#!/usr/bin/env python

# coding: utf-8

# In[1]:

import os

# In[2]:

import random

# In[3]:

import numpy as np

# In[4]:

import pandas as pd

# In[5]:

import plotly.express as px

# In[6]:

import plotly.graph\_objects as go

# In[7]:

from plotly.subplots import make\_subplots

# In[8]:

import matplotlib.pyplot as plt

# In[9]:

import seaborn as sns

# In[10]:

import statsmodels.api as sm

# In[11]:

from statsmodels.tsa.seasonal import seasonal\_decompose

# In[12]:

energy\_df = pd.read\_csv("Dataset/Energy.csv", parse\_dates=['time'])

# In[13]:

weather\_df = pd.read\_csv("Dataset/Weather.csv", parse\_dates=['dt\_iso'])

# In[14]:

energy\_df.head()

# In[15]:

energy\_df.info()

# In[16]:

weather\_df.head()

# In[17]:

weather\_df["city\_name"].unique()

# In[18]:

weather\_df.query("city\_name == 'Madrid' ").head()

# In[19]:

weather\_df.info()

# In[20]:

city\_counts = weather\_df["city\_name"].value\_counts()

# In[21]:

def display\_bar\_chart(data, xvalues, yvalues, graph\_title, xtitle, ytitle):

fig = px.bar(data, x=xvalues, y=yvalues, color=xvalues)

fig.update\_layout(title=graph\_title, xaxis\_title=xtitle, yaxis\_title=ytitle)

fig.show()

# In[22]:

display\_bar\_chart(city\_counts, city\_counts.index, city\_counts.values, "City wise count", "City Name", "Count")

# In[23]:

energy\_df["year"]=energy\_df["time"].apply(lambda x: x.year)

# In[24]:

energy\_df['time']=pd.to\_datetime(energy\_df['time'], infer\_datetime\_format=True, utc=True)

energy\_df = energy\_df.set\_index('time')

# In[25]:

energy\_year\_counts = energy\_df["year"].value\_counts()

# In[26]:

display\_bar\_chart(energy\_year\_counts, energy\_year\_counts.index, energy\_year\_counts.values, "Energy Year wise count", "Year", "Count")

# In[27]:

weather\_df["year"]=weather\_df.dt\_iso.apply(lambda x: x.year)

# In[28]:

weather\_year\_counts = weather\_df["year"].value\_counts()

# In[29]:

display\_bar\_chart(weather\_year\_counts, weather\_year\_counts.index, weather\_year\_counts.values, "Weather Year wise count", "Year", "Count")

# In[30]:

energy\_df.index.min(),energy\_df.index.max()

energy\_df1 = energy\_df.copy()

# In[31]:

len(energy\_df)

# In[32]:

energy\_df.isnull().sum()

# In[33]:

energy\_df1 = energy\_df.copy()

# In[34]:

energy\_df1.index = pd.to\_datetime(energy\_df1.index, utc=True)

# In[35]:

load\_series = energy\_df1["total load actual"]

# In[36]:

load\_series.sample(15)

# In[37]:

forecast\_series = energy\_df1["total load forecast"]

# In[38]:

forecast\_series.sample(15)

# In[39]:

energy\_demand\_df = pd.concat([load\_series.dropna(), forecast\_series.dropna()], axis=1)

energy\_demand\_df.columns = ['Energy\_Load', 'Energy\_Forecast']

# In[40]:

energy\_demand\_df

# In[41]:

energy\_demand\_df.describe()

# In[42]:

fig = go.Figure()

fig.add\_trace(go.Histogram(x=energy\_demand\_df["Energy\_Load"],name='Energy Load',cumulative\_enabled=False))

fig.add\_trace(go.Histogram(x=energy\_demand\_df["Energy\_Forecast"],name='Energy Forecast',cumulative\_enabled=False))

fig.update\_traces(opacity=1)

fig.update\_layout(barmode='stack',

title\_text='Distribution of energy demanded and TSO 1 day forecasts', xaxis\_title\_text='MW Power',

yaxis\_title\_text='Energy', bargap=0.1, bargroupgap=0 )

fig.show()

# In[43]:

energy\_groups = energy\_demand\_df['Energy\_Load'].groupby(pd.Grouper(freq='A'))

# In[44]:

fig = go.Figure()

fig = make\_subplots(rows=len(energy\_groups), cols=1)

row = 1

for name, group in energy\_groups:

pd.Series(group.values)

fig.add\_trace(go.Scatter(y=group.values, name=name.year), row=row, col=1)

row = row + 1

fig.update\_layout(height=1200, width=800, title\_text="Yearly Consumption")

fig.show()

# In[45]:

energy\_demand\_df

# In[46]:

energy\_df.columns=[x.replace('/','\_') for x in energy\_df.columns]

energy\_df.columns=[x.replace('-','\_') for x in energy\_df.columns]

energy\_df.columns=[x.replace(' ','\_') for x in energy\_df.columns]

# In[47]:

energy\_df.apply(lambda x: len(x.unique()))

# In[48]:

energy\_df = energy\_df.drop(['generation\_fossil\_coal\_derived\_gas','generation\_fossil\_oil\_shale',

'generation\_fossil\_peat', 'generation\_geothermal',

'generation\_hydro\_pumped\_storage\_aggregated', 'generation\_marine',

'generation\_wind\_offshore', 'forecast\_wind\_offshore\_eday\_ahead',

'forecast\_solar\_day\_ahead', 'forecast\_wind\_onshore\_day\_ahead'], axis=1)

# In[49]:

def show\_raw\_visualization(data, nrows, width, height):

#time\_data1 = data.index

fig, axes = plt.subplots(

nrows=nrows, ncols=2, figsize=(width, height), dpi=80, facecolor="w", edgecolor="k"

)

for i in range(len(feature\_keys)):

key = feature\_keys[i]

c = colors[i % (len(colors))]

t\_data = data[key]

t\_data.index = data.index

t\_data.head()

ax = t\_data.plot(

ax=axes[i // 2, i % 2],

color=c,

title="{} - {}".format(key, titles[i]),

rot=25,

)

ax.legend([titles[i]])

plt.tight\_layout()

colors = ["blue", "orange", "green", "red", "purple", "brown", "pink", "gray", "olive", "cyan"]

titles = [ 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh',

'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'MWh', 'Euro', 'Euro']

# In[50]:

feature\_keys = ['generation\_biomass', 'generation\_fossil\_brown\_coal\_lignite', 'generation\_fossil\_gas', 'generation\_fossil\_hard\_coal']

show\_raw\_visualization(energy\_df[:24\*7\*4],nrows=2,width=12,height=4)

# In[51]:

energy\_df.describe().transpose()

# In[52]:

def show\_hist\_boxplot(cols):

cols = cols

for col in cols:

plt.figure(figsize=(12,2))

plt.subplot(1,2,1)

sns.histplot(data=energy\_df, x=col, bins=30, kde=True)

plt.subplot(1,2,2)

sns.boxplot(data=energy\_df, x=col)

plt.show()

# In[53]:

energy\_df.loc[energy\_df.generation\_fossil\_brown\_coal\_lignite ==0, 'generation\_fossil\_brown\_coal\_lignite'] = np.nan

# In[54]:

energy\_df.loc[energy\_df.generation\_biomass <110, 'generation\_biomass'] = np.nan

energy\_df.loc[energy\_df.generation\_fossil\_gas ==0, 'generation\_fossil\_gas'] = np.nan

energy\_df.loc[energy\_df.generation\_fossil\_oil <50, 'generation\_fossil\_oil'] = np.nan

energy\_df.loc[energy\_df.generation\_nuclear <2000, 'generation\_nuclear'] = np.nan

energy\_df.loc[energy\_df.generation\_other\_renewable <20, 'generation\_other\_renewable'] = np.nan

energy\_df.loc[energy\_df.generation\_waste <50, 'generation\_waste'] = np.nan

# In[55]:

def show\_heat\_map(df,cols, width=5, height=5):

width=width

height=height

df=df

cols=cols

plt.figure(figsize=(width,height))

sns.heatmap(df[cols].corr(), annot=True)

plt.show()

# In[56]:

cols =['generation\_hydro\_pumped\_storage\_consumption', 'generation\_fossil\_gas', 'generation\_fossil\_oil',

'generation\_hydro\_water\_reservoir', 'price\_actual', 'generation\_fossil\_hard\_coal', 'generation\_fossil\_brown\_coal\_lignite', 'total\_load\_actual']

show\_heat\_map(energy\_df,cols,10,2)

# In[57]:

cols = ['pressure', 'humidity', 'wind\_speed', 'wind\_deg', 'clouds\_all', 'weather\_id']

for col in cols:

weather\_df[col]=weather\_df[col].values.astype(np.float64)

weather\_df['time']=pd.to\_datetime(weather\_df['dt\_iso'], infer\_datetime\_format=True, utc=True)

weather\_df = weather\_df.set\_index('time')

weather\_df = weather\_df.drop(['dt\_iso'], axis=1)

# In[58]:

weather\_df.apply(lambda x: len(x.unique()))

# In[59]:

weather\_df = weather\_df.drop(['rain\_1h','weather\_main', 'weather\_description','weather\_icon', 'weather\_id'], axis=1)

# In[60]:

feature\_keys = ['temp', 'pressure', 'humidity','wind\_speed']

show\_raw\_visualization(weather\_df[:24\*7\*4],nrows=2,width=12,height=4)

# In[61]:

weather\_df.describe().transpose()

# In[62]:

print(weather\_df.duplicated().sum())

# In[63]:

features = ['pressure','wind\_speed', 'wind\_deg', 'rain\_3h']

def show\_hist\_boxplot2(features):

features = features

for feature in features:

plt.figure(figsize=(12,3))

plt.subplot(1,2,1)

sns.histplot(data=weather\_df, x=feature, bins=30, kde=True, hue='city\_name')

plt.subplot(1,2,2)

sns.boxplot(data=weather\_df, x=feature, y='city\_name')

plt.show()

# In[64]:

weather\_df.loc[weather\_df.pressure >1030, 'pressure'] = np.nan

weather\_df.loc[weather\_df.pressure <800, 'pressure'] = np.nan

weather\_df.loc[weather\_df.wind\_speed >100, 'wind\_speed'] = np.nan

weather\_df.loc[weather\_df.rain\_3h >2, 'rain\_3h'] = np.nan

# In[65]:

weather\_df.interpolate(method='linear', limit\_direction='forward', inplace=True, axis=0)

# In[66]:

wv = weather\_df['wind\_speed']

wd\_rad = weather\_df['wind\_deg']\*np.pi / 180

weather\_df['Wx'] = wv\*np.cos(wd\_rad)

weather\_df['Wy'] = wv\*np.sin(wd\_rad)

# In[67]:

df\_bcn, df\_bil, df\_mdr, df\_sev, df\_val = [x for \_, x in weather\_df.groupby('city\_name')]

dfs = [df\_bcn, df\_bil, df\_mdr, df\_sev, df\_val]

# In[68]:

data = energy\_df

for df in dfs:

city = df['city\_name'].unique()

city\_str = str(city).replace("'", "").replace('[', '').replace(']', '').replace(' ', '')

df = df.add\_suffix('\_{}'.format(city\_str))

data = data.merge(df, on=['time'], how='outer')

data = data.drop('city\_name\_{}'.format(city\_str), axis=1)

# In[69]:

cols =['humidity\_Valencia','humidity\_Barcelona', 'humidity\_Seville','humidity\_Bilbao',

'temp\_min\_Valencia','temp\_Valencia','temp\_max\_Valencia', 'temp\_Seville',

'humidity\_Madrid','temp\_Madrid','wind\_speed\_Valencia', 'total\_load\_actual']

show\_heat\_map(data, cols, 10, 2)

# In[70]:

import missingno as msno

# In[71]:

def process\_missing\_data(data):

missing\_df=pd.DataFrame()

missing\_df["missing\_row\_count"]=data.isnull().sum()

missing\_df["missing\_row\_data"]=data.isnull().sum()/len(data)

missing\_df=missing\_df.loc[missing\_df["missing\_row\_data"]>0].sort\_values(by="missing\_row\_data", ascending=False)

missing\_df["top"]=missing\_df["missing\_row\_data"].map(lambda x:1 if x>=0.05 else 0)

return missing\_df

# In[72]:

process\_missing\_data(energy\_df)

# In[73]:

energy\_df.eq(0)

# In[74]:

energy\_df.eq(0).sum().to\_frame(name="Missing Count")

# In[75]:

msno.bar(energy\_df,figsize=(12,5),

sort="descending",

fontsize=12);

# In[76]:

msno.dendrogram(df=energy\_df,figsize=(12,5),fontsize=12)

# In[77]:

msno.heatmap(df=energy\_df)

# In[78]:

energy\_df.isnull().sum()

# In[79]:

energy\_df.dropna(inplace=True)

# In[80]:

energy\_df.isnull().sum()

# In[81]:

def show\_decompose(result):

result=result

s1 = result.observed

s2 = result.seasonal

s3 = result.trend

s4 = result.resid

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(16, 8), sharex=True)

fig.suptitle('Descomposition of multiplicative time series\n jan 2015')

ax1.plot(t, s2, 'k')

ax1.set\_ylabel('Seasonal')

ax2.plot(t, s3, 'k')

ax2.set\_ylabel('Trend')

ax3.plot(t, s4, 'k')

ax3.set\_xlabel('time (H)')

ax3.set\_ylabel('Resid')

plt.show()

# In[82]:

def show\_correlogramas():

fig, axs = plt.subplots(1, 2, figsize=(16, 3))

sm.graphics.tsa.plot\_acf(energy\_df['total\_load\_actual'].values.squeeze(), ax=axs[0], title=(f"Autocorrelation ({'total\_load\_actual'})"), lags=40)

sm.graphics.tsa.plot\_pacf(energy\_df['total\_load\_actual'].values.squeeze(), ax=axs[1], title=(f"Partial Autocorrelation ({'total\_load\_actual'})"), lags=40)

plt.grid(color='gray', linestyle='-', linewidth=1)

plt.show()

# In[83]:

show\_correlogramas()

# In[84]:

mape = np.mean(np.abs((energy\_df['total\_load\_actual'] - energy\_df['total\_load\_forecast']) / energy\_df['total\_load\_actual'])) \* 100

print('MAPE of the forecasted data present in DataFrame:', mape)

# In[85]:

energy\_df = energy\_df1.copy()

temp\_df = energy\_df.copy()

# In[86]:

from sklearn.preprocessing import MinMaxScaler

# In[87]:

dataset = temp\_df['total load actual'].dropna().values

dataset = dataset.astype('float32')

dataset = np.reshape(dataset, (-1, 1))

# In[88]:

scaler = MinMaxScaler(feature\_range=(0, 1))

# In[89]:

dataset = scaler.fit\_transform(dataset)

# In[90]:

train\_size = int(len(dataset) \* 0.80)

# In[91]:

test\_size = len(dataset) - train\_size

# In[92]:

train, test = dataset[0:train\_size,:], dataset[train\_size:len(dataset),:]

# In[93]:

def create\_dataset(dataset, look\_back=1):

X, Y = [], []

for i in range(len(dataset)-look\_back-1):

a = dataset[i:(i+look\_back), 0]

X.append(a)

Y.append(dataset[i + look\_back, 0])

return np.array(X), np.array(Y)

# In[94]:

look\_back = 25

# In[95]:

X\_train, trainY = create\_dataset(train, look\_back)

# In[96]:

X\_test, testY = create\_dataset(test, look\_back)

# In[97]:

X\_train = np.reshape(X\_train, (X\_train.shape[0], 1, X\_train.shape[1]))

# In[98]:

X\_test = np.reshape(X\_test, (X\_test.shape[0], 1, X\_test.shape[1]))

# In[99]:

print("Shapes: \nTraining set: {}, Testing set: {}".format(X\_train.shape, X\_test.shape))

print("Sample from training set: \n{}".format(X\_train[0]))

# In[100]:

from statsmodels.tsa.ar\_model import AR

# In[101]:

from statsmodels.tsa.arima\_model import ARMA

# In[102]:

from statsmodels.tsa.arima\_model import ARIMA

# In[103]:

from sklearn.metrics import mean\_squared\_error

# In[104]:

from sklearn.metrics import mean\_absolute\_error

# In[105]:

model = AR(train)

model\_fit = model.fit()

# In[106]:

test\_predict = model\_fit.predict(start=len(train), end=len(train)+len(test)-1, dynamic=False)

# In[107]:

test\_predict = scaler.inverse\_transform(test\_predict.reshape(-1, 1))

# In[108]:

Y\_test = scaler.inverse\_transform(test)

# In[109]:

print('Test Mean Absolute Error:', mean\_absolute\_error(Y\_test, test\_predict))

print('Test Root Mean Squared Error:',np.sqrt(mean\_squared\_error(Y\_test, test\_predict)))

# In[110]:

print(len(Y\_test), len(test\_predict))

# In[111]:

mape = np.mean(np.abs((Y\_test - test\_predict) / Y\_test)) \* 100

print("Testing MAPE: {}".format(mape))

# In[112]:

idx = 200

time\_step\_range=[x for x in range(idx)]

# In[113]:

plt.figure(figsize=(8,4))

plt.plot(time\_step\_range, Y\_test[:idx], marker='.', label="actual")

plt.plot(time\_step\_range, test\_predict[:idx], 'r', label="prediction")

# plt.tick\_params(left=False, labelleft=True) #remove ticks

plt.tight\_layout()

sns.despine(top=True)

plt.subplots\_adjust(left=0.07)

plt.ylabel('TOTAL Load', size=15)

plt.xlabel('Time step', size=15)

plt.legend(fontsize=15)

plt.show();

# In[114]:

model = ARMA(train, order=(2, 1))

model\_fit = model.fit(disp=False)

# In[115]:

test\_predict = model\_fit.predict(start=len(train), end=len(train)+len(test)-1, dynamic=False)

# In[116]:

test\_predict = scaler.inverse\_transform(test\_predict.reshape(-1, 1))

Y\_test = scaler.inverse\_transform(test)

# In[117]:

print('Test Mean Absolute Error:', mean\_absolute\_error(Y\_test, test\_predict))

print('Test Root Mean Squared Error:',np.sqrt(mean\_squared\_error(Y\_test, test\_predict)))

# In[118]:

mape = np.mean(np.abs((Y\_test - test\_predict) / Y\_test)) \* 100

print("Testing MAPE: {}".format(mape))

# In[119]:

idx = 200

time\_step\_range=[x for x in range(idx)]

# In[120]:

plt.figure(figsize=(8,4))

plt.plot(time\_step\_range, Y\_test[:idx], marker='.', label="actual")

plt.plot(time\_step\_range, test\_predict[:idx], 'r', label="prediction")

plt.tight\_layout()

sns.despine(top=True)

plt.subplots\_adjust(left=0.07)

plt.ylabel('TOTAL Load', size=15)

plt.xlabel('Time step', size=15)

plt.legend(fontsize=15)

plt.show();

# In[121]:

model = ARIMA(train, order=(1, 1, 1))

model\_fit = model.fit(disp=False)

# In[122]:

test\_predict = model\_fit.predict(start=len(train), end=len(train)+len(test)-1, dynamic=False)

# In[123]:

test\_predict = scaler.inverse\_transform(test\_predict.reshape(-1, 1))

Y\_test = scaler.inverse\_transform(test)

# In[124]:

print('Test Mean Absolute Error:', mean\_absolute\_error(Y\_test, test\_predict))

print('Test Root Mean Squared Error:',np.sqrt(mean\_squared\_error(Y\_test, test\_predict)))

# In[125]:

mape = np.mean(np.abs((Y\_test - test\_predict) / Y\_test)) \* 100

print("Testing MAPE: {}".format(mape))

# In[126]:

idx = 200

time\_step\_range=[x for x in range(idx)]

# In[127]:

plt.figure(figsize=(8,4))

plt.plot(time\_step\_range, Y\_test[:idx], marker='.', label="actual")

plt.plot(time\_step\_range, test\_predict[:idx], 'r', label="prediction")

# plt.tick\_params(left=False, labelleft=True) #remove ticks

plt.tight\_layout()

sns.despine(top=True)

plt.subplots\_adjust(left=0.07)

plt.ylabel('TOTAL Load', size=15)

plt.xlabel('Time step', size=15)

plt.legend(fontsize=15)

plt.show()

# In[128]:

from tensorflow.keras.models import Sequential

# In[129]:

from tensorflow.keras.layers import Dense, LSTM, Dropout

# In[130]:

temp\_df = energy\_df.copy()

# In[131]:

dataset = temp\_df['total load actual'].dropna().values

# In[132]:

dataset = dataset.astype('float32')

# In[133]:

dataset = np.reshape(dataset, (-1, 1))

# In[134]:

scaler = MinMaxScaler(feature\_range=(0, 1))

# In[135]:

dataset = scaler.fit\_transform(dataset)

# In[136]:

train\_size = int(len(dataset) \* 0.80)

# In[137]:

test\_size = len(dataset) - train\_size

# In[138]:

train, test = dataset[0:train\_size,:], dataset[train\_size:len(dataset),:]

# In[139]:

def create\_dataset(dataset, look\_back=1):

X, Y = [], []

for i in range(len(dataset)-look\_back-1):

a = dataset[i:(i+look\_back), 0]

X.append(a)

Y.append(dataset[i + look\_back, 0])

return np.array(X), np.array(Y)

# In[140]:

look\_back = 25

X\_train, Y\_train = create\_dataset(train, look\_back)

X\_test, Y\_test = create\_dataset(test, look\_back)

# In[141]:

X\_train = np.reshape(X\_train, (X\_train.shape[0], 1, X\_train.shape[1]))

X\_test = np.reshape(X\_test, (X\_test.shape[0], 1, X\_test.shape[1]))

# In[142]:

def create\_lstm\_model():

model = Sequential()

model.add(LSTM(100, input\_shape=(X\_train.shape[1], X\_train.shape[2])))

model.add(Dropout(0.2))

model.add(Dense(1))

return model

# In[143]:

model = create\_lstm\_model()

# In[144]:

model.compile(loss='mean\_squared\_error', optimizer='adam')

# In[145]:

history = model.fit(X\_train, Y\_train, epochs=120, batch\_size=70, validation\_data=(X\_test, Y\_test),verbose=1, shuffle=False)

# In[146]:

model.summary()

# In[147]:

train\_predict = model.predict(X\_train)

test\_predict = model.predict(X\_test)

# In[148]:

train\_predict = scaler.inverse\_transform(train\_predict)

Y\_train = scaler.inverse\_transform([Y\_train])

test\_predict = scaler.inverse\_transform(test\_predict)

Y\_test = scaler.inverse\_transform([Y\_test])

# In[149]:

print('Train Mean Absolute Error:', mean\_absolute\_error(Y\_train[0], train\_predict[:,0]))

print('Train Root Mean Squared Error:',np.sqrt(mean\_squared\_error(Y\_train[0], train\_predict[:,0])))

print('Test Mean Absolute Error:', mean\_absolute\_error(Y\_test[0], test\_predict[:,0]))

print('Test Root Mean Squared Error:',np.sqrt(mean\_squared\_error(Y\_test[0], test\_predict[:,0])))

# In[150]:

mape\_train = np.mean(np.abs((Y\_train[0] - train\_predict[:,0]) / Y\_train[0])) \* 100

mape\_test = np.mean(np.abs((Y\_test[0] - test\_predict[:,0]) / Y\_test[0])) \* 100

# In[151]:

print("Train MAPE: {}, Test MAPE: {}".format(mape\_train, mape\_test))

# In[152]:

plt.figure(figsize=(8,4))

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Test Loss')

plt.title('model loss')

plt.ylabel('loss')

plt.xlabel('epochs')

plt.legend(loc='upper right')

plt.show()

# In[153]:

idx = 200

time\_step\_range=[x for x in range(idx)]

# In[154]:

plt.figure(figsize=(8,4))

plt.plot(time\_step\_range, Y\_test[0][:idx], marker='.', label="actual")

plt.plot(time\_step\_range, test\_predict[:,0][:idx], 'r', label="prediction")

plt.tight\_layout()

sns.despine(top=True)

plt.subplots\_adjust(left=0.07)

plt.ylabel('TOTAL Load', size=15)

plt.xlabel('Time step', size=15)

plt.legend(fontsize=15)

plt.show();