ANALYSIS OF TIME SERIES DATA USING MACHINE LEARNING

PROJECT REPORT

HARDWARE DESIGN PROJECT

By

PARAS WAHI **E18ECE026**

Submitted to

Dr. Ankit Kumar Pandey

BENNETT UNIVERSITY

INDEX

INTRODUCTION

Time series analysis is a statistical technique that deals with time series data, or trend analysis. Time series data means that data is in a series of particular time periods or intervals.

Time series are one of the most common data types encountered in daily life. Financial prices, weather, home energy usage, and even weight are all examples of data that can be collected at regular intervals.

This project uses various machine learning techniques like SARIMAX, XGBOOST, FACEBOOK's PROPHET and FEED FORWARD NEURAL NETWORKS to learn the properties of time series data and analyze it.

WHAT IS TIME SERIES ?

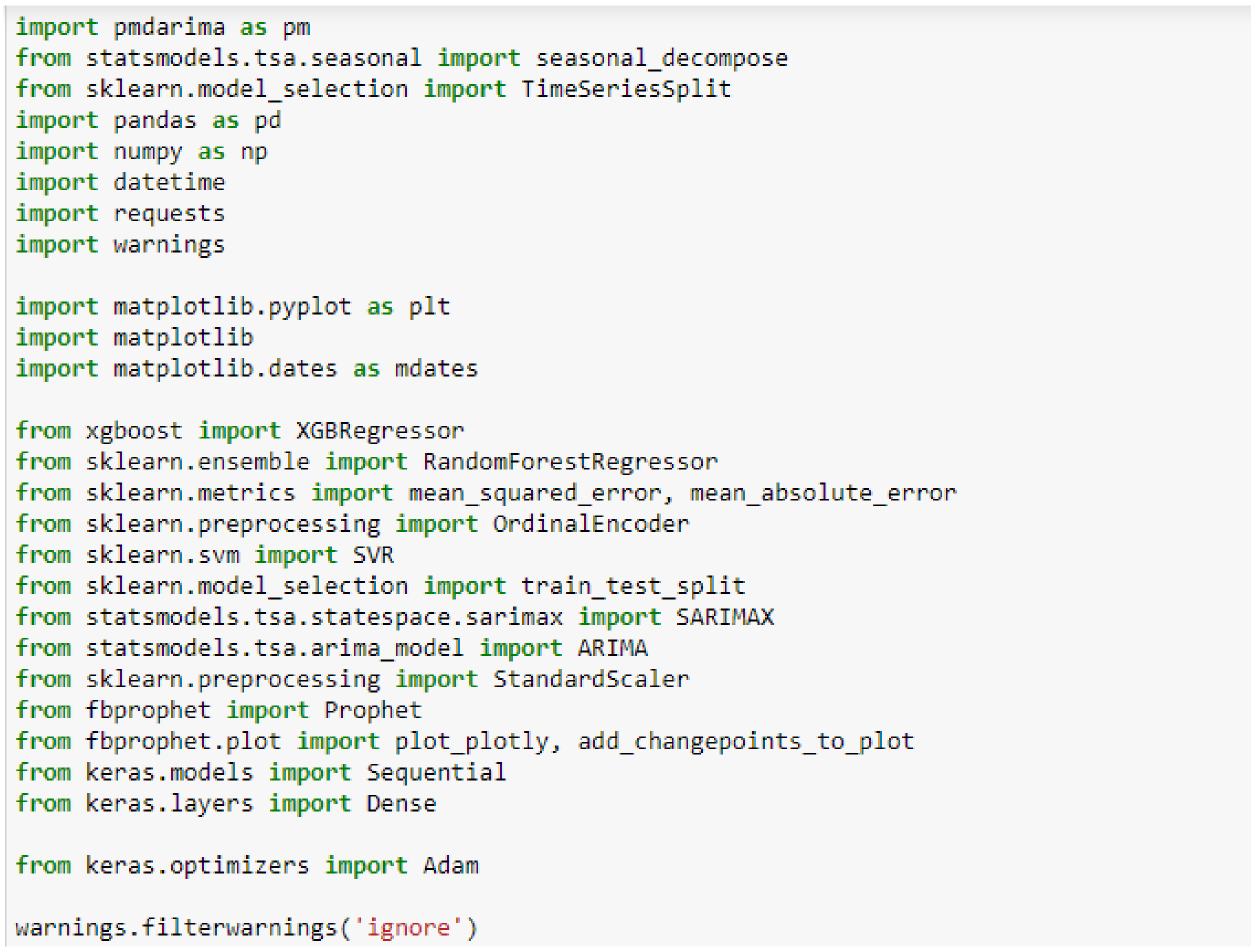
Time series data is a collection of observations obtained through repeated measurements over time. Plot the points on a graph, and one of your axes would always be time.

Time series data is everywhere, since time is a constituent of everything that is observable. As our world gets increasingly instrumented, sensors and systems are constantly emitting a relentless stream of time series data. Such data has numerous applications across various industries.

Time series data can be useful for:

- Tracking daily, hourly, or weekly weather data
- Tracking changes in application performance
- Medical devices to visualize vitals in real time
- Tracking network logs.

IMPORTING LIBRARIES AND PREPARING DATASET FRAME



```python
import pmdarima as pm
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.model_selection import TimeSeriesSplit
import pandas as pd
import numpy as np
import datetime
import requests
import warnings

import matplotlib.pyplot as plt
import matplotlib
import matplotlib.dates as mdates

from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import OrdinalEncoder
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima_model import ARIMA
from sklearn.preprocessing import StandardScaler
from fbprophet import Prophet
from fbprophet.plot import plot_plotly, add_changepoints_to_plot
from keras.models import Sequential
from keras.layers import Dense

from keras.optimizers import Adam

warnings.filterwarnings('ignore')
```

```
deaths_df = pd.read_csv('time_series_covid19_deaths_global.csv')
```

First, get the data for the daily deaths in all countries in the report

```
d = deaths_df.loc[:, '1/22/20':]
```

Transpose the data frame

```
d = d.transpose()
```

Then sum row-wise

```
d = d.sum(axis=1)
```

We only need the numeric values here, so we convert `d` to a list

```
d = d.to_list()
```

Create a new data frame with two columns, which will be our dataset:

```
dataset = pd.DataFrame(columns=['ds', 'y'])
```

Get the dates from the columns in `deaths_df` data frame, starting from the fifth column. They will be obtained in string format.

```
dates = list(deaths_df.columns[4:])
```

Convert the string dates into the datetime format, so that we can perform datetime operations on them

```
dates = list(pd.to_datetime(dates))
```

Now, assign the dates and deaths data to the columns in the new dataframe:

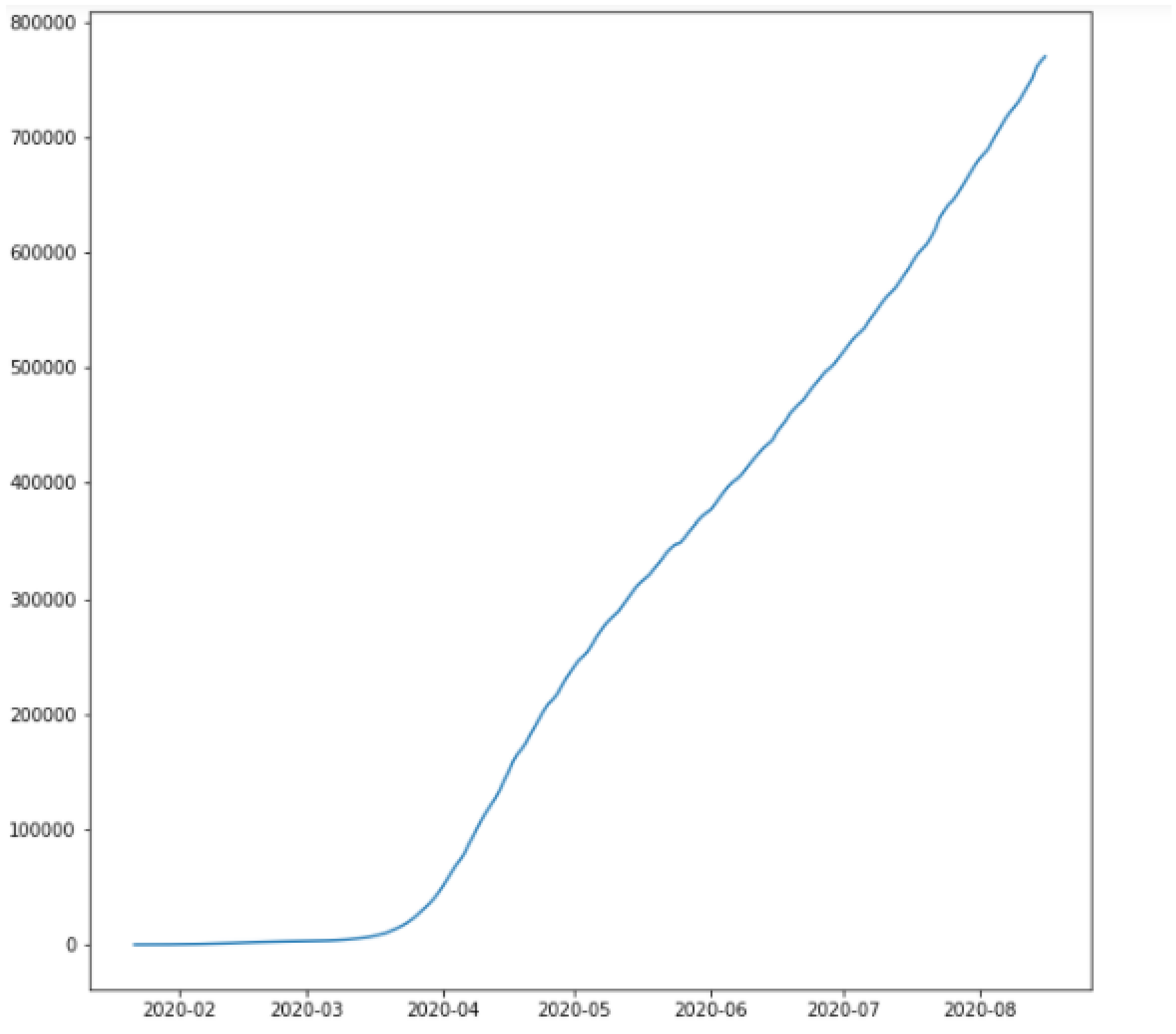
```
dataset['ds'] = dates
dataset['y'] = d
```

We will need to have only 1 data column, y, with the index being ds

```
dataset=dataset.set_index('ds')
```

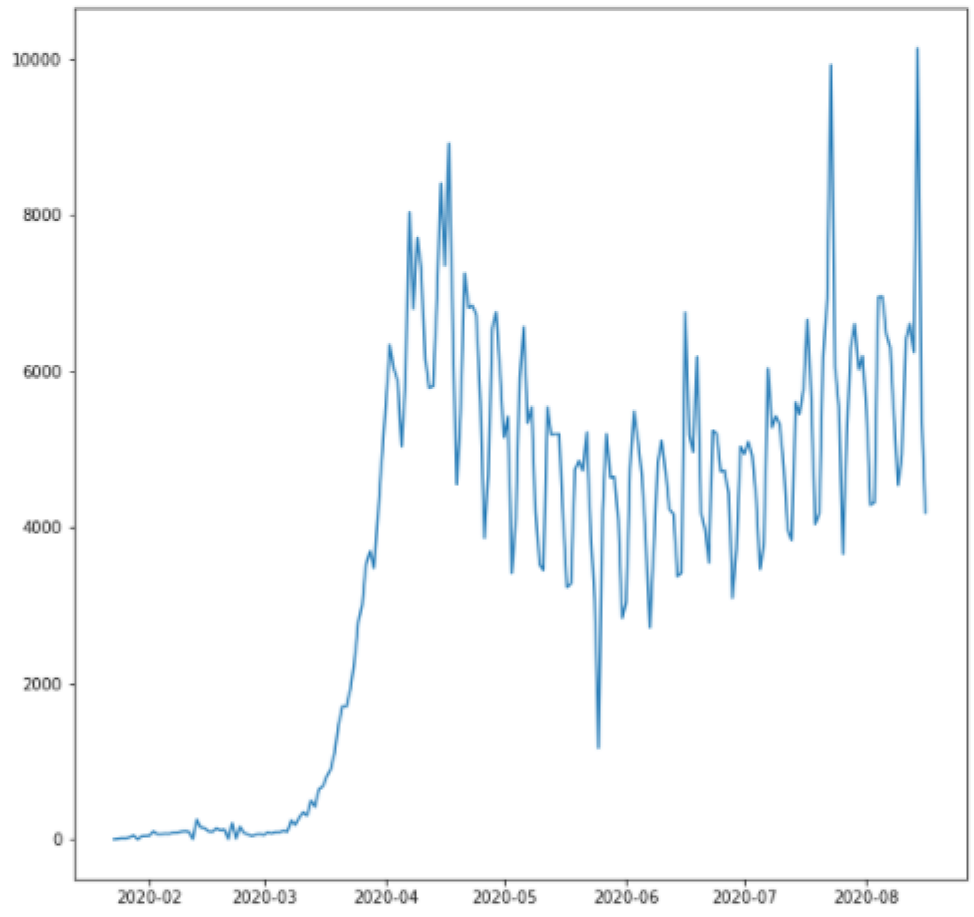
Let's plot the daily number of deaths

```
plt.figure(figsize=(10, 10))
plt.plot(dataset)
plt.savefig('Cummulative daily deaths', bbox_inches='tight', transparent=False)
```



Plotting daily increase in deaths using "diff" method from dataframe object.

```
plt.figure(figsize=(10, 10))
plt.plot(dataset.diff())
plt.savefig('Daily deaths', bbox_inches='tight', transparent=False)
```



In this project, we are going to analyze the daily reported deaths

```
dataset = dataset.diff()
```

We need to remove the first data point here, which will be a None value

```
dataset = dataset.loc['2020-01-23':'2020-08-13']
```

THE SARIMAX MODEL

- Autoregressive Integrated Moving Average, or ARIMA, is one of the most widely used forecasting methods for univariate time series data forecasting.
- Although the method can handle data with a trend, it does not support time series with a seasonal component.
- An extension to ARIMA that supports the direct modeling of the seasonal component of the series is called SARIMA.

Seasonal Autoregressive Integrated Moving Average, SARIMA or Seasonal ARIMA, is an extension of ARIMA that explicitly supports univariate time series data with a seasonal component.

It adds three new hyperparameters to specify the autoregression (AR), differencing (I) and moving average (MA) for the seasonal component of the series, as well as an additional parameter for the period of the seasonality.

But before we apply the model, we will split our dataset into training and test sets. We will do this by taking all dates prior to 31st of July as a training set, on which the model will be optimized, and all dates from 31st July as the test set, on which we will test the accuracy of the predictions that the optimized model will make.

```
start_date = '2020-07-31'

train = dataset.loc[dataset.index < pd.to_datetime(start_date)]
test = dataset.loc[dataset.index >= pd.to_datetime(start_date)]
```

Now let's talk about SARIMAX. There are three hyperparameters that go into the order tuple: p, q and d. Here we've used p=2, q=1 and d=3.

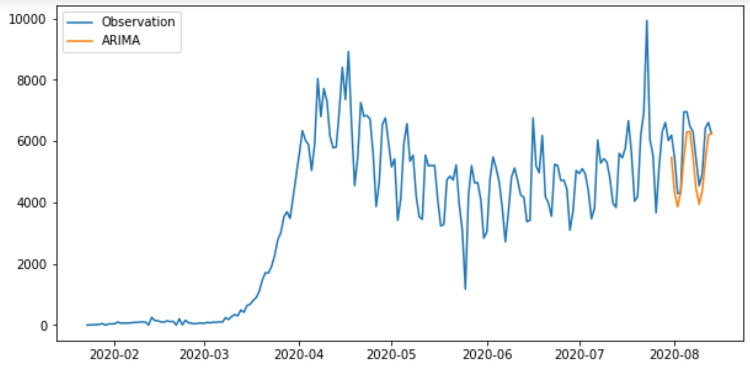
```
model = SARIMAX(train, order=(2, 1, 3))
```

Next, we call the fit method to optimize the model.

```
results = model.fit(disp=True)
```

Now let's make predictions using the model, and compare those against the values in the test set.

```
sarimax_prediction = results.predict(
    start=start_date, end='2020-08-13', dynamic=False)
plt.figure(figsize=(10, 5))
l1, = plt.plot(dataset, label='Observation')
l2, = plt.plot(sarimax_prediction, label='ARIMA')
plt.legend(handles=[l1, l2])
plt.savefig('SARIMAX prediction', bbox_inches='tight', transparent=False)
```

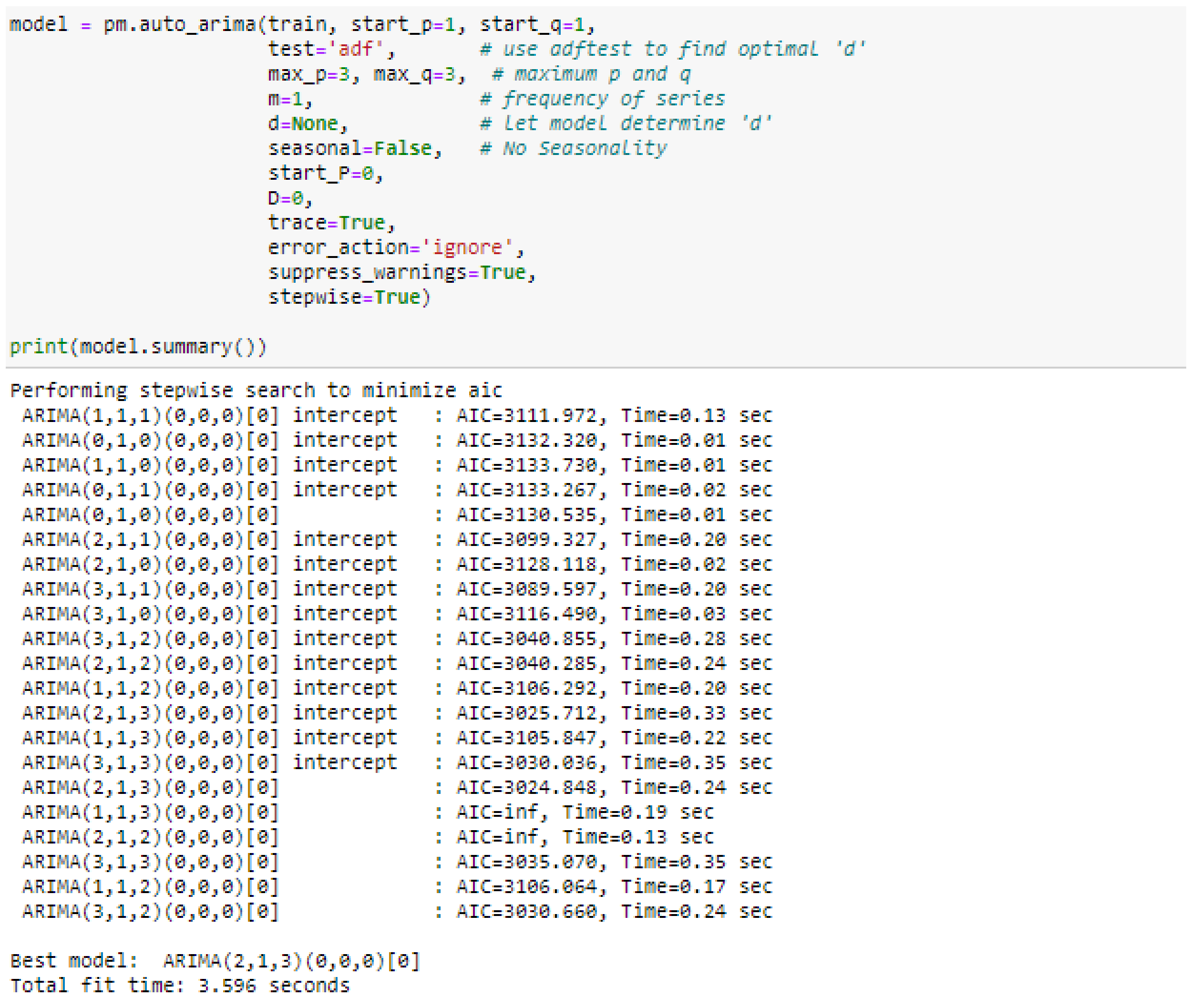


Since that we are interested in comparing between the different time series analysis approaches, we are going to use one of the validation measures: mean absolute error.

```
print('SARIMAX MAE = ', mean_absolute_error(sarimax_prediction, test))
```

```
SARIMAX MAE =  650.6548918305232
```

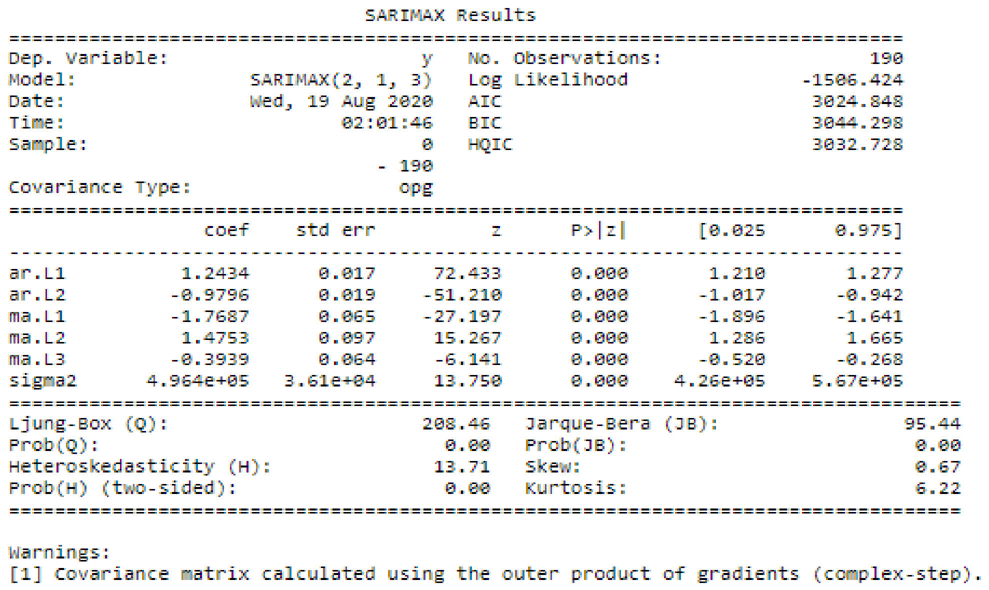
However, in a typical machine learning workflow, we should find the best values of p, q and r that will minimize the error. We can use the auto_arima function in the pmdarima module to do that. This will find the optimal parameter combination and return the best model.



```
model = pm.auto_arima(train, start_p=1, start_q=1,
                      test='adf',       # use adftest to find optimal 'd'
                      max_p=3, max_q=3,  # maximum p and q
                      m=1,              # frequency of series
                      d=None,           # let model determine 'd'
                      seasonal=False,   # No Seasonality
                      start_P=0,
                      D=0,
                      trace=True,
                      error_action='ignore',
                      suppress_warnings=True,
                      stepwise=True)

print(model.summary())
```

```
Performing stepwise search to minimize aic
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=3111.972, Time=0.13 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=3132.320, Time=0.01 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=3133.730, Time=0.01 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=3133.267, Time=0.02 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=3130.535, Time=0.01 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=3099.327, Time=0.20 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=3128.118, Time=0.02 sec
 ARIMA(3,1,1)(0,0,0)[0] intercept   : AIC=3089.597, Time=0.20 sec
 ARIMA(3,1,0)(0,0,0)[0] intercept   : AIC=3116.490, Time=0.03 sec
 ARIMA(3,1,2)(0,0,0)[0] intercept   : AIC=3040.855, Time=0.28 sec
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=3040.285, Time=0.24 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=3106.292, Time=0.20 sec
 ARIMA(2,1,3)(0,0,0)[0] intercept   : AIC=3025.712, Time=0.33 sec
 ARIMA(1,1,3)(0,0,0)[0] intercept   : AIC=3105.847, Time=0.22 sec
 ARIMA(3,1,3)(0,0,0)[0] intercept   : AIC=3030.036, Time=0.35 sec
 ARIMA(2,1,3)(0,0,0)[0]             : AIC=3024.848, Time=0.24 sec
 ARIMA(1,1,3)(0,0,0)[0]             : AIC=inf, Time=0.19 sec
 ARIMA(2,1,2)(0,0,0)[0]             : AIC=inf, Time=0.13 sec
 ARIMA(3,1,3)(0,0,0)[0]             : AIC=3035.070, Time=0.35 sec
 ARIMA(1,1,2)(0,0,0)[0]             : AIC=3106.064, Time=0.17 sec
 ARIMA(3,1,2)(0,0,0)[0]             : AIC=3030.660, Time=0.24 sec

Best model:  ARIMA(2,1,3)(0,0,0)[0]
Total fit time: 3.596 seconds
```

SARIMAX RESULTS



```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  190
Model:               SARIMAX(2, 1, 3)   Log Likelihood               -1506.424
Date:                Wed, 19 Aug 2020   AIC                           3024.848
Time:                        02:01:46   BIC                           3044.298
Sample:                             0   HQIC                          3032.728
                                - 190
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.2434      0.017     72.433      0.000       1.210       1.277
ar.L2         -0.9796      0.019    -51.210      0.000      -1.017      -0.942
ma.L1         -1.7687      0.065    -27.197      0.000      -1.896      -1.641
ma.L2          1.4753      0.097     15.267      0.000       1.286       1.665
ma.L3         -0.3939      0.064     -6.141      0.000      -0.520      -0.268
sigma2      4.964e+05   3.61e+04     13.750      0.000    4.26e+05    5.67e+05
===================================================================================
Ljung-Box (Q):                      208.46   Jarque-Bera (JB):                95.44
Prob(Q):                              0.00   Prob(JB):                         0.00
Heteroskedasticity (H):              13.71   Skew:                             0.67
Prob(H) (two-sided):                  0.00   Kurtosis:                         6.22
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

FACEBOOK’S PROPHET MODEL

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

This is the open source time series library released by Facebook. It is also widely used by Facebook in their own time series analysis tasks. Facebook prophet does not require that you specify or search for hyperparameters. The model can act as a black box that does all the required computations on its own. And it works with the same object-fit-predict API.

Prophet expects the data frame to have 2 columns, unlike SARIMAX.

```
train['ds'] = train.index.values
```

Then we create a new Prophet object and call the `fit()` method

```
m = Prophet()
m.fit(train)
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

```
<fbprophet.forecaster.Prophet at 0x14c828cd0>
```

Now let's forecast:

```
future = m.make_future_dataframe(periods=dataset.shape[0]-train.shape[0])
prophet_prediction = m.predict(future)
```

Now let us calculate the mean absolute error for our predictions.

```
prophet_prediction = prophet_prediction.set_index('ds')
prophet_future = prophet_prediction.yhat.loc[prophet_prediction.index >= start_date]
print('Prophet MAE = ', mean_absolute_error(prophet_future, test))
```

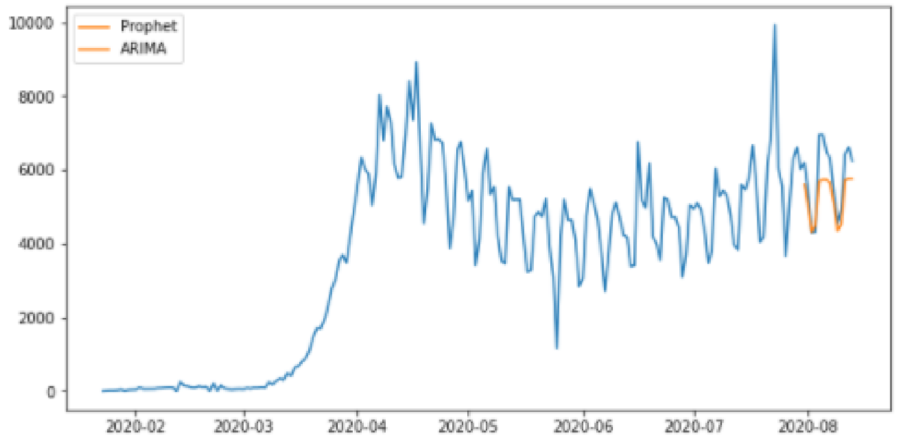
```
Prophet MAE =  572.5560073843841
```

Next, let's visualize the predictions.

```
plt.figure(figsize=(10, 5))
l1, = plt.plot(dataset, label='Observation')
l1, = plt.plot(prophet_future, label='Prophet')
plt.legend(handles=[l1, l2])
plt.savefig('Prophet predictions',
            bbox_inches='tight', transparent=False)
```

VISUALIZE PREDICTIONS

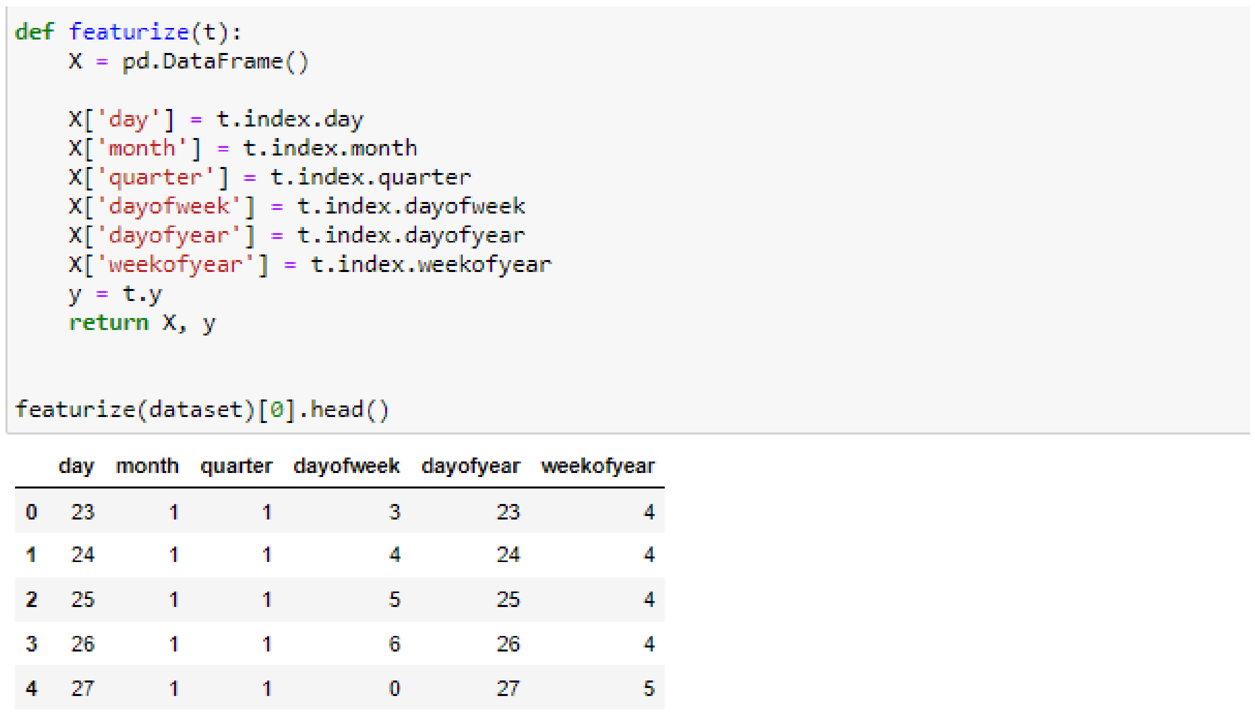
```
plt.figure(figsize=(10, 5))
l1, = plt.plot(dataset, label='Observation')
l1, = plt.plot(prophet_future, label='Prophet')
plt.legend(handles=[l1, l2])
plt.savefig('Prophet predictions',
            bbox_inches='tight', transparent=False)
```



PREPARING DATASET FOR XGBOOST AND NN

Unlike the prophet and SAIMAX models, the two models we will train in Task 6, namely XGBOOST and NN, are supervised machine learning models that deal with independent data points, or examples. It assumes that each data point is totally independent from the rest of the data points in the dataset.

Here is a method that extracts these features from a given dataframe object.



```
def featurize(t):
    X = pd.DataFrame()

    X['day'] = t.index.day
    X['month'] = t.index.month
    X['quarter'] = t.index.quarter
    X['dayofweek'] = t.index.dayofweek
    X['dayofyear'] = t.index.dayofyear
    X['weekofyear'] = t.index.weekofyear
    y = t.y
    return X, y


featurize(dataset)[0].head()
```

	day	month	quarter	dayofweek	dayofyear	weekofyear
0	23	1	1	3	23	4
1	24	1	1	4	24	4
2	25	1	1	5	25	4
3	26	1	1	6	26	4
4	27	1	1	0	27	5

CREATING TRAINING AND TEST DATASETS BY SPLITTING THE DATASET, AND PERFORM DATA NORMALIZATION.

We have done a splitting operation of a dataset in machine learning, where one takes a randomly selected portion of the dataset, say 20%, as a test set, while the remaining 80% is the training set. It is randomly selected because the whole dataset is randomly shuffled before the selection. Another popular approach is the k-fold cross validation.

However, those two methods won't work with time series data. The reason is: when we train the model on the training set, the purpose is to predict the target values in the future, which corresponds to date values that are outside of the date values in the training set.

```
X_train, y_train = featurize(
    dataset.loc[dataset.index < pd.to_datetime(start_date)])
X_test, y_test = featurize(
    dataset.loc[dataset.index >= pd.to_datetime(start_date)])
```

We perform data normalization so as to make the range of values of the features, or the columns in the X_train table, as close as possible. For example, we have the features dayofweek and dayofyear. The range of values of dayofweek is from 1 to 7, whereas dayofyear is from 1 to 365. Having such large differences in the ranges of values will either slow down the training of the machine learning model or make it quite difficult. We solve this problem by applying normalization. There are several ways we can normalize the data with. Here I will choose the StandardScaler, which applies the following equation on each of the columns.

$z = (x - u) / s$

Here x is the column before scaling, u is the mean and s is the standard deviation. So basically, we subtract the mean of each column from itself, then divide by the standard deviation of that column. To apply StandardScaler, we first fit the scalar object to the dataset:

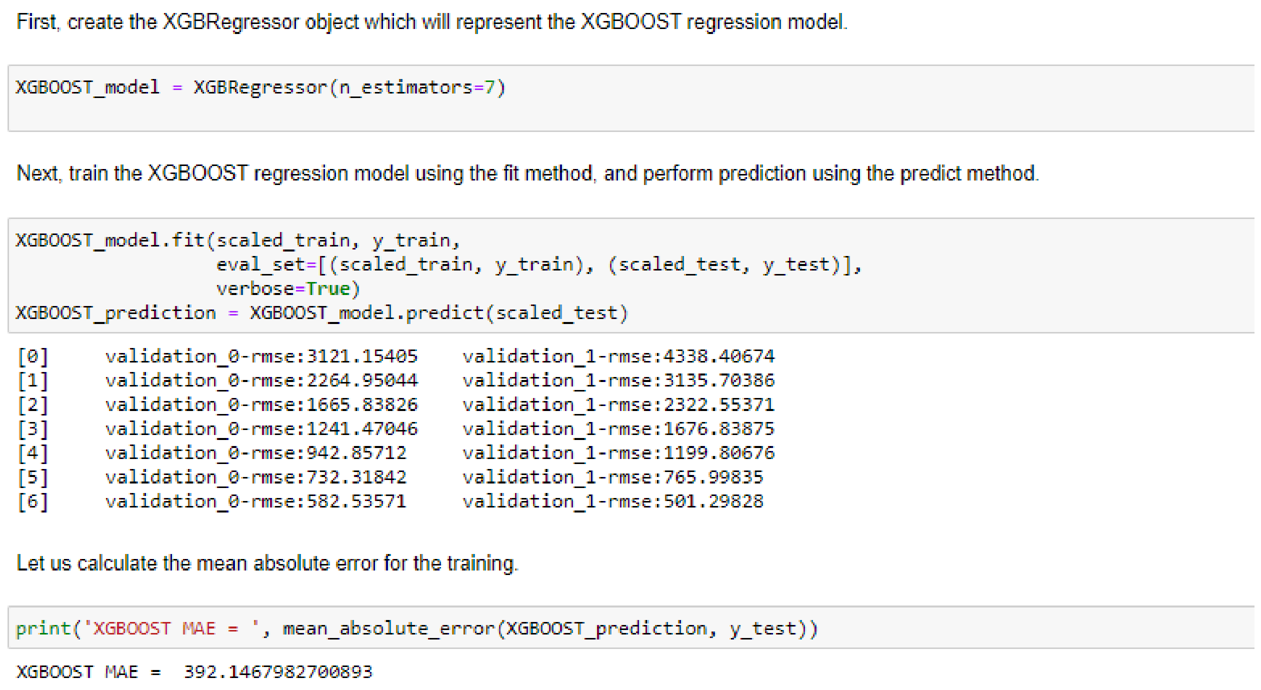
```
scaler = StandardScaler()
scaler.fit(X_train)
```

```
StandardScaler()
```

Apply the scaling to both the training and test sets, as follows.

```
scaled_train = scaler.transform(X_train)
scaled_test = scaler.transform(X_test)
```

TRAINING THE XGBOOST AND NN MODELS



First, create the XGBRegressor object which will represent the XGBOOST regression model.

```
XGBOOST_model = XGBRegressor(n_estimators=7)
```

Next, train the XGBOOST regression model using the fit method, and perform prediction using the predict method.

```
XGBOOST_model.fit(scaled_train, y_train,
                  eval_set=[(scaled_train, y_train), (scaled_test, y_test)],
                  verbose=True)
XGBOOST_prediction = XGBOOST_model.predict(scaled_test)
```

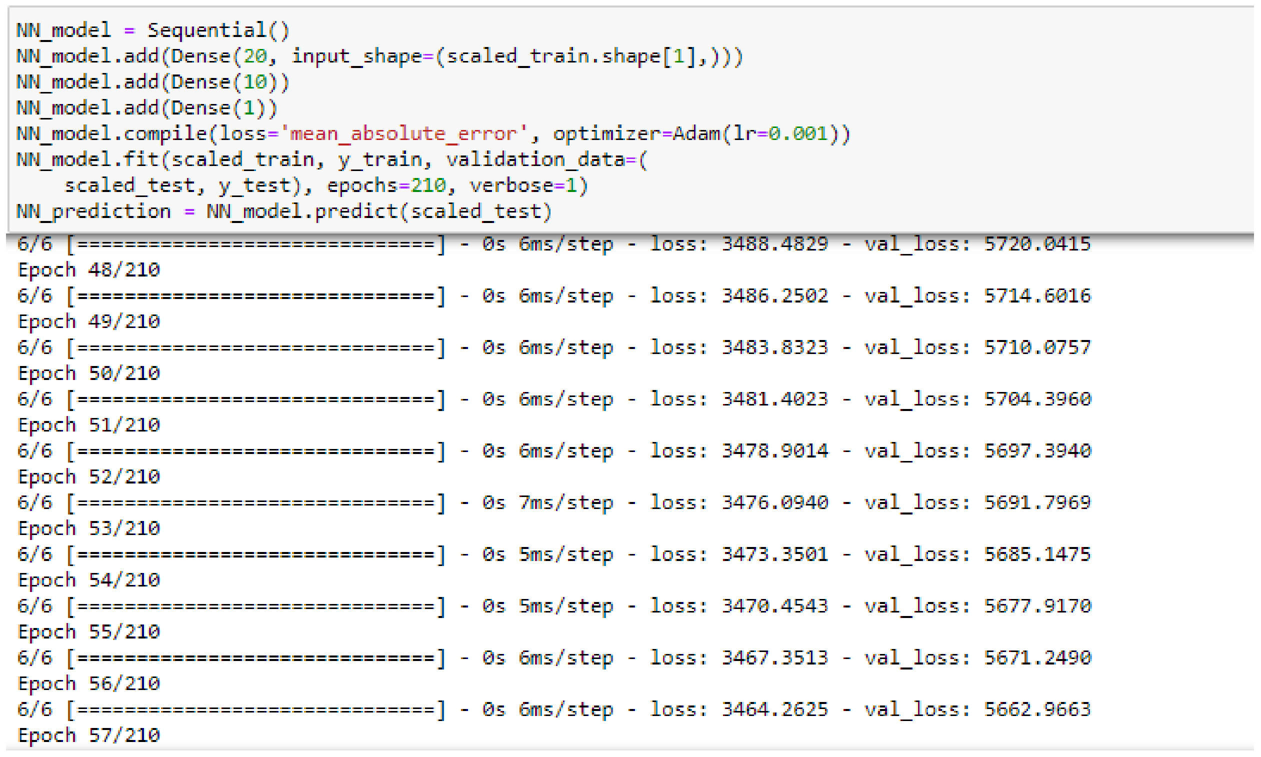
```
[0]	validation_0-rmse:3121.15405	validation_1-rmse:4338.40674
[1]	validation_0-rmse:2264.95044	validation_1-rmse:3135.70386
[2]	validation_0-rmse:1665.83826	validation_1-rmse:2322.55371
[3]	validation_0-rmse:1241.47046	validation_1-rmse:1676.83875
[4]	validation_0-rmse:942.85712	validation_1-rmse:1199.80676
[5]	validation_0-rmse:732.31842	validation_1-rmse:765.99835
[6]	validation_0-rmse:582.53571	validation_1-rmse:501.29828
```

Let us calculate the mean absolute error for the training.

```
print('XGBOOST MAE = ', mean_absolute_error(XGBOOST_prediction, y_test))
```

```
XGBOOST MAE =  392.1467982700893
```

CREATING AND TRAINING THE FEED FORWARD NEURAL NETWORK



```
NN_model = Sequential()
NN_model.add(Dense(20, input_shape=(scaled_train.shape[1],)))
NN_model.add(Dense(10))
NN_model.add(Dense(1))
NN_model.compile(loss='mean_absolute_error', optimizer=Adam(lr=0.001))
NN_model.fit(scaled_train, y_train, validation_data=(
    scaled_test, y_test), epochs=210, verbose=1)
NN_prediction = NN_model.predict(scaled_test)
```

```
6/6 [==============================] - 0s 6ms/step - loss: 3488.4829 - val_loss: 5720.0415
Epoch 48/210
6/6 [==============================] - 0s 6ms/step - loss: 3486.2502 - val_loss: 5714.6016
Epoch 49/210
6/6 [==============================] - 0s 6ms/step - loss: 3483.8323 - val_loss: 5710.0757
Epoch 50/210
6/6 [==============================] - 0s 6ms/step - loss: 3481.4023 - val_loss: 5704.3960
Epoch 51/210
6/6 [==============================] - 0s 6ms/step - loss: 3478.9014 - val_loss: 5697.3940
Epoch 52/210
6/6 [==============================] - 0s 7ms/step - loss: 3476.0940 - val_loss: 5691.7969
Epoch 53/210
6/6 [==============================] - 0s 5ms/step - loss: 3473.3501 - val_loss: 5685.1475
Epoch 54/210
6/6 [==============================] - 0s 5ms/step - loss: 3470.4543 - val_loss: 5677.9170
Epoch 55/210
6/6 [==============================] - 0s 6ms/step - loss: 3467.3513 - val_loss: 5671.2490
Epoch 56/210
6/6 [==============================] - 0s 6ms/step - loss: 3464.2625 - val_loss: 5662.9663
Epoch 57/210
```

COMPARING ALL MAE VALUES

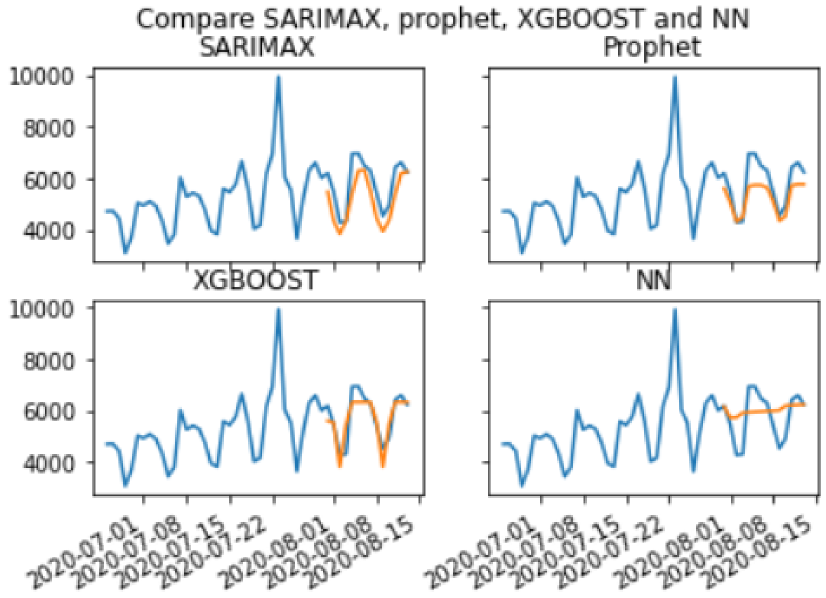
```
print('XGBOOST MAE = ', mean_absolute_error(XGBOOST_prediction, y_test))
print('Prophet MAE = ', mean_absolute_error(prophet_future, test))
print('SARIMAX MAE = ', mean_absolute_error(sarimax_prediction, test))
print('NN MAE = ', mean_absolute_error(NN_prediction, test))
```

```
XGBOOST MAE =  392.1467982700893
Prophet MAE =  572.5560073843841
SARIMAX MAE =  650.6548918305232
NN MAE =  722.49755859375
```

VISUALIZING ALL FOUR MODELS

```

XGBOOST_df = pd.DataFrame({'y': XGBOOST_prediction.tolist()})
XGBOOST_df.index = y_test.index

NN_df = pd.DataFrame(NN_prediction)
NN_df.index = y_test.index
plt.figure(figsize=(20, 20))
fig, axs = plt.subplots(2, 2)
fig.suptitle('Compare SARIMAX, prophet, XGBOOST and NN')
axs[0, 0].plot(dataset.tail(50))
axs[0, 0].plot(sarimax_prediction.tail(50))
axs[0, 0].set_title("SARIMAX")
axs[0, 1].plot(dataset.tail(50))
axs[0, 1].plot(prophet_future.tail(50))
axs[0, 1].set_title("Prophet")
axs[1, 0].plot(dataset.tail(50))
axs[1, 0].plot(XGBOOST_df.tail(50))
axs[1, 0].set_title("XGBOOST")
axs[1, 1].plot(dataset.tail(50))
axs[1, 1].plot(NN_df.tail(50))
axs[1, 1].set_title("NN")

for ax in fig.get_axes():
    ax.label_outer()
fig.autofmt_xdate()

plt.savefig('Comparison',
            bbox_inches='tight', transparent=False)
```

```
<Figure size 1440x1440 with 0 Axes>
```


Compare SARIMAX, prophet, XGBOOST and NN
SARIMAX
Prophet
XGBOOST
NN
10000
8000
6000
4000
2020-07-01
2020-07-08
2020-07-15
2020-07-22
2020-08-01
2020-08-08
2020-08-15