MANAI MOHAMED MORTADHA - 3GII/SSE

```
# Import necessary libraries
import pandas as pd  # For data manipulation and analysis
import matplotlib.pyplot as plt  # For plotting graphs
import numpy as np  # For numerical operations
import tensorflow as tf  # For machine learning

# Import specific modules from TensorFlow
from tensorflow.keras.models import Sequential  # Sequential model for stacking layers
from tensorflow.keras.layers import Dropout, Dense, LSTM  # Different types of neural network layers
```

```
# Read the Excel file into a Pandas DataFrame
df = pd.read_excel("/content/production.xlsx")
```

```
df.head() #"df.head()" is a method used in Pandas to display the first few rows of a DataFrame named "df".
```

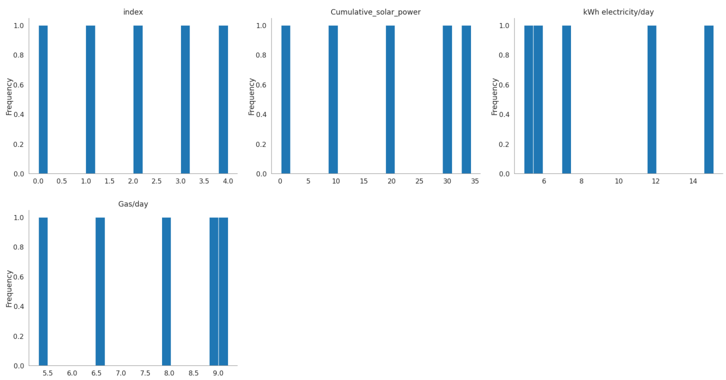
1 to 5 of 5 entries Filter

index	date	Cumulative_solar_power	kWh electricity/day	Gas/day
0	2011-10-26 00:00:00	0.1	15.1	9.0
1	2011-10-27 00:00:00	10.2	7.4	9.2
2	2011-10-28 00:00:00	20.2	5.8	8.0
3	2011-10-29 00:00:00	29.6	4.9	6.6
4	2011-10-30 00:00:00	34.2	11.7	5.3

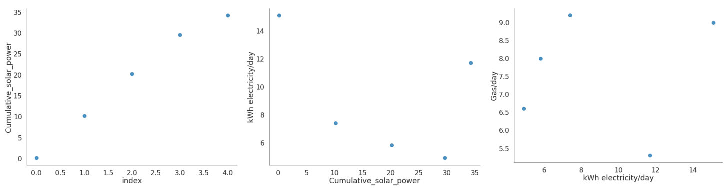
Show 25 per page

Like what you see? Visit the data table notebook to learn more about interactive tables.

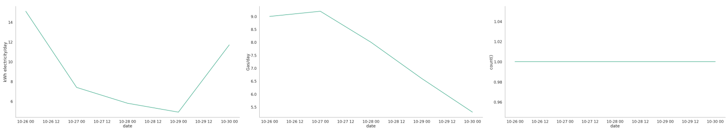
Distributions



2-d distributions



Time series



```
df.tail() #"df.tail()" is used in Pandas to display the last few rows of a DataFrame named "df".
```

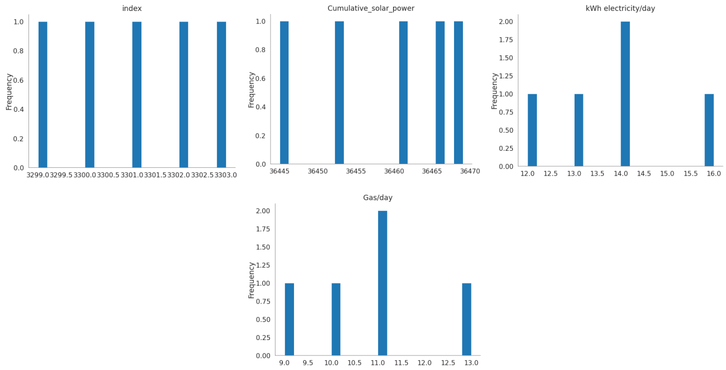
1 to 5 of 5 entries Filter

index	date	Cumulative_solar_power ▲	kWh electricity/day	Gas/day
3299	2020-11-06 00:00:00	36445.0	16.0	11.0
3300	2020-11-07 00:00:00	36453.0	13.0	13.0
3301	2020-11-08 00:00:00	36461.0	12.0	11.0
3302	2020-11-09 00:00:00	36466.0	14.0	10.0
3303	2020-11-10 00:00:00	36469.0	14.0	9.0

Show 25 per page

Like what you see? Visit the [data table notebook](#) to learn more about interactive tables.

Distributions



2-d distributions

```
# Convert the 'datetime' column to a datetime data type and 'Global_active_power' column to numeric,
# handling any errors by converting them to 'NaN' (Not a Number)
df['date'] = pd.to_datetime(df['date'])
df['Cumulative_solar_power'] = pd.to_numeric(df['Cumulative_solar_power'], errors='coerce')
```

```
# Display the data types of each column in the DataFrame
print(df.dtypes)
```

```
date                      datetime64[ns]
Cumulative_solar_power           float64
kWh electricity/day              float64
Gas/day                          float64
dtype: object
```

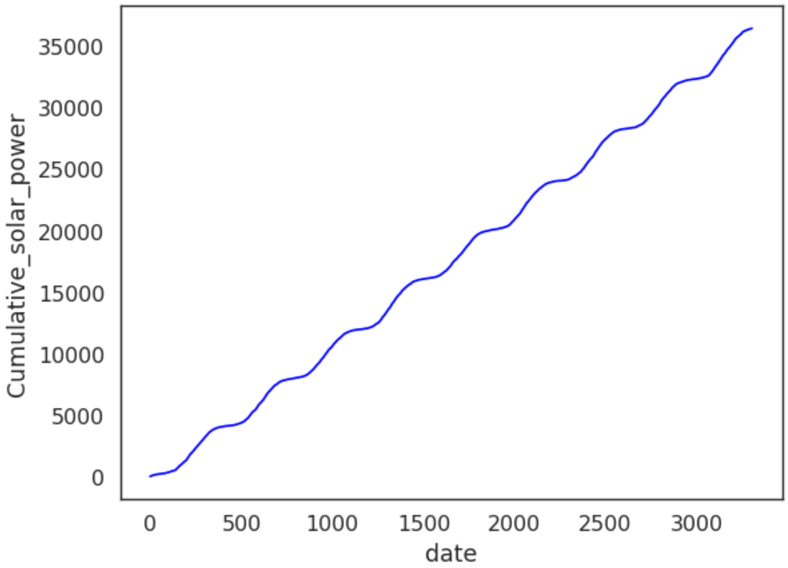
```
# Display the shape of the DataFrame, indicating the number of rows and columns
print(df.shape)
```

```
(3304, 4)
```

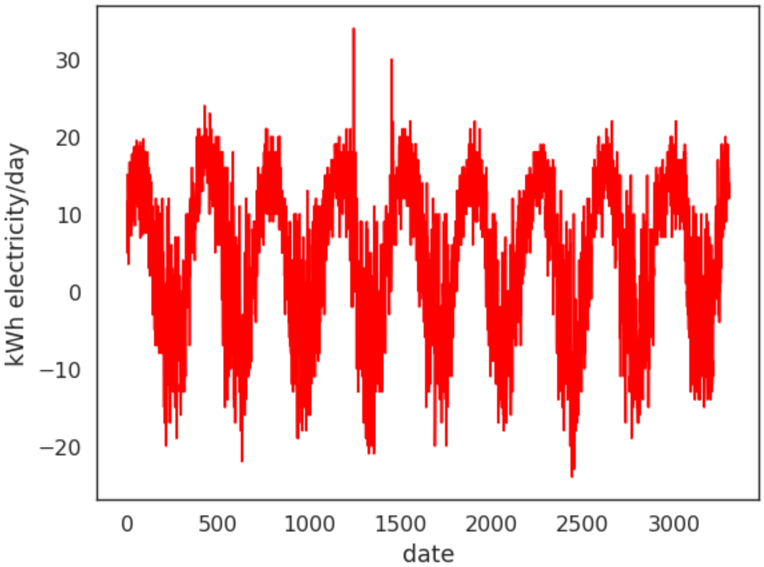
```
# Generate a concise summary of the DataFrame's information
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3304 entries, 0 to 3303
Data columns (total 4 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   date                    3304 non-null   datetime64[ns]
 1   Cumulative_solar_power  3304 non-null   float64
 2   kWh electricity/day     3304 non-null   float64
 3   Gas/day                 3304 non-null   float64
dtypes: datetime64[ns](1), float64(3)
memory usage: 103.4 KB
```

```
# Plotting the 'Global_active_power' against 'datetime'
plt.xlabel("date")  # Label for the x-axis indicating datetime
plt.ylabel("Cumulative_solar_power")  # Label for the y-axis indicating Global_active_power
plt.plot(df['Cumulative_solar_power'], color='blue')  # Plotting Global_active_power values in blue
plt.show()  # Displaying the plot
```




```
# Plotting the 'Global_active_power' against 'datetime'
plt.xlabel("date")  # Label for the x-axis indicating datetime
plt.ylabel("kWh electricity/day")  # Label for the y-axis indicating Global_active_power
plt.plot(df['kWh electricity/day'], color='Red')  # Plotting Global_active_power values in blue
plt.show()  # Displaying the plot
```




```
# Plotting the 'Global_active_power' against 'datetime'
plt.xlabel("date")  # Label for the x-axis indicating datetime
plt.ylabel("Gas/day")  # Label for the y-axis indicating Global_active_power
plt.plot(df['Gas/day'], color='Green')  # Plotting Global_active_power values in blue
plt.show()  # Displaying the plot
```

```
# Calculating the training size for the dataset, which is 80% of the 'Global_active_power' column's length
training_size = int(len(df['Cumulative_solar_power']) * 0.8)
training_size  # Displaying the calculated training size
```

```
2643
```

```
# Function to load data for a sequence model
def load_data(data, seq_len):
    x = []  # List to store input sequences
    y = []  # List to store output sequences
    for i in range(seq_len, len(data)):
        # Append sequences of length 'seq_len' to 'x' and the corresponding next value to 'y'
        x.append(data.iloc[i - seq_len: i, 1])  # Input sequence
        y.append(data.iloc[i, 1])  # Corresponding output value

    return x, y  # Return the input sequences and output values
```

```
# Generating input sequences ('x') and corresponding output values ('y') using the 'load_data' function
x, y = load_data(df, 20)
```

```
# Determining the number of input sequences generated ('x')
len(x)
```

```
3284
```

```
# Splitting the generated sequences and corresponding output values into training and test sets
x_train = x[:training_size]  # Training input sequences
y_train = y[:training_size]  # Corresponding output values for training
x_test = x[training_size:]   # Test input sequences
y_test = y[training_size:]   # Corresponding output values for testing
```

```
# Converting the training and test sets from lists to NumPy arrays
x_train = np.array(x_train)  # Training input sequences as a NumPy array
y_train = np.array(y_train)  # Corresponding output values for training as a NumPy array
x_test = np.array(x_test)    # Test input sequences as a NumPy array
y_test = np.array(y_test)    # Corresponding output values for testing as a NumPy array
```

```
# Displaying the shapes of the training and test sets
print('x_train.shape = ', x_train.shape)  # Shape of the training input sequences
print('y_train.shape = ', y_train.shape)  # Shape of the corresponding output values for training
print('x_test.shape = ', x_test.shape)    # Shape of the test input sequences
print('y_test.shape = ', y_test.shape)    # Shape of the corresponding output values for testing
```

```
x_train.shape =  (2643, 20)
y_train.shape =  (2643,)
x_test.shape =  (641, 20)
y_test.shape =  (641,)
```

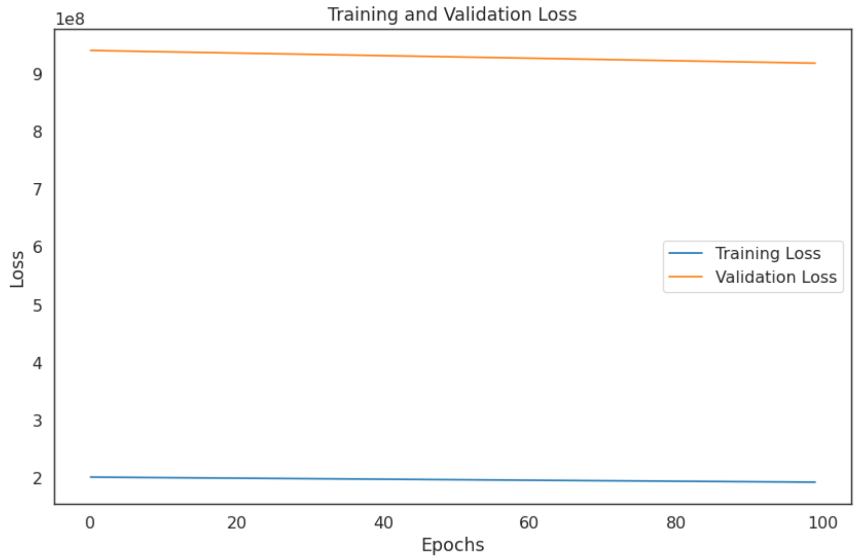
```
# Reshaping the input sequences for compatibility with LSTM model
x_train = np.reshape(x_train, (training_size, 20, 1))  # Reshaping training input sequences to (training_size, 20, 1)
x_test = np.reshape(x_test, (x_test.shape[0], 20, 1))  # Reshaping test input sequences to match the LSTM input shape
```

```
# Displaying the shapes of the reshaped training and test sets
print('x_train.shape = ', x_train.shape)  # Shape of the reshaped training input sequences
print('y_train.shape = ', y_train.shape)  # Shape of the corresponding output values for training
print('x_test.shape = ', x_test.shape)    # Shape of the reshaped test input sequences
print('y_test.shape = ', y_test.shape)    # Shape of the corresponding output values for testing
```

```
x_train.shape =  (2643, 20, 1)
y_train.shape =  (2643,)
x_test.shape =  (641, 20, 1)
y_test.shape =  (641,)
```

```
# Prepare input and output sequences for LSTM
sequence_length = 10  # Length of the sequence to consider
data = df['Cumulative_solar_power'].values
timestamps = df['date'].values.astype(np.int64) // 10**9  # Convert to UNIX timestamp
X, y = [], []
for i in range(len(data) - sequence_length):
    X.append(timestamps[i:i+sequence_length])  # Use datetime as the sequence
    y.append(data[i+sequence_length])  # Target value after the sequence

X = np.array(X)
y = np.array(y)

# Reshape input for LSTM (samples, time steps, features)
X = np.reshape(X, (X.shape[0], sequence_length, 1))

# Splitting the data into training and test sets
train_size = int(len(X) * 0.7)
test_size = len(X) - train_size
X_train, X_test = X[0:train_size], X[train_size:len(X)]
y_train, y_test = y[0:train_size], y[train_size:len(y)]

# Building the LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units=1))

# Compiling the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Training the LSTM model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test), verbose=1)

# Plotting the training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 85/100
73/73 [==============================] - 1s 19ms/step - loss: 194082560.0000
Epoch 86/100
73/73 [==============================] - 2s 23ms/step - loss: 194020272.0000
Epoch 87/100
73/73 [==============================] - 1s 18ms/step - loss: 193954752.0000
Epoch 88/100
73/73 [==============================] - 1s 14ms/step - loss: 193859424.0000
Epoch 89/100
73/73 [==============================] - 1s 14ms/step - loss: 193751920.0000
Epoch 90/100
73/73 [==============================] - 1s 14ms/step - loss: 193688688.0000
Epoch 91/100
73/73 [==============================] - 1s 14ms/step - loss: 193592768.0000
Epoch 92/100
73/73 [==============================] - 1s 14ms/step - loss: 193495088.0000
Epoch 93/100
73/73 [==============================] - 1s 14ms/step - loss: 193420416.0000
Epoch 94/100
73/73 [==============================] - 1s 14ms/step - loss: 193310992.0000
Epoch 95/100
73/73 [==============================] - 1s 14ms/step - loss: 193250992.0000
Epoch 96/100
73/73 [==============================] - 1s 14ms/step - loss: 193164848.0000
Epoch 97/100
73/73 [==============================] - 1s 20ms/step - loss: 193048480.0000
Epoch 98/100
73/73 [==============================] - 2s 23ms/step - loss: 192979728.0000
Epoch 99/100
73/73 [==============================] - 1s 16ms/step - loss: 192885744.0000
Epoch 100/100
73/73 [==============================] - 1s 14ms/step - loss: 192811648.0000
```



```
# Prepare input and output sequences for LSTM
sequence_length = 10  # Length of the sequence to consider
data = df['kWh electricity/day'].values
timestamps = df['date'].values.astype(np.int64) // 10**9  # Convert to UNIX timestamp
X, y = [], []
for i in range(len(data) - sequence_length):
    X.append(timestamps[i:i+sequence_length])  # Use datetime as the sequence
    y.append(data[i+sequence_length])  # Target value after the sequence

X = np.array(X)
y = np.array(y)

# Reshape input for LSTM (samples, time steps, features)
X = np.reshape(X, (X.shape[0], sequence_length, 1))

# Splitting the data into training and test sets
train_size = int(len(X) * 0.7)
test_size = len(X) - train_size
X_train, X_test = X[0:train_size], X[train_size:len(X)]
y_train, y_test = y[0:train_size], y[train_size:len(y)]

# Building the LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units=1))

# Compiling the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Training the LSTM model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test), verbose=1)

# Plotting the training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
73/73 [==============================] - 1s 14ms/step - loss: 95.8705 - val_l
Epoch 85/100
73/73 [==============================] - 1s 14ms/step - loss: 95.7643 - val_l
Epoch 86/100
73/73 [==============================] - 1s 14ms/step - loss: 95.7997 - val_l
Epoch 87/100
73/73 [==============================] - 1s 14ms/step - loss: 95.8559 - val_l
Epoch 88/100
73/73 [==============================] - 1s 14ms/step - loss: 95.9247 - val_l
Epoch 89/100
73/73 [==============================] - 1s 14ms/step - loss: 95.9806 - val_l
Epoch 90/100
73/73 [==============================] - 1s 19ms/step - loss: 95.7204 - val_l
Epoch 91/100
73/73 [==============================] - 1s 20ms/step - loss: 95.7275 - val_l
Epoch 92/100
73/73 [==============================] - 1s 20ms/step - loss: 95.5194 - val_l
Epoch 93/100
73/73 [==============================] - 1s 14ms/step - loss: 95.6127 - val_l
Epoch 94/100
73/73 [==============================] - 1s 14ms/step - loss: 95.6991 - val_l
Epoch 95/100
73/73 [==============================] - 1s 14ms/step - loss: 95.4659 - val_l
Epoch 96/100
73/73 [==============================] - 1s 14ms/step - loss: 95.7856 - val_l
Epoch 97/100
73/73 [==============================] - 1s 14ms/step - loss: 95.7036 - val_l
Epoch 98/100
73/73 [==============================] - 1s 14ms/step - loss: 96.0482 - val_l
Epoch 99/100
73/73 [==============================] - 1s 14ms/step - loss: 95.9393 - val_l
Epoch 100/100
73/73 [==============================] - 1s 14ms/step - loss: 95.6625 - val_l
```