A study on the timeseries forecasting method: ARIMA

```python
# initial imports
import pandas as pd

import numpy as np

import plotly.express as px
import plotly.graph_objects as go

import plotly.io as pio
pio.renderers.default='notebook'

import holidays

import warnings
warnings.filterwarnings("ignore")

from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

import pmdarima as pm
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

To run the same notebook for other countries, youll need to change the country name in 3 places:

1. country variable while reading dataset
2. creating holidays array for the exog variable in the training section
3. creating holidays array for the exog variable in the forecasting section

but,

NOTE:

1. Since the study was done for India, the parameters were optimized for that particular series, and you might see high errors in other countries unless you change those parameters for yourself before running all cells

2. It was not possible to write a common generic functional flow for this due to the way the holiday library is implemented, where the country is passed as a method and not a argument

Data Preparation

The daily data (for confirmed cases in each) has already been preprocessed in update_data.py and is stored the data directory

Now we write a function that takes this cumulative time series and returns a new time series with number of new daily cases for a singule country, for:

1. Easier trend, seasonality identification, and to
2. Reduce chances for the cumulative number to go down

```python
confirmed_global = pd.read_csv(r"./data/country_confirmed.csv")
```

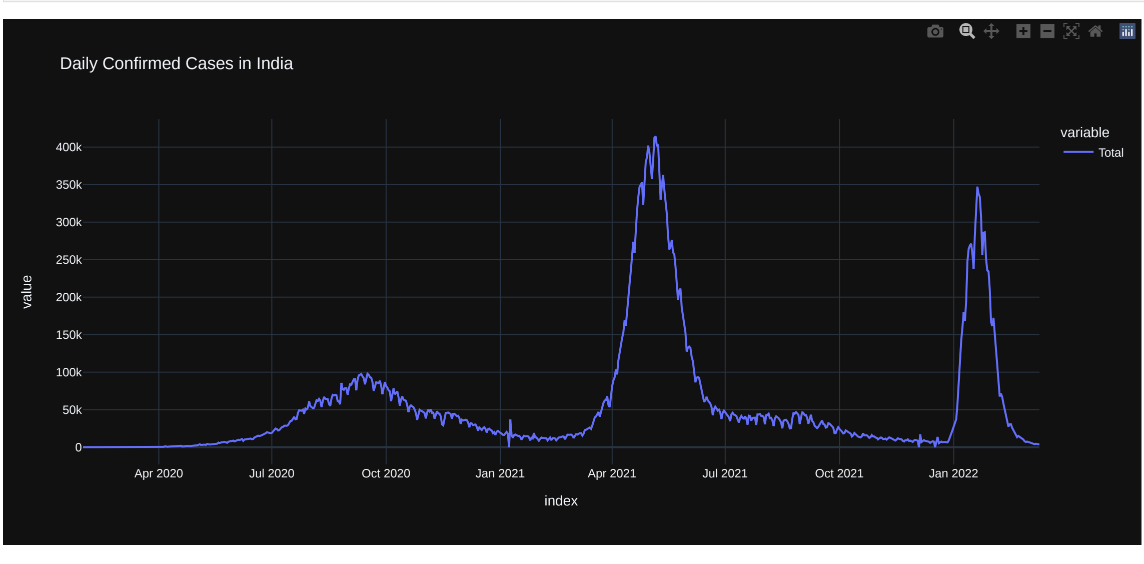
```python
def get_data(country, confirmed=confirmed_global):
    confirmed = confirmed.groupby("country").sum().T
    confirmed.index = pd.to_datetime(confirmed.index, infer_datetime_format=True)
    data = pd.DataFrame(
            index=confirmed.index, data=confirmed[country].values, columns=["Total"]
        )
    # we remove the beginninhg 0s as they do not contribute
    data = data[(data != 0).all(1)]

    data_diff = data.diff()

    # removing the first value from data_diff
    # it had no previous value and is a NaN after taking the difference
    data_diff = data_diff[1:]

    return data, data_diff
```

```python
country = "India"
confirmed_dfs = get_data(country)
confirmed_daily = confirmed_dfs[1] # taking the daily data, not the cumulative
```

```python
confirmed_daily.tail(10)
```

	Total
2022-03-02	6561.0
2022-03-03	6396.0
2022-03-04	5921.0
2022-03-05	5476.0
2022-03-06	4362.0
2022-03-07	3993.0
2022-03-08	4575.0
2022-03-09	4184.0
2022-03-10	4194.0
2022-03-11	3614.0

```python
# plotting our daily cases
px.line(confirmed_daily, title=f"Daily Confirmed Cases in {country}", template="plotly_dark")
```



We will be using holidays of that particular country as an additional feature (besides the past data), for the model to identify patterns.

```python
holiday = [1 if i==True else 0 for i in [day in holidays.India() for day in confirmed_daily.index]]
confirmed_daily["holiday"] = holiday
```

```python
df = confirmed_daily.copy()
df = df[["holiday", "Total"]]
df[df["holiday"] == 1].tail() # making sure there are holidays in the data that we are using
```

	holiday	Total
2021-10-02	1	22842.0
2021-11-04	1	12729.0
2021-12-25	1	6987.0
2022-01-14	1	268833.0
2022-01-26	1	286384.0

Model Identification

Depending on the type of our time series, there are 2 to 7 parameters that we will need to identify:

1. **Stationary time series:**

A staionary time series is a time series whose value(and covariance) does not depend on time at which the series is observed rather just the lag ***'k'*** wrt to some other point in the series, i.e one that has a constant mean and variance (and thus by default cannot have a non zero trend). An example of a perfectly stationary time series is a sine wave. But as time series are rarely perfectly stationary, we will set a threshold(p-value of 0.05 in adfuller) and use a test (Augmented Dicky-Fuller or adfuller) to identify if the time series is stationary or not.

Forecast of a stationary time series can be found using an ARMA model. It consists of two parts Autoreggressive(AR) and Moving Average(MA). Moving average is a technique that forecasts the future value of a time series data using the average (or of needed weighted average) of the past *n* values. The AR part is a regression on the time series itself measures/observed at different points with respect to a specified lag *k*. For example if we were use a AR model with lag 1 or AR(1), the model's equation would be given as follows:

Y(t+1) = μ + βY(t) + ε(t+1)

where μ is the mean of the time series, β is the coefficient of the previous value of the time series, and ε(t+1) is the extra residual term

In this model, we need to identify only 2 parameters/orders: **p** and **q**, the lags for the AR and MA processes respectively. The final model will be a ARMA(p,q) or ARIMA(p,0,q)

1. **Non stationary and non seasonal:**

If our time series does not satisfy the conditions for the stationary time series, we will use the method of differencing to convert our time series to a stationary time series. This is what the extra ***'I'*** in ARIMA stands for: Integration, i.e, order of differencing.

In this model, the 3 parameters/orders we will need to identify are: **p**, **q** (same as that from ARMA) and **d**, the order of differencing. The final model will be ARIMA(p,d,q).

1. **Seasonal time series:**

If our time series has an additional seasonality component, we will need to use SARIMA/SARIMAX model to forecast (Seasonal ARIMA) for the forecast, as seasonality doesnt work well with the standard ARIMA model. For this, in addition to the 3 parameters we found in ARIMA, we will need to identify:

1. ***P*** and ***Q***: the lags of the **seasonal component** of the time series
2. ***D***: the order of differencing for the seasonal component
3. ***s or m***: the number of seasonal periods in the time series. (i.e, if m is 4, each period will be 1/4 of a year, so quarterly. If 12, then monthly etc.)

Step 1

Checking if our time series is stationary or not, and if not, identify the order of differencing.

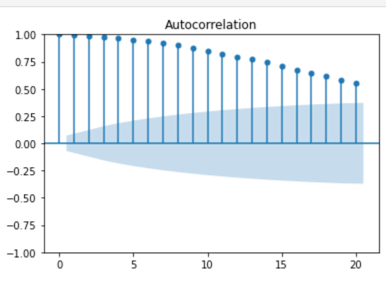
```python
result=adfuller(df['Total'].dropna())
print(f'p-value: {result[1]}')
```

```
p-value: 0.08187455463886278
```

As our p-value is > 0.05, the time series is not stationary.

Another show of this, is that the decrease/decline in the auto correlations is very slow (shown below)

```python
acf = plot_acf(df['Total'].dropna(), lags=20)
```

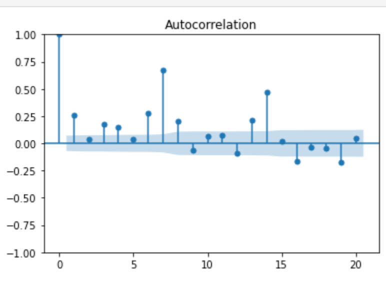


We will now try the same with the same series after a single difference (d = 1)

```python
result=adfuller(df['Total'].diff().dropna())
print(f'p-value: {result[1]}')
```

```
p-value: 6.3930081355302866e-09
```

```python
acf = plot_acf(df['Total'].diff().dropna(), lags=20)
```



After this single difference, we find that our time series satisfies both conditions for being stationary.

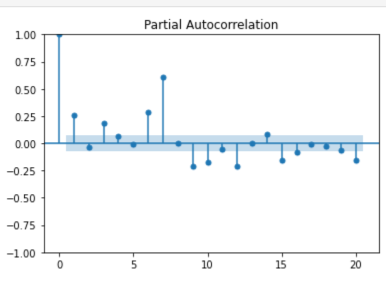
We have successfully identified the order of differencing to be 1. (i.e, **d = 1**)

Step 2:

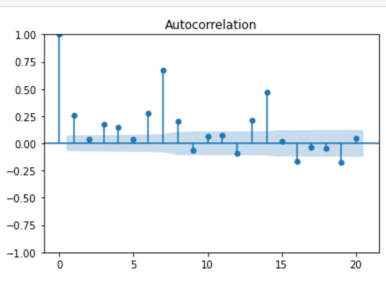
To identify the lags **p** and **q** manually, we can follow the steps:

1. The partial autocorrelation is significant only for the first p-values/lags and cuts off to zero.
2. The autocorrelation values is significant only for the first q-values/lags and cuts off to zero.

```python
# manually identifying p

pacf = plot_pacf(df['Total'].diff().dropna(), lags=20)
```




```python
# manually identifying q

acf = plot_acf(df['Total'].diff().dropna(), lags=20)
```



From these plots we can see the lags upto 2 for both acf and pacf are significant, therfore our manual arrived values of p and q are 2 and 2 respectively.

So the model we arrived at from our analysis is ARIMA(2,1,2) (NOTE: we do not need to and so haven't yet checked or accounted for the seasonal component)

Model Evaluation

```python
df.index.freq = "D"
arima = SARIMAX(df['Total'], order=(2,1,2), exog=df['holiday'])
model = arima.fit(method="powell")
```

```
Optimization terminated successfully.
         Current function value: 10.390369
         Iterations: 2
         Function evaluations: 145
```

```python
# we now evaluate our model
pred = model.predict(start=len(df['Total'])-28, end=len(df['Total'])-1, exog=df["holiday"][-28:],dynamic=False)
```

```python
# error functions
def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
def mape(predictions, targets):
    return np.mean(np.abs((predictions - targets) / targets)) * 100
```

```python
# mape
print(f"MAPE: {mape(pred, df['Total'][-28:]).round(2)}%")
```

```
MAPE: 10.1%
```

```python
# rmse
print(f"RMSE: {rmse(pred, df['Total'][-28:]).round(2)}")
```

```
RMSE: 2323.1
```

```python
fig = go.Figure()
fig.add_trace(go.Scatter(x=df[-28:].index, y=df[-28:]['Total'], name="Actual"))
fig.add_trace(go.Scatter(x=df[-28:].index, y=pred, name="Predicted"))
fig.update_layout(title_text=f"{country} Confirmed Cases", xaxis_title="Date", yaxis_title="Cases", template="plotly_dark", hovermode="x")
```

Additionally, to both comapre with out analysis and to optimize our model, we can use auto_arima from pmdarima, which works like a grid search for the most optimal parameters

```python
results=pm.auto_arima(df['Total'], start_p=1, d=1, start_q=1, max_p=2, max_q=2,
                      information_criterion='bic', trace=True, error_action='ignore',
                      exog=df['holiday'], stepwise=True, freq="D")
```

```
Performing stepwise search to minimize bic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : BIC=16106.310, Time=0.31 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : BIC=16150.275, Time=0.04 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : BIC=16105.306, Time=0.05 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : BIC=16100.836, Time=0.13 sec
 ARIMA(0,1,0)(0,0,0)[0]             : BIC=16143.630, Time=0.05 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : BIC=16107.017, Time=0.20 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : BIC=16078.000, Time=0.45 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : BIC=16061.090, Time=0.46 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : BIC=16100.489, Time=0.55 sec
 ARIMA(2,1,2)(0,0,0)[0]             : BIC=16054.443, Time=0.71 sec
 ARIMA(1,1,2)(0,0,0)[0]             : BIC=16071.350, Time=0.58 sec
 ARIMA(2,1,1)(0,0,0)[0]             : BIC=16093.843, Time=0.78 sec
 ARIMA(1,1,1)(0,0,0)[0]             : BIC=16099.663, Time=0.35 sec

Best model:  ARIMA(2,1,2)(0,0,0)[0]
Total fit time: 4.676 seconds
```

This agrees with our analysis!!

NOTES:

1. Setting a higher max_p and max_q might result in auto_arima resulting in a different model, but by experimenting with those values, I observed that the model was getting overfitted , and hence stuck to theor max values being 2 and 2 respectively.

2. We do not take into account seasonality here as ARIMA/SARIMAX is known to face many problems in high frequency data (even weekly data *(m=52)* causes problems, and ours is daily data *(m=365)*) and in data where the seasonal cycles are too long, as mentioned in these links:

a. Poor perfomance on long cycles,

b. SARIMAX too blunt for high frequncy data(Answer written by one of the authors of statsmodels himself!!)

(Same reason accounts for a flat trend predicted in the test set if a train-test split is used to evalute the model. To backup this claim I did the train test split down to 90-10 split in data, with the same parameters for the ARIMA model to get a ~60-70% MAPE, as the model just coninues the trend from the point the split was made on (refer second link above))

Forecasting

```python
datelist = pd.date_range(start=df.index[-1], periods= 8, freq="D")[1:]
```

```python
is_holiday = [1 if i==True else 0 for i in [day in holidays.India() for day in datelist]]
```

```python
forecast = model.get_forecast(steps=7, exog = is_holiday)
mean_forecast=forecast.predicted_mean
```

```python
print(mean_forecast)
```

```
2022-03-12    3100.513606
2022-03-13    2769.706399
2022-03-14    2370.934411
2022-03-15    2041.129035
2022-03-16    1702.817324
2022-03-17    1397.277760
2022-03-18    3514.084271
Freq: D, Name: predicted_mean, dtype: float64
```

```python
# converting to cumulative form for final plot and prediction format
start = confirmed_dfs[0]["Total"][-1]
predictions_cumulative = []
for i in mean_forecast :
    start = start + i
    predictions_cumulative.append(start)
```

```python
fig = go.Figure()

fig.add_trace(go.Scatter(x=confirmed_dfs[0].index[-60:], y=confirmed_dfs[0]["Total"][-60:],
                         mode='lines',
                         name='Up till now'))

fig.add_trace(go.Scatter(x=datelist, y=predictions_cumulative,
                         mode='lines',
                         name='Prediction*'))

fig.update_layout(title_text=f"{country} Confirmed Cases", xaxis_title="Date", yaxis_title="Cases", template="plotly_dark", hovermode="x")
```


```python
# function to format the number for easy readability
def format_number(number):
    s, *d = str(number).partition(".")
    r = ",".join([s[x-3:x] for x in range(-3, -len(s), -3)][::-1] + [s[-3:]])
    return "".join([r] + d)
```

```python
predictions = pd.DataFrame()
predictions.index = datelist
predictions["Total"] = [format_number(str(int(i))) for i in predictions_cumulative]
```

```python
predictions
```

	Total
2022-03-12	42,990,975
2022-03-13	42,993,745
2022-03-14	42,996,116
2022-03-15	42,998,157
2022-03-16	42,999,860
2022-03-17	43,001,257
2022-03-18	43,004,771