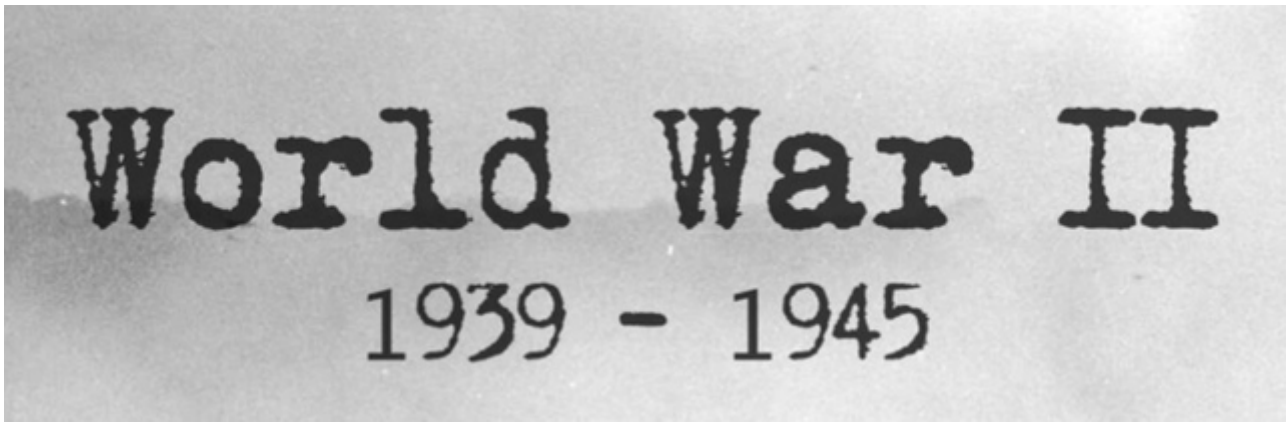
INTRODUCTION



- Hi guys, I hope you are doing fine.
- In this kernel, we use multipla data sources that are **aerial bombing operations** and **weather conditions in world war 2**.
- After this point, I will use acronym ww2 for world war 2.
- We will start with **data description and cleaning**, then we will visualize our data to understand better. These processes can be called **EDA (Exploratory Data Analysis)**.
- After that, we will focus on **time series prediction** to predict when bombing operations are done.
- For time series prediction, we will use **ARIMA** method that will be a tutorial.

Content:

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns # visualization library
import matplotlib.pyplot as plt # visualization library
import plotly.plotly as py # visualization library
from plotly.offline import init_notebook_mode, iplot # plotly offline mode
init_notebook_mode(connected=True)
import plotly.graph_objs as go # plotly graphical object

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list the files
import os
print(os.listdir("../input"))
# import warnings library
import warnings
# ignore filters
warnings.filterwarnings("ignore") # if there is a warning after some codes, this will avoi
plt.style.use('ggplot') # style of plots. ggplot is one of the most used style, I also lik
# Any results you write to the current directory are saved as output.
```

```
['world-war-ii', 'weatherww2']
```

Load the Data

- As I mentioned at introduction, we use multiple data sources.
  - Aerial Bombing Operations in WW2
    - Shortly, this data includes bombing operations. For example, USA who use ponte olivo airfield bomb Germany (Berlin) with A36 air craft in 1945.
  - Wether Conditions in WW2
    - Shortly, weather conditions during ww2. For example, according to george town weather station, average temperature is 23.88 in 1/7/1942.
    - This data set has 2 subset in it. First one includes weather station locations like country, latitude and longitude.
    - Second one includes measured min, max and mean temperatures from weather stations.

```
# bombing data
aerial = pd.read_csv("../input/world-war-ii/operations.csv")
# first weather data that includes locations like country, latitude and longitude.
weather_station_location = pd.read_csv("../input/weatherww2/Weather Station Locations.csv"
# Second weather data that includes measured min, max and mean temperatures
weather = pd.read_csv("../input/weatherww2/Summary of Weather.csv")
```

Data Description

I only explain data features that we will use in this kernel.

- **Aerial bombing Data description:**
    - Mission Date: Date of mission
    - Theater of Operations: Region in which active military operations are in progress; "the army was in the field awaiting action"; Example: "he served in the Vietnam theater for three years"
    - Country: Country that makes mission or operation like USA
    - Air Force: Name or id of air force unity like 5AF
    - Aircraft Series: Model or type of aircraft like B24
    - Callsign: Before bomb attack, message, code, announcement, or tune that is broadcast by radio.
    - Takeoff Base: Takeoff airport name like Ponte Olivo Airfield
    - Takeoff Location: takeoff region Sicily
    - Takeoff Latitude: Latitude of takeoff region
    - Takeoff Longitude: Longitude of takeoff region
    - Target Country: Target country like Germany
    - Target City: Target city like Berlin
    - Target Type: Type of target like city area
    - Target Industry: Target industy like town or urban
    - Target Priority: Target priority like 1 (most)
    - Target Latitude: Latitude of target
    - Target Longitude: Longitude of target
- **Weather Condition data description:**
    - Weather station location:
        - WBAN: Weather station number
        - NAME: weather station name
        - STATE/COUNTRY ID: acronym of countries
        - Latitude: Latitude of weather station
        - Longitude: Longitude of weather station
    - Weather:

- STA: eather station number (WBAN)
- Date: Date of temperature measurement
- MeanTemp: Mean temperature

Data Cleaning

- Aerial Bombing data includes a lot of NaN value. Instead of usign them, I drop some NaN values. It does not only remove the uncertainty but it also easa visualization process.
  - Drop countries that are NaN
  - Drop if target longitude is NaN
  - Drop if takeoff longitude is NaN
  - Drop unused features
- Weather Condition data does not need any cleaning. According to exploratory data analysis and visualization, we will choose certain location to examine deeper. However, lets put our data variables what we use only.

```
# drop countries that are NaN
aerial = aerial[pd.isna(aerial.Country)==False]
# drop if target longitude is NaN
aerial = aerial[pd.isna(aerial['Target Longitude'])==False]
# Drop if takeoff longitude is NaN
aerial = aerial[pd.isna(aerial['Takeoff Longitude'])==False]
# drop unused features
drop_list = ['Mission ID','Unit ID','Target ID','Altitude (Hundreds of Feet)','Airborne Ai
             'Attacking Aircraft', 'Bombing Aircraft', 'Aircraft Returned',
             'Aircraft Failed', 'Aircraft Damaged', 'Aircraft Lost',
             'High Explosives', 'High Explosives Type','Mission Type',
             'High Explosives Weight (Pounds)', 'High Explosives Weight (Tons)',
             'Incendiary Devices', 'Incendiary Devices Type',
             'Incendiary Devices Weight (Pounds)',
             'Incendiary Devices Weight (Tons)', 'Fragmentation Devices',
             'Fragmentation Devices Type', 'Fragmentation Devices Weight (Pounds)',
             'Fragmentation Devices Weight (Tons)', 'Total Weight (Pounds)',
             'Total Weight (Tons)', 'Time Over Target', 'Bomb Damage Assessment','Source I
aerial.drop(drop_list, axis=1,inplace = True)
aerial = aerial[ aerial.iloc[:,8]!="4248"] # drop this takeoff latitude
aerial = aerial[ aerial.iloc[:,9]!=1355]   # drop this takeoff longitude
```

```
aerial.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2555 entries, 0 to 178080
Data columns (total 17 columns):
Mission Date             2555 non-null object
Theater of Operations    2555 non-null object
Country                  2555 non-null object
```

```
Air Force               2505 non-null object
Aircraft Series         2528 non-null object
Callsign                10 non-null object
Takeoff Base            2555 non-null object
Takeoff Location        2555 non-null object
Takeoff Latitude        2555 non-null object
Takeoff Longitude       2555 non-null float64
Target Country          2499 non-null object
Target City             2552 non-null object
Target Type             602 non-null object
Target Industry         81 non-null object
Target Priority         230 non-null object
Target Latitude         2555 non-null float64
Target Longitude        2555 non-null float64
dtypes: float64(3), object(14)
memory usage: 359.3+ KB
```

```
# what we will use only
weather_station_location = weather_station_location.loc[:,["WBAN","NAME","STATE/COUNTRY ID
weather_station_location.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 161 entries, 0 to 160
Data columns (total 5 columns):
WBAN                161 non-null int64
NAME                161 non-null object
STATE/COUNTRY ID    161 non-null object
Latitude            161 non-null float64
Longitude           161 non-null float64
dtypes: float64(2), int64(1), object(2)
memory usage: 6.4+ KB
```

```
# what we will use only
weather = weather.loc[:,["STA","Date","MeanTemp"] ]
weather.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 119040 entries, 0 to 119039
Data columns (total 3 columns):
STA         119040 non-null int64
Date        119040 non-null object
MeanTemp    119040 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 2.7+ MB
```

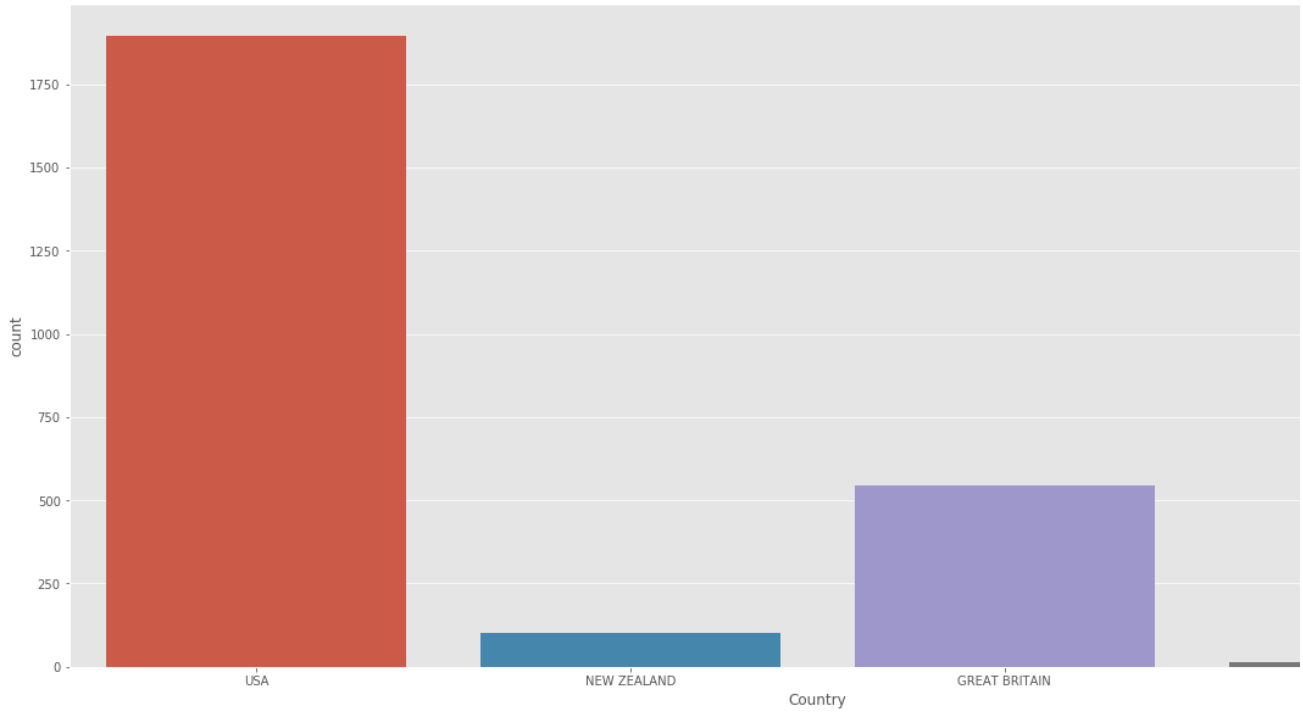
Data Visualization

- Lets start with basics of visualization that is understanding data.
  - How many country which attacks
  - Top target countries
  - Top 10 aircraft series

- Takeoff base locations (Attacjk countries)
- Target locations (If you do not understand methods of pyplot look at my pyplot tutorial: https://www.kaggle.com/kanncaa1/plotly-tutorial-for-beginners)
- Bombing paths
- Theater of Operations
- Weather station locations

```
# country
print(aerial['Country'].value_counts())
plt.figure(figsize=(22,10))
sns.countplot(aerial['Country'])
plt.show()
```

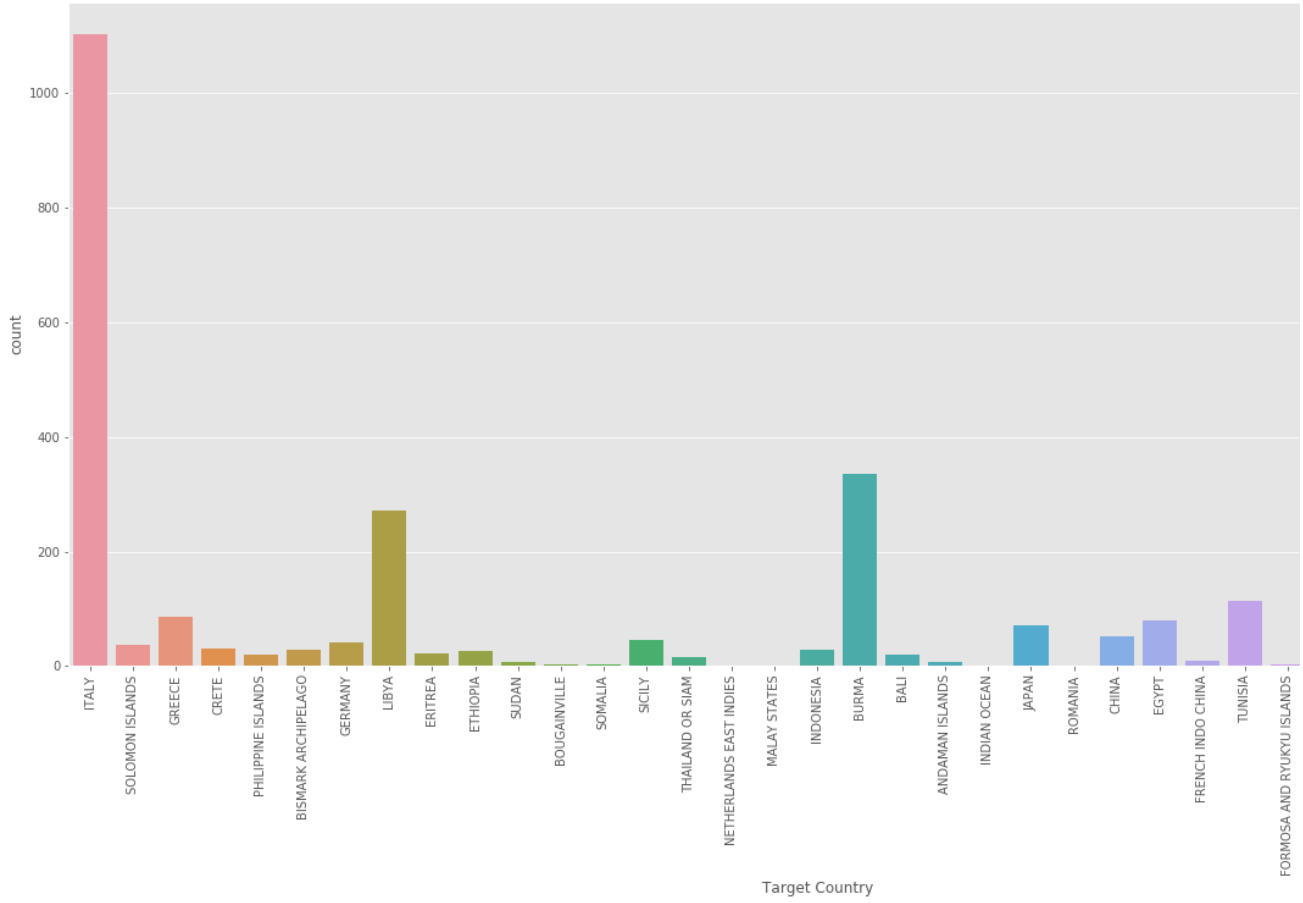
```
USA              1895
GREAT BRITAIN     544
NEW ZEALAND       102
SOUTH AFRICA       14
Name: Country, dtype: int64
```




```
# Top target countries
print(aerial['Target Country'].value_counts()[:10])
```

```
plt.figure(figsize=(22,10))
sns.countplot(aerial['Target Country'])
plt.xticks(rotation=90)
plt.show()
```

```
ITALY      1104
BURMA       335
LIBYA       272
TUNISIA     113
GREECE       87
EGYPT        80
JAPAN        71
CHINA        52
SICILY       46
GERMANY      41
Name: Target Country, dtype: int64
```



```
# Aircraft Series
data = aerial['Aircraft Series'].value_counts()
print(data[:10])
data = [go.Bar(
            x=data[:10].index,
            y=data[:10].values,
            hoverinfo = 'text',
            marker = dict(color = 'rgba(177, 14, 22, 0.5)',
                          line=dict(color='rgb(0,0,0)',width=1.5)),
    )]

layout = dict(
    title = 'Aircraft Series',
)
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

```
A36             990
B25             416
```

- **Most used air craft: A36**
  - 
- Now lets visualize take off bases of countries who attack
  - In plot below, blue color draw the attention, it is USA and red color is Great Britain

```
aerial.head()
```

	Mission Date	Theater of Operations	Country	Air Force	Aircraft Series	Callsign	Takeoff Base	Takeoff Location	Tal Lat:
0	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.1:
2	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.1:
3	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.1:
8	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.1:
9	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.1:

```
# ATTACK
aerial["color"] = ""
aerial.color[aerial.Country == "USA"] = "rgb(0,116,217)"
```

```
aerial.color[aerial.Country == "GREAT BRITAIN"] = "rgb(255,65,54)"
aerial.color[aerial.Country == "NEW ZEALAND"] = "rgb(133,20,75)"
aerial.color[aerial.Country == "SOUTH AFRICA"] = "rgb(255,133,27)"

data = [dict(
    type='scattergeo',
    lon = aerial['Takeoff Longitude'],
    lat = aerial['Takeoff Latitude'],
    hoverinfo = 'text',
    text = "Country: " + aerial.Country + " Takeoff Location: "+aerial["Takeoff Location"]
    mode = 'markers',
    marker=dict(
        sizemode = 'area',
        sizeref = 1,
        size= 10 ,
        line = dict(width=1,color = "white"),
        color = aerial["color"],
        opacity = 0.7),
)]
layout = dict(
    title = 'Countries Take Off Bases ',
    hovermode='closest',
    geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
               countrywidth=1, projection=dict(type='Mercator'),
              landcolor = 'rgb(217, 217, 217)',
              subunitwidth=1,
              showlakes = True,
              lakecolor = 'rgb(255, 255, 255)',
              countrycolor="rgb(5, 5, 5)")
)
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

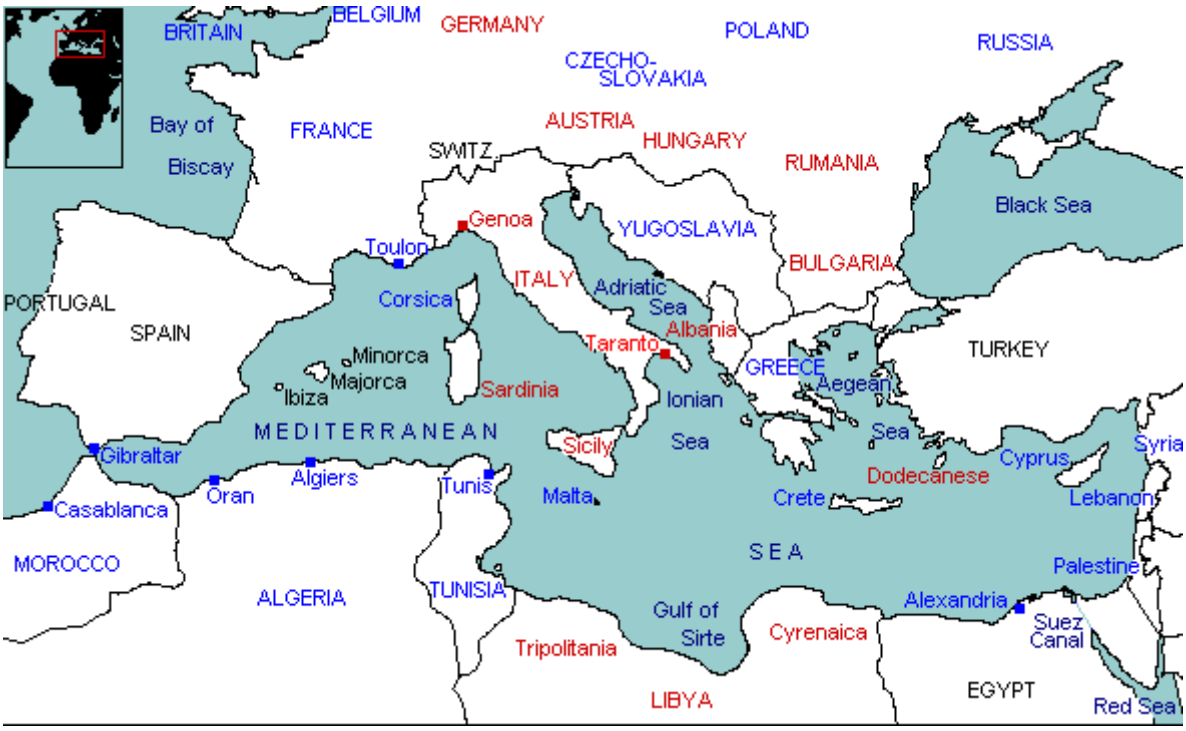
- Okey, now lets visualize bombing paths which country from which take off base bomb the which countries and cities.

```
# Bombing paths
# trace1
airports = [ dict(
        type = 'scattergeo',
        lon = aerial['Takeoff Longitude'],
        lat = aerial['Takeoff Latitude'],
        hoverinfo = 'text',
        text = "Country: " + aerial.Country + " Takeoff Location: "+aerial["Takeoff Locati
        mode = 'markers',
        marker = dict(
            size=5,
            color = aerial["color"],
            line = dict(
                width=1,
                color = "white"
            )
        ))]
# trace2
targets = [ dict(
        type = 'scattergeo',
        lon = aerial['Target Longitude'],
        lat = aerial['Target Latitude'],
        hoverinfo = 'text',
        text = "Target Country: "+aerial["Target Country"]+" Target City: "+aerial["Target
        mode = 'markers',
        marker = dict(
            size=1,
            color = "red",
            line = dict(
                width=0.5,
                color = "red"
            )
        ))]

# trace3
flight_paths = []
for i in range( len( aerial['Target Longitude'] ) ):
    flight_paths.append(
        dict(
            type = 'scattergeo',
            lon = [ aerial.iloc[i,9], aerial.iloc[i,16] ],
            lat = [ aerial.iloc[i,8], aerial.iloc[i,15] ],
            mode = 'lines',
```

```
                line = dict(
                    width = 0.7,
                    color = 'black',
                ),
                opacity = 0.6,
            )
        )

    layout = dict(
        title = 'Bombing Paths from Attacker Country to Target ',
        hovermode='closest',
        geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
                   countrywidth=1, projection=dict(type='Mercator'),
                  landcolor = 'rgb(217, 217, 217)',
                  subunitwidth=1,
                  showlakes = True,
                  lakecolor = 'rgb(255, 255, 255)',
                  countrycolor="rgb(5, 5, 5)")
    )

    fig = dict( data=flight_paths + airports+targets, layout=layout )
    iplot( fig )
```

As you ca see from bombing paths, most of the bombing attack is done in Mediterranean theater of operations. **Theater of Operations:**

- ETO: European Theater of Operations
- PTO: Pasific Theater of Operations
- MTO: Mediterranean Theater of Operations
- CBI: China-Burma-India Theater of Operations
- EAST AFRICA: East Africa Theater of Operations




```
#Theater of Operations
print(aerial['Theater of Operations'].value_counts())
plt.figure(figsize=(22,10))
sns.countplot(aerial['Theater of Operations'])
plt.show()
```

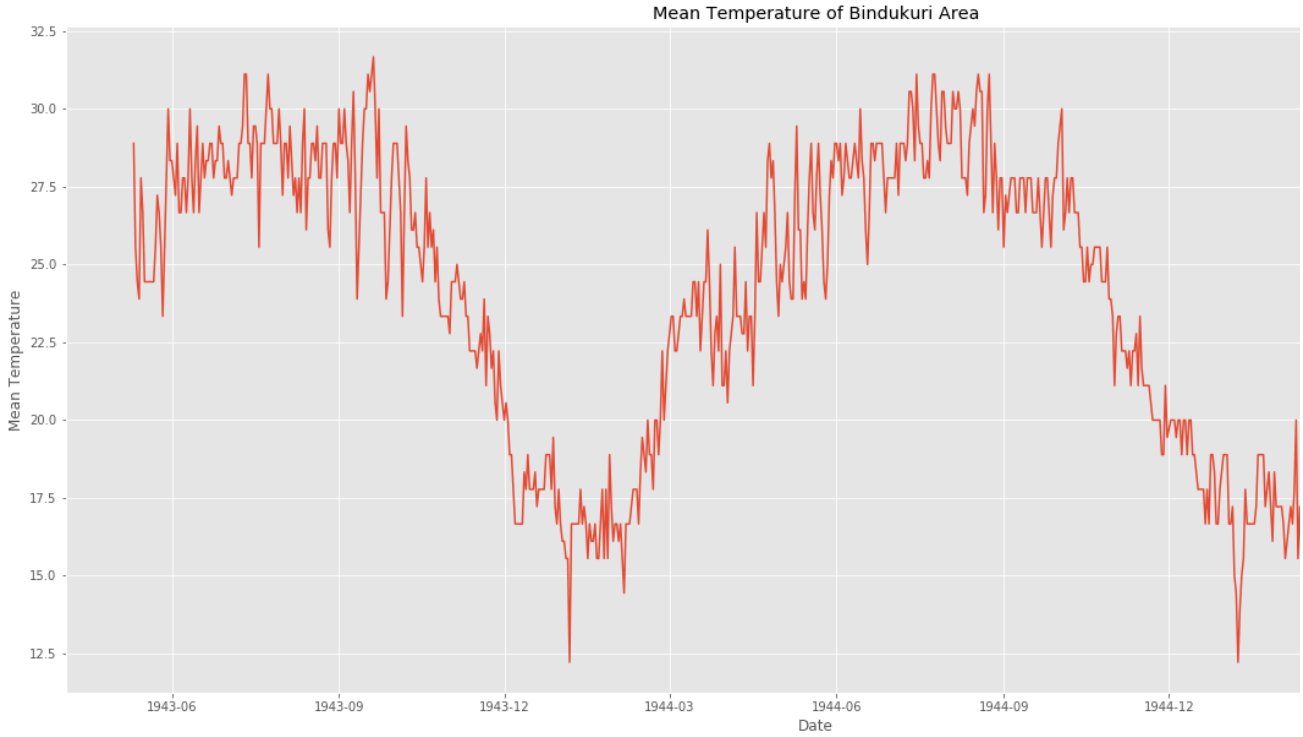
```
MTO            1802
CBI             425
PTO             247
ETO              44
EAST AFRICA      37
Name: Theater of Operations, dtype: int64
```

- Weather station locations are in below

```
# weather station locations

data = [dict(
    type='scattergeo',
    lon = weather_station_location.Longitude,
    lat = weather_station_location.Latitude,
    hoverinfo = 'text',
    text = "Name: " + weather_station_location.NAME + " Country: " + weather_station_locat
    mode = 'markers',
    marker=dict(
        sizemode = 'area',
        sizeref = 1,
        size= 8 ,
        line = dict(width=1,color = "white"),
        color = "blue",
        opacity = 0.7),
)]
layout = dict(
    title = 'Weather Station Locations ',
    hovermode='closest',
    geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
               countrywidth=1, projection=dict(type='Mercator'),
              landcolor = 'rgb(217, 217, 217)',
              subunitwidth=1,
              showlakes = True,
              lakecolor = 'rgb(255, 255, 255)',
              countrycolor="rgb(5, 5, 5)")
)
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

- Lets focus **USA and BURMA war**
- In this war USA bomb BURMA(KATHA city) from 1942 to 1945.
- The closest weather station to this war is **BINDUKURI** and it has temperature record from 1943 to 1945.
- Now lets visualize this situation. But before visualization, we need to make date features date time object.

```
weather_station_id = weather_station_location[weather_station_location.NAME == "BINDUKURI"
weather_bin = weather[weather.STA == 32907]
weather_bin["Date"] = pd.to_datetime(weather_bin["Date"])
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp)
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.show()
```



- As you can see, we have temperature measurement from 1943 to 1945.
- Temperature ossilates between 12 and 32 degrees.
- Temperature of winter months is colder than temperature of summer months.

```
aerial = pd.read_csv("../input/world-war-ii/operations.csv")
aerial["year"] = [ each.split("/")[2] for each in aerial["Mission Date"]]
aerial["month"] = [ each.split("/")[0] for each in aerial["Mission Date"]]
aerial = aerial[aerial["year"]>="1943"]
aerial = aerial[aerial["month"]>="8"]

aerial["Mission Date"] = pd.to_datetime(aerial["Mission Date"])

attack = "USA"
target = "BURMA"
city = "KATHA"

aerial_war = aerial[aerial.Country == attack]
aerial_war = aerial_war[aerial_war["Target Country"] == target]
aerial_war = aerial_war[aerial_war["Target City"] == city]



# I get very tired while writing this part, so sorry for this dummy code But I guess you g
liste = []
aa = []
for each in aerial_war["Mission Date"]:
    dummy = weather_bin[weather_bin.Date == each]
    liste.append(dummy["MeanTemp"].values)
aerial_war["dene"] = liste
```

```
for each in aerial_war.dene.values:
    aa.append(each[0])

# Create a trace
trace = go.Scatter(
    x = weather_bin.Date,
    mode = "lines",
    y = weather_bin.MeanTemp,
    marker = dict(color = 'rgba(16, 112, 2, 0.8)'),
    name = "Mean Temperature"
)
trace1 = go.Scatter(
    x = aerial_war["Mission Date"],
    mode = "markers",
    y = aa,
    marker = dict(color = 'rgba(16, 0, 200, 1)'),
    name = "Bombing temperature"
)
layout = dict(title = 'Mean Temperature --- Bombing Dates and Mean Temperature at this Dat
data = [trace,trace1]

fig = dict(data = data, layout = layout)
iplot(fig)
```

- Green line is mean temperature that is measured in Bindukuri.
- Blue markers are bombing dates and bombing date temperature.
- As it can be seen from plot, USA bomb at high temperatures.
    - The question is that can we predict future weather and according to this prediction can we know whether bombing will be done or not.
    - In order to answer this question lets first start with time series prediction.

Time Series Prediction with ARIMA

- We will use most used method ARIMA
- ARIMA : AutoRegressive Integrated Moving Average. I will explain it detailed at next parts.
- The way that we will follow:
    - What is Time Series ?
    - Stationarity of a Time Series
    - Make a Time Series Stationary?
    - Forecasting a Time Series

What is time series?

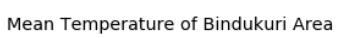
- Time series is a collection of data points that are collected at constant time intervals.
- It is time dependent.
- Most of time series have some form of **seasonality trends**. For example, if we sale ice cream, most probably there will be higher sales in summer seasons. Therefore, this time series has seasonality trends.
- Another example, lets think we dice one time every day during 1 year. As you guess, there will be no scenario like that number six is appeared mostly in summer season or number five is mostly appeared in January. Therefore, this time series does not have seasonality trends.

Stationarity of a Time Series

- There are three basic criterion for a time series to understand whether it is stationary series or not.
    - Statistical properties of time series such as mean, variance should remain constant over time to call **time series is stationary**

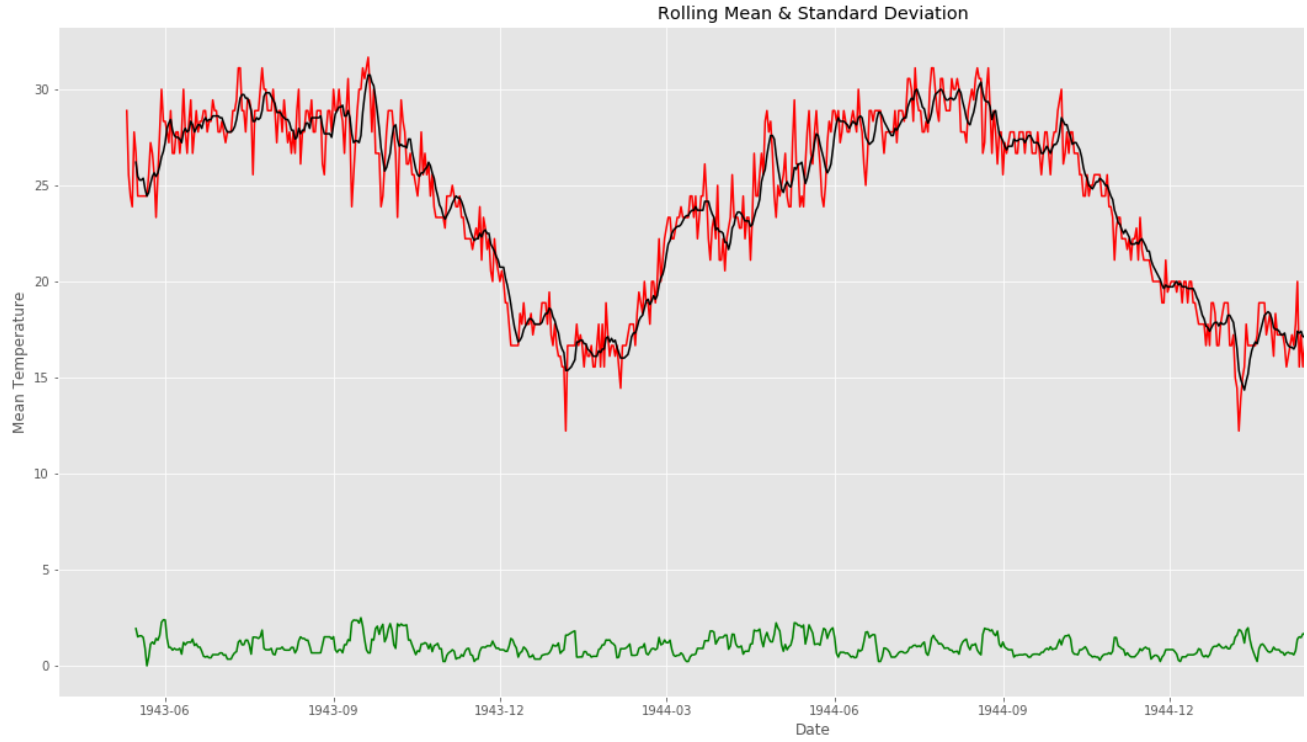
        - constant mean
        - constant variance
        - autocovariance that does not depend on time. autocovariance is covariance between time series and lagged time series.
- Lets visualize and check seasonality trend of our time series.

```
# Mean temperature of Bindikuri area
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp)
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.show()

# lets create time series from weather
timeSeries = weather_bin.loc[:, ["Date","MeanTemp"]]
timeSeries.index = timeSeries.Date
ts = timeSeries.drop("Date",axis=1)
```



- As you can see from plot above, our time series has seasonal variation. In summer, mean temperature is higher and in winter mean temperature is lower for each year.
- Now lets check stationary of time series. We can check stationarity using the following methods:
    - Plotting Rolling Statistics: We have a window lets say window size is 6 and then we find rolling mean and variance to check stationary.
    - Dickey-Fuller Test: The test results comprise of a **Test Statistic** and some **Critical Values** for difference confidence levels. If the **test statistic** is less than the **critical value**, we can say that time series is stationary.

```
# adfuller library
from statsmodels.tsa.stattools import adfuller
# check_adfuller
def check_adfuller(ts):
    # Dickey-Fuller test
    result = adfuller(ts, autolag='AIC')
    print('Test statistic: ' , result[0])
    print('p-value: '  ,result[1])
    print('Critical Values:' ,result[4])
# check_mean_std
def check_mean_std(ts):
    #Rolling statistics
    rolmean = pd.rolling_mean(ts, window=6)
    rolstd = pd.rolling_std(ts, window=6)
    plt.figure(figsize=(22,10))
    orig = plt.plot(ts, color='red',label='Original')
    mean = plt.plot(rolmean, color='black', label='Rolling Mean')
    std = plt.plot(rolstd, color='green', label = 'Rolling Std')
    plt.xlabel("Date")
    plt.ylabel("Mean Temperature")
    plt.title('Rolling Mean & Standard Deviation')
    plt.legend()
    plt.show()

# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts)
check_adfuller(ts.MeanTemp)
```




```
Test statistic:  -1.409596674588769
p-value:  0.5776668028526388
Critical Values: {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.56885
```

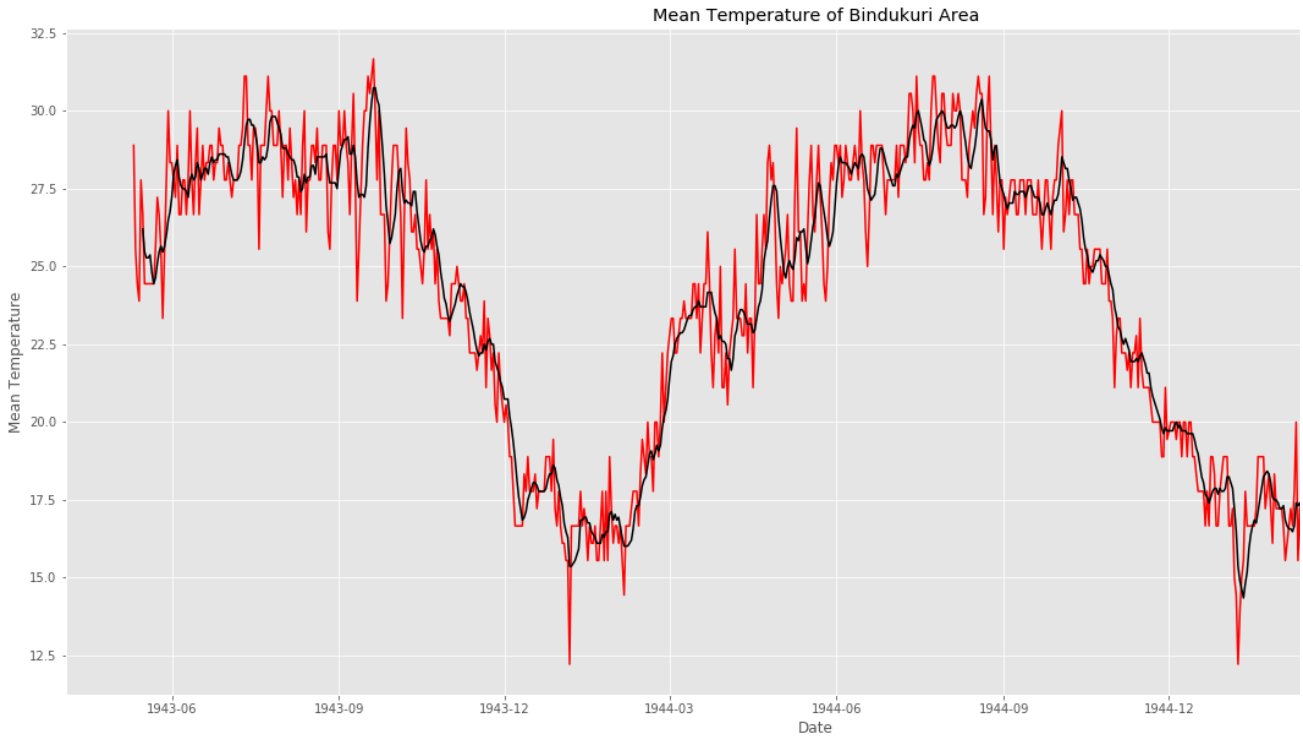
- Our first criteria for stationary is constant mean. So we fail because mean is not constant as you can see from plot(black line) above . (no stationary)
- Second one is constant variance. It looks like constant. (yes stationary)
- Third one is that If the **test statistic** is less than the **critical value**, we can say that time series is stationary. Lets look:
    - test statistic = -1.4 and critical values = {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.5688568756191392}. Test statistic is bigger than the critical values. (no stationary)
- As a result, we sure that our time series is not stationary.
- Lets make time series stationary at the next part.

Make a Time Series Stationary?

- As we mentioned before, there are 2 reasons behind non-stationarity of time series
    - Trend: varying mean over time. We need constant mean for stationary of time series.
    - Seasonality: variations at specific time. We need constant variations for stationary of time series.
- First solve **trend(constant mean)** problem

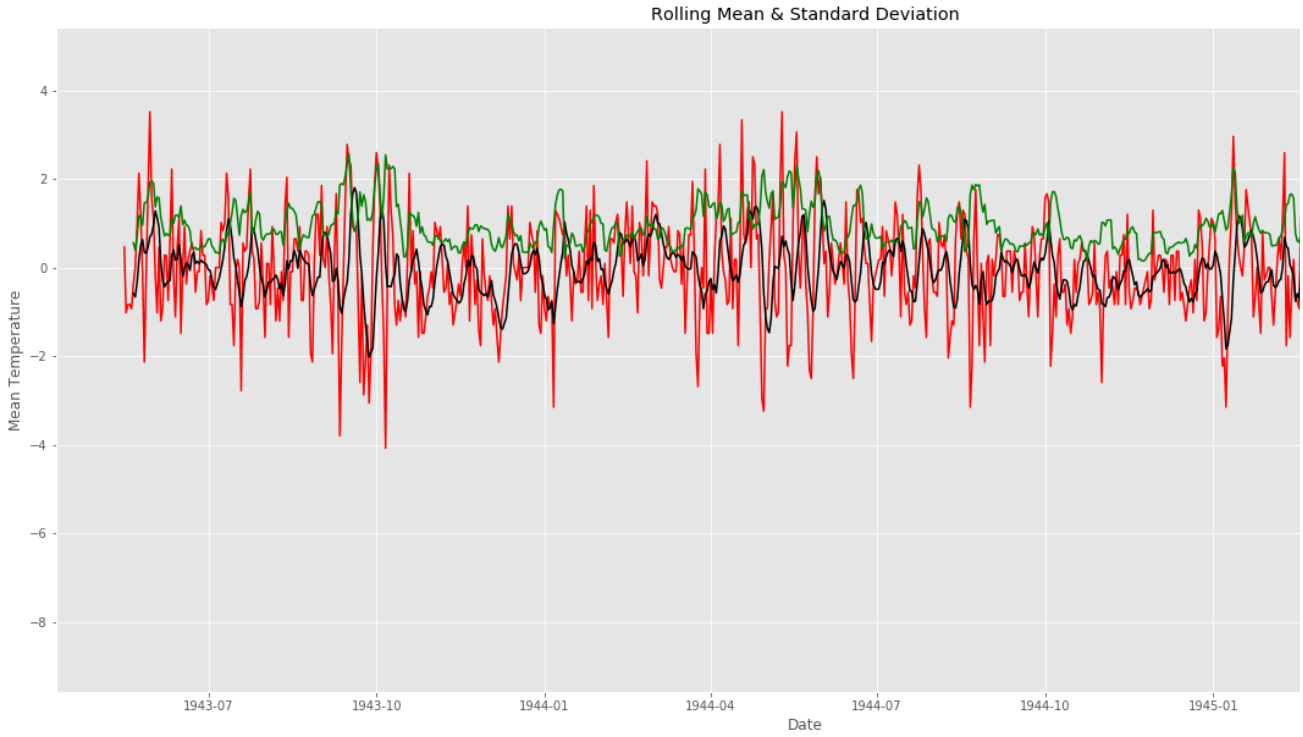
- Most popular method is moving average.
  - Moving average: We have window that take the average over the past 'n' sample. 'n' is window size.

```
# Moving average method
window_size = 6
moving_avg = pd.rolling_mean(ts,window_size)
plt.figure(figsize=(22,10))
plt.plot(ts, color = "red",label = "Original")
plt.plot(moving_avg, color='black', label = "moving_avg_mean")
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.show()
```




```
ts_moving_avg_diff = ts - moving_avg
ts_moving_avg_diff.dropna(inplace=True) # first 6 is nan value due to window size
```

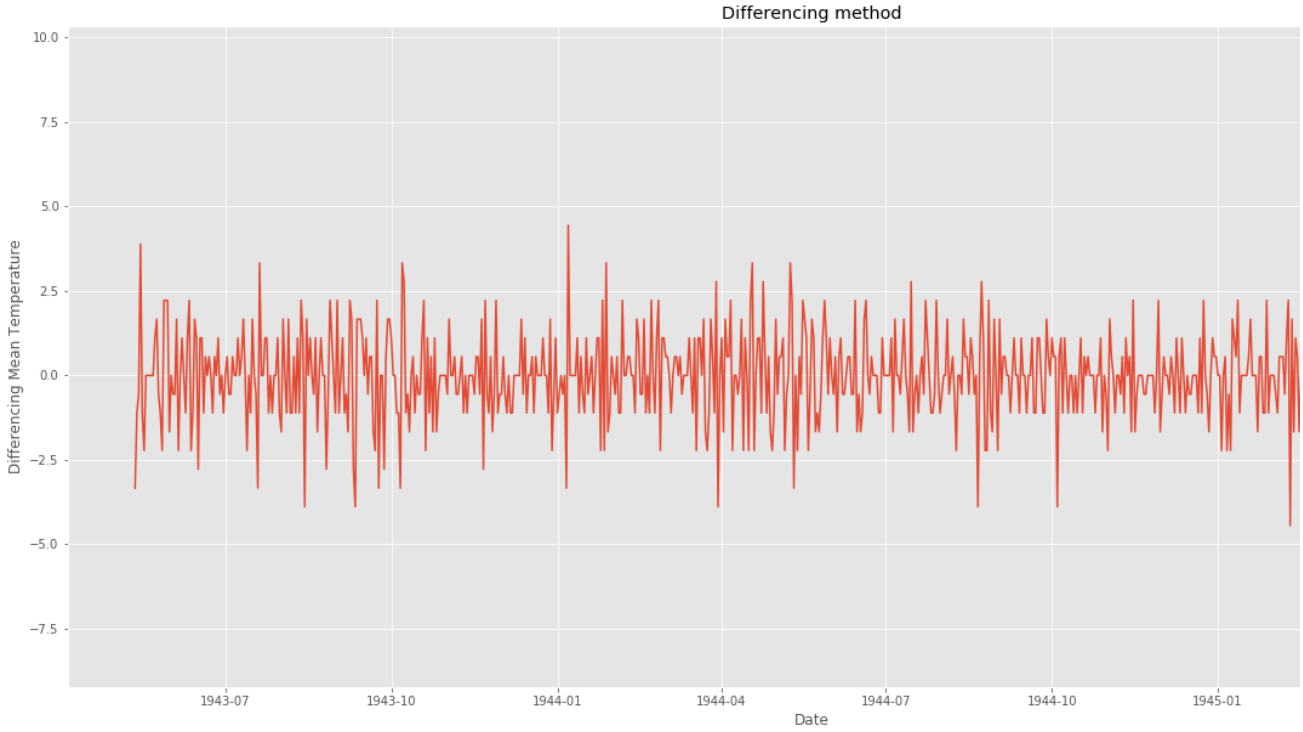
```
# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts_moving_avg_diff)
check_adfuller(ts_moving_avg_diff.MeanTemp)
```



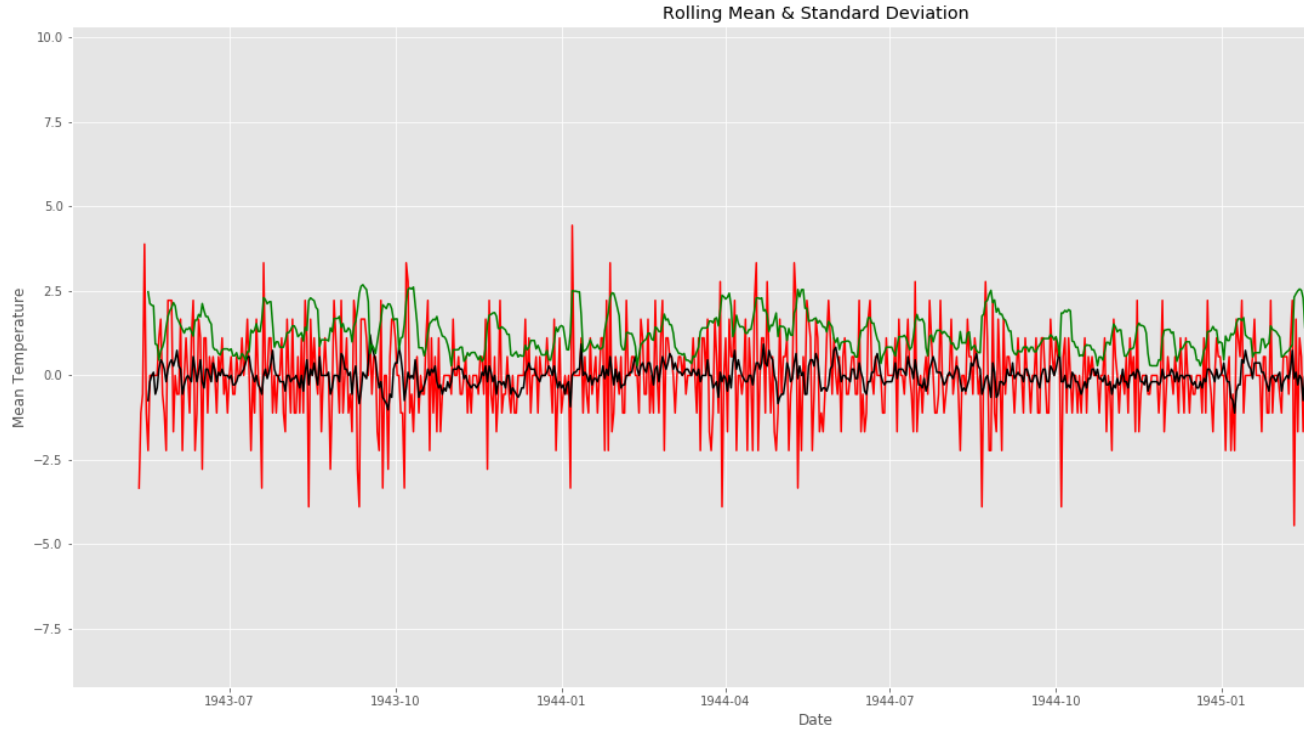

```
Test statistic:  -11.138514335138474
p-value:  3.150868563164652e-20
Critical Values: {'1%': -3.4392539652094154, '5%': -2.86546960465041, '10%': -2.56886
```

- Constant mean criteria: mean looks like constant as you can see from plot(black line) above . (yes stationary)
- Second one is constant variance. It looks like constant. (yes stationary)
- The test statistic is smaller than the 1% critical values so we can say with 99% confidence that this is a stationary series. (yes stationary)
- We achieve stationary time series. However lets look at one more method to avoid trend and seasonality.
    - Differencing method: It is one of the most common method. Idea is that take difference between time series and shifted time series.

```
# differencing method
ts_diff = ts - ts.shift()
plt.figure(figsize=(22,10))
plt.plot(ts_diff)
plt.title("Differencing method")
plt.xlabel("Date")
plt.ylabel("Differencing Mean Temperature")
plt.show()
```




```
ts_diff.dropna(inplace=True) # due to shifting there is nan values
# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts_diff)
check_adfuller(ts_diff.MeanTemp)
```




```
Test statistic:  -11.678955575105382
p-value:  1.7602075693558453e-21
Critical Values: {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.568856
```

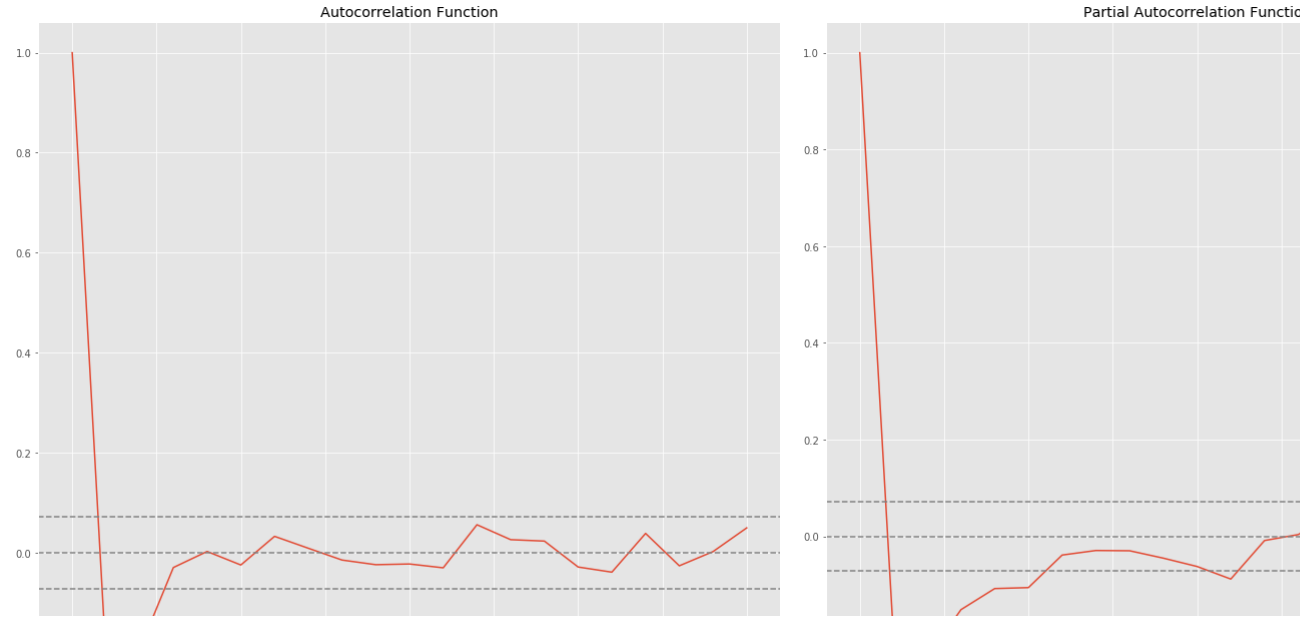
- Constant mean criteria: mean looks like constant as you can see from plot(black line) above . (yes stationary)
- Second one is constant variance. It looks like constant. (yes stationary)
- The test statistic is smaller than the 1% critical values so we can say with 99% confidence that this is a stationary series. (yes stationary)

Forecasting a Time Series

- We learn two different methodsthat are **moving average and differencing** methods to avoid trend and seasonality problem
- For prediction(forecasting) we will use ts_diff time series that is result of differencing method. There is no reason I only choose it.
- Also prediction method is ARIMA that is Auto-Regressive Integrated Moving Averages.
    - AR: Auto-Regressive (p): AR terms are just lags of dependent variable. For example lets say p is 3, we will use x(t-1), x(t-2) and x(t-3) to predict x(t)
    - I: Integrated (d): These are the number of nonseasonal differences. For example, in our case we take the first order difference. So we pass that variable and put d=0

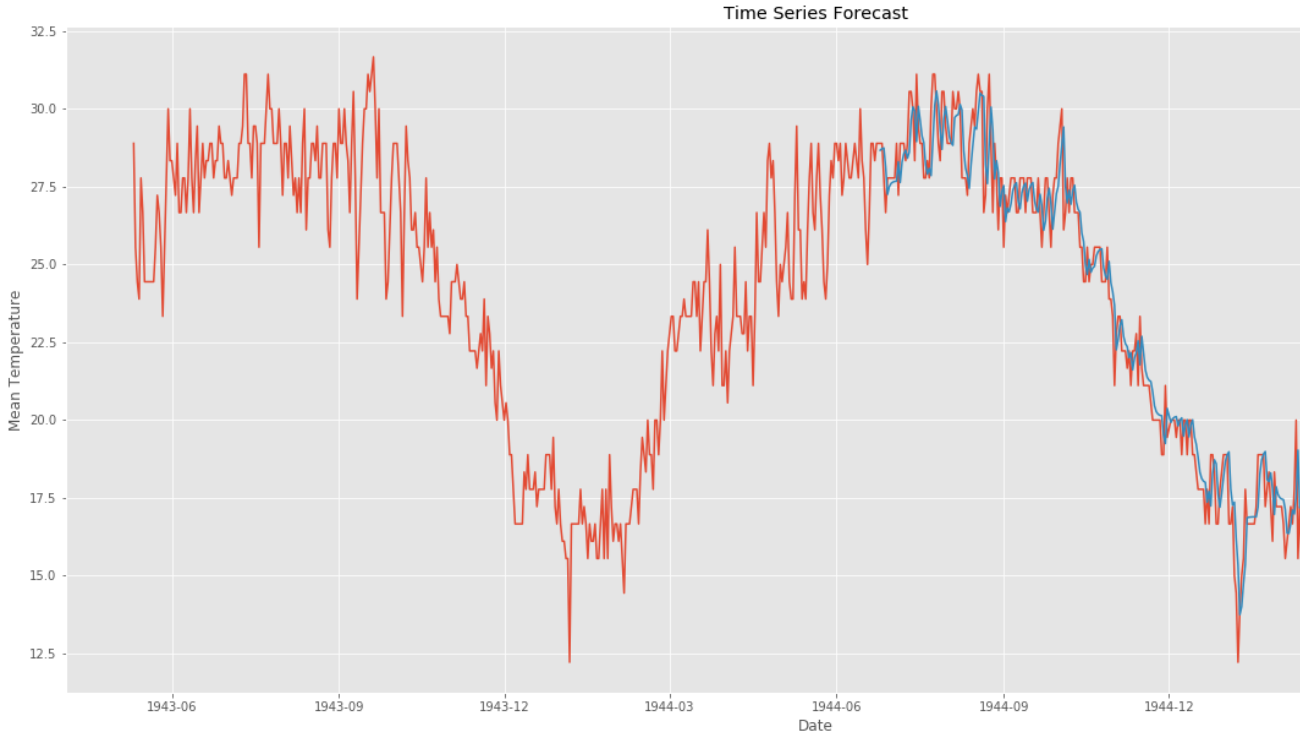
    - MA: Moving Averages (q): MA terms are lagged forecast errors in prediction equation.
- (p,d,q) is parameters of ARIMA model.
- In order to choose p,d,q parameters we will use two different plots.
    - Autocorrelation Function (ACF): Measurement of the correlation between time series and lagged version of time series.
    - Partial Autocorrelation Function (PACF): This measures the correlation between the time series and lagged version of time series but after eliminating the variations already explained by the intervening comparisons.

```
# ACF and PACF
from statsmodels.tsa.stattools import acf, pacf
lag_acf = acf(ts_diff, nlags=20)
lag_pacf = pacf(ts_diff, nlags=20, method='ols')
# ACF
plt.figure(figsize=(22,10))

plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.title('Autocorrelation Function')

# PACF
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function')
plt.tight_layout()
```



- Two dotted lines are the confidence interevals. We use these lines to determine the ‘p’ and ‘q’ values
    - Choosing p: The lag value where the PACF chart crosses the upper confidence interval for the first time. p=1.
    - Choosing q: The lag value where the ACF chart crosses the upper confidence interval for the first time. q=1.
- Now lets use (1,0,1) as parameters of ARIMA models and predict
    - ARIMA: from statsmodels libarary
    - datetime: we will use it start and end indexes of predict method

```
# ARIMA LİBRARY
from statsmodels.tsa.arima_model import ARIMA
from pandas import datetime

# fit model
model = ARIMA(ts, order=(1,0,1)) # (ARMA) = (1,0,1)
model_fit = model.fit(disp=0)

# predict
start_index = datetime(1944, 6, 25)
end_index = datetime(1945, 5, 31)
forecast = model_fit.predict(start=start_index, end=end_index)

# visualization
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp,label = "original")
plt.plot(forecast,label = "predicted")
plt.title("Time Series Forecast")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.show()
```

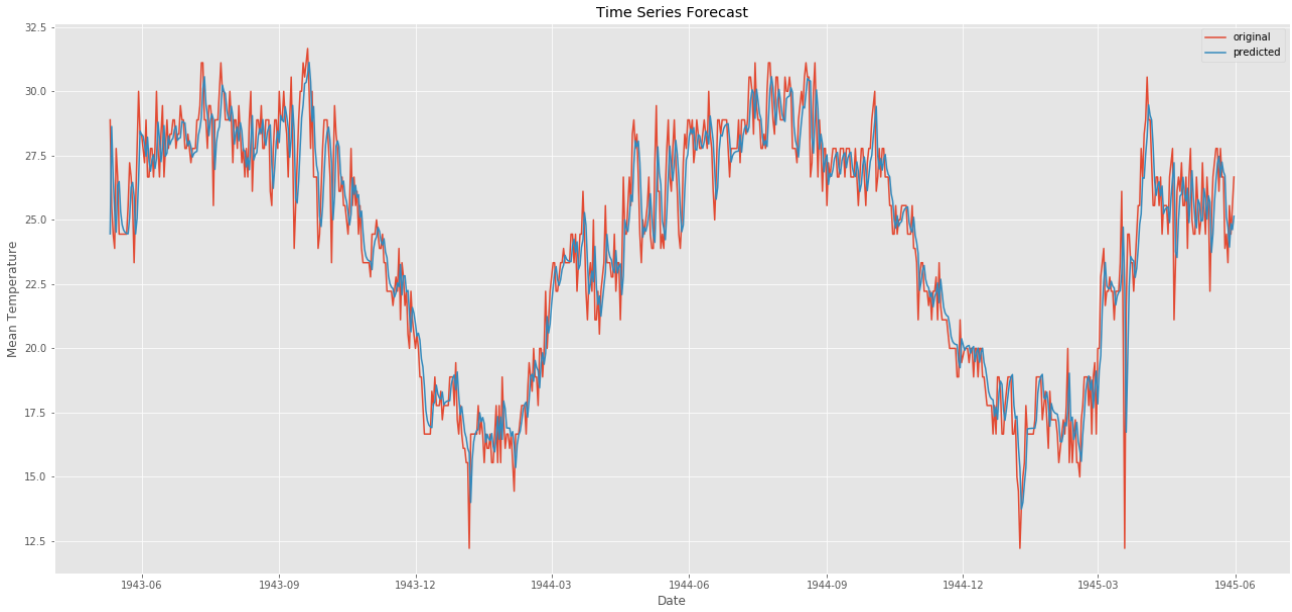


- lets predict and visualize all path and find mean squared error

```
# predict all path
from sklearn.metrics import mean_squared_error
# fit model
model2 = ARIMA(ts, order=(1,0,1)) # (ARMA) = (1,0,1)
model_fit2 = model2.fit(disp=0)
forecast2 = model_fit2.predict()
error = mean_squared_error(ts, forecast2)
print("error: " ,error)
# visualization
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp,label = "original")
plt.plot(forecast2,label = "predicted")
plt.title("Time Series Forecast")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.savefig('graph.png')
```

```
plt.show()
```

```
error:  1.8625819952314109
```



Conclusion

- In this tutorial, I want ot make tutorial about ARIMA and make some visualization before it.
- We learn how to make map plots with pyplot.
- We learn how to make time series forecast.
- **If you have any question advise or feedback, I will be very happy to hear it**