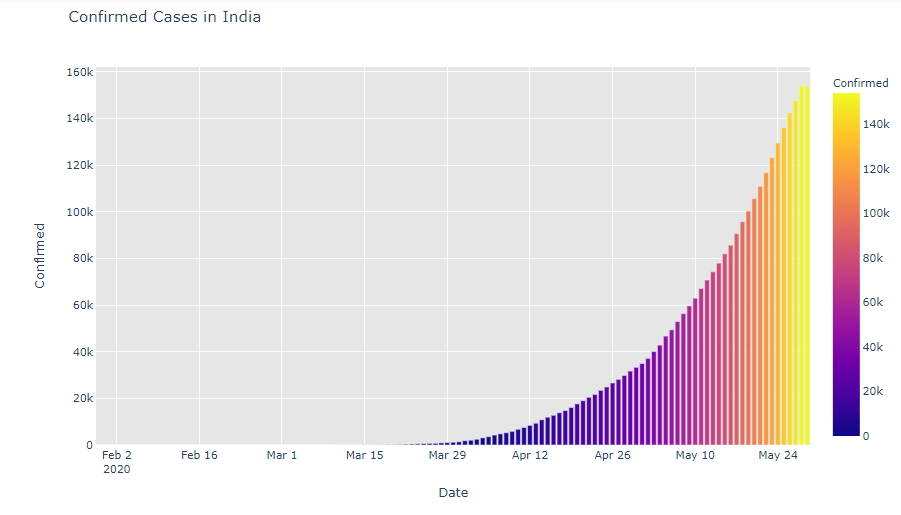


* **fig = px.bar(df\_daywise\_India, x="Date", y="Confirmed", color='Confirmed', orientation='v', height=600, title='Confirmed Cases in India', color\_discrete\_sequence = px.colors.cyclical.IceFire)**

**fig.update\_layout(plot\_bgcolor='rgb(230, 230, 230)')**

**fig.show()**

#Here we have changed the color scale, we have used discrete color sequence.

Lighter is the color more the number of cases.

* **fig = px.line(x=df\_daywise\_India['Date'], y=df\_daywise\_India['New Cases'], labels = {'x': "Dates",'y': "Counts"})**

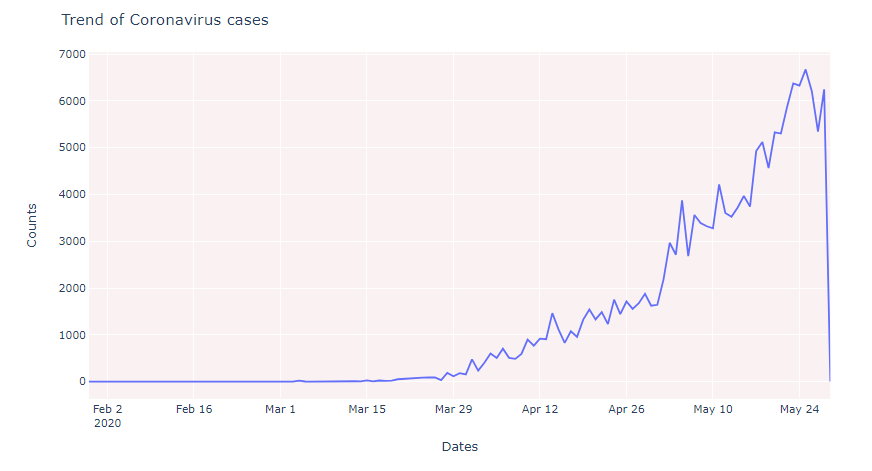
**fig.update\_layout( showlegend=False,title\_text="Trend of Coronavirus cases")**

**fig.update\_layout(plot\_bgcolor='rgb(250, 242, 242)')**

**fig.show()**

**#** With px.line, each data point is represented as a vertex (which location is given by the x and y columns) of a **polyline mark** in 2D space.

Data labels are the names of the data points that are displayed on the x-axis and y-axis of a chart.



Forecasting Using fbprophet

* **from fbprophet import Prophet**

We import FBPROPHET library to predict the future analysis on the corona virus outbreak in India.

FBPROPHET library- Time series forecasting is an important task for effective and efficient planning in many fields like finance, weather and energy. Facebook researchers recently released FBProphet, a time series forecast supporting both Python and R. FBProphet provides a decomposition re model that is extendable and configurable with interpretable par Prophet follows the sklearn model API.

* **df = df\_daywise\_India.iloc[:-1,]**

**df\_train = df.loc[df['Date']<= "2020-05-23",:]**

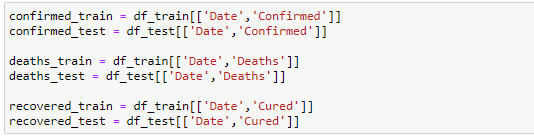
**df\_test = df.loc[df['Date'] > "2020-05-23",:]**

DataFrame.iloc[] is purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

We are dividing the data into two parts train and test data.Train data is stored in df\_train and test data is present is df\_test.

Here till 23rd may we are storing the data in df\_train and after 23rd may we are storing the data in df\_test.



Here we have declared some variable that contains different data-

* **confirmed\_train = df\_train[['Date','Confirmed']]**

This contains the data of Date and Confirmed column of df\_train dataset

* **confirmed\_test = df\_test[['Date','Confirmed']]**

This contains the data of Date and Confirmed column of df\_test dataset

* **deaths\_train = df\_train[['Date','Deaths']]**

This contains the data of Date and Deaths column of df\_train dataset

* **deaths\_test = df\_test[['Date','Deaths']]**

This contains the data of Date and Deaths column of df\_test dataset

* **recovered\_train = df\_train[['Date','Cured']]**

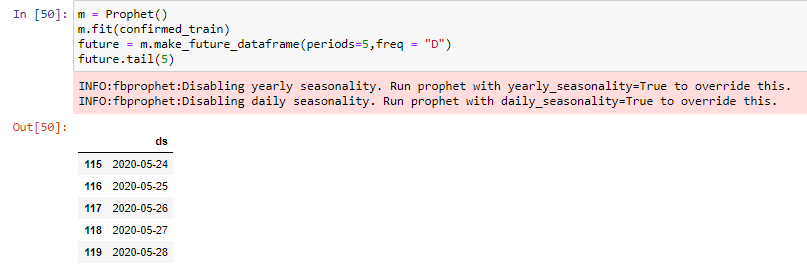
This contains the data of Date and Cured column of df\_train dataset

* **recovered\_test = df\_test[['Date','Cured']]**

This contains the data of Date and Cured column of df\_test dataset

* **confirmed\_train.columns = ['ds','y']**

**confirmed\_train.tail()**

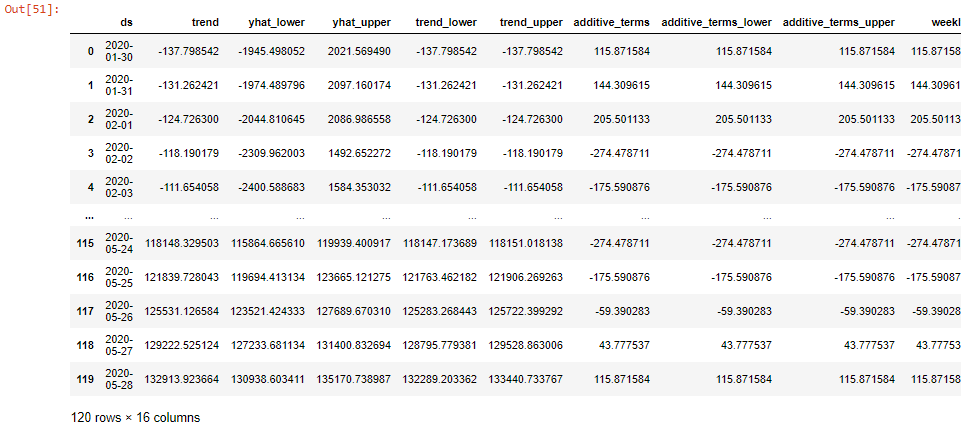
Here we are checking the confirmed cases

We instantiate a new Prophet object "m" of the instance library Prophet from the module FbProphet, using the command: m=Prophet().

We fit the model by instantiating the m object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds. The fit is used to train the module to use data in confirmed\_train, using the command: m.fit(confirmed\_train).

The module Prophet has a method called make\_future\_dataframe() for and future predictions. Predictions are then made on a dataframe with a column ds containing the dates for which a prediction is to be made. You can get a suitable dataframe that extends into the future a specified number of days using the helper method Prophet.make\_future\_dataframe. By default it will also include the dates from the history, so we will see the model fit as well.

* **forecast = m.predict(future)**

**forecast**

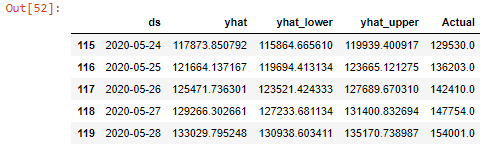
The predict method will assign each row in future a predicted value which it names yhat. If you pass in historical dates, it will provide an in-sample fit. The forecast object here is a new dataframe that includes a column yhat with the forecast, as well as columns for components and uncertainty intervals.

* **result\_df = forecast[['ds', 'yhat', 'yhat\_lower', 'yhat\_upper']].tail(5)**

**result\_df['Actual'] = confirmed\_test['Confirmed']**

**result\_df**

The actual data that we need are in the column ds. yhat, vhat\_lower,yhat\_upper of the whole dataframe ‘forecast’. Thus, we create a new dataframe named 'result\_df’ that contain the above columns.



* **trace0 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['Actual'],**

**mode = 'lines+markers',**

**name='Actuals',**

**line = dict(color = '#dd0000', shape = 'linear'),**

**opacity = 0.3,**

**connectgaps=True)**

**trace1 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['yhat'],**

**name='Predicted',**

**mode = 'lines+markers',**

**marker = dict(**

**size = 10,**

**color = '#44dd00'),**

**opacity = 0.3**

**)**

**data = [trace0, trace1]**

**layout = go.Layout(**

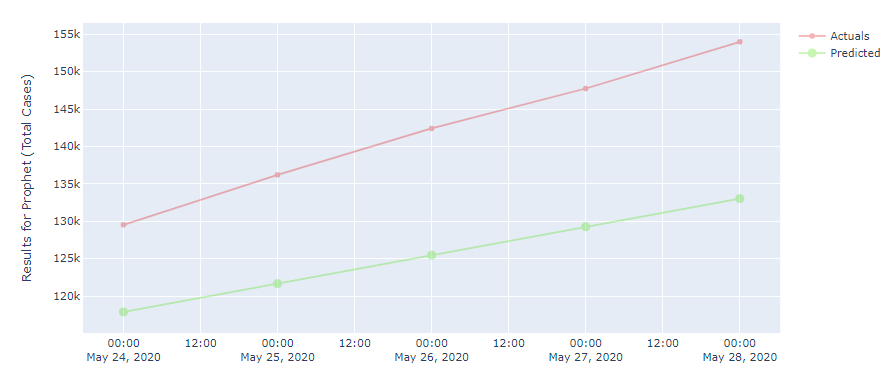
**yaxis=dict(**

**title="Results for Prophet (Total Cases)"**

**)**

**)**

**fig = go.Figure(data=data, layout=layout)**

**fig.show()**

In the above graph we have plotted the scatter plot using go.Scatter() method.

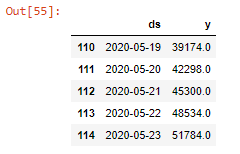
The trace0 shown in Red is the actual no of confirmed cases over the span from 24th

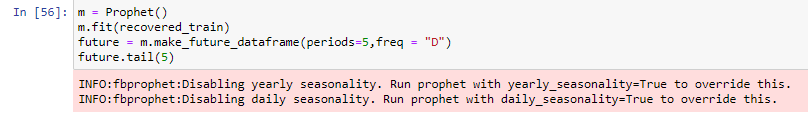
May 2020 to 28th May 2020. The trace1 shown in green is the predicted number of confirmed cases over the span from 24th May 2020 to 28th May 2020.

We can observe from this graph the number of actual confirmed cases on 28 May,2020 is 154.001k and number of predicted confirmed cases on 28 May,2020 is 133.0298k. There is a slight variation this is because of some difference in the module.

* **recovered\_train.columns = ['ds','y']**

**recovered\_train.tail()**

Here we are checking the recovered cases.



We instantiate a new Prophet object "m" of the instance library Prophet from the module FbProphet, using the command: m=Prophet().

We fit the model by instantiating the m object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds. The fit is used to train the module to use data in recovered\_train, using the command: m.fit(recovered\_train).

The module Prophet has a method called make\_future\_dataframe() for and future predictions. Predictions are then made on a dataframe with a column ds containing the dates for which a prediction is to be made. You can get a suitable dataframe that extends into the future a specified number of days using the helper method. Prophet.make\_future\_dataframe. By default it will also include the dates from the history, so we will see the model fit as well.

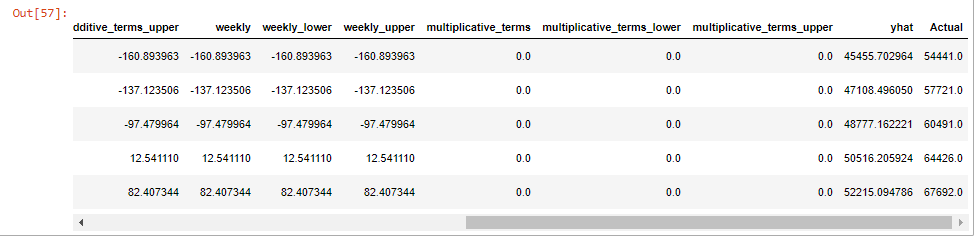
* **forecast = m.predict(future)**

**forecast[['ds', 'yhat', 'yhat\_lower', 'yhat\_upper']].tail(5)**

**result\_df = forecast.tail(5)**

**result\_df['Actual'] = recovered\_test['Cured']**

**result\_df**



The actual data that we need are in the column ds. yhat, vhat\_lower,yhat\_upper of the whole dataframe ‘forecast’. Thus, we create a new dataframe named 'result\_df’ that contain the above columns.

* **trace0 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['Actual'],**

**mode = 'lines+markers',**

**name='Actuals',**

**line = dict(color = '#dd0000', shape = 'linear'),**

**opacity = 0.3,**

**connectgaps=True**

**)**

**trace1 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['yhat'],**

**name='Predicted',**

**mode = 'lines+markers',**

**marker = dict(**

**size = 10,**

**color = '#44dd00'),**

**opacity = 0.3**

**)**

**data = [trace0, trace1]**

**layout = go.Layout(**

**yaxis=dict(**

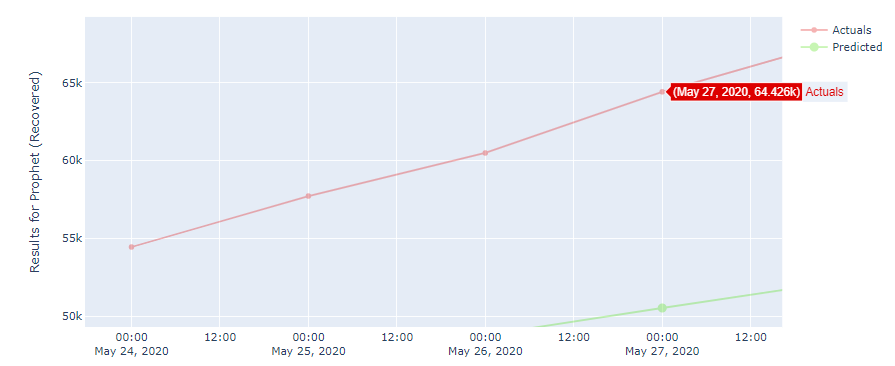
**title="Results for Prophet (Recovered)"**

**)**

**)**

**fig = go.Figure(data=data, layout=layout)**

**fig.show()**



In the above graph we have plotted the scatter plot using go.Scatter() method.

The trace0 shown in Red is the actual no of recovered cases over the span from 24th

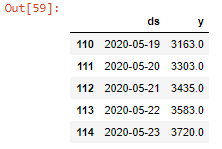
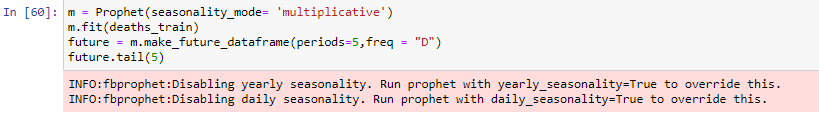
May 2020 to 27th May 2020. The trace1 shown in green is the predicted number of recovered cases over the span from 24th May 2020 to 27th May 2020.

We can observe from this graph the number of actual recovered cases on 27 May,2020 is 64.426k and number of predicted recovered cases on 27 May,2020 is 50.51621k. There is a slight variation this is because of some difference in the module.

* **deaths\_train.columns = ['ds','y']**

**deaths\_train.tail()**

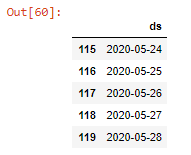
Here we are checking the death cases.



We instantiate a new Prophet object "m" of the instance library Prophet from the module FbProphet, using the command: m=Prophet().

We fit the model by instantiating the m object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds. The fit is used to train the module to use data in deaths\_train, using the command: m.fit(deaths\_train).

The module Prophet has a method called make\_future\_dataframe() for and future predictions. Predictions are then made on a dataframe with a column ds containing the dates for which a prediction is to be made. You can get a suitable dataframe that extends into the future a specified number of days using the helper method. Prophet.make\_future\_dataframe. By default it will also include the dates from the history, so we will see the model fit as well.



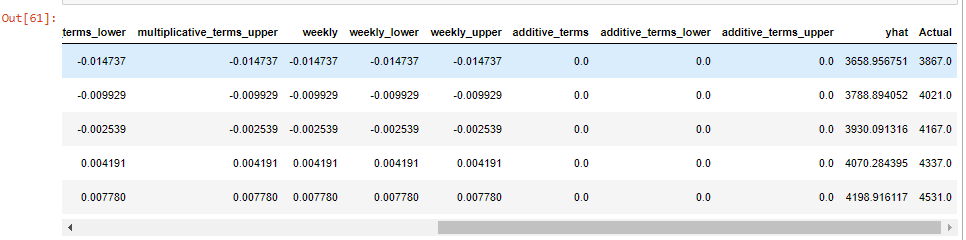
* **forecast = m.predict(future)**

**forecast[['ds', 'yhat', 'yhat\_lower', 'yhat\_upper']].tail(5)**

**result\_df = forecast.tail(5)**

**result\_df['Actual'] = deaths\_test['Deaths']**

**result\_df**

The predict method will assign each row in future a predicted value which it names yhat. If you pass in historical dates, it will provide an in-sample fit. The forecast object here is a new dataframe that includes a column yhat with the forecast, as well as columns for components and uncertainty intervals.

The actual data that we need are in the column ds. yhat, vhat\_lower,yhat\_upper of the whole dataframe ‘forecast’. Thus, we create a new dataframe named 'result\_df’ that contain the above columns.

* **trace0 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['Actual'],**

**mode = 'lines+markers',**

**name='Actuals',**

**line = dict(color = '#dd0000', shape = 'linear'),**

**opacity = 0.3,**

**connectgaps=True**

**)**

**trace1 = go.Scatter(**

**x = result\_df['ds'],**

**y = result\_df['yhat'],**

**name='Predicted',**

**mode = 'lines+markers',**

**marker = dict(**

**size = 10,**

**color = '#44dd00'),**

**opacity = 0.3**

**)**

**data = [trace0, trace1]**

**layout = go.Layout(**

**yaxis=dict(**

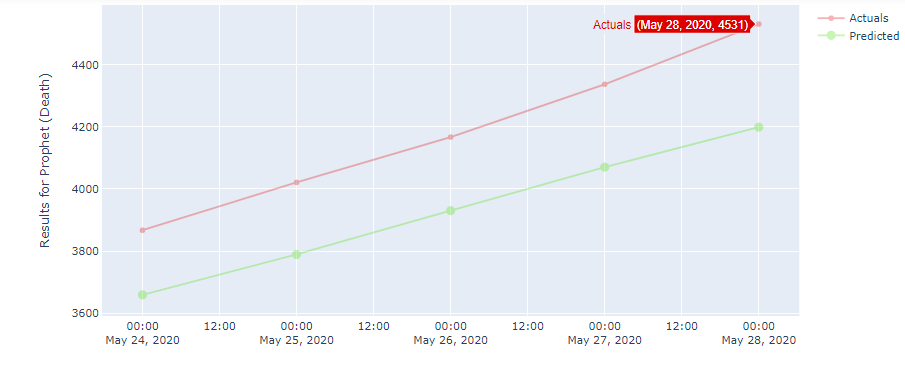
**title="Results for Prophet (Death)"**

**)**

**)**

**fig = go.Figure(data=data, layout=layout)**

**fig.show()**



In the above graph we have plotted the scatter plot using go.Scatter() method.

The trace0 shown in Red is the actual no of Death cases over the span from 24th

May 2020 to 28th May 2020. The trace1 shown in green is the predicted number of Death cases over the span from 24th May 2020 to 28th May 2020.

We can observe from this graph the number of actual Death cases on 28 May, 2020 is 4531 and number of predicted Death cases on 28 May, 2020 is 4198.916. There is a slight variation this is because of some difference in the module.

CONCLUSION

In our project, we import two files

1. Covid\_19\_india.csv

2. Indian Coordinates.xls

* We then prepare our data for analysis with the help of data cleaning by removing or modifying data that is incorrect, incomplete, irrelevant, duplicated, or improperly formatted.
* Then we perform the merging operation wherein we join the two dataframes. In this way we get all the information about a particular entity common in both the two datasets.
* Each state's cases are analyzed using a pie chart. Also, overall country's cases are analyzed.
  1. States like Maharashtra, Gujarat, and Madhya Pradesh West Bengal have death rate nearly 5 percent.
  2. States like Tamil Nadu Madhya Pradesh, Uttar Pradesh, Rajasthan, Andhra Pradesh, Telangana, Haryana, Kerela have recovered more than 50 percent.
  3. Punjab has almost 90 percent cases recovered.
  4. No death cases have been recorded in Tripura, Goa, Ladakh, Pondicherry and Manipura.
  5. Andaman and Nicobar Islands have all the cases recovered.
  6. Dadra and Nagar Haveli, Sikkim, Nagaland has all the cases active.
  7. Maharashtra has highest number of cases recorded.
  8. Overall in India 53 % are active, 3% have died due the virus, 44% have recovered.
* While merging two datasets, some data gets repeated or may be missing, data cleaning is an important step to obtain appropriate data for analysis
* The maps plotted become a clear visual indicator of COVID hotspots.
* From this we can interpret that:

1. First table lists the number of confirmed, cured, deaths and number of new cases per day. From this we can conclude that the number of COVID cases soared from mid-April.
2. The first graph shows that the curve was flat until mid-March and then it grew exponentially thereafter.
3. The bar plots in the second graph also shows that the new cases rise exponentially by March ending.
4. On the similar lines from the third graph we understand that curve was flat until April start and it grew exponentially thereafter.
5. This graph clearly states that India witnessed it 1st COVID case on March 2nd 2020. Till May 1st the graph was gradually rising and on 3rd May it witnessed a steep rise in cases nearly doubling the previous day’s cases. On 25th May 2020 India witnessed maximum number of fresh cases recorded till date.

The Fbprophet library used the predicting and forecasting the Corona Virus Spread has helped in estimating the rough figures of no of cases of Corona. These prediction done for 5 days are plotted in Graphs. The following conclusions can be drawn:

1. Considering the data set used being small, the prediction ate quiet correct.

2. In all three cases, the actual no of cases outnumbers the predicted no of cases.

3. The slope of actual cases curve is more than that of the predicted cases, stating the increasing coefficient is higher in the actual case scenarios.

4. The area between both the plots in the graph tends to become divergent, depicting that the future prediction in further days will become more inaccurate.