National College of Ireland

Configuration Manual

MSc Research Project
Programme Name

Pooja Singh
Student ID: X19234236

School of Computing
National College of Ireland

Supervisor: Aaloka Anant

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Pooja Singh

Student ID: X19234236

Programme: Data Analytics **Year:** 2022

Module: MSc Research Project

Lecturer: Aaloka Anant

Submission Due Date: 31/01/2022

Project Title: Configuration Manual

Word Count: 1347 **Page Count:** 12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Pooja Singh……

Date: 31st January 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Pooja Singh
X19234236

1 Hardware Requirements

The machine specifications used for this project execution are Windows 10 64-bit, and the processor is AMD Ryzen 5 4600HS, having Radeon Graphics 3.00 GHz, and 16 GB of RAM.

Figure 1: Hardware Configuration

2 Software Requirement

Python version 3.8 is used for all code areas in this research. We used Jupyter Notebook on Anaconda Navigator to build Python scripts and code. The Anaconda App must first be installed. The 64-bit version of the configuration file is downloaded as per the requirement and should be consistent to 64 bit Windows Operating System.

Figure 2: Anaconda Specification

After installation, select Start -> Anaconda Navigator. The Jupyter Lab is available directly from Anaconda Navigator.

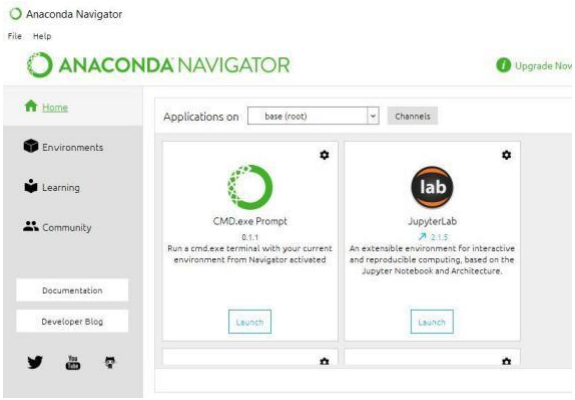


Figure 3: Anaconda Navigator

3 Python Library Package Requirement

The list of python package are installed using pip command in the python environmental command prompt,

- keras library
- matplotlib
- numpy
- pandas
- plotly
- Scikit-learn
- Statsmodels
- Sarimax
- sklearn
- Seaborn

4 Dataset Description

- The dataset for emission from different countries is obtained from Emissions Database for Global Atmospheric Research (EDGAR[1]) website.
- The csv file is taken and transposed for achieving the visualization.
- The dataset for carbon emission from different sector of power industry in United States is obtained from Energy Information Administration (EIA[2]) website.
- The dataset is extracted in the form of csv.

5 Data Preparation

EDGAR - Emissions Database for Global Atmospheric Research identifies the emitting nation and sector. The second dataset utilized in this study is publicly accessible for download and reuse on the EIA website and has no ethical considerations. Monthly CO2 emissions from 9 sectors from January 1973 to August 2021. (MtCO2). The data has 6 columns and 5256 rows, with NA values.

The first step is to import the libraries needed for data loading and preparation.

```
import itertools
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from sklearn.metrics import mean_absolute_error
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.tsa.api as smt
from statsmodels.tsa.statespace.sarimax import SARIMAX

warnings.filterwarnings('ignore')
```

Dataset 1 for visualizing the top emitter countries and sectors-

The specified dataset is then imported into the data frame known as “df”.

```
df = pd.read_csv('emission data.csv')
```

Due to the present dataframe's form, we will convert it to a vertical dataframe. This will make working with data and visualising pollution easier.

```
# Use the pandas melt function to change the horizontal dataframe to a vertical dataframe
df = pd.melt(
    frame=df,
    id_vars=['Country'],
    var_name='Year',
    value_name='Emission'
)
```

The redundant country information is removed.

[1] https://ourworldindata.org/co2-and-other-greenhouse-gas-emissions
[2] https://www.eia.gov/totalenergy/data/browser/?tbl=T11.06#/

```
# Check all the unique countries in the dataframe
df['Country'].unique()

# Filter out all the listings that are not countries and safe this in a list
drop_list = ['Africa', 'Americas (other)','Asia and Pacific (other)', 'EU-28', 'Europe (other)', 'World']

# Drop all the columns that are in our drop list
df = df[~df['Country'].isin(drop_list)]

# Check the output for success
df['Country'].unique()
```

Dataset 2 for predicting future emissions: This dataset contains monthly US atmospheric carbon dioxide (CO2) concentrations.

The dataset is first imported in a dataframe. To read the dataset as a time series, we use the read_csv. Convert the index to datetime, coerce errors, and filter NaT.

```
dateparse = lambda x: pd.to_datetime(x, format='%Y%m', errors = 'coerce')
df = pd.read_csv("Power_Sector_CO2_emission.csv", parse_dates=['YYYYMM'], index_col='YYYYMM', date_parser=dateparse)
df.head()
```

The dataset consists of data as shown below:

```
ts.head()
```

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	73.112	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	65.369	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	CLEIEUS	65.006	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	61.717	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	62.686	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

First, let's transform the emission value to a number.

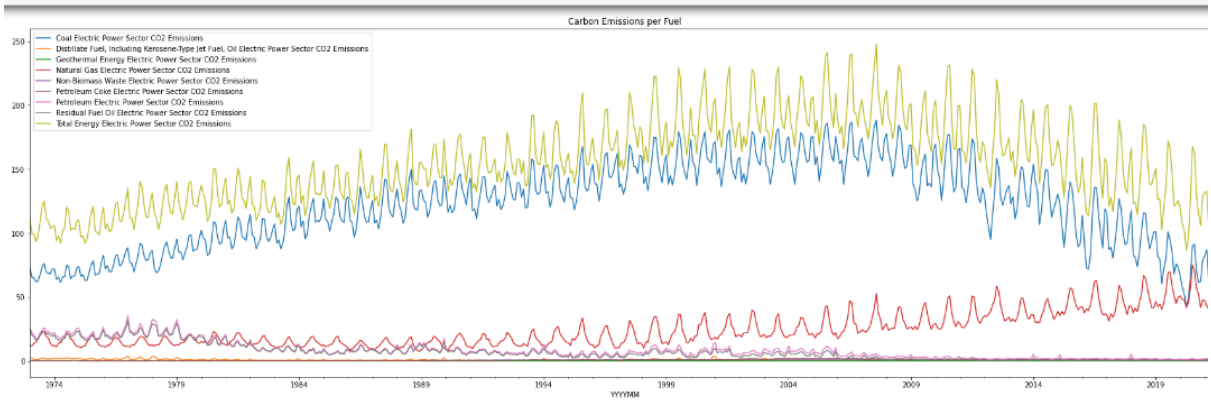
```
ts['Value'] = pd.to_numeric(ts['Value'] , errors='coerce')
ts.head()
```

Since 4872 observations include emissions values, we need to remove the empty rows.

```
ts.dropna(inplace = True)
```

Then the sector with highest emission is visualized.

```
fuels = data.groupby('Description')
fig, ax = plt.subplots(figsize=(30,9))
for desc, group in fuels:
    group.plot(x='YYYYMM', y='Value', label=desc, ax=ax, title='Carbon Emissions per Fuel')
    group.plot(x='YYYYMM', y='Value', title=desc,figsize=(30,9))
```



Each sector emission data can be stored in separate files by the below process, this research stores the sectors information with highest emissions.

```
Emissions = ts.iloc[:,1:]   # Monthly total emissions (mte)
Emissions= Emissions.groupby(['Description', pd.Grouper(freq = 'M')])['Value'].sum().unstack(level = 0)
mte = Emissions['Natural Gas Electric Power Sector CO2 Emissions'] # monthly total emissions (mte)
mte.head()
```

The information for every sector can be stored using to_csv function.

```
natural_gas_emission_ts.to_csv("natural_gas_emission.csv")
```

6 Model Preparation

6.1 ARIMA:

The data is initially put into Jupyter Notebook and parsed using the date parse function. Dickey Fuller test is used to test the stationarity of this time series.

A mull hypothesis that the time series is stationary is made, and then if the test statistic value comes out to be less than critical value then the null hypothesis is rejected.

```
def TestStationaryAdfuller(ts, cutoff = 0.01):
    ts_test = adfuller(ts, autolag = 'AIC')
    ts_test_output = pd.Series(ts_test[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])

    for key,value in ts_test[4].items():
        ts_test_output['Critical Value (%s)'%key] = value
    print(ts_test_output)

    if ts_test[1] <= cutoff:
        print("Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is
    else:
        print("Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary ")
```

Time series has a unit root, suggesting it is non-stationary.

```
Test Statistic                  1.127619
p-value                         0.995446
#Lags Used                     18.000000
Number of Observations Used    565.000000
Critical Value (1%)             -3.441977
Critical Value (5%)             -2.866669
Critical Value (10%)            -2.569502
dtype: float64
Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

Now the dataset is transformed to stationary. And the trend and seasonality is modelled and removed using decompose method.

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(mte)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(mte, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

The tuning parameters (p and q) of the model will be determined by looking at the autocorrelation and partial autocorrelation graphs. It shows how to read autocorrelation and partial autocorrelation graphs to pick parameters.

```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(mte_seasonal_first_difference.iloc[13:], lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(mte_seasonal_first_difference.iloc[13:], lags=40, ax=ax2)
```

Finding appropriate ARIMA model parameters via the graphical technique is difficult and time intensive. We will use grid search (hyperparameter optimization) to choose optimal parameter values.

```
p = d = q = range(0, 2) # Define the p, d and q parameters to take any value between 0 and 2
pdq = list(itertools.product(p, d, q)) # Generate all different combinations of p, q and q triplets
pdq_x_QDQs = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))] # Generate all different combinations of s
print('Examples of Seasonal ARIMA parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], pdq_x_QDQs[1]))
print('SARIMAX: {} x {}'.format(pdq[2], pdq_x_QDQs[2]))
```

Hyperparameter optimization is performed for enhanced performance.

```
p = d = q = range(0, 4)
pdq = list(itertools.product(p, d, q))

def arima_optimizer_aic(train, orders):
    best_aic, best_params = float("inf"), None
    for order in orders:
        try:
            arma_model_result = ARIMA(train, order).fit(disp=0)
            aic = arma_model_result.aic
            if aic < best_aic:
                best_aic, best_params = aic, order
            print('ARIMA%s AIC=%.2f' % (order, aic))
        except:
            continue
    print('Best ARIMA%s AIC=%.2f' % (best_params, best_aic))
    return best_params


best_params_aic = arima_optimizer_aic(train, pdq)
```

Final model implementation of ARIMA is as below.

```
############################
# Final Model
############################

arima_model = ARIMA(train, best_params_aic).fit(disp=0)
y_pred = arima_model.forecast(12)[0]
y_pred = pd.Series(y_pred, index=test.index)

plot_co2(train, test, y_pred, "ARIMA")
```

6.2 SARIMA:

Then the integration order I. (d). For series stabilization, d defines the lot of variations required. Finally, we include seasonality S(P, D, Q, s), where s is the season's duration. This component requires P and Q, which are the same as p and q except for the seasonal component. Use D to remove seasonality from a series. The seasonality and non stationarity behaviour has already been removed from this time series. Then the hyper parameter optimization is done for good performance.

```
model = SARIMAX(train, order=best_order, seasonal_order=best_seasonal_order)
sarima_final_model = model.fit(disp=0)

y_pred_test = sarima_final_model.get_forecast(steps=12)

# MAE
y_pred = y_pred_test.predicted_mean
y_pred = pd.Series(y_pred, index=test.index)

plot_co2(train, test, y_pred, "SARIMA")
```

The evaluation metric values are lower for SARIMA and are as below.

```
from sklearn.metrics import mean_squared_error
df['SArima']=y_pred.values
sarima_rmse_error = np.sqrt(mean_squared_error(test,y_pred))
sarima_mse_error = sarima_rmse_error**2
from sklearn.metrics import mean_absolute_error
sarima_mae = mean_absolute_error(test, y_pred)
sarima_mape = (np.mean(np.abs(test-y_pred)/test)*100)
print(f'MAE: {sarima_mae}')
print(f'MSE Error: {sarima_mse_error}\nRMSE Error: {sarima_rmse_error}')
print(f"MAPE: {sarima_mape}" )
```

```
MAE: 3.727696039171868
MSE Error: 18.934173860059087
RMSE Error: 4.351341616106358
MAPE: 7.960878589133895
```

6.3 LSTM (Long short-term memory)

First, MinMaxScaler is used to scale the data.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
scaler.fit(train_data)
scaled_train_data = scaler.transform(train_data)
scaled_test_data = scaler.transform(test_data)
```

Now, using keras library, the time series generator object is created since it will be required in lstm model.

```
from keras.preprocessing.sequence import TimeseriesGenerator

n_input = 12
n_features= 1
generator = TimeseriesGenerator(scaled_train_data, scaled_train_data, length=n_input, batch_size=1)
```

The model is implemented as below.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

lstm_model = Sequential()
lstm_model.add(LSTM(200, activation='relu', input_shape=(n_input, n_features)))
lstm_model.add(Dense(1))
lstm_model.compile(optimizer='adam', loss='mse')

lstm_model.summary()
lstm_model.fit_generator(generator,epochs=20)
```

```
Epoch 1/20
560/560 [==============================] - 3s 3ms/step - loss: 0.0228
Epoch 2/20
560/560 [==============================] - 2s 3ms/step - loss: 0.0100
Epoch 3/20
560/560 [==============================] - 2s 3ms/step - loss: 0.0073
Epoch 4/20
560/560 [==============================] - 2s 3ms/step - loss: 0.0032
Epoch 5/20
560/560 [==============================] - 2s 3ms/step - loss: 0.0029
```

Because we scaled our data, we must reverse it to see accurate forecasts.

```
lstm_predictions = scaler.inverse_transform(lstm_predictions_scaled)
```

A new column is created in test_data and the lstm_predictions are saved in the newly created column for comparison of projected values with other models.

```
test_data['LSTM_Predictions'] = lstm_predictions
```

The results are as follows.

```
lstm_rmse_error = rmse(test_data['Value'], test_data["LSTM_Predictions"])
lstm_mse_error = lstm_rmse_error**2
lstm_mae = mean_absolute_error(test_data['Value'], test_data["LSTM_Predictions"])
lstm_mape = (np.mean(np.abs(test_data['Value']-test_data["LSTM_Predictions"])/test_data['Value'])*100)
print(f'MAE: {lstm_mae}')
print(f'MSE Error: {lstm_mse_error}\nRMSE Error: {lstm_rmse_error}')
print(f"MAPE : {lstm_mape}" )
```

```
MAE: 7.777783396273847
MSE Error: 66.94487937099214
RMSE Error: 8.181985050768068
MAPE : 16.00763936490417
```

6.4 Prophet

For Python, have used the fbprophet module. NA When the date parse() function changes the year column 'YYYMM' to datetime, the data is erased. Change the column names "YYYYMM" for years and "Value" for values so the Prophet model can forecast. In this case, the model forecasts monthly. Predict may be used to calculate emission values for future dates from a dataframe built using the.make future dataframe() function.

```
pip install pystan
```

```
Requirement already satisfied: pystan in c:\softwares\lib\site-packages (2.19.1.1)
Requirement already satisfied: Cython!=0.25.1,>=0.22 in c:\softwares\lib\site-packages (from pystan) (0.29.21)
Requirement already satisfied: numpy>=1.7 in c:\softwares\lib\site-packages (from pystan) (1.19.2)
Note: you may need to restart the kernel to use updated packages.
```

```
pip install fbprophet
```

```
Requirement already satisfied: fbprophet in c:\softwares\lib\site-packages (0.7.1)
Requirement already satisfied: numpy>=1.15.4 in c:\softwares\lib\site-packages (from fbprophet) (1.19.2)
Requirement already satisfied: matplotlib>=2.0.0 in c:\softwares\lib\site-packages (from fbprophet) (3.3.2)
Requirement already satisfied: convertdate>=2.1.2 in c:\softwares\lib\site-packages (from fbprophet) (2.3.2)
Requirement already satisfied: LunarCalendar>=0.0.9 in c:\softwares\lib\site-packages (from fbprophet) (0.0.9)
Requirement already satisfied: python-dateutil>=2.8.0 in c:\softwares\lib\site-packages (from fbprophet) (2.8.1)
```

Column names are given as ‘ds’ and ‘y’

```
df_pr.columns = ['ds','y'] # To use prophet column names should be like that
```

Prophet is imported used fbprophet library.

```
from fbprophet import Prophet
```

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

Making future projections by using the implemented prophet model.

```
future = m.make_future_dataframe(periods=12,freq='MS')
prophet_pred = m.predict(future)
```

Setting other parameters for predicted values.

```
prophet_pred = pd.DataFrame({"Date" : prophet_pred[-12:]['ds'], "Pred" : prophet_pred[-12:]["yhat"]})
```

```
prophet_pred = prophet_pred.set_index("Date")
```

```
prophet_pred.index.freq = "MS"
```

```
prophet_pred
```

Saving predictions for future comparison.

```
test_data["Prophet_Predictions"] = prophet_pred['Pred'].values
```

The values of error metrics are as below.

```
prophet_rmse_error = rmse(test_data['Value'], test_data["Prophet_Predictions"])
prophet_mse_error = prophet_rmse_error**2
mean_value = df['Value'].mean()
from sklearn.metrics import mean_absolute_error
prophet_mae = mean_absolute_error(test_data['Value'], test_data["Prophet_Predictions"])
prophet_mape = (np.mean(np.abs(test_data['Value']-test_data["Prophet_Predictions"])/test_data['Value'])*100)
print(f'MAE: {prophet_mae}')
print(f'MSE Error: {prophet_mse_error}\nRMSE Error: {prophet_rmse_error}')
print(f"MAPE: {prophet_mape}" )
```

```
MAE: 4.107523728623316
MSE Error: 25.183780349223333
RMSE Error: 5.018344383282532
MAPE: 8.846700427513671
```

6.5 Exponential Smoothing

For data with a systematic trend or seasonal component, exponential smoothing is a time series forecasting approach for univariate data.

Other exponential smoothing model hyperparameters can also be automatically optimized, such as whether to model the trend and seasonal component, and if so, whether to model them additively or multiplicatively.

Triple Exponential Smoothing approach uses level, trend, and seasonality to forecast. First, it is optimized as below.

```
def tes_optimizer(train, abg, step=12):
    best_alpha, best_beta, best_gamma, best_mae = None, None, None, float("inf")

    for comb in abg:
        tes_model = ExponentialSmoothing(train, trend="add", seasonal="add", seasonal_periods=12).\
            fit(smoothing_level=comb[0], smoothing_slope=comb[1], smoothing_seasonal=comb[2])
        y_pred = tes_model.forecast(step)
        mae = mean_absolute_error(test, y_pred)
        if mae < best_mae:
            best_alpha, best_beta, best_gamma, best_mae = comb[0], comb[1], comb[2], mae
        print([round(comb[0], 2), round(comb[1], 2), round(comb[2], 2), round(mae, 2)])

    print("best_alpha:", round(best_alpha, 2), "best_beta:", round(best_beta, 2), "best_gamma:", round(best_gamma, 2),
          "best_mae:", round(best_mae, 4))

    return best_alpha, best_beta, best_gamma, best_mae


alphas = betas = gammas = np.arange(0.10, 1, 0.20)
abg = list(itertools.product(alphas, betas, gammas))

best_alpha, best_beta, best_gamma, best_mae = tes_optimizer(train, abg)
```

```
############################
# Final TES Model
############################

final_tes_model = ExponentialSmoothing(train, trend="add", seasonal="add", seasonal_periods=12).
            fit(smoothing_level=best_alpha, smoothing_slope=best_beta, smoothing_seasonal=best_gamma)

y_pred = final_tes_model.forecast(12)

plot_co2(train, test, y_pred, "Triple Exponential Smoothing")
```

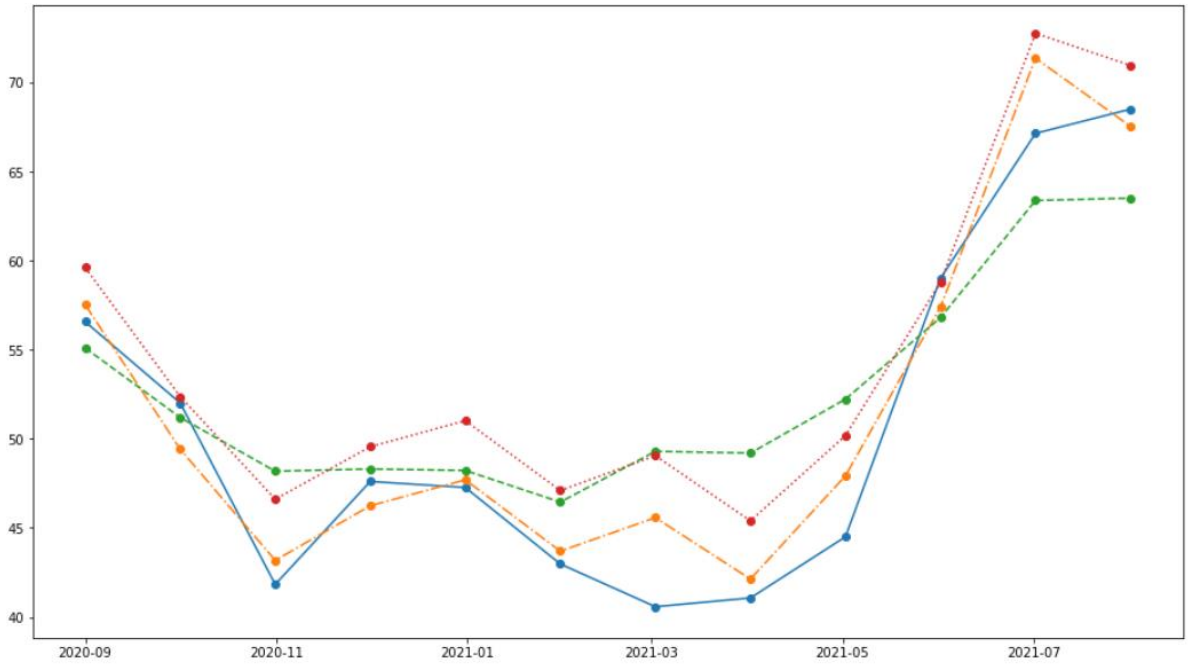
The values of MAE, RMSE and MAPE are as follow.

```
from sklearn.metrics import mean_squared_error
df['TES']=y_pred.values
tes_rmse_error = np.sqrt(mean_squared_error(test,y_pred))
tes_mse_error = tes_rmse_error**2
from sklearn.metrics import mean_absolute_error
tes_mae = mean_absolute_error(test, y_pred)
tes_mape = (np.mean(np.abs(test-y_pred)/test)*100)
print(f'MAE: {tes_mae}')
print(f'MSE Error: {tes_mse_error}\nRMSE Error: {tes_rmse_error}')
print(f"MAPE: {tes_mape}" )
```

```
MAE: 1.974551017064026
MSE Error: 5.929473256682219
RMSE Error: 2.435050976197874
MAPE: 4.030616903290366
```

After comparing all the models, it is evident that the Triple Exponential Model performed the best with projections being the closest to actual values as also seen from the graph below. The comparison is made between the top 3 performing models.

```
plt.figure(figsize=(16,9))
plt.plot_date(df.index, df["Value"], linestyle="-")
plt.plot_date(df.index, df["TES"], linestyle="-.")
plt.plot_date(df.index, df["Prophet"], linestyle="--")
plt.plot_date(df.index, df["SArima"], linestyle=":")
plt.show()
```



As we can see here the Triple exponential model follows a trajectory very similar to actual values. The MAE is as low as 1.97 for this model. Hence the forecasting is done using this method in this research.