```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings

warnings.filterwarnings('ignore')

data = pd.read_csv('energy_dataset.csv', parse_dates = ['time'])
data.head()
```

{"type":"dataframe","variable_name":"data"}

```python
data.time = pd.to_datetime(data.time, utc = True,
infer_datetime_format= True)
data = data.set_index('time')
data.head()
```

{"type":"dataframe","variable_name":"data"}

```python
data.isnull().sum()
```

```
generation biomass                                 19
generation fossil brown coal/lignite               18
generation fossil coal-derived gas                 18
generation fossil gas                              18
generation fossil hard coal                        18
generation fossil oil                              19
generation fossil oil shale                        18
generation fossil peat                             18
generation geothermal                              18
generation hydro pumped storage aggregated      35064
generation hydro pumped storage consumption        19
generation hydro run-of-river and poundage         19
generation hydro water reservoir                   18
generation marine                                  19
generation nuclear                                 17
generation other                                   18
generation other renewable                         18
generation solar                                   18
generation waste                                   19
generation wind offshore                           18
generation wind onshore                            18
forecast solar day ahead                            0
forecast wind offshore eday ahead               35064
forecast wind onshore day ahead                     0
total load forecast                                 0
total load actual                                  36
price day ahead                                     0
price actual                                        0
dtype: int64
```

```python
# Count number of zeros in all columns of Dataframe
for column_name in data.columns:
    column = data[column_name]
    # Get the count of Zeros in column
    count = (column == 0).sum()
    print(f"{column_name:{50}} : {count}")
```

```
generation biomass                                  : 4
generation fossil brown coal/lignite                : 10517
generation fossil coal-derived gas                  : 35046
generation fossil gas                               : 1
generation fossil hard coal                         : 3
generation fossil oil                               : 3
generation fossil oil shale                         : 35046
generation fossil peat                              : 35046
generation geothermal                               : 35046
generation hydro pumped storage aggregated          : 0
generation hydro pumped storage consumption         : 12607
generation hydro run-of-river and poundage          : 3
generation hydro water reservoir                    : 3
generation marine                                   : 35045
generation nuclear                                  : 3
generation other                                    : 4
generation other renewable                          : 3
generation solar                                    : 3
generation waste                                    : 3
generation wind offshore                            : 35046
generation wind onshore                             : 3
forecast solar day ahead                            : 539
forecast wind offshore eday ahead                   : 0
forecast wind onshore day ahead                     : 0
total load forecast                                 : 0
total load actual                                   : 0
price day ahead                                     : 0
price actual                                        : 0
```

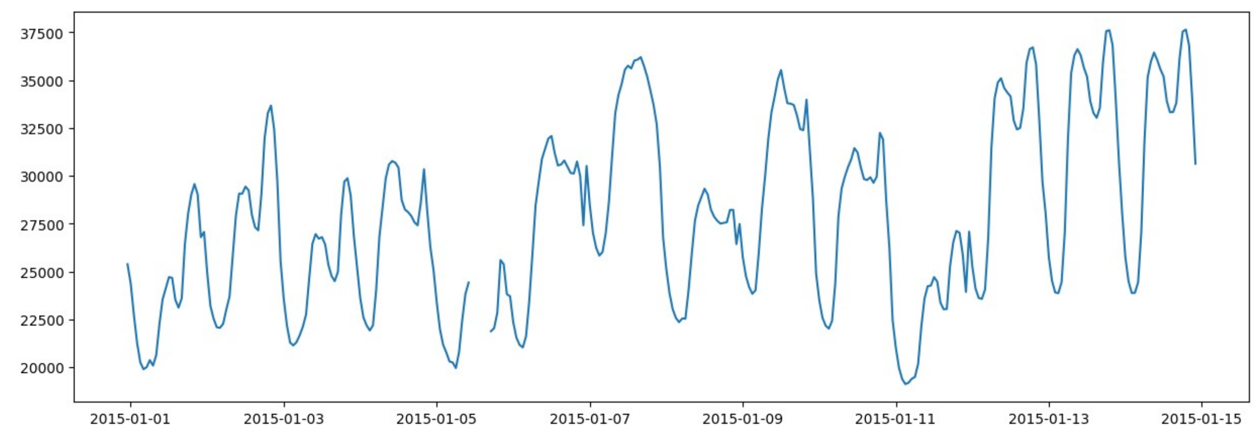
```python
data.drop(['generation hydro pumped storage aggregated', 'forecast 
wind offshore eday ahead',
           'generation wind offshore', 'generation fossil coal-derived 
gas',
           'generation fossil oil shale', 'generation fossil peat', 
'generation marine',
           'generation wind offshore', 'generation geothermal'], 
inplace = True, axis = 1)

data.isnull().sum()
```

```
generation biomass                              19
generation fossil brown coal/lignite            18
generation fossil gas                           18
```

```
generation fossil hard coal                        18
generation fossil oil                              19
generation hydro pumped storage consumption        19
generation hydro run-of-river and poundage         19
generation hydro water reservoir                   18
generation nuclear                                 17
generation other                                   18
generation other renewable                         18
generation solar                                   18
generation waste                                   19
generation wind onshore                            18
forecast solar day ahead                            0
forecast wind onshore day ahead                     0
total load forecast                                 0
total load actual                                  36
price day ahead                                     0
price actual                                        0
dtype: int64
```

```python
plt.rcParams['figure.figsize'] = (15, 5)
plt.plot(data['total load actual'][:24*7*2])
```

```
[<matplotlib.lines.Line2D at 0x7c4025abda50>]
```




```python
# Linear Interpolate the missing values in the dataset
data.interpolate(method='linear', limit_direction='forward',
inplace=True, axis=0)
data.isnull().sum()
```

```
generation biomass                                 0
generation fossil brown coal/lignite               0
generation fossil gas                              0
generation fossil hard coal                        0
generation fossil oil                              0
generation hydro pumped storage consumption        0
generation hydro run-of-river and poundage         0
```

```
generation hydro water reservoir                 0
generation nuclear                               0
generation other                                 0
generation other renewable                       0
generation solar                                 0
generation waste                                 0
generation wind onshore                          0
forecast solar day ahead                         0
forecast wind onshore day ahead                  0
total load forecast                              0
total load actual                                0
price day ahead                                  0
price actual                                     0
dtype: int64
```

```
data.describe()
```

{"summary":"{\n \"name\": \"data\",\n \"rows\": 8,\n \"fields\": [\
n   {\n   \"column\": \"generation biomass\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
12287.643775014392,\n   \"min\": 0.0,\n   \"max\": 35064.0,\
n   \"num_unique_values\": 8,\n   \"samples\": [\n
383.53134268765683,\n   367.0,\n   35064.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"generation fossil brown
coal/lignite\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": 12246.791587740387,\n   \"min\":
0.0,\n   \"max\": 35064.0,\n   \"num_unique_values\": 7,\n
\"samples\": [\n   35064.0,\n   448.0945699292722,\n
757.0\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"generation fossil gas\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 11838.935109591645,\n
\"min\": 0.0,\n   \"max\": 35064.0,\n
\"num_unique_values\": 8,\n   \"samples\": [\n
5622.7006473876345,\n   4969.5,\n   35064.0\n   ],\
n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   },\n   {\n   \"column\": \"generation fossil hard coal\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
11299.94655238462,\n   \"min\": 0.0,\n   \"max\": 35064.0,\n
\"num_unique_values\": 8,\n   \"samples\": [\n
4256.531271389459,\n   4475.0,\n   35064.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"generation fossil oil\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
12312.375692604463,\n   \"min\": 0.0,\n   \"max\": 35064.0,\
n   \"num_unique_values\": 8,\n   \"samples\": [\n
298.34241672370524,\n   300.0,\n   35064.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"generation hydro pumped storage

```
consumption\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 12162.557674775831,\n        \"min\":
0.0,\n        \"max\": 35064.0,\n        \"num_unique_values\": 7,\n
\"samples\": [\n          35064.0,\n          475.5827059091946,\n
616.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"generation hydro run-of-river and poundage\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
12100.12231169857,\n        \"min\": 0.0,\n        \"max\": 35064.0,\n
\"num_unique_values\": 8,\n        \"samples\": [\n
972.2019022359115,\n          906.0,\n          35064.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"generation hydro water
reservoir\",\n      \"properties\": {\n        \"dtype\": \"number\",\
n        \"std\": 11704.503848110502,\n        \"min\": 0.0,\n
\"max\": 35064.0,\n        \"num_unique_values\": 8,\n
\"samples\": [\n          2605.5341233173626,\n          2165.0,\n
35064.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"generation nuclear\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 11063.898652933607,\n        \"min\":
0.0,\n        \"max\": 35064.0,\n        \"num_unique_values\": 8,\n
\"samples\": [\n          6263.483430298882,\n          6564.0,\n
35064.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"generation other\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 12378.024931525102,\n        \"min\":
0.0,\n        \"max\": 35064.0,\n        \"num_unique_values\": 8,\n
\"samples\": [\n          60.22602954597308,\n          57.0,\n
35064.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"generation other renewable\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 12372.986131487201,\n
\"min\": 0.0,\n        \"max\": 35064.0,\n
\"num_unique_values\": 8,\n        \"samples\": [\n
85.63432580424367,\n          88.0,\n          35064.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"generation solar\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
11928.667629518644,\n        \"min\": 0.0,\n        \"max\": 35064.0,\
n        \"num_unique_values\": 8,\n        \"samples\": [\n
1432.818546087155,\n          616.0,\n          35064.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"generation waste\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
12321.589064303862,\n        \"min\": 0.0,\n        \"max\": 35064.0,\
n        \"num_unique_values\": 8,\n        \"samples\": [\n
269.418691535478,\n          279.0,\n          35064.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
```

n   },\n   {\n   \"column\": \"generation wind onshore\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 11536.87913038231,\n   \"min\": 0.0,\n   \"max\": 35064.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   5464.980450034223,\n   4849.5,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"forecast solar day ahead\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 11928.829092827906,\n   \"min\": 0.0,\n   \"max\": 35064.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   1439.0667351129364,\n   576.0,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"forecast wind onshore day ahead\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 11508.353339672747,\n   \"min\": 237.0,\n   \"max\": 35064.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   5471.21668948209,\n   4855.0,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"total load forecast\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 11287.094586882546,\n   \"min\": 4594.100854174024,\n   \"max\": 41390.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   28712.129962354553,\n   28906.0,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"total load actual\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 11224.55004891461,\n   \"min\": 4575.828853961431,\n   \"max\": 41015.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   28698.281385466576,\n   28902.0,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"price day ahead\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 12380.815380727721,\n   \"min\": 2.06,\n   \"max\": 35064.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   49.87434120465434,\n   50.52,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"price actual\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 12378.171493557733,\n   \"min\": 9.33,\n   \"max\": 35064.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   57.88402292950034,\n   58.02,\n   35064.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}

```
data.columns
```

```
Index(['generation biomass', 'generation fossil brown coal/lignite',
       'generation fossil gas', 'generation fossil hard coal',
```

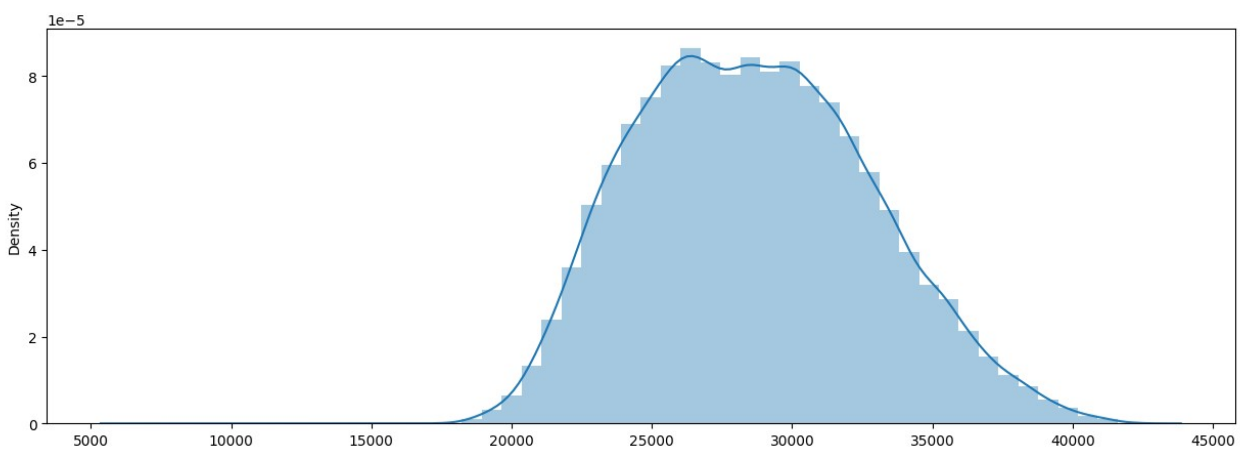
```
       'generation fossil oil', 'generation hydro pumped storage
consumption',
       'generation hydro run-of-river and poundage',
       'generation hydro water reservoir', 'generation nuclear',
       'generation other', 'generation other renewable', 'generation
solar',
       'generation waste', 'generation wind onshore',
       'forecast solar day ahead', 'forecast wind onshore day ahead',
       'total load forecast', 'total load actual', 'price day ahead',
       'price actual'],
      dtype='object')
```

```
# creating a new column to sum the total Generationof power
data['total generation'] = data['generation biomass'] +
data['generation fossil brown coal/lignite'] + data['generation fossil
gas'] + data['generation fossil hard coal'] + data['generation fossil
oil'] + data['generation hydro pumped storage consumption'] +
data['generation hydro run-of-river and poundage'] + data['generation
hydro water reservoir'] + data['generation nuclear'] +
data['generation other'] + data['generation other renewable'] +
data['generation solar'] + data['generation waste'] + data['generation
wind onshore']


data.head()
```

{"type":"dataframe","variable_name":"data"}

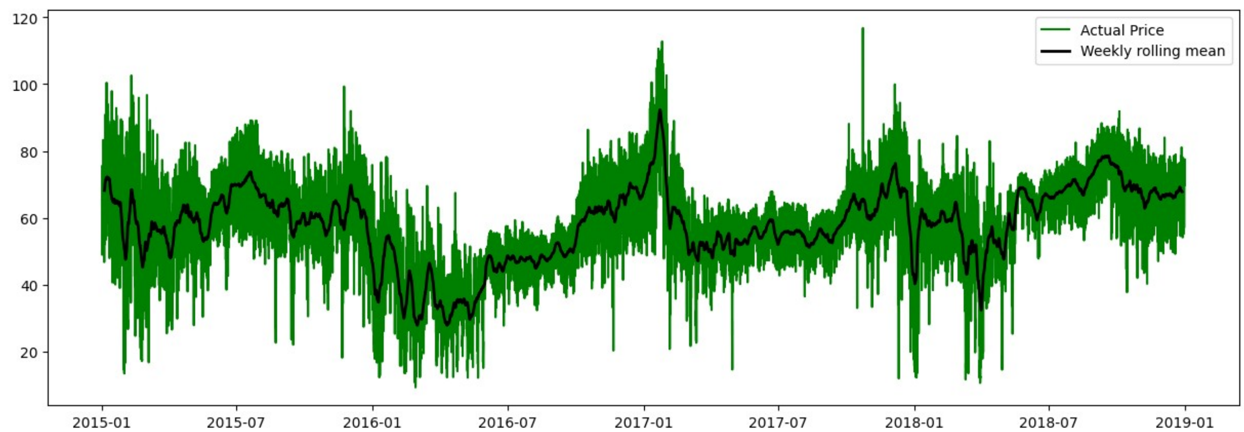
```
# Total Generation
sns.distplot(x= data['total generation'], kde = True)
```

```
<Axes: ylabel='Density'>
```

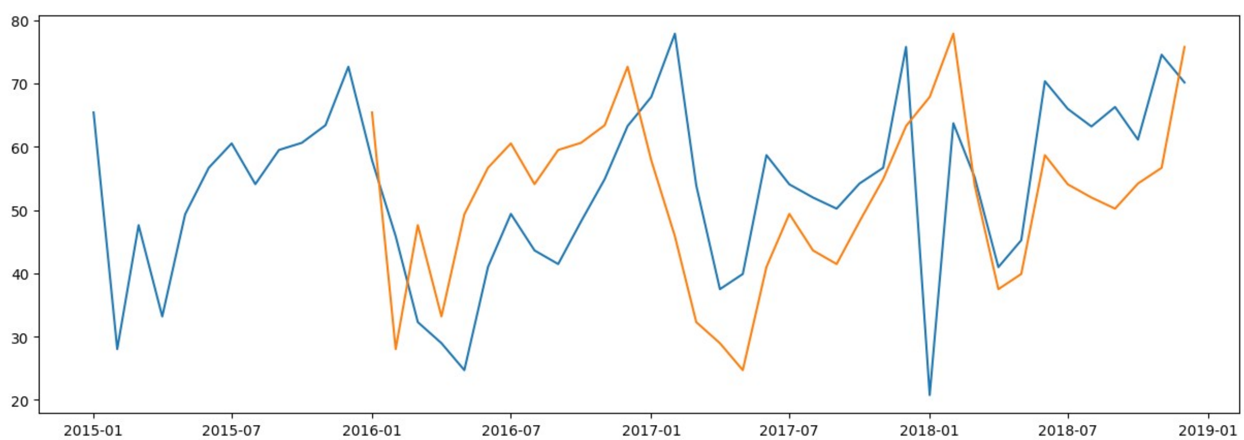



```
#Ploting the actual hourly electricity price and its rolling mean over
a week
fig, ax = plt.subplots(1,1)
```

```
rolling = data['price actual'].rolling(24*7, center = True).mean()
ax.plot(data['price actual'], color = 'g', label='Actual Price')
ax.plot(rolling, color = 'black', linestyle='-', linewidth=2,
label='Weekly rolling mean')
plt.legend()
plt.show()
```



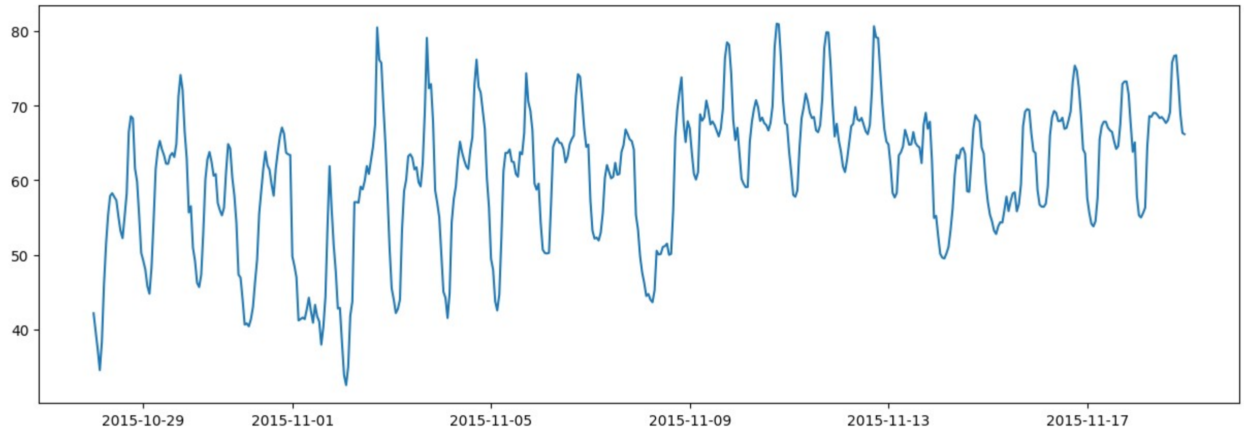

```
# Plot the electricity prie (month wise) along with 1st year lagg
monthly_price = data['price actual'].asfreq('M')
lagged = monthly_price.shift(12)

fig, ax = plt.subplots(1,1)
ax.plot(monthly_price, label = 'Monthly Price')
ax.plot(lagged, label ='1 yr lagged')
plt.plot()

[]
```




```
# Plotting hourly data of 3 weeks
start = 1+ 24*300
end = 1+ 24*322
```

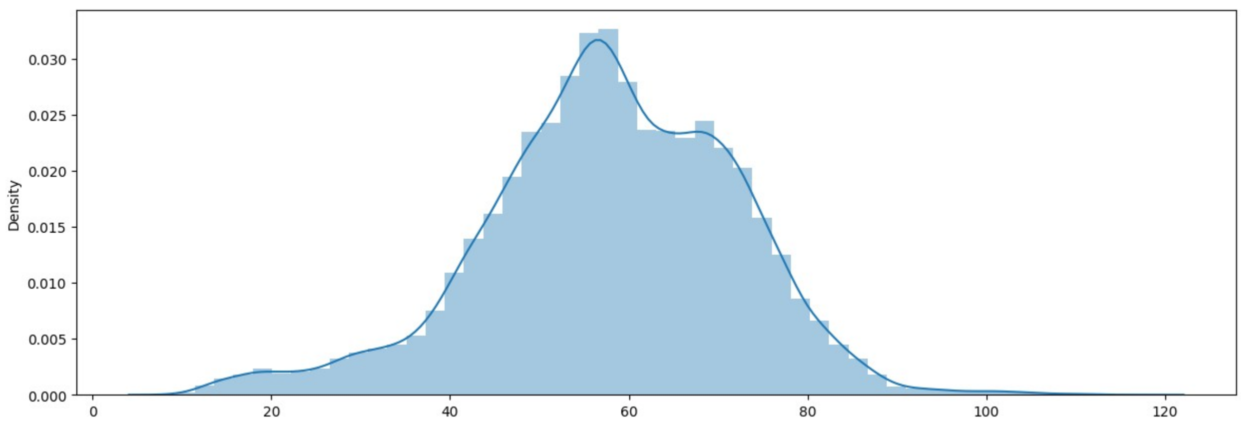
```
plt.plot(data['price actual'][start:end])
```

```
[<matplotlib.lines.Line2D at 0x7c401cf63700>]
```

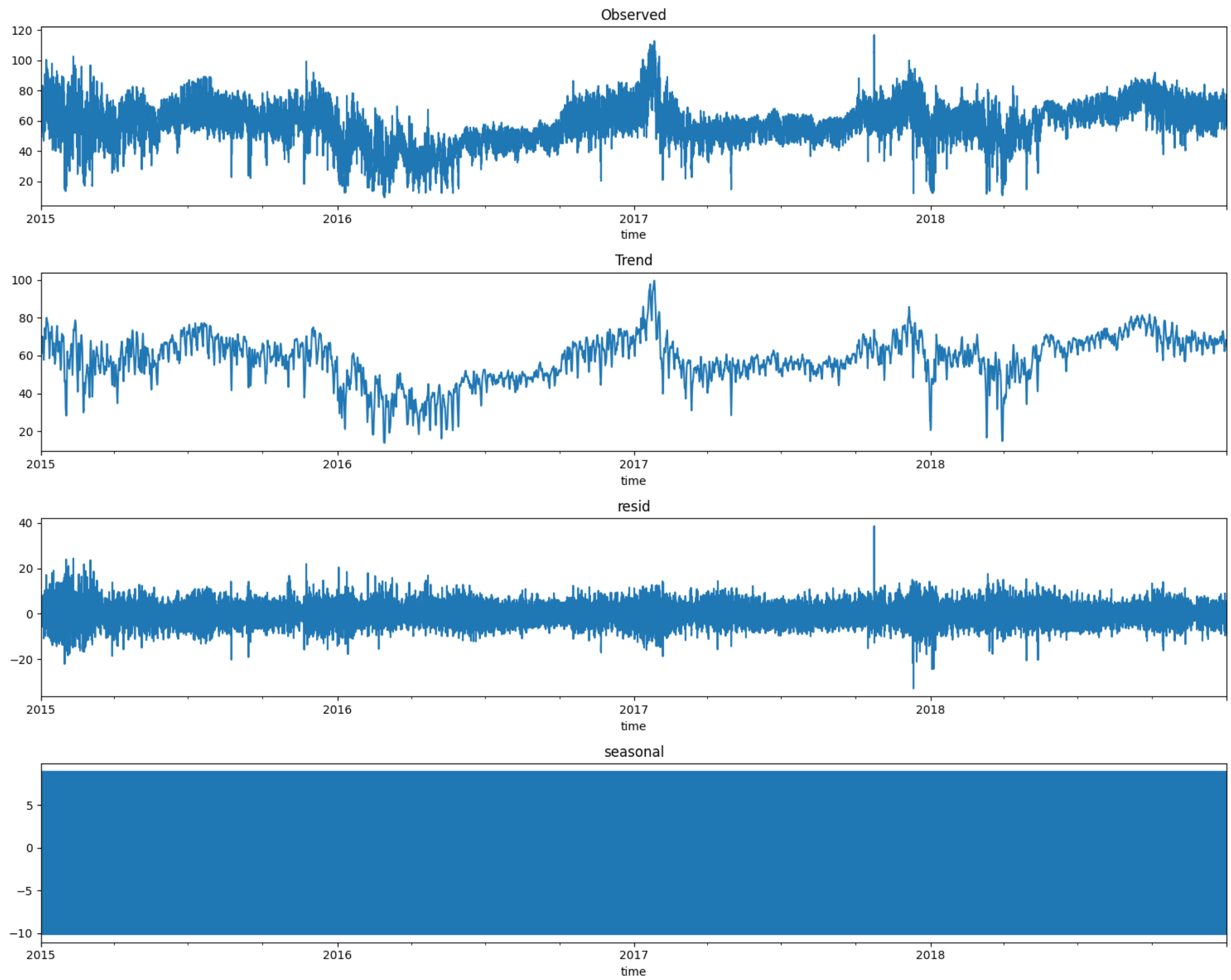



```
# Plotting the histogram
sns.distplot(x = data['price actual'], kde = True)
```

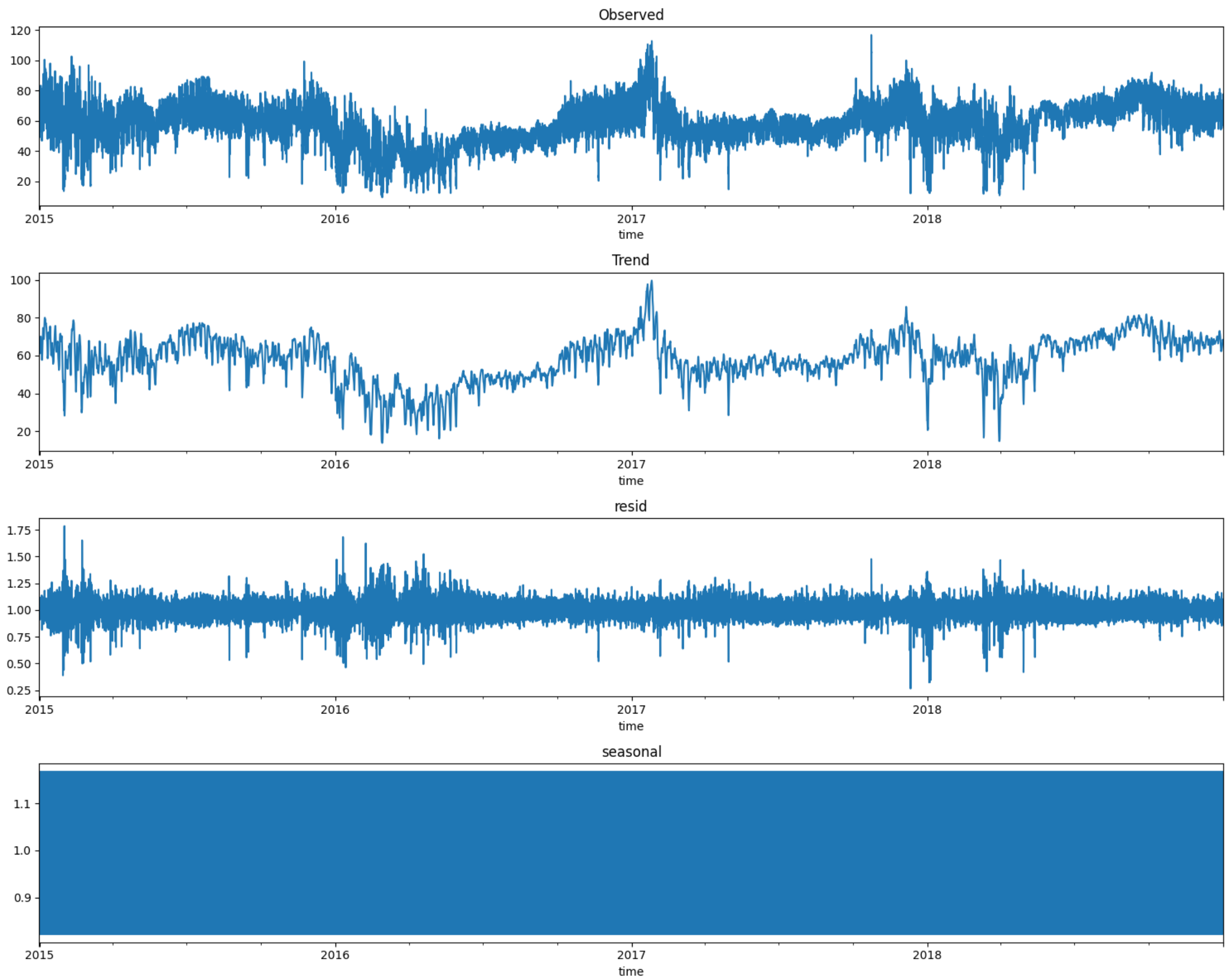
```
<Axes: ylabel='Density'>
```




```
import statsmodels.api as sm
res = sm.tsa.seasonal_decompose(data['price actual'], model =
'additive')
fig, (ax1,ax2,ax3,ax4) = plt.subplots(4,1, figsize = (15,12))
res.observed.plot(ax=ax1, title= 'Observed')
res.trend.plot(ax=ax2, title = 'Trend')
res.resid.plot(ax=ax3, title = 'resid')
res.seasonal.plot(ax= ax4, title = 'seasonal')
plt.tight_layout()
plt.show()
```




```
res = sm.tsa.seasonal_decompose(data['price actual'], model =
'multiplicative')
fig, (ax1,ax2,ax3,ax4) = plt.subplots(4,1, figsize = (15,12))
res.observed.plot(ax=ax1, title= 'Observed')
res.trend.plot(ax=ax2, title = 'Trend')
res.resid.plot(ax=ax3, title = 'resid')
res.seasonal.plot(ax= ax4, title = 'seasonal')
plt.tight_layout()
plt.show()
```



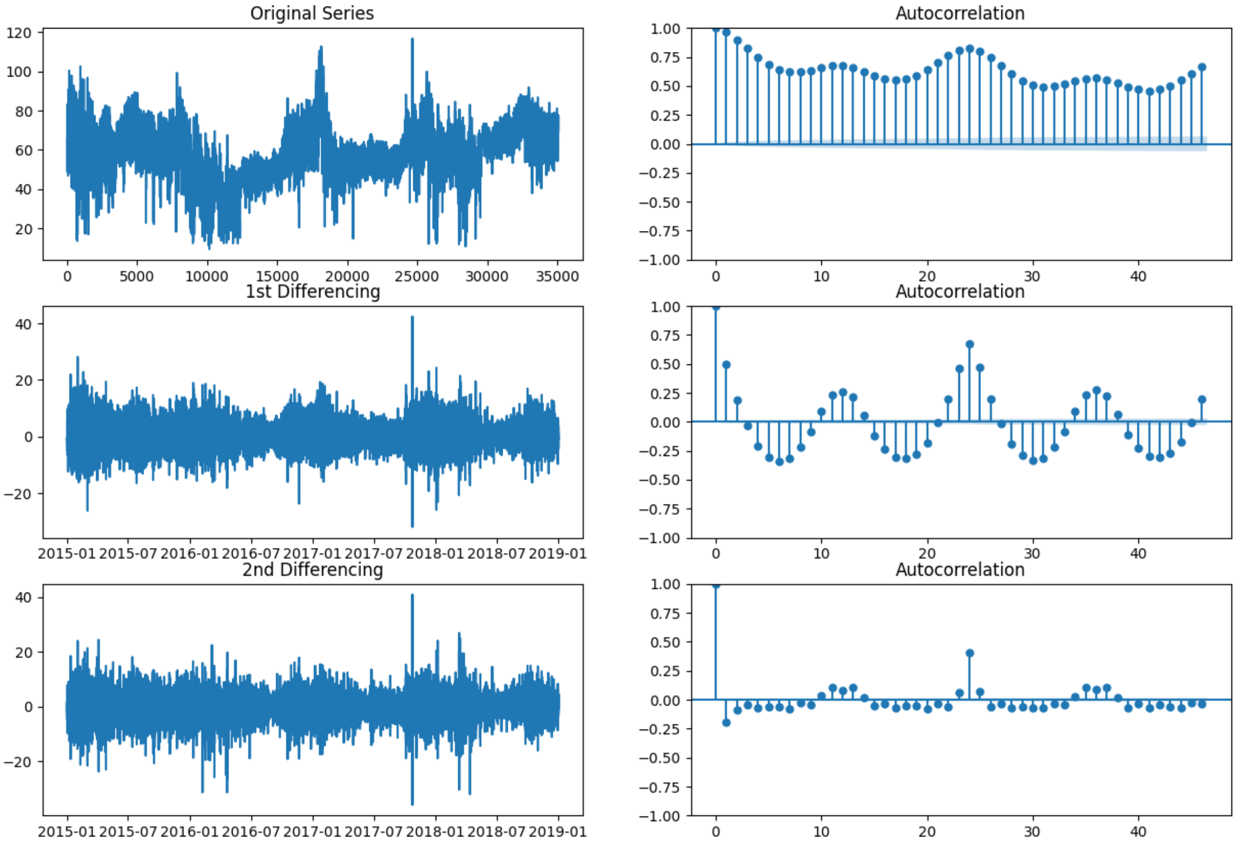

```
#ADFuller
from statsmodels.tsa.stattools import adfuller
result = adfuller(data['price actual'], autolag = 'AIC')
print(f'ADF Stats: {result[0]}')
print(f'n-lags: {result[2]}')
print(f'p-value: {round(result[1], 6)}')

for key, value in result[4].items():
  print(f'Critical Values:')
  print(f' {key} :  {value}')

ADF Stats: -9.147016232851248
n-lags: 50
p-value: 0.0
Critical Values:
 1% :  -3.4305367814665044
Critical Values:
 5% :  -2.8616225527935106
Critical Values:
 10% :  -2.566813940257257
```

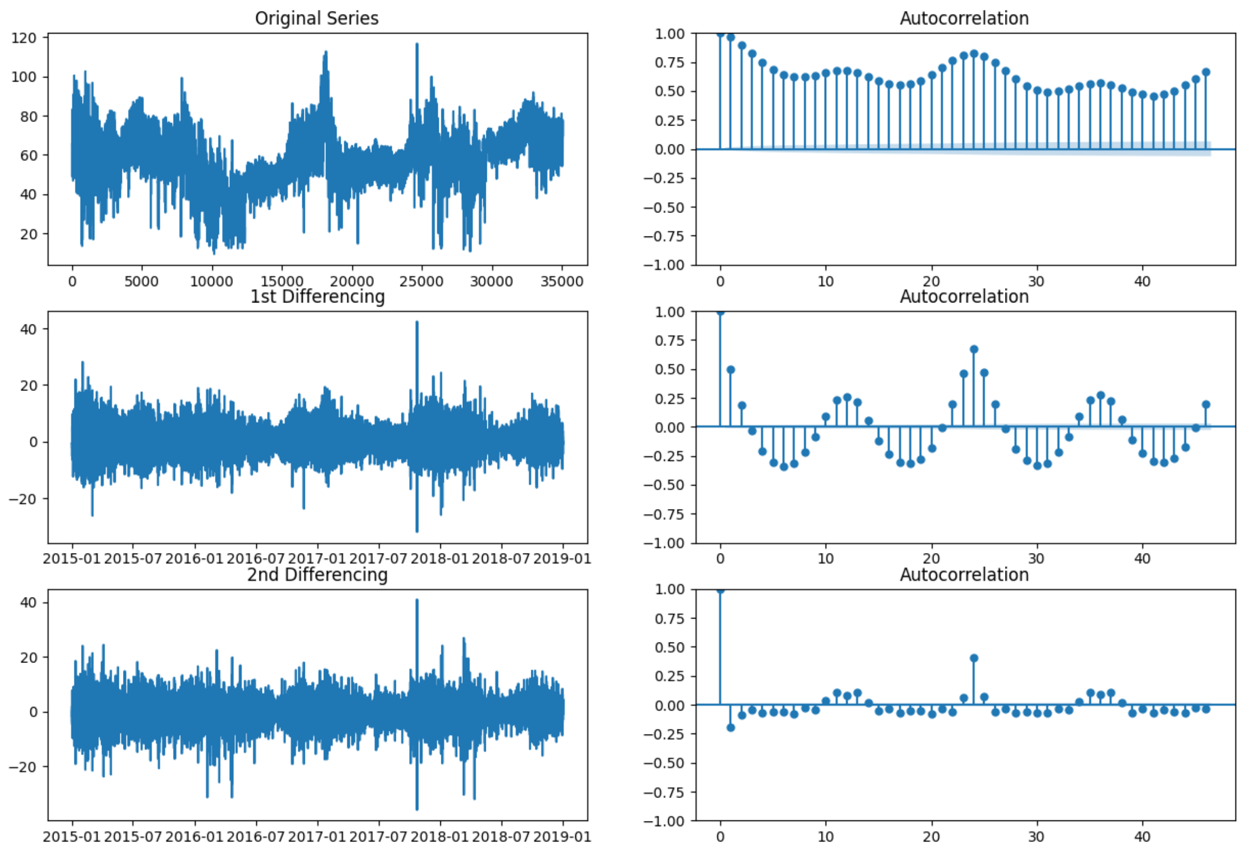
```python
from prompt_toolkit.key_binding import key_processor
# KPSS
from statsmodels.tsa.stattools import kpss
result = kpss(data['price actual'])
print(f'ADF Stats: {result[0]}')
print(f'n-lags: {result[2]}')
print(f'p-value: {result[1]}')

for key, value in result[3].items():
  print(f'Critical Values:')
  print(f' {key} :  {value}')
```

```
ADF Stats: 4.330033575195487
n-lags: 105
p-value: 0.01
Critical Values:
 10% :  0.347
Critical Values:
 5% :  0.463
Critical Values:
 2.5% :  0.574
Critical Values:
 1% :  0.739
```

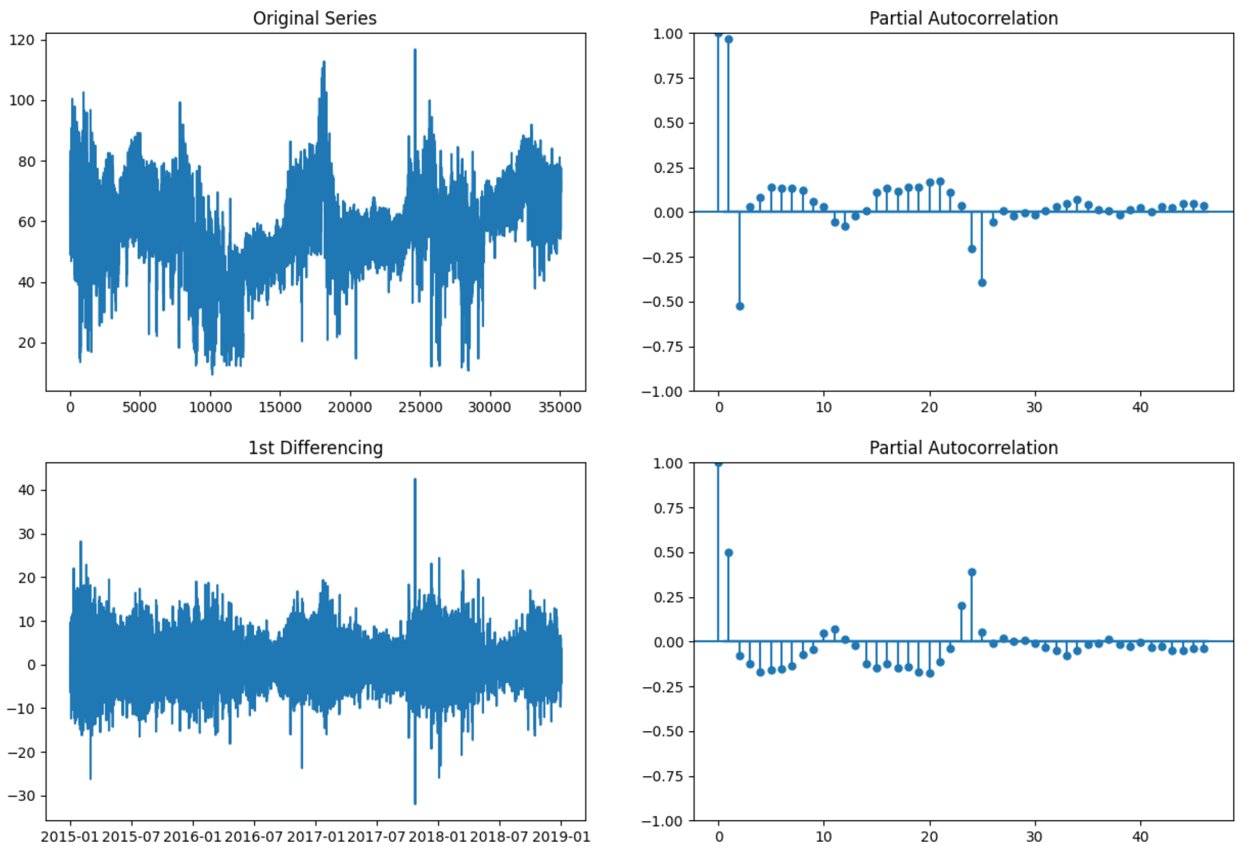
```
<ipython-input-25-dabaa2717dc6>:4: InterpolationWarning: The test
statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value
returned.

  result = kpss(data['price actual'])
```

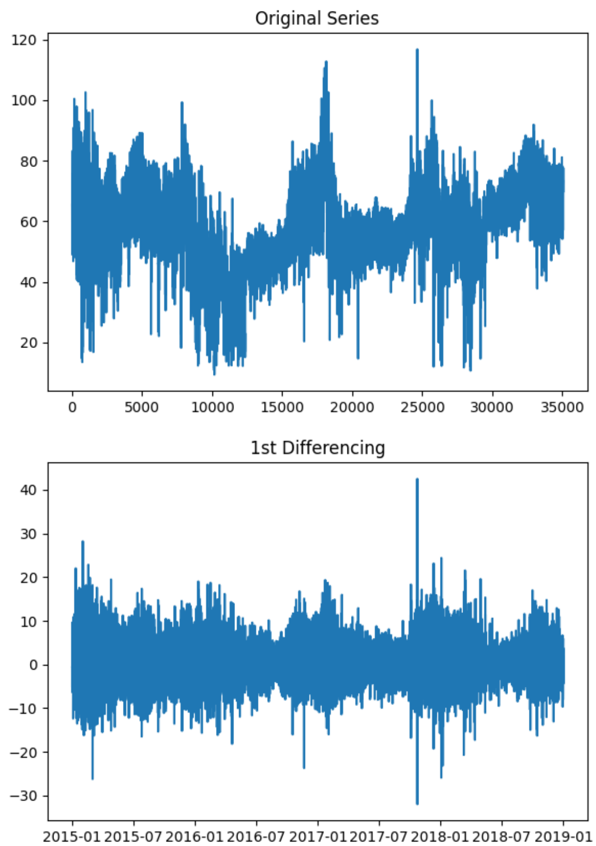
```python
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
#Plotting ACF graph with original series, 1st Differencing, 2nd
Diferencing

fig, axes = plt.subplots(3,2, figsize = (15,10))

axes[0,0].plot(data['price actual'].values)
axes[0,0].set_title('Original Series')
plot_acf(data['price actual'].dropna(), ax = axes[0,1])

axes[1,0].plot(data['price actual'].diff())
axes[1,0].set_title('1st Differencing')
plot_acf(data['price actual'].diff().dropna(), ax = axes[1,1])

axes[2,0].plot(data['price actual'].diff().diff())
axes[2,0].set_title('2nd Differencing')
plot_acf(data['price actual'].diff().diff().dropna(), ax = axes[2,1])
```


Original Series
Autocorrelation
1st Differencing
Autocorrelation
2nd Differencing
Autocorrelation
0
5000
10000
15000
20000
25000
30000
35000
2015-01
2015-07
2016-01
2016-07
2017-01
2017-07
2018-01
2018-07
2019-01
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40




```python
#Plotting PACF graph with the 1st Differenced values of the column
fig, axes = plt.subplots(2,2, figsize = (15,10))

axes[0,0].plot(data['price actual'].values)
axes[0,0].set_title('Original Series')
plot_pacf(data['price actual'].dropna(), ax = axes[0,1])

axes[1,0].plot(data['price actual'].diff())
axes[1,0].set_title('1st Differencing')
plot_pacf(data['price actual'].diff().dropna(), ax = axes[1,1])
```


Original Series
120
100
80
60
40
20
0
5000
10000
15000
20000
25000
30000
35000
Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
1st Differencing
40
30
20
10
0
−10
−20
−30
2015-01
2015-07
2016-01
2016-07
2017-01
2017-07
2018-01
2018-07
2019-01
Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40




```
def prepare_dataset(data, size):
  x_data = []
  y_data = []

  l = len(data) - size

  for i in range(l):
    x = data[i:i+size]
    y = data[i+size]
    x_data.append(x)
    y_data.append(y)

  return np.array(x_data), np.array(y_data)

def plot_model_rmse_and_loss(history, title):

    # Evaluate train and validation accuracies and losses

    train_rmse = history.history['root_mean_squared_error']
    val_rmse = history.history['val_root_mean_squared_error']

    train_loss = history.history['loss']
    val_loss = history.history['val_loss']
```

```
    # Visualize epochs vs. train and validation accuracies and losses

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 2, 1)
    plt.plot(train_rmse, label='Training RMSE')
    plt.plot(val_rmse, label='Validation RMSE')
    plt.legend()
    plt.title('Epochs vs. Training and Validation RMSE')

    plt.subplot(1, 2, 2)
    plt.plot(train_loss, label='Training Loss')
    plt.plot(val_loss, label='Validation Loss')
    plt.legend()
    plt.title('Epochs vs. Training and Validation Loss')

    plt.suptitle(title, fontweight = 'bold',  fontsize= 15)

    plt.show()

from sklearn.preprocessing import MinMaxScaler

data_filtered = data['price actual'].values

scaler = MinMaxScaler(feature_range = (0,1))

scaled_data = scaler.fit_transform(data_filtered.reshape(-1,1))
scaled_data.shape
```

```
(35064, 1)
```

```
train_size = int(np.ceil(len(scaled_data) * 0.8))
test_size = int((len(scaled_data) - train_size) *0.5)
print(train_size, test_size)
```

```
28052 3506
```

```
xtrain, ytrain = prepare_dataset(scaled_data[:train_size], 25)
xval, yval = prepare_dataset(scaled_data[train_size-25:train_size
+test_size], 25)
xtest, ytest = prepare_dataset(scaled_data[train_size + test_size-
25:], 25)
print(xtrain.shape)
print(xval.shape)
print(xtest.shape)
```

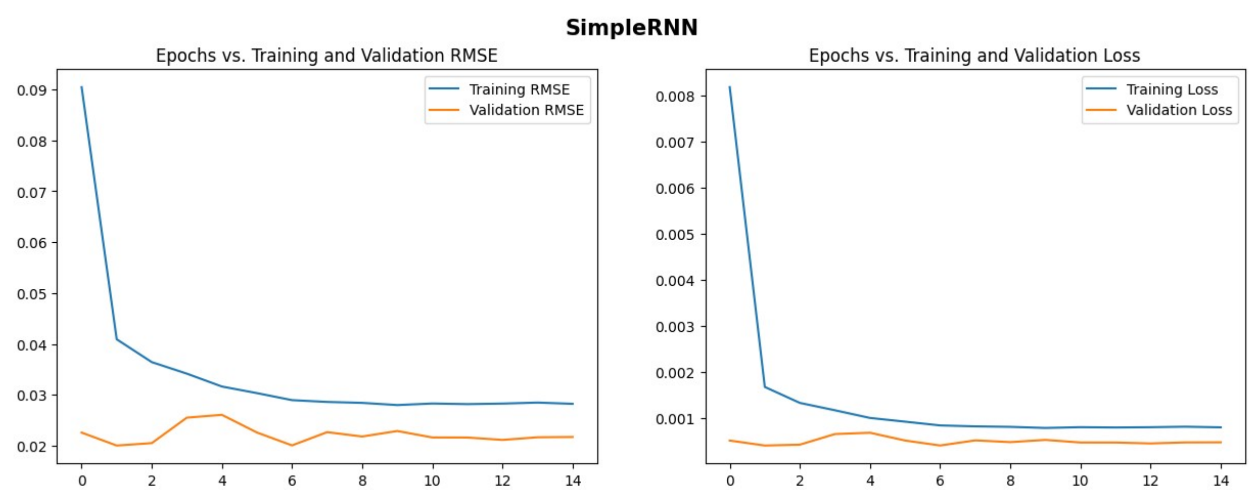
```
(28027, 25, 1)
(3506, 25, 1)
(3506, 25, 1)
```

```
import tensorflow as tf
from tensorflow.keras import Sequential
```

```
from tensorflow.keras.layers import Dense, LSTM, Dropout, Conv1D,
Flatten, SimpleRNN
loss = tf.keras.losses.MeanSquaredError()
metric = [tf.keras.metrics.RootMeanSquaredError()]
optimizer = tf.keras.optimizers.Adam()
early_stopping = [tf.keras.callbacks.EarlyStopping(monitor = 'loss',
patience = 5)]

model_SimpleRNN = Sequential()
model_SimpleRNN.add(SimpleRNN(28, re1turn_sequences = True,
input_shape = (xtrain.shape[1], 1)))
model_SimpleRNN.add(SimpleRNN(64, return_sequences = False))
model_SimpleRNN.add(Dense(64))
model_SimpleRNN.add(Dropout(0.2))
model_SimpleRNN.add(Dense(1))
model_SimpleRNN.compile(loss = loss, metrics = metric, optimizer =
optimizer)

history = model_SimpleRNN.fit(xtrain, ytrain, epochs = 60,
validation_data =(xval,yval), callbacks = early_stopping)
```

```
Epoch 1/60
876/876 [==============================] - 19s 18ms/step - loss:
0.0082 - root_mean_squared_error: 0.0905 - val_loss: 5.0991e-04 -
val_root_mean_squared_error: 0.0226
Epoch 2/60
876/876 [==============================] - 19s 22ms/step - loss:
0.0017 - root_mean_squared_error: 0.0409 - val_loss: 4.0135e-04 -
val_root_mean_squared_error: 0.0200
Epoch 3/60
876/876 [==============================] - 18s 20ms/step - loss:
0.0013 - root_mean_squared_error: 0.0364 - val_loss: 4.2073e-04 -
val_root_mean_squared_error: 0.0205
Epoch 4/60
876/876 [==============================] - 15s 17ms/step - loss:
0.0012 - root_mean_squared_error: 0.0342 - val_loss: 6.5107e-04 -
val_root_mean_squared_error: 0.0255
Epoch 5/60
876/876 [==============================] - 17s 20ms/step - loss:
0.0010 - root_mean_squared_error: 0.0316 - val_loss: 6.7946e-04 -
val_root_mean_squared_error: 0.0261
Epoch 6/60
876/876 [==============================] - 15s 17ms/step - loss:
9.2018e-04 - root_mean_squared_error: 0.0303 - val_loss: 5.1058e-04 -
val_root_mean_squared_error: 0.0226
Epoch 7/60
876/876 [==============================] - 17s 20ms/step - loss:
8.3849e-04 - root_mean_squared_error: 0.0290 - val_loss: 4.0250e-04 -
val_root_mean_squared_error: 0.0201
Epoch 8/60
```

```
876/876 [==============================] - 15s 17ms/step - loss:
8.1803e-04 - root_mean_squared_error: 0.0286 - val_loss: 5.1379e-04 -
val_root_mean_squared_error: 0.0227
Epoch 9/60
876/876 [==============================] - 15s 17ms/step - loss:
8.0754e-04 - root_mean_squared_error: 0.0284 - val_loss: 4.7566e-04 -
val_root_mean_squared_error: 0.0218
Epoch 10/60
876/876 [==============================] - 15s 17ms/step - loss:
7.8335e-04 - root_mean_squared_error: 0.0280 - val_loss: 5.2406e-04 -
val_root_mean_squared_error: 0.0229
Epoch 11/60
876/876 [==============================] - 15s 17ms/step - loss:
8.0037e-04 - root_mean_squared_error: 0.0283 - val_loss: 4.6762e-04 -
val_root_mean_squared_error: 0.0216
Epoch 12/60
876/876 [==============================] - 16s 19ms/step - loss:
7.9368e-04 - root_mean_squared_error: 0.0282 - val_loss: 4.6693e-04 -
val_root_mean_squared_error: 0.0216
Epoch 13/60
876/876 [==============================] - 16s 19ms/step - loss:
7.9967e-04 - root_mean_squared_error: 0.0283 - val_loss: 4.4696e-04 -
val_root_mean_squared_error: 0.0211
Epoch 14/60
876/876 [==============================] - 16s 18ms/step - loss:
8.1080e-04 - root_mean_squared_error: 0.0285 - val_loss: 4.6921e-04 -
val_root_mean_squared_error: 0.0217
Epoch 15/60
876/876 [==============================] - 15s 17ms/step - loss:
7.9690e-04 - root_mean_squared_error: 0.0282 - val_loss: 4.7157e-04 -
val_root_mean_squared_error: 0.0217

plot_model_rmse_and_loss(history,"SimpleRNN")
```



```
predictions = model_SimpleRNN.predict(xtest)
predictions = scaler.inverse_transform(predictions)
simplernn_rmse = np.sqrt(np.mean(((predictions - ytest) ** 2)))
print(f"Root Mean Squarred Error for SimpleRNN = {simplernn_rmse}")
```

```
110/110 [==============================] - 3s 13ms/step
Root Mean Squarred Error for SimpleRNN = 70.96717141535989
```

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Conv1D,
Flatten, SimpleRNN
loss = tf.keras.losses.MeanSquaredError()
metric = [tf.keras.metrics.RootMeanSquaredError()]
optimizer = tf.keras.optimizers.Adam()
early_stopping = [tf.keras.callbacks.EarlyStopping(monitor = 'loss',
patience = 5)]

model_LSTM = Sequential()
model_LSTM.add(LSTM(128, input_shape = (xtrain.shape[1], 1)))
model_LSTM.add(Flatten())
model_LSTM.add(Dense(128, activation = 'relu'))
model_LSTM.add(Dropout(0.2))
model_LSTM.add(Dense(64, activation = 'relu'))
model_LSTM.add(Dropout(0.2))
model_LSTM.add(Dense(1))
model_LSTM.compile(loss = loss, metrics = metric, optimizer =
optimizer)

history = model_LSTM.fit(xtrain, ytrain, epochs = 60, validation_data
=(xval , yval), callbacks = early_stopping)
```

```
Epoch 1/60
876/876 [==============================] - 35s 37ms/step - loss:
0.0045 - root_mean_squared_error: 0.0668 - val_loss: 7.4714e-04 -
val_root_mean_squared_error: 0.0273
Epoch 2/60
876/876 [==============================] - 32s 37ms/step - loss:
0.0019 - root_mean_squared_error: 0.0431 - val_loss: 5.4536e-04 -
val_root_mean_squared_error: 0.0234
Epoch 3/60
876/876 [==============================] - 31s 36ms/step - loss:
0.0014 - root_mean_squared_error: 0.0380 - val_loss: 5.5125e-04 -
val_root_mean_squared_error: 0.0235
Epoch 4/60
876/876 [==============================] - 31s 35ms/step - loss:
0.0012 - root_mean_squared_error: 0.0340 - val_loss: 5.3291e-04 -
val_root_mean_squared_error: 0.0231
Epoch 5/60
876/876 [==============================] - 34s 39ms/step - loss:
```

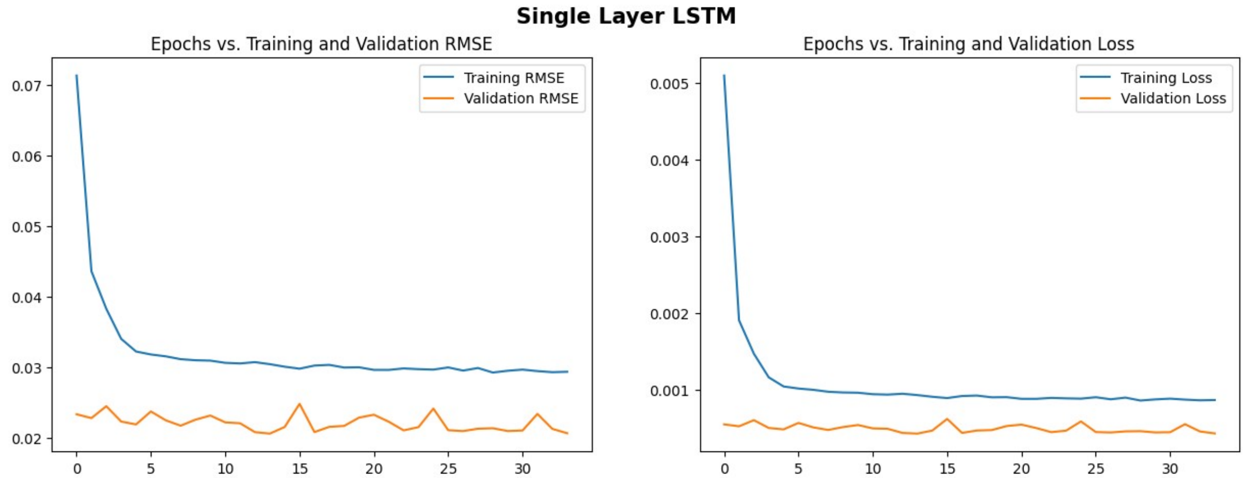
```
0.0011 - root_mean_squared_error: 0.0327 - val_loss: 4.6875e-04 -
val_root_mean_squared_error: 0.0217
Epoch 6/60
876/876 [==============================] - 31s 35ms/step - loss:
0.0010 - root_mean_squared_error: 0.0318 - val_loss: 4.7215e-04 -
val_root_mean_squared_error: 0.0217
Epoch 7/60
876/876 [==============================] - 32s 37ms/step - loss:
0.0010 - root_mean_squared_error: 0.0318 - val_loss: 6.8331e-04 -
val_root_mean_squared_error: 0.0261
Epoch 8/60
876/876 [==============================] - 31s 35ms/step - loss:
9.6250e-04 - root_mean_squared_error: 0.0310 - val_loss: 4.6928e-04 -
val_root_mean_squared_error: 0.0217
Epoch 9/60
876/876 [==============================] - 31s 36ms/step - loss:
9.6679e-04 - root_mean_squared_error: 0.0311 - val_loss: 4.9043e-04 -
val_root_mean_squared_error: 0.0221
Epoch 10/60
876/876 [==============================] - 34s 39ms/step - loss:
9.5602e-04 - root_mean_squared_error: 0.0309 - val_loss: 5.1559e-04 -
val_root_mean_squared_error: 0.0227
Epoch 11/60
876/876 [==============================] - 31s 36ms/step - loss:
9.6043e-04 - root_mean_squared_error: 0.0310 - val_loss: 5.6515e-04 -
val_root_mean_squared_error: 0.0238
Epoch 12/60
876/876 [==============================] - 32s 37ms/step - loss:
9.3997e-04 - root_mean_squared_error: 0.0307 - val_loss: 4.8095e-04 -
val_root_mean_squared_error: 0.0219
Epoch 13/60
876/876 [==============================] - 31s 35ms/step - loss:
9.2981e-04 - root_mean_squared_error: 0.0305 - val_loss: 5.2390e-04 -
val_root_mean_squared_error: 0.0229
Epoch 14/60
876/876 [==============================] - 32s 36ms/step - loss:
9.2219e-04 - root_mean_squared_error: 0.0304 - val_loss: 6.8676e-04 -
val_root_mean_squared_error: 0.0262
Epoch 15/60
876/876 [==============================] - 32s 37ms/step - loss:
9.3052e-04 - root_mean_squared_error: 0.0305 - val_loss: 6.4687e-04 -
val_root_mean_squared_error: 0.0254
Epoch 16/60
876/876 [==============================] - 30s 34ms/step - loss:
9.2931e-04 - root_mean_squared_error: 0.0305 - val_loss: 5.4575e-04 -
val_root_mean_squared_error: 0.0234
Epoch 17/60
876/876 [==============================] - 32s 36ms/step - loss:
9.1643e-04 - root_mean_squared_error: 0.0303 - val_loss: 5.3940e-04 -
```

```
val_root_mean_squared_error: 0.0232
Epoch 18/60
876/876 [==============================] - 31s 35ms/step - loss: 
8.9688e-04 - root_mean_squared_error: 0.0299 - val_loss: 4.8070e-04 - 
val_root_mean_squared_error: 0.0219
Epoch 19/60
876/876 [==============================] - 32s 36ms/step - loss: 
9.2252e-04 - root_mean_squared_error: 0.0304 - val_loss: 5.6422e-04 - 
val_root_mean_squared_error: 0.0238
Epoch 20/60
876/876 [==============================] - 32s 36ms/step - loss: 
8.8242e-04 - root_mean_squared_error: 0.0297 - val_loss: 4.7268e-04 - 
val_root_mean_squared_error: 0.0217
Epoch 21/60
876/876 [==============================] - 30s 35ms/step - loss: 
8.8639e-04 - root_mean_squared_error: 0.0298 - val_loss: 5.5236e-04 - 
val_root_mean_squared_error: 0.0235
Epoch 22/60
876/876 [==============================] - 31s 35ms/step - loss: 
8.7835e-04 - root_mean_squared_error: 0.0296 - val_loss: 4.6424e-04 - 
val_root_mean_squared_error: 0.0215
Epoch 23/60
876/876 [==============================] - 32s 36ms/step - loss: 
8.6753e-04 - root_mean_squared_error: 0.0295 - val_loss: 6.0569e-04 - 
val_root_mean_squared_error: 0.0246
Epoch 24/60
876/876 [==============================] - 31s 35ms/step - loss: 
8.6555e-04 - root_mean_squared_error: 0.0294 - val_loss: 4.5242e-04 - 
val_root_mean_squared_error: 0.0213
Epoch 25/60
876/876 [==============================] - 32s 36ms/step - loss: 
8.6097e-04 - root_mean_squared_error: 0.0293 - val_loss: 5.5718e-04 - 
val_root_mean_squared_error: 0.0236
Epoch 26/60
876/876 [==============================] - 31s 35ms/step - loss: 
8.7360e-04 - root_mean_squared_error: 0.0296 - val_loss: 5.6987e-04 - 
val_root_mean_squared_error: 0.0239
Epoch 27/60
876/876 [==============================] - 32s 37ms/step - loss: 
8.7479e-04 - root_mean_squared_error: 0.0296 - val_loss: 5.6030e-04 - 
val_root_mean_squared_error: 0.0237
Epoch 28/60
876/876 [==============================] - 32s 36ms/step - loss: 
8.5617e-04 - root_mean_squared_error: 0.0293 - val_loss: 5.4180e-04 - 
val_root_mean_squared_error: 0.0233
Epoch 29/60
876/876 [==============================] - 32s 36ms/step - loss: 
8.6650e-04 - root_mean_squared_error: 0.0294 - val_loss: 4.7718e-04 - 
val_root_mean_squared_error: 0.0218
```

```
Epoch 30/60
876/876 [==============================] - 30s 35ms/step - loss:
8.4934e-04 - root_mean_squared_error: 0.0291 - val_loss: 5.3747e-04 -
val_root_mean_squared_error: 0.0232
Epoch 31/60
876/876 [==============================] - 30s 35ms/step - loss:
8.5076e-04 - root_mean_squared_error: 0.0292 - val_loss: 6.0269e-04 -
val_root_mean_squared_error: 0.0245
Epoch 32/60
876/876 [==============================] - 33s 38ms/step - loss:
8.5102e-04 - root_mean_squared_error: 0.0292 - val_loss: 5.0659e-04 -
val_root_mean_squared_error: 0.0225
Epoch 33/60
876/876 [==============================] - 30s 35ms/step - loss:
8.3227e-04 - root_mean_squared_error: 0.0288 - val_loss: 5.6528e-04 -
val_root_mean_squared_error: 0.0238
Epoch 34/60
876/876 [==============================] - 33s 37ms/step - loss:
8.4753e-04 - root_mean_squared_error: 0.0291 - val_loss: 4.9392e-04 -
val_root_mean_squared_error: 0.0222
Epoch 35/60
876/876 [==============================] - 35s 40ms/step - loss:
8.3154e-04 - root_mean_squared_error: 0.0288 - val_loss: 5.0247e-04 -
val_root_mean_squared_error: 0.0224
Epoch 36/60
876/876 [==============================] - 33s 37ms/step - loss:
8.2938e-04 - root_mean_squared_error: 0.0288 - val_loss: 5.5840e-04 -
val_root_mean_squared_error: 0.0236
Epoch 37/60
876/876 [==============================] - 32s 36ms/step - loss:
8.2434e-04 - root_mean_squared_error: 0.0287 - val_loss: 5.0178e-04 -
val_root_mean_squared_error: 0.0224
Epoch 38/60
876/876 [==============================] - 33s 37ms/step - loss:
8.2345e-04 - root_mean_squared_error: 0.0287 - val_loss: 4.5644e-04 -
val_root_mean_squared_error: 0.0214
Epoch 39/60
876/876 [==============================] - 30s 35ms/step - loss:
8.1116e-04 - root_mean_squared_error: 0.0285 - val_loss: 6.0788e-04 -
val_root_mean_squared_error: 0.0247
Epoch 40/60
876/876 [==============================] - 32s 36ms/step - loss:
7.9799e-04 - root_mean_squared_error: 0.0282 - val_loss: 4.8540e-04 -
val_root_mean_squared_error: 0.0220
Epoch 41/60
876/876 [==============================] - 32s 37ms/step - loss:
7.9738e-04 - root_mean_squared_error: 0.0282 - val_loss: 4.6433e-04 -
val_root_mean_squared_error: 0.0215
Epoch 42/60
```

```
876/876 [==============================] - 32s 37ms/step - loss:
8.0084e-04 - root_mean_squared_error: 0.0283 - val_loss: 4.4117e-04 -
val_root_mean_squared_error: 0.0210
Epoch 43/60
876/876 [==============================] - 31s 36ms/step - loss:
7.8370e-04 - root_mean_squared_error: 0.0280 - val_loss: 4.5928e-04 -
val_root_mean_squared_error: 0.0214
Epoch 44/60
876/876 [==============================] - 31s 36ms/step - loss:
7.7475e-04 - root_mean_squared_error: 0.0278 - val_loss: 4.4581e-04 -
val_root_mean_squared_error: 0.0211
Epoch 45/60
876/876 [==============================] - 33s 38ms/step - loss:
7.8673e-04 - root_mean_squared_error: 0.0280 - val_loss: 4.7011e-04 -
val_root_mean_squared_error: 0.0217
Epoch 46/60
876/876 [==============================] - 31s 35ms/step - loss:
7.8854e-04 - root_mean_squared_error: 0.0281 - val_loss: 4.3610e-04 -
val_root_mean_squared_error: 0.0209
Epoch 47/60
876/876 [==============================] - 32s 37ms/step - loss:
7.8600e-04 - root_mean_squared_error: 0.0280 - val_loss: 4.7830e-04 -
val_root_mean_squared_error: 0.0219
Epoch 48/60
876/876 [==============================] - 31s 35ms/step - loss:
7.7274e-04 - root_mean_squared_error: 0.0278 - val_loss: 4.8285e-04 -
val_root_mean_squared_error: 0.0220
Epoch 49/60
876/876 [==============================] - 33s 37ms/step - loss:
7.5268e-04 - root_mean_squared_error: 0.0274 - val_loss: 6.8271e-04 -
val_root_mean_squared_error: 0.0261
Epoch 50/60
876/876 [==============================] - 32s 37ms/step - loss:
7.6882e-04 - root_mean_squared_error: 0.0277 - val_loss: 4.2675e-04 -
val_root_mean_squared_error: 0.0207
Epoch 51/60
876/876 [==============================] - 32s 36ms/step - loss:
7.5445e-04 - root_mean_squared_error: 0.0275 - val_loss: 5.0851e-04 -
val_root_mean_squared_error: 0.0226
Epoch 52/60
876/876 [==============================] - 30s 35ms/step - loss:
7.6315e-04 - root_mean_squared_error: 0.0276 - val_loss: 4.2679e-04 -
val_root_mean_squared_error: 0.0207
Epoch 53/60
876/876 [==============================] - 31s 35ms/step - loss:
7.6418e-04 - root_mean_squared_error: 0.0276 - val_loss: 6.2239e-04 -
val_root_mean_squared_error: 0.0249
Epoch 54/60
876/876 [==============================] - 34s 38ms/step - loss:
```

```
7.5709e-04 - root_mean_squared_error: 0.0275 - val_loss: 5.9508e-04 -
val_root_mean_squared_error: 0.0244

plot_model_rmse_and_loss(history,"Single Layer LSTM")
```




```
predictions = model_LSTM.predict(xtest)
predictions = scaler.inverse_transform(predictions)
singleLSTM_rmse = np.sqrt(np.mean(((predictions - ytest) ** 2)))
print(f"Root Mean Squared Error for Single Layer LSTM =
{singleLSTM_rmse}")

110/110 [==============================] - 3s 22ms/step
Root Mean Squared Error for Single Layer LSTM = 71.47716636740277

import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense,LSTM, Dropout, Conv1D,
Flatten, SimpleRNN
loss = tf.keras.losses.MeanSquaredError()
metric = [tf.keras.metrics.RootMeanSquaredError()]
optimizer = tf.keras.optimizers.Adam()
early_stopping = [tf.keras.callbacks.EarlyStopping(monitor = 'loss',
patience = 5)]

model_StackedLSTM = Sequential()
model_StackedLSTM.add(LSTM(128, return_sequences=True, input_shape=
(xtrain.shape[1], 1)))
model_StackedLSTM.add(LSTM(64, return_sequences=False))
model_StackedLSTM.add(Dense(64))
model_StackedLSTM.add(Dropout(0.2))
model_StackedLSTM.add(Dense(1))
model_StackedLSTM.compile(loss = loss, metrics = metric, optimizer =
optimizer)
history = model_StackedLSTM.fit(xtrain, ytrain, epochs = 40,
validation_data =(xval , yval), callbacks = early_stopping)
```

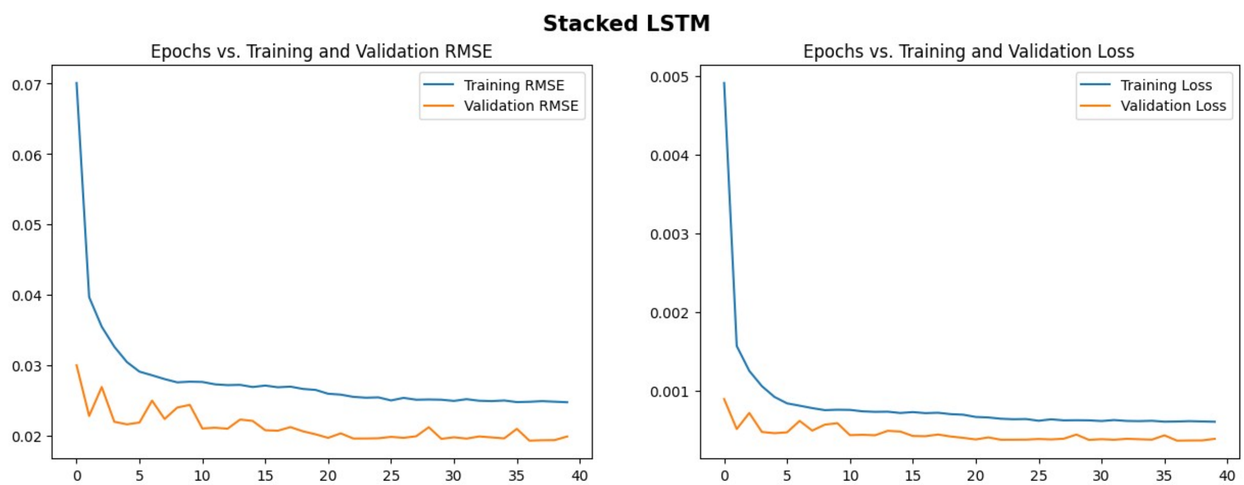
```
Epoch 1/40
876/876 [==============================] - 73s 79ms/step - loss:
0.0049 - root_mean_squared_error: 0.0701 - val_loss: 8.9824e-04 -
val_root_mean_squared_error: 0.0300
Epoch 2/40
876/876 [==============================] - 50s 57ms/step - loss:
0.0016 - root_mean_squared_error: 0.0397 - val_loss: 5.1778e-04 -
val_root_mean_squared_error: 0.0228
Epoch 3/40
876/876 [==============================] - 50s 57ms/step - loss:
0.0013 - root_mean_squared_error: 0.0354 - val_loss: 7.2206e-04 -
val_root_mean_squared_error: 0.0269
Epoch 4/40
876/876 [==============================] - 46s 53ms/step - loss:
0.0011 - root_mean_squared_error: 0.0326 - val_loss: 4.8083e-04 -
val_root_mean_squared_error: 0.0219
Epoch 5/40
876/876 [==============================] - 50s 57ms/step - loss:
9.2563e-04 - root_mean_squared_error: 0.0304 - val_loss: 4.6490e-04 -
val_root_mean_squared_error: 0.0216
Epoch 6/40
876/876 [==============================] - 47s 54ms/step - loss:
8.4419e-04 - root_mean_squared_error: 0.0291 - val_loss: 4.7602e-04 -
val_root_mean_squared_error: 0.0218
Epoch 7/40
876/876 [==============================] - 47s 54ms/step - loss:
8.1442e-04 - root_mean_squared_error: 0.0285 - val_loss: 6.2164e-04 -
val_root_mean_squared_error: 0.0249
Epoch 8/40
876/876 [==============================] - 50s 57ms/step - loss:
7.8379e-04 - root_mean_squared_error: 0.0280 - val_loss: 4.9818e-04 -
val_root_mean_squared_error: 0.0223
Epoch 9/40
876/876 [==============================] - 48s 54ms/step - loss:
7.5807e-04 - root_mean_squared_error: 0.0275 - val_loss: 5.7348e-04 -
val_root_mean_squared_error: 0.0239
Epoch 10/40
876/876 [==============================] - 49s 56ms/step - loss:
7.6323e-04 - root_mean_squared_error: 0.0276 - val_loss: 5.9207e-04 -
val_root_mean_squared_error: 0.0243
Epoch 11/40
876/876 [==============================] - 49s 56ms/step - loss:
7.6118e-04 - root_mean_squared_error: 0.0276 - val_loss: 4.3987e-04 -
val_root_mean_squared_error: 0.0210
Epoch 12/40
876/876 [==============================] - 53s 61ms/step - loss:
7.4266e-04 - root_mean_squared_error: 0.0273 - val_loss: 4.4445e-04 -
val_root_mean_squared_error: 0.0211
Epoch 13/40
876/876 [==============================] - 48s 55ms/step - loss:
```

```
7.3636e-04 - root_mean_squared_error: 0.0271 - val_loss: 4.3847e-04 -
val_root_mean_squared_error: 0.0209
Epoch 14/40
876/876 [==============================] - 47s 54ms/step - loss:
7.3839e-04 - root_mean_squared_error: 0.0272 - val_loss: 4.9478e-04 -
val_root_mean_squared_error: 0.0222
Epoch 15/40
876/876 [==============================] - 47s 53ms/step - loss:
7.2197e-04 - root_mean_squared_error: 0.0269 - val_loss: 4.8607e-04 -
val_root_mean_squared_error: 0.0220
Epoch 16/40
876/876 [==============================] - 49s 56ms/step - loss:
7.3306e-04 - root_mean_squared_error: 0.0271 - val_loss: 4.2946e-04 -
val_root_mean_squared_error: 0.0207
Epoch 17/40
876/876 [==============================] - 47s 54ms/step - loss:
7.1989e-04 - root_mean_squared_error: 0.0268 - val_loss: 4.2658e-04 -
val_root_mean_squared_error: 0.0207
Epoch 18/40
876/876 [==============================] - 46s 53ms/step - loss:
7.2489e-04 - root_mean_squared_error: 0.0269 - val_loss: 4.4804e-04 -
val_root_mean_squared_error: 0.0212
Epoch 19/40
876/876 [==============================] - 49s 56ms/step - loss:
7.0726e-04 - root_mean_squared_error: 0.0266 - val_loss: 4.2312e-04 -
val_root_mean_squared_error: 0.0206
Epoch 20/40
876/876 [==============================] - 46s 53ms/step - loss:
6.9998e-04 - root_mean_squared_error: 0.0265 - val_loss: 4.0573e-04 -
val_root_mean_squared_error: 0.0201
Epoch 21/40
876/876 [==============================] - 48s 55ms/step - loss:
6.7137e-04 - root_mean_squared_error: 0.0259 - val_loss: 3.8561e-04 -
val_root_mean_squared_error: 0.0196
Epoch 22/40
876/876 [==============================] - 48s 55ms/step - loss:
6.6492e-04 - root_mean_squared_error: 0.0258 - val_loss: 4.1129e-04 -
val_root_mean_squared_error: 0.0203
Epoch 23/40
876/876 [==============================] - 46s 53ms/step - loss:
6.4868e-04 - root_mean_squared_error: 0.0255 - val_loss: 3.8195e-04 -
val_root_mean_squared_error: 0.0195
Epoch 24/40
876/876 [==============================] - 48s 54ms/step - loss:
6.4240e-04 - root_mean_squared_error: 0.0253 - val_loss: 3.8212e-04 -
val_root_mean_squared_error: 0.0195
Epoch 25/40
876/876 [==============================] - 47s 54ms/step - loss:
6.4481e-04 - root_mean_squared_error: 0.0254 - val_loss: 3.8332e-04 -
val_root_mean_squared_error: 0.0196
```

```
Epoch 26/40
876/876 [==============================] - 48s 55ms/step - loss:
6.2309e-04 - root_mean_squared_error: 0.0250 - val_loss: 3.9122e-04 -
val_root_mean_squared_error: 0.0198
Epoch 27/40
876/876 [==============================] - 46s 53ms/step - loss:
6.4129e-04 - root_mean_squared_error: 0.0253 - val_loss: 3.8583e-04 -
val_root_mean_squared_error: 0.0196
Epoch 28/40
876/876 [==============================] - 48s 55ms/step - loss:
6.2798e-04 - root_mean_squared_error: 0.0251 - val_loss: 3.9473e-04 -
val_root_mean_squared_error: 0.0199
Epoch 29/40
876/876 [==============================] - 47s 54ms/step - loss:
6.2975e-04 - root_mean_squared_error: 0.0251 - val_loss: 4.4732e-04 -
val_root_mean_squared_error: 0.0211
Epoch 30/40
876/876 [==============================] - 46s 53ms/step - loss:
6.2807e-04 - root_mean_squared_error: 0.0251 - val_loss: 3.8007e-04 -
val_root_mean_squared_error: 0.0195
Epoch 31/40
876/876 [==============================] - 49s 56ms/step - loss:
6.1975e-04 - root_mean_squared_error: 0.0249 - val_loss: 3.8891e-04 -
val_root_mean_squared_error: 0.0197
Epoch 32/40
876/876 [==============================] - 46s 52ms/step - loss:
6.3197e-04 - root_mean_squared_error: 0.0251 - val_loss: 3.8143e-04 -
val_root_mean_squared_error: 0.0195
Epoch 33/40
876/876 [==============================] - 48s 54ms/step - loss:
6.2085e-04 - root_mean_squared_error: 0.0249 - val_loss: 3.9368e-04 -
val_root_mean_squared_error: 0.0198
Epoch 34/40
876/876 [==============================] - 48s 55ms/step - loss:
6.1825e-04 - root_mean_squared_error: 0.0249 - val_loss: 3.8822e-04 -
val_root_mean_squared_error: 0.0197
Epoch 35/40
876/876 [==============================] - 48s 55ms/step - loss:
6.2270e-04 - root_mean_squared_error: 0.0250 - val_loss: 3.8271e-04 -
val_root_mean_squared_error: 0.0196
Epoch 36/40
876/876 [==============================] - 48s 55ms/step - loss:
6.1129e-04 - root_mean_squared_error: 0.0247 - val_loss: 4.3765e-04 -
val_root_mean_squared_error: 0.0209
Epoch 37/40
876/876 [==============================] - 48s 54ms/step - loss:
6.1344e-04 - root_mean_squared_error: 0.0248 - val_loss: 3.7027e-04 -
val_root_mean_squared_error: 0.0192
Epoch 38/40
876/876 [==============================] - 47s 54ms/step - loss:
```

```
6.1810e-04 - root_mean_squared_error: 0.0249 - val_loss: 3.7248e-04 -
val_root_mean_squared_error: 0.0193
Epoch 39/40
876/876 [==============================] - 46s 53ms/step - loss:
6.1412e-04 - root_mean_squared_error: 0.0248 - val_loss: 3.7321e-04 -
val_root_mean_squared_error: 0.0193
Epoch 40/40
876/876 [==============================] - 48s 54ms/step - loss:
6.1031e-04 - root_mean_squared_error: 0.0247 - val_loss: 3.9330e-04 -
val_root_mean_squared_error: 0.0198
```

```
plot_model_rmse_and_loss(history,"Stacked LSTM")
predictions = model_StackedLSTM.predict(xtest)
predictions = scaler.inverse_transform(predictions)
stackedLSTM_rmse = np.sqrt(np.mean(((predictions - ytest) ** 2)))
print(f"\nRoot Mean Squarred Error for Stacked LSTM =
{stackedLSTM_rmse}")
```




```
110/110 [==============================] - 2s 18ms/step

Root Mean Squarred Error for Stacked LSTM = 69.67299637608207
```

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Conv1D,
Flatten, SimpleRNN
loss = tf.keras.losses.MeanSquaredError()
metric = [tf.keras.metrics.RootMeanSquaredError()]
optimizer = tf.keras.optimizers.Adam()
early_stopping = [tf.keras.callbacks.EarlyStopping(monitor = 'loss',
patience = 5)]

model_CNN = Sequential()
model_CNN.add(Conv1D(filters = 48, kernel_size =2, padding = 'causal',
```

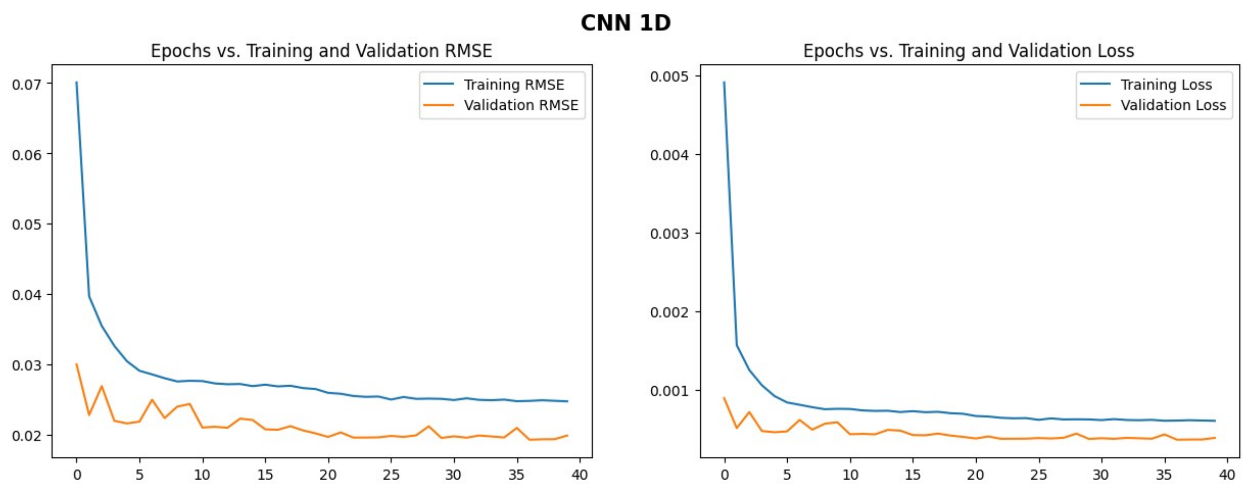
```
activation = 'relu', input_shape = (xtrain.shape[1], 1)))
model_CNN.add(Flatten())
model_CNN.add(Dense(48, activation = 'relu'))
model_CNN.add(Dropout(0.2))
model_CNN.add(Dense(1))
model_CNN.compile(loss = loss, metrics = metric, optimizer =
optimizer)
history = model_CNN.fit(xtrain, ytrain, epochs = 60, validation_data
=(xval , yval), callbacks = early_stopping)
```

```
Epoch 1/60
876/876 [==============================] - 4s 4ms/step - loss: 0.0055
- root_mean_squared_error: 0.0741 - val_loss: 6.9899e-04 -
val_root_mean_squared_error: 0.0264
Epoch 2/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0021
- root_mean_squared_error: 0.0457 - val_loss: 4.5606e-04 -
val_root_mean_squared_error: 0.0214
Epoch 3/60
876/876 [==============================] - 4s 5ms/step - loss: 0.0013
- root_mean_squared_error: 0.0365 - val_loss: 4.6647e-04 -
val_root_mean_squared_error: 0.0216
Epoch 4/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0011
- root_mean_squared_error: 0.0336 - val_loss: 3.9875e-04 -
val_root_mean_squared_error: 0.0200
Epoch 5/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0011
- root_mean_squared_error: 0.0326 - val_loss: 3.8243e-04 -
val_root_mean_squared_error: 0.0196
Epoch 6/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0011
- root_mean_squared_error: 0.0325 - val_loss: 4.3977e-04 -
val_root_mean_squared_error: 0.0210
Epoch 7/60
876/876 [==============================] - 4s 5ms/step - loss: 0.0011
- root_mean_squared_error: 0.0324 - val_loss: 4.3031e-04 -
val_root_mean_squared_error: 0.0207
Epoch 8/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0010
- root_mean_squared_error: 0.0319 - val_loss: 6.8154e-04 -
val_root_mean_squared_error: 0.0261
Epoch 9/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0010
- root_mean_squared_error: 0.0320 - val_loss: 4.1752e-04 -
val_root_mean_squared_error: 0.0204
Epoch 10/60
876/876 [==============================] - 3s 3ms/step - loss: 0.0010
- root_mean_squared_error: 0.0317 - val_loss: 4.2286e-04 -
val_root_mean_squared_error: 0.0206
```

```
Epoch 11/60
876/876 [==============================] - 4s 4ms/step - loss:
9.8724e-04 - root_mean_squared_error: 0.0314 - val_loss: 5.7432e-04 -
val_root_mean_squared_error: 0.0240
Epoch 12/60
876/876 [==============================] - 3s 4ms/step - loss:
9.7810e-04 - root_mean_squared_error: 0.0313 - val_loss: 4.5987e-04 -
val_root_mean_squared_error: 0.0214
Epoch 13/60
876/876 [==============================] - 3s 3ms/step - loss:
9.6840e-04 - root_mean_squared_error: 0.0311 - val_loss: 3.9517e-04 -
val_root_mean_squared_error: 0.0199
Epoch 14/60
876/876 [==============================] - 3s 3ms/step - loss:
9.7520e-04 - root_mean_squared_error: 0.0312 - val_loss: 5.0368e-04 -
val_root_mean_squared_error: 0.0224
Epoch 15/60
876/876 [==============================] - 3s 3ms/step - loss:
9.7998e-04 - root_mean_squared_error: 0.0313 - val_loss: 3.9395e-04 -
val_root_mean_squared_error: 0.0198
Epoch 16/60
876/876 [==============================] - 4s 5ms/step - loss:
9.6863e-04 - root_mean_squared_error: 0.0311 - val_loss: 4.0000e-04 -
val_root_mean_squared_error: 0.0200
Epoch 17/60
876/876 [==============================] - 4s 5ms/step - loss:
9.5934e-04 - root_mean_squared_error: 0.0310 - val_loss: 4.1861e-04 -
val_root_mean_squared_error: 0.0205
Epoch 18/60
876/876 [==============================] - 3s 3ms/step - loss:
9.5281e-04 - root_mean_squared_error: 0.0309 - val_loss: 3.8190e-04 -
val_root_mean_squared_error: 0.0195
Epoch 19/60
876/876 [==============================] - 4s 4ms/step - loss:
9.4327e-04 - root_mean_squared_error: 0.0307 - val_loss: 4.9303e-04 -
val_root_mean_squared_error: 0.0222
Epoch 20/60
876/876 [==============================] - 3s 4ms/step - loss:
9.4575e-04 - root_mean_squared_error: 0.0308 - val_loss: 6.8376e-04 -
val_root_mean_squared_error: 0.0261
Epoch 21/60
876/876 [==============================] - 3s 3ms/step - loss:
9.4571e-04 - root_mean_squared_error: 0.0308 - val_loss: 4.0160e-04 -
val_root_mean_squared_error: 0.0200
Epoch 22/60
876/876 [==============================] - 3s 3ms/step - loss:
9.4982e-04 - root_mean_squared_error: 0.0308 - val_loss: 4.6550e-04 -
val_root_mean_squared_error: 0.0216
Epoch 23/60
```

```
876/876 [==============================] - 3s 4ms/step - loss: 
9.6357e-04 - root_mean_squared_error: 0.0310 - val_loss: 4.3761e-04 - 
val_root_mean_squared_error: 0.0209
Epoch 24/60
876/876 [==============================] - 4s 4ms/step - loss: 
9.4596e-04 - root_mean_squared_error: 0.0308 - val_loss: 4.9047e-04 - 
val_root_mean_squared_error: 0.0221
```

```
plot_model_rmse_and_loss(history,"CNN 1D")
predictions = model_CNN.predict(xtest)
predictions = scaler.inverse_transform(predictions)
CNN_rmse = np.sqrt(np.mean(((predictions - ytest) ** 2)))
print(f"\nRoot Mean Squarred Error for CNN 1D = {CNN_rmse}")
```




```
110/110 [==============================] - 0s 2ms/step

Root Mean Squarred Error for CNN 1D = 71.06840319933407
```

```
print(f"Root Mean Squarred Error for Single Layer LSTM = 
{singleLSTM_rmse}")
print(f"Root Mean Squarred Error for Stacked LSTM = 
{stackedLSTM_rmse}")
print(f"Root Mean Squarred Error for CNN 1D = {CNN_rmse}")
```

```
Root Mean Squarred Error for Single Layer LSTM = 71.47716636740277
Root Mean Squarred Error for Stacked LSTM = 69.67299637608207
Root Mean Squarred Error for CNN 1D = 71.06840319933407
```