**Introduction**

This project deals with prediction of consumption of electricity in the coming future. Time series analysis is important to know trend seasonality and other characteristics of the data for efficient modelling. The Dataset have two columns “Date” and “IPG221A2N”. This Columns shows the Actual Production of electricity throughout the Dates. Now we must analyse the given data and Time Series Model to predict the future Electricity production of the Country.

**Code Explain:**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal\_decompose

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

import warnings

warnings.filterwarnings("ignore")

In the above imported the libraries or python module for the further analysis.

df=pd.read\_csv(r"/content/drive/MyDrive/Electric\_Production.csv",index\_col="DATE", parse\_dates=True)

In This above column I`ve gave the path to the Dataset and the **“DATE”** column as index.

df = df.rename(columns={"IPG2211A2N":"EP"})

Its very hard to understand the meaning of the name of column such as **“IPD2211A2N”** so I`ve renamed the column to **“EP”** means *Electric Production*.

sns.boxplot(df["EP"],orient="h",color="g")

plt.title("Electric Production")

The Boxplot is plotted to check any outliers in the column of Electric Production.

df.info()

Shows the information about the dataset.

df.plot(figsize=(14,5))

plt.xlabel("Date")

plt.ylabel("Electric Production")

plt.show()

I`ve Plotting the graph Date vs Electric production through lineplot.

df.index.freq = "MS"

result = seasonal\_decompose(df["EP"])

result.plot();

**df.index.freq = "MS"** sets the frequency of the DataFrame index to "MS", which stands for Month Start. This means that each entry in the index represents the start of a month.

The **seasonal\_decompose** function is then applied to decompose the time series data in the "EP" column of the DataFrame **df** into its constituent components: trend, seasonality, and residuals.

Finally, **result.plot()** is used to visualize the decomposed time series data, showing the original series along with its trend, seasonality, and residual components. This visualization helps in understanding the underlying patterns and fluctuations in the time series data.

train=df.iloc[:325]

test= df.iloc[325:]

len(train)

len(test)

Splitting the Dataset into Train and Test and checking the length of Train and Test.

scaler= MinMaxScaler()

scaler.fit(train)

scaled\_train = scaler.transform(train)

scaled\_test = scaler.transform(test)

1. **scaler = MinMaxScaler()**: This line creates an instance of the MinMaxScaler class, which will be used to scale the features of the dataset to a specified range.
2. **scaler.fit(train)**: This line fits the scaler to the training data, computing the minimum and maximum values of each feature in the training set.
3. **scaled\_train = scaler.transform(train)**: This line applies the scaling transformation to the training data, transforming each feature to a scaled version according to the computed minimum and maximum values.
4. **scaled\_test = scaler.transform(test)**: This line applies the same scaling transformation to the test data, using the same minimum and maximum values computed from the training data.

n\_input = 12

generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)

len(generator)

X, y = generator[0]

X.ravel(), y

1. **n\_input = 12**: This variable specifies the number of lag observations to use as input for predicting the next time step. In this case, the model will use the previous 12 time steps as input features to predict the next time step.
2. **generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)**: This line creates an instance of the **TimeseriesGenerator** class. It takes the following arguments:
   * **data**: The input time series data, which is the scaled training data.
   * **targets**: The target time series data, which is also the scaled training data in this case. This parameter is used when you want to predict a different time series than the input data, but in this case, we're predicting the same series.
   * **length**: The number of lag observations to use as input for each sample, which is set to **n\_input**.
   * **batch\_size**: The number of samples to include in each batch, which is set to 1. This means that each batch will contain one input-output pair.
3. **len(generator)**: This line calculates the total number of batches that the generator will yield. Each batch represents a sequence of input-output pairs based on the specified length and batch size.
4. **X, y = generator[0]**: This line retrieves the first batch of input-output pairs from the generator. **X** contains the input features (lag observations) for the first sample, and **y** contains the corresponding target value (the next time step) for prediction.
5. **X.ravel(), y**: This line flattens the input features **X** into a 1D array using the **ravel()** method and returns both the flattened input features and the target value **y**.

model=Sequential()

model.add(LSTM(1000, activation="relu", input\_shape=(12,1)))

model.add(Dense(1))

model.compile(optimizer="adam", loss="mse")

model.fit(generator, epochs= 40)

1. **model = Sequential()**: This line initializes a sequential model, which is a linear stack of layers. In this model, layers are added sequentially, one on top of the other.
2. **model.add(LSTM(1000, activation="relu", input\_shape=(12,1)))**: This line adds an LSTM layer to the model. The arguments passed to the LSTM layer are as follows:
   * **1000**: This specifies the number of LSTM units (also known as neurons or cells) in the layer. Increasing the number of units allows the model to learn more complex patterns in the data.
   * **activation="relu"**: This specifies the activation function used by the LSTM units. In this case, the Rectified Linear Unit (ReLU) activation function is used.
   * **input\_shape=(12,1)**: This specifies the shape of the input data expected by the LSTM layer. The input shape is a tuple **(12, 1)**, indicating that each input sample has 12 time steps (lag observations) and each time step has 1 feature.
3. **model.add(Dense(1))**: This line adds a dense (fully connected) layer to the model with a single neuron. This layer is used to output the predicted value for each input sequence.
4. **model.compile(optimizer="adam", loss="mse")**: This line compiles the model. The arguments passed to the **compile** method are:
   * **optimizer="adam"**: This specifies the optimization algorithm used during training. Adam is a popular optimization algorithm known for its efficiency and effectiveness in training neural networks.
   * **loss="mse"**: This specifies the loss function used to measure the difference between the predicted values and the actual values. Mean Squared Error (MSE) is a common loss function used for regression problems.
5. **model.fit(generator, epochs=40)**: This line trains the model using the **fit** method. The **fit** method takes the following arguments:
   * **generator**: This is the data generator object that yields batches of input-output pairs generated from the time series data.
   * **epochs=40**: This specifies the number of training epochs, i.e., the number of times the model will be trained on the entire dataset.

loss= model.history.history["loss"]

plt.plot(loss)

plt.xlabel("Epoches")

plt.ylabel("loss")

plt.title("Epoches vs Loss")

plt.show()

Above graph plot the Loss Vs Epochs of the model. How much loss decrease’s as per epochs.

last\_train\_batch= scaled\_train[-12:]

last\_train\_batch

last\_train\_batch= last\_train\_batch.reshape((1,12,1))

last\_train\_batch

model.predict(last\_train\_batch)

scaled\_test[0]

1. **last\_train\_batch = scaled\_train[-12:]**: This line selects the last 12 scaled values from the training data, representing the most recent observations. These values will be used as input to generate predictions for the first time step in the test data.
2. **last\_train\_batch**: This line displays the selected last 12 scaled values from the training data.
3. **last\_train\_batch = last\_train\_batch.reshape((1, 12, 1))**: This line reshapes the **last\_train\_batch** array to match the input shape expected by the LSTM model. The reshaping is done to add an extra dimension to represent the batch size (which is 1 in this case), resulting in a shape of **(1, 12, 1)**.
4. **last\_train\_batch**: This line displays the reshaped **last\_train\_batch** array.
5. **model.predict(last\_train\_batch)**: This line generates predictions for the next time step using the trained LSTM model. The reshaped **last\_train\_batch** array is passed as input to the model, and it returns the predicted scaled value for the next time step.
6. **scaled\_test[0]**: This line displays the actual scaled value for the next time step in the test data.

test\_pred = []

first\_eval\_batch = scaled\_train[-12:]

current\_batch = first\_eval\_batch.reshape((1, 12, 1))

for i in range(len(test)):

  current\_pred = model.predict(current\_batch)[0]

  test\_pred.append(current\_pred)

  current\_batch = np.append(current\_batch[:,1:,:], [[current\_pred]], axis=1)

This loop iteratively predicts the scaled values for each time step in the test data, updating the current batch at each iteration to include the most recent prediction. Finally, the list **test\_pred** contains the predicted scaled values for the entire test dataset.

true\_pred = scaler.inverse\_transform(test\_pred)

test["predicted production"]= true\_pred

test.head()

The Above code convert the Predicted test data Into Inverse transformation and create the column in the DataFrame **“Predicted Production”**

sns.lineplot(test)

plt.xlabel("Date")

plt.ylabel("Electric Production")

plt.show()

The code plots the graph of *Electricity Production vs Date* of **Predicted Electricity production vs Actual Electric Production**

rsme= np.sqrt(mean\_squared\_error(test["EP"], test["predicted production"]))

rsme

The last Cell is for the Root Mean Squared Error of the Model of LSTM Time Series.

**Conclusion:**

We have successfully developed the LSTM model for Time Series analysis on the Electricity production of the Country the Root Mean Squared Error is about 7.49. That is means the Model perform well with our dataset and achieved the good accuracy.