In [1032]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 15, 10
import numpy as np
import seaborn as sns
import datetime
import sys
from IPython.display import Markdown as md
import missingno as msno
import warnings
warnings.filterwarnings("ignore")
from sklearn.metrics import mean_absolute_error
from datetime import datetime, timedelta
from pandas.tseries.holiday import USFederalHolidayCalendar
from sklearn import linear_model
from statsmodels.tsa.arima_model import ARIMA
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
    from sklearn.ensemble import AdaBoostRegressor
```

In [1033]:
```
# X_Axis values for time stamp in a plot
global x_duration
```

Getting Input from the User

In [1335]:
```
basePath = './data/'

# Take input from the user for the type of household he wants the prediction for
typeofhome = input("Select a home (B, C, F) ")

if typeofhome == 'B':
    print('B')
    selected_home_kwh_B = pd.read_csv(basePath + "Home B - 2014/HomeB-meter1_2014.csv")
    selected_home_weather_B = pd.read_csv(basePath + "Home B - 2014/homeB2014.csv")
elif typeofhome == 'C':
    print('C')
    selected_home_kwh_C = pd.read_csv(basePath + "Home C -2015/HomeC-meter1_2015.csv")
    selected_home_weather_C = pd.read_csv(basePath + "Home C -2015/homeC2015.csv")
elif typeofhome == 'F':
    print('F')
    selected_home_kwh_F = pd.read_csv(basePath + "Home F - 2016/HomeF-meter3_2016.csv")
    selected_home_weather_F = pd.read_csv(basePath + "Home F - 2016/homeF2016.csv")
```

```
Select a home (B, C, F) B
B
```

In [1034]:
```
selected_home_kwh_B = pd.read_csv(basePath + "Home B - 2014/HomeB-meter1_2014.csv")
selected_home_weather_B = pd.read_csv(basePath + "Home B - 2014/homeB2014.csv")
selected_home_kwh_C = pd.read_csv(basePath + "Home C -2015/HomeC-meter1_2015.csv")
selected_home_weather_C = pd.read_csv(basePath + "Home C -2015/homeC2015.csv")
selected_home_kwh_F = pd.read_csv(basePath + "Home F - 2016/HomeF-meter3_2016.csv")
selected_home_weather_F = pd.read_csv(basePath + "Home F - 2016/homeF2016.csv")
```

In [1035]:
```
# Change epoch time to datetime

def convertToDatetime(df):
    index = 0
    df['DateTime'] = df['time']
    for row in df['time']:
        df['DateTime'][index] = datetime.fromtimestamp(row).strftime('%Y-%m-%d %H:%M:%S')
        index = index + 1

convertToDatetime(selected_home_weather_B)
convertToDatetime(selected_home_weather_C)
convertToDatetime(selected_home_weather_F)
```

In [ ]:

In [ ]:

In [ ]:

Merging the Power Consumption data with the Weather Data for each Home

In [1036]:
```
temp_power_df_B = selected_home_kwh_B.filter(['Date & Time','use [kW]'],axis=1)
temp_power_df_C = selected_home_kwh_C.filter(['Date & Time','use [kW]'],axis=1)
temp_power_df_F = selected_home_kwh_F.filter(['Date & Time','Usage [kW]'],axis=1)

temp_power_df_F['use [kW]'] = temp_power_df_F['Usage [kW]']
temp_power_df_F = temp_power_df_F.drop('Usage [kW]',axis=1)

def convertToCommonTimeFormat_HouseF(df):
    indices = []
    for i in range(0, len(df), 30):
        indices.append(i)
    common = df.iloc[indices]
    common = common.reset_index()
    del common['index']
    return common

temp_power_df_F = convertToCommonTimeFormat_HouseF(temp_power_df_F)


weather_df_B = selected_home_weather_B.copy()
weather_df_C = selected_home_weather_C.copy()
weather_df_F = selected_home_weather_F.copy()

temp_power_df_B['DateTime'] = temp_power_df_B['Date & Time']
temp_power_df_C['DateTime'] = temp_power_df_C['Date & Time']
temp_power_df_F['DateTime'] = temp_power_df_F['Date & Time']


def add_datetime_powerdf(df):
    for i in range(0, df.shape[0]):
        df['DateTime'][i] = str(pd.to_datetime(df['DateTime'][i]).replace(minute=0))

add_datetime_powerdf(temp_power_df_B)
add_datetime_powerdf(temp_power_df_C)
add_datetime_powerdf(temp_power_df_F)


# Merging the power dataframe and the weather dataframe
merged_df_B = pd.merge(temp_power_df_B, weather_df_B, on='DateTime')
merged_df_C = pd.merge(temp_power_df_C, weather_df_C, on='DateTime')
merged_df_F = pd.merge(temp_power_df_F, weather_df_F, on='DateTime')


# Converting Non-Numerical Categorical features to Numerical Features

merged_df_B = pd.get_dummies(merged_df_B, columns=['summary'])
merged_df_B.head()

merged_df_C = pd.get_dummies(merged_df_C, columns=['summary'])
merged_df_C.head()

merged_df_F = pd.get_dummies(merged_df_F, columns=['summary'])
merged_df_F.head()

print("Dataframes are massaged and merged.")
```

```
Dataframes are massaged and merged.
```

In [ ]:

In [1037]:
```
# Adding Seasons to the dataframe for each home Type

def add_seasons(df):
    # Initializing seasons with random values
    df['Spring'] = df['windBearing']
    df['Summer'] = df['windBearing']
    df['Fall']   = df['windBearing']
    df['Winter'] = df['windBearing']

    c1 = 0
    c2 = 0
    c3 = 0
    c4 = 0
    for index, row in df.iterrows():

        month = pd.to_datetime(df['Date & Time'][index]).month
        if month in range(3,6):
            c1 = c1 + 1
            df['Spring'][index] = 1
            df['Summer'][index] = 0
            df['Fall'][index] = 0
            df['Winter'][index] = 0
        elif month in range(6,9):
            c2 = c2 + 1
            df['Spring'][index] = 0
            df['Summer'][index] = 1
            df['Fall'][index] = 0
            df['Winter'][index] = 0
        elif month in range(9,12):
            c3 = c3 + 1
            df['Spring'][index] = 0
            df['Summer'][index] = 0
            df['Fall'][index] = 1
            df['Winter'][index] = 0
        elif month in range(12,13) or month in range(1,3):
            c4 = c4 + 1
            df['Spring'][index] = 0
            df['Summer'][index] = 0
            df['Fall'][index] = 0
            df['Winter'][index] = 1

    print(c1,' ',c2,' ',c3, ' ',c4)


# Creating Seasons feature for Home Type B
df_B = merged_df_B.copy()
add_seasons(df_B)

# Creating Seasons feature for Home Type C
df_C = merged_df_C.copy()
add_seasons(df_C)

# Creating Seasons feature for Home Type F
df_F = merged_df_F.copy()
add_seasons(df_F)
```

```
4414   4416   4374   4320
4414   4416   4374   26564
4414   4416   4374   3598
```

In [1038]:
```
# Creating Weekday or Weekend column for each Home Type

def add_day_type(df):
    df['Weekday'] = df['windBearing']
    df['Weekend'] = df['windBearing']

    for index, row in df.iterrows():
        dayofweek = pd.to_datetime(merged_df['Date & Time'][index]).dayofweek
        if dayofweek in range(0,5):
            df['Weekday'][index] = 1
            df['Weekend'][index] = 0
        else:
            df['Weekday'][index] = 0
            df['Weekend'][index] = 1

# For Home Type B
add_day_type(df_B)

# For Home Type C
add_day_type(df_C)

# For Home Type F
add_day_type(df_F)
```

In [1039]:
```
# Creating isUSHoliday column for each Home Type

def convert_to_date(hol):
    for i in range(0, len(hol)):
        hol[i] = pd.to_datetime(hol[i]).date()

def add_isUSHoliday(df):
    df['isUSHoliday'] = df.windBearing
    for index, row in df.iterrows():
        if pd.to_datetime(row['Date & Time']).date() in holidays:
            df['isUSHoliday'][index] = 1
        else:
            df['isUSHoliday'][index] = 0

convert_to_date(holidays)

cal = USFederalHolidayCalendar()
holidays = cal.holidays(start=str(minDate), end=str(maxDate)).to_pydatetime()

minDate = pd.to_datetime(df_B['Date & Time']).min().date()
maxDate = pd.to_datetime(df_B['Date & Time']).max().date()
add_isUSHoliday(df_B)

minDate = pd.to_datetime(df_C['Date & Time']).min().date()
maxDate = pd.to_datetime(df_C['Date & Time']).max().date()
add_isUSHoliday(df_C)

minDate = pd.to_datetime(df_F['Date & Time']).min().date()
maxDate = pd.to_datetime(df_F['Date & Time']).max().date()
add_isUSHoliday(df_F)
```

In [1200]:
```
df_B.shape
```

Out[1200]: (17524, 42)

In [1201]:
```
df_C.shape
```

Out[1201]: (39768, 42)

In [1202]:
```
df_F.shape
```

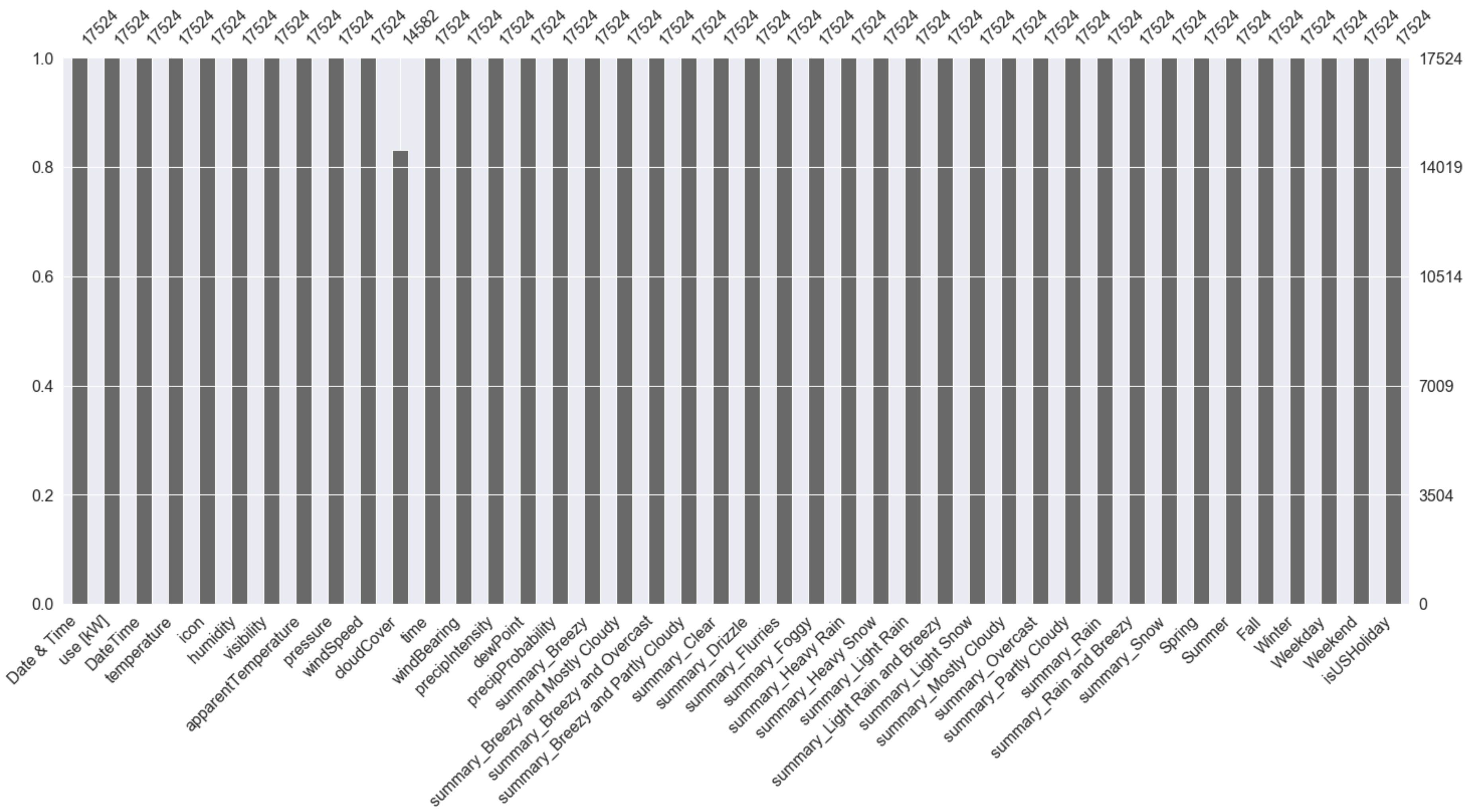
Out[1202]: (16802, 41)

Nullity Check

For Home Type B

In [1040]:
```
# Checking if null values are present in the dataset
msno.bar(df_B)
```

Out[1040]: <matplotlib.axes._subplots.AxesSubplot at 0x1c522de890>

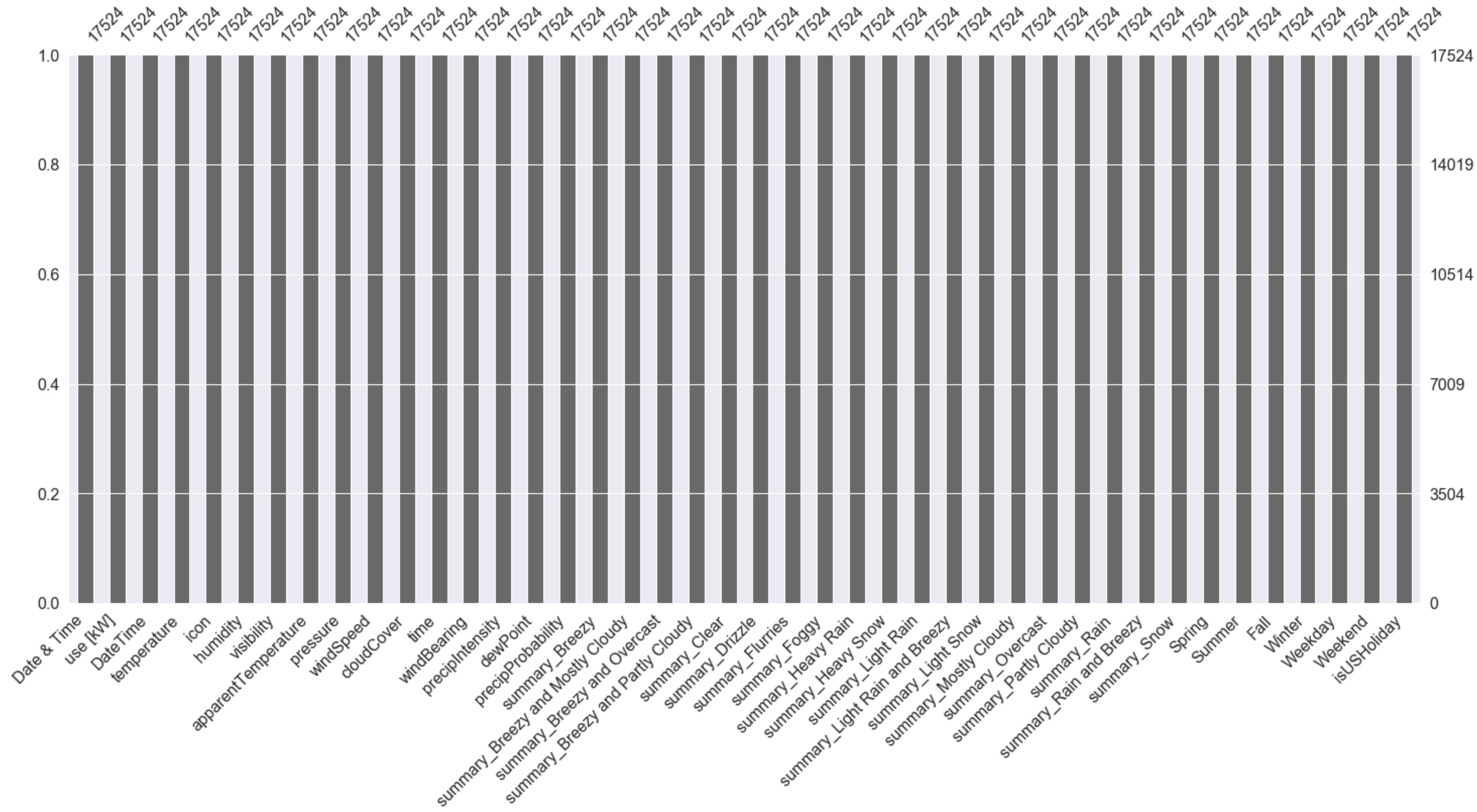


Null values found in the 'cloudCover' feature in the dataset.

In [1041]:
```
df_B['cloudCover'].fillna((df_B['cloudCover'].mean()), inplace=True)
```

In [1042]:
```
msno.bar(df_B)
```

Out[1042]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1c479f6ad0>
```



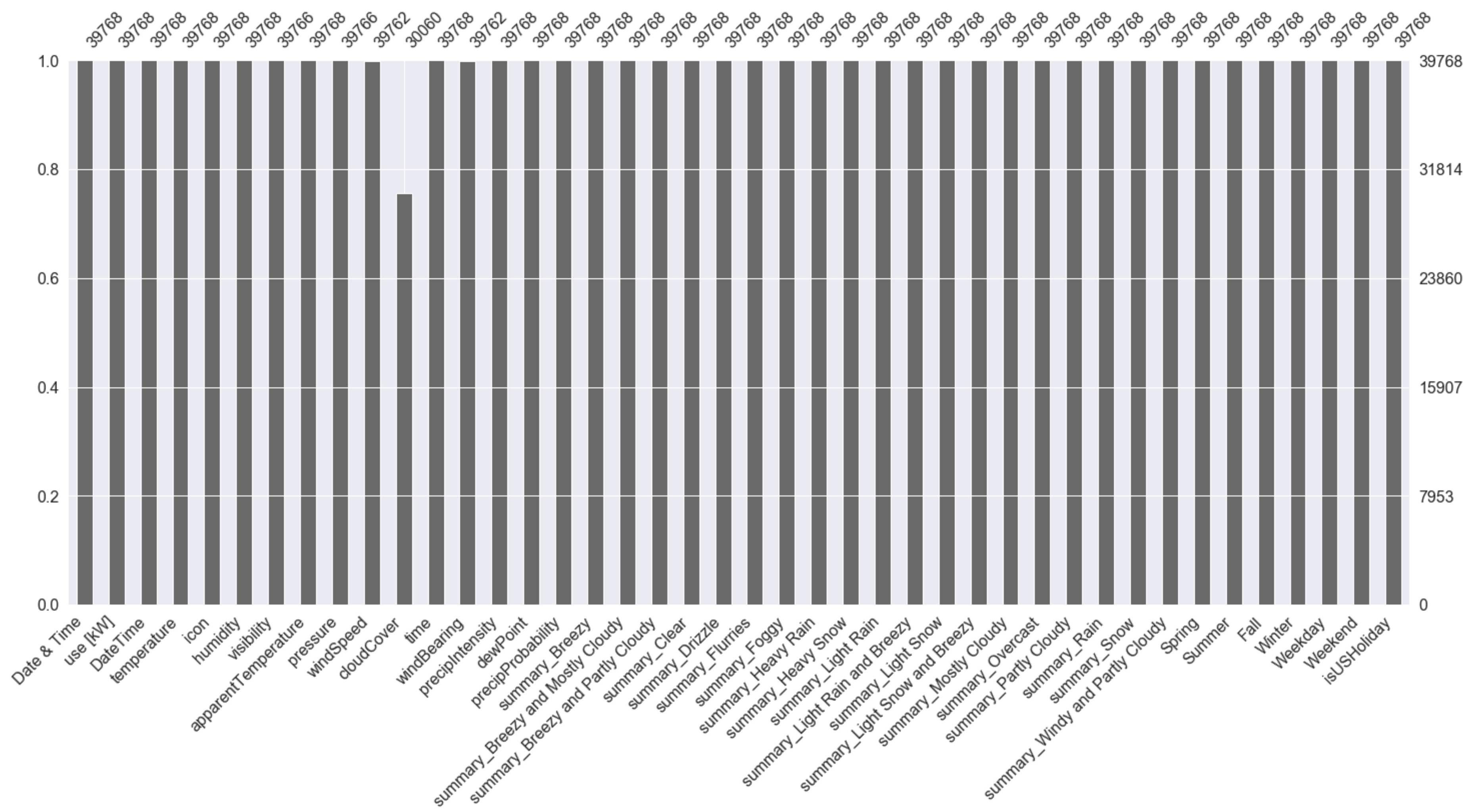
Replaced the null values with the mean of the feature values.

In [ ]:

For Home Type C

In [1043]:
```
# Checking if null values are present in the dataset
msno.bar(df_C)
```

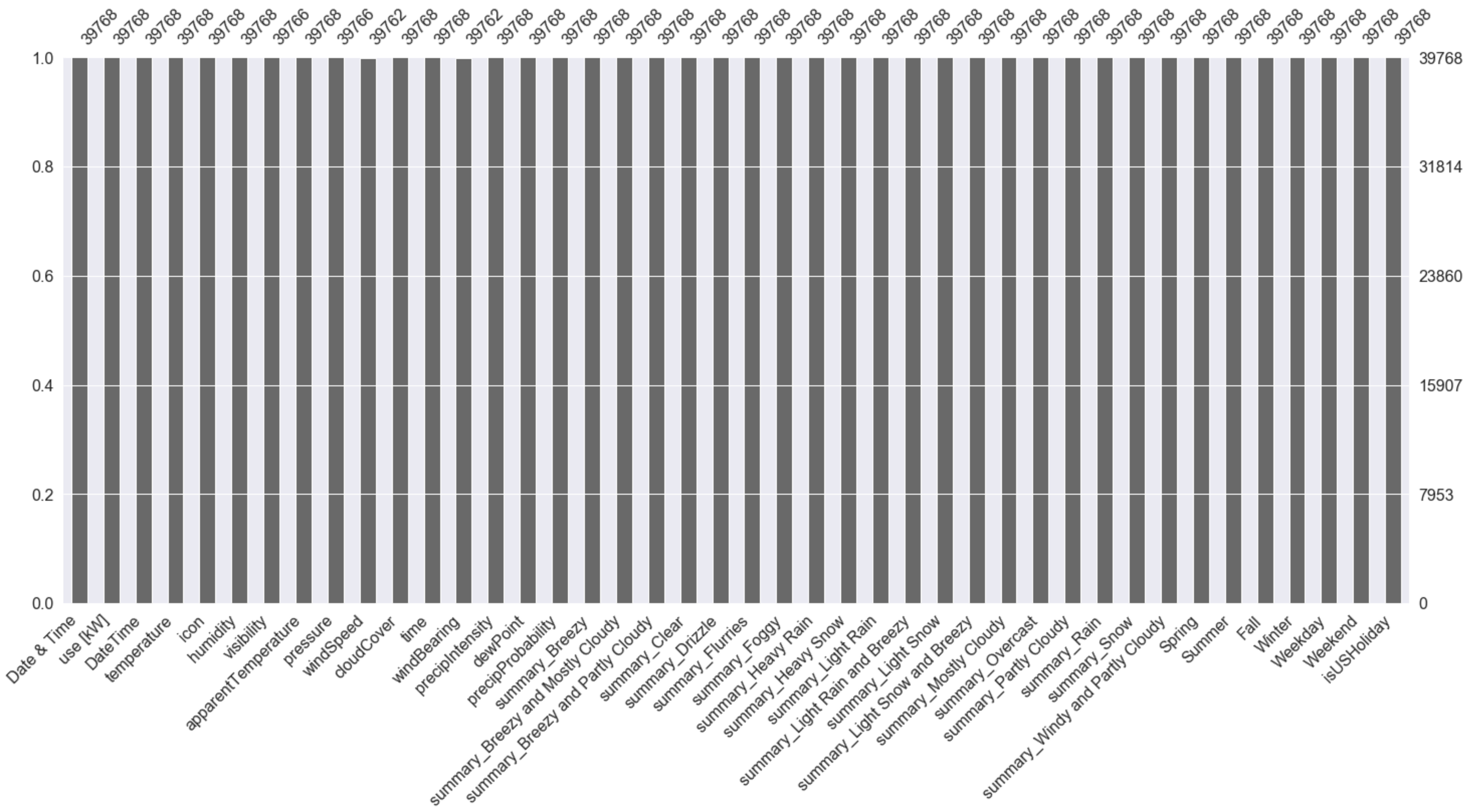
Out[1043]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a3ee92590>
```



In [1044]:
```
df_C['cloudCover'].fillna((df_C['cloudCover'].mean()), inplace=True)
```

In [1045]:
```
msno.bar(df_C)
```

Out[1045]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5b511490>



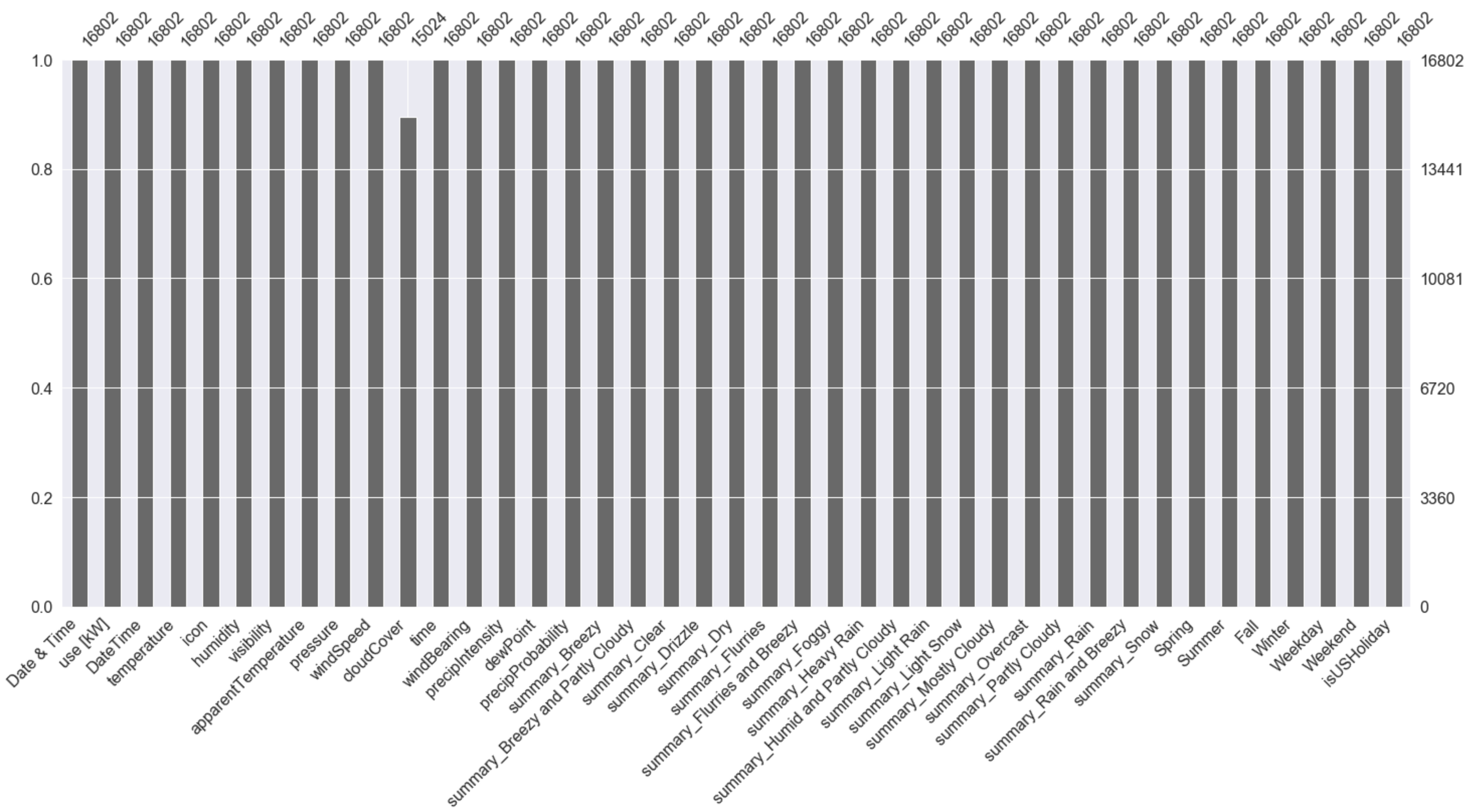
Replaced the null values with the mean of the feature values.

In [ ]:

For Home Type F

In [1046]:
```
# Checking if null values are present in the dataset
msno.bar(df_F)
```

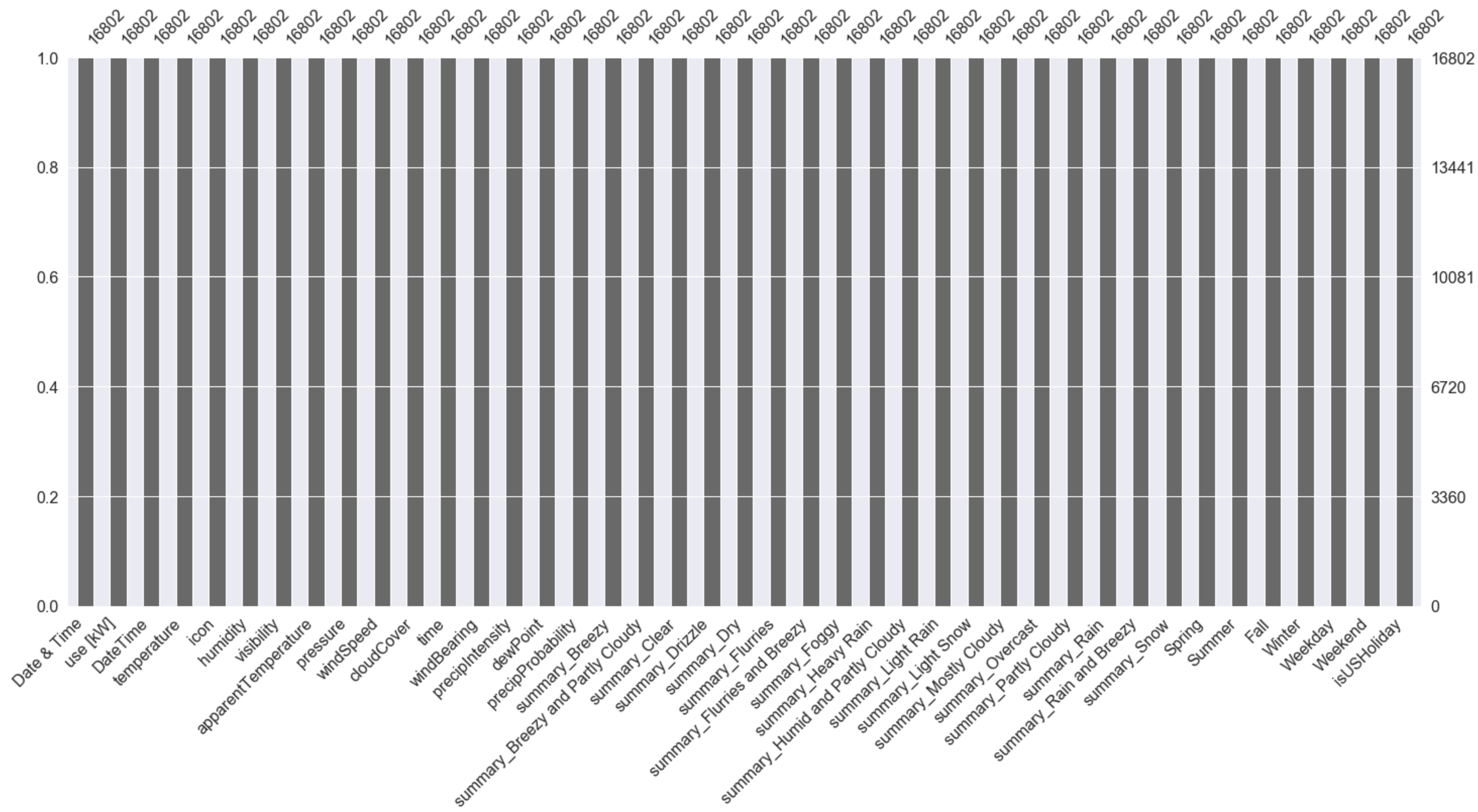
Out[1046]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5b51b810>



In [1047]:
```
df_F['cloudCover'].fillna((df_F['cloudCover'].mean()), inplace=True)
```

In [1048]:
```
msno.bar(df_F)
```

Out[1048]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1c5d9f3950>
```



Replaced the null values with the mean of the feature values.

Applying the Naive Method of Prediction

For Home Type B

In [1049]:
```
# Getting input date from the user
naive_dataframe_B = df_B.copy()

print('Enter Date between ', min(naive_dataframe_B['Date & Time']),' and ', max(naive_dataframe_B['Date & Time'
Year, Month, Day, Hour, Minute = input("Enter Timestamp YYYY,MM,DD,HH,MM: ").split()

input_date_B = datetime(int(Year), int(Month), int(Day), int(Hour), int(Minute) , 0)
print("You entered date = ",str(input_date_B))
```

```
Enter Date between  2014-01-01 00:00:00  and  2014-12-31 23:30:00
Enter Timestamp YYYY,MM,DD,HH,MM: 2014 10 01 00 00
You entered date =  2014-10-01 00:00:00
```

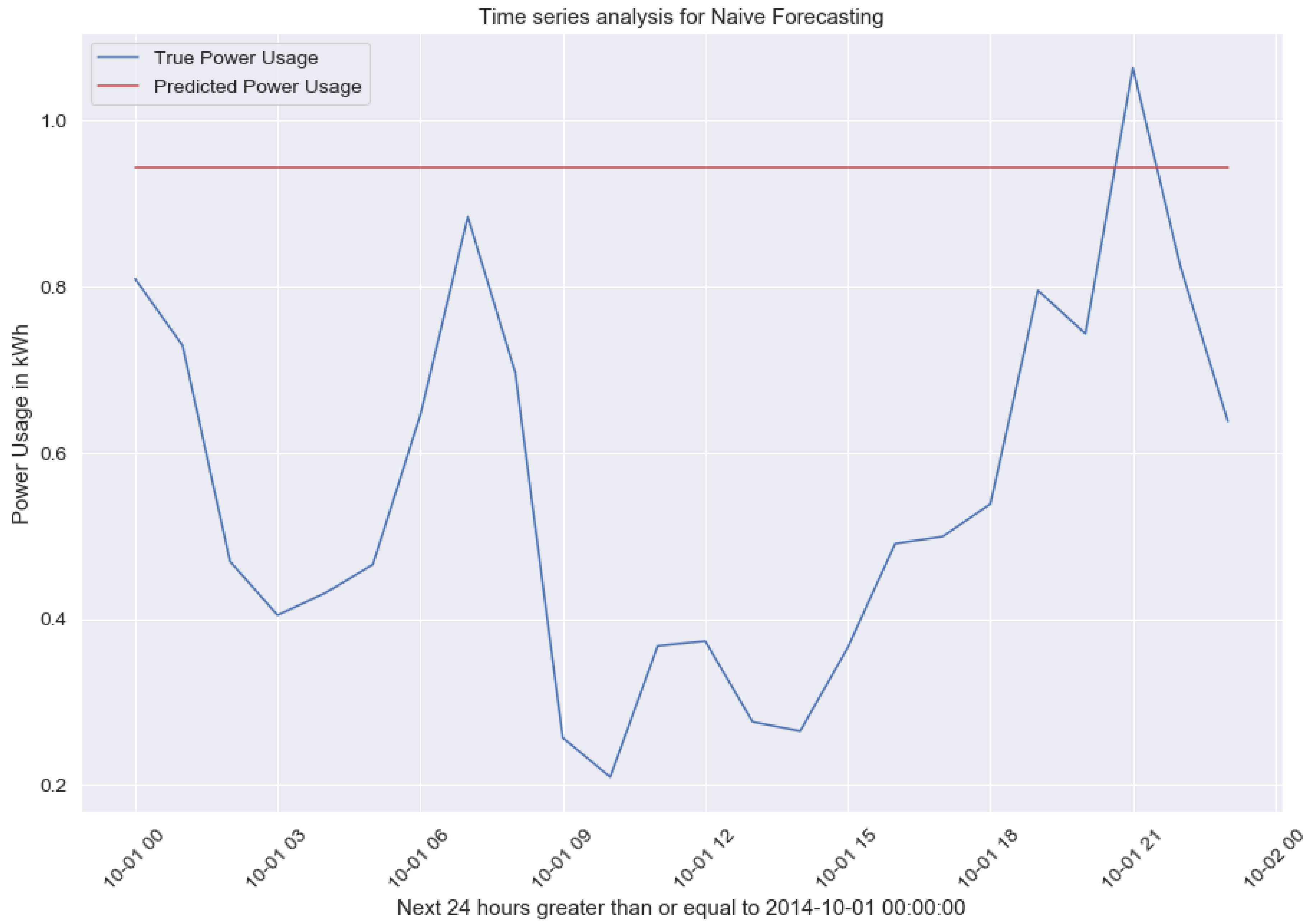
In [1050]:
```
# Adding the predicted usage column based on Naive Prediction
naive_dataframe_B['predicted_usage'] = naive_dataframe_B['use [kW]']

def add_naive_prediction(df):
    df['predicted_usage'] = df['use [kW]']
    df['predicted_usage'][0] = 0

    predicted_date = df['Date & Time'][0]
    for i in range(1, df['predicted_usage'].shape[0]):
        if pd.to_datetime(df['Date & Time'][i]) < input_date_B:
            df['predicted_usage'][i] = df['use [kW]'][i-1]
            predicted_date = df['use [kW]'][i]
        else:
            df['predicted_usage'][i] = predicted_date

add_naive_prediction(naive_dataframe_B)
naive_dataframe_B = naive_dataframe_B.filter(['Date & Time','use [kW]','predicted_usage'], axis=1)
naive_dataframe_B['Date & Time'] = pd.to_datetime(naive_dataframe_B['Date & Time'])
```

In [1051]:
```
# Training and Testing

# Training data is the portion before the input date
# Testing data is the one after the input date

datapoints = naive_dataframe_B.copy()

training_data = datapoints[datapoints['Date & Time'] < input_date_B]
predicted_testing_data = datapoints[datapoints['Date & Time'] >= input_date_B]

print("Dataframe of size = ",len(datapoints)," has training size = ",len(training_data)," ",int(len(training_data
```

```
Dataframe of size =  17524  has training size =  13102   74 %
```

In [1052]:
```
# Getting the next 24 hours data

def getNext24HourData(df):
    count = 1
    indices = []
    for i in range(0, len(df), 2):
        if count <=24:
           indices.append(i)
        count = count + 1
    next_24h_data = df.iloc[indices]
    return next_24h_data

next_24h_data_B = getNext24HourData(predicted_testing_data)

print('The Naive Forecast Data for input date = ',input_date_B)
x_val = next_24h_data_B['Date & Time']
Y_true = next_24h_data_B['use [kW]']
Y_pred = next_24h_data_B['predicted_usage']

def plotGraph(x_val,Y_true, Y_pred):
    x_val = np.array(x_val.values)
    Y_true = np.array(Y_true.values)
    Y_pred = np.array(Y_pred.values)
    plt.title('Time series analysis for Naive Forecasting')
    plt.xlabel('Next 24 hours greater than or equal to ' + str(input_date_B))
    plt.ylabel('Power Usage in kWh')
    plt.plot(x_val, Y_true,'b', label='True Power Usage')
    plt.plot(x_val, Y_pred,'r', label='Predicted Power Usage')
    plt.xticks(rotation=45)
    plt.legend(loc='upper left')
    plt.show()

plotGraph(x_val, Y_true, Y_pred)
```

```
The Naive Forecast Data for input date =  2014-10-01 00:00:00
```



In [1053]:
```
print('The Predicted Data dimensions = ',next_24h_data_B.shape)
next_24h_data_B
```

```
The Predicted Data dimensions =  (24, 3)
```

Out[1053]:

	Date & Time	use [kW]	predicted_usage
13102	2014-10-01 00:00:00	0.809155	0.943985
13104	2014-10-01 01:00:00	0.728769	0.943985
13106	2014-10-01 02:00:00	0.469251	0.943985
13108	2014-10-01 03:00:00	0.404612	0.943985
13110	2014-10-01 04:00:00	0.431194	0.943985
13112	2014-10-01 05:00:00	0.465531	0.943985
13114	2014-10-01 06:00:00	0.644938	0.943985
13116	2014-10-01 07:00:00	0.883777	0.943985
13118	2014-10-01 08:00:00	0.696323	0.943985
13120	2014-10-01 09:00:00	0.257019	0.943985
13122	2014-10-01 10:00:00	0.209837	0.943985
13124	2014-10-01 11:00:00	0.367699	0.943985
13126	2014-10-01 12:00:00	0.373417	0.943985
13128	2014-10-01 13:00:00	0.276246	0.943985
13130	2014-10-01 14:00:00	0.265015	0.943985
13132	2014-10-01 15:00:00	0.365273	0.943985
13134	2014-10-01 16:00:00	0.490715	0.943985
13136	2014-10-01 17:00:00	0.499203	0.943985
13138	2014-10-01 18:00:00	0.538178	0.943985
13140	2014-10-01 19:00:00	0.795120	0.943985
13142	2014-10-01 20:00:00	0.743074	0.943985
13144	2014-10-01 21:00:00	1.063002	0.943985
13146	2014-10-01 22:00:00	0.824203	0.943985
13148	2014-10-01 23:00:00	0.637488	0.943985

In [1054]:
```
# Calculating the Mean Absolute Error
print("The Mean Absolute Error for Naive Forecasting is : %.4f "%mean_absolute_error(Y_true, Y_pred))
```

```
The Mean Absolute Error for Naive Forecasting is : 0.4023
```

In [ ]:

In [ ]:

In [ ]:

For Home Type C

In [1226]:
```
df = df_C.copy()
newIndices = []
d12 = pd.to_datetime('2015-12-16 00:00:00')
for i in range(0,len(df)):
    if pd.to_datetime(df.iloc[i]['Date & Time'])>=d12 and (pd.to_datetime(df.iloc[i]['Date & Time']).minute ==
        newIndices.append(i)
    elif pd.to_datetime(df.iloc[i]['Date & Time'])<d12:
        newIndices.append(i)
```

In [1230]:
```
df_C = df_C.iloc[newIndices]
```

In [ ]:

In [1251]:
```
# Getting input date from the user
naive_dataframe_C = df_C.copy()

print('Enter Date between ', min(naive_dataframe_C['Date & Time']),' and ', max(naive_dataframe_C['Date & Time'
Year, Month, Day, Hour, Minute = input("Enter Timestamp YYYY,MM,DD,HH,MM: ").split()

input_date_C = datetime(int(Year), int(Month), int(Day), int(Hour), int(Minute) , 0)
print("You entered date = ",str(input_date_C))
```

```
Enter Date between  2015-01-01 00:00:00  and  2015-12-31 23:30:00
Enter Timestamp YYYY,MM,DD,HH,MM: 2015 10 20 00 00
You entered date =  2015-10-20 00:00:00
```

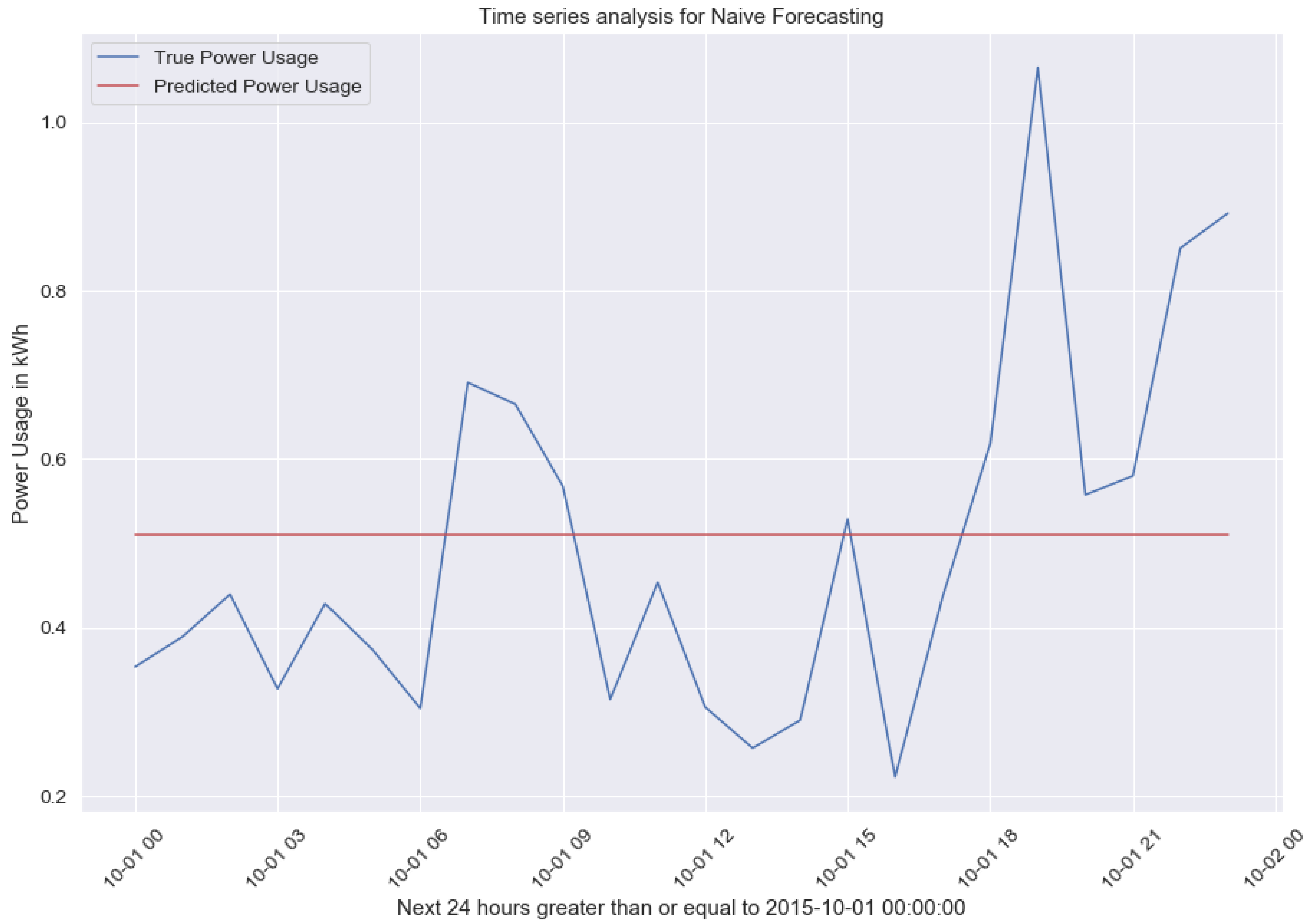
In [1252]:
```
# Adding the predicted usage column based on Naive Prediction
naive_dataframe_C['predicted_usage'] = naive_dataframe_C['use [kW]']

def add_naive_prediction(df):
    df['predicted_usage'] = df['use [kW]']
    df['predicted_usage'][0] = 0

    predicted_date = df['Date & Time'][0]
    for i in range(1, len(df['predicted_usage'])):
        if pd.to_datetime(df['Date & Time'].iloc[i]) < input_date_C:
            df['predicted_usage'].iloc[i] = df['use [kW]'].iloc[i-1]
            predicted_date = df['use [kW]'].iloc[i]
        else:
            df['predicted_usage'].iloc[i] = predicted_date

add_naive_prediction(naive_dataframe_C)
naive_dataframe_C = naive_dataframe_C.filter(['Date & Time','use [kW]','predicted_usage'], axis=1)
naive_dataframe_C['Date & Time'] = pd.to_datetime(naive_dataframe_C['Date & Time'])
```

In [1253]:
```
Training and Testing

Training data is the portion before the input date
Testing data is the one after the input date

tapoints = naive_dataframe_C.copy()

aining_data = datapoints[datapoints['Date & Time'] < input_date_C]
edicted_testing_data = datapoints[datapoints['Date & Time'] >= input_date_C]

int("Dataframe of size = ",len(datapoints)," has training size = ",len(training_data)," ",int(len(training_data)
```

```
Dataframe of size =  17524  has training size =  14014   79 %
```

In [1247]:
```
# Getting the next 24 hours data
# next_24h_data = predicted_testing_data.iloc[:48:2, :]

def getNext24HourData(df):
    count = 1
    indices = []
    for i in range(0, len(df), 2):
        if count <=24:
           indices.append(i)
        count = count + 1
    next_24h_data = df.iloc[indices]
    return next_24h_data

next_24h_data_C = getNext24HourData(predicted_testing_data)

print('The Naive Forecast Data for input date = ',input_date_C)
x_val = next_24h_data_C['Date & Time']
Y_true = next_24h_data_C['use [kW]']
Y_pred = next_24h_data_C['predicted_usage']

def plotGraph(x_val,Y_true, Y_pred):
    x_val = np.array(x_val.values)
    Y_true = np.array(Y_true.values)
    Y_pred = np.array(Y_pred.values)
    plt.title('Time series analysis for Naive Forecasting')
    plt.xlabel('Next 24 hours greater than or equal to ' + str(input_date_C))
    plt.ylabel('Power Usage in kWh')
    plt.plot(x_val, Y_true,'b', label='True Power Usage')
    plt.plot(x_val, Y_pred,'r', label='Predicted Power Usage')
    plt.xticks(rotation=45)
    plt.legend(loc='upper left')
    plt.show()

plotGraph(x_val, Y_true, Y_pred)
```

```
The Naive Forecast Data for input date =  2015-10-01 00:00:00
```



In [1248]:
```
print('The Predicted Data dimensions = ',next_24h_data_C.shape)
next_24h_data_C
```

```
The Predicted Data dimensions =  (24, 3)
```

Out[1248]:

	Date & Time	use [kW]	predicted_usage
13102	2015-10-01 00:00:00	0.353275	0.510292
13104	2015-10-01 01:00:00	0.389069	0.510292
13106	2015-10-01 02:00:00	0.439365	0.510292
13108	2015-10-01 03:00:00	0.327237	0.510292
13110	2015-10-01 04:00:00	0.428229	0.510292
13112	2015-10-01 05:00:00	0.373331	0.510292
13114	2015-10-01 06:00:00	0.303963	0.510292
13116	2015-10-01 07:00:00	0.690503	0.510292
13118	2015-10-01 08:00:00	0.664884	0.510292
13120	2015-10-01 09:00:00	0.568016	0.510292
13122	2015-10-01 10:00:00	0.314633	0.510292
13124	2015-10-01 11:00:00	0.453560	0.510292
13126	2015-10-01 12:00:00	0.305413	0.510292
13128	2015-10-01 13:00:00	0.256811	0.510292
13130	2015-10-01 14:00:00	0.289835	0.510292
13132	2015-10-01 15:00:00	0.528908	0.510292
13134	2015-10-01 16:00:00	0.222486	0.510292
13136	2015-10-01 17:00:00	0.436332	0.510292
13138	2015-10-01 18:00:00	0.617346	0.510292
13140	2015-10-01 19:00:00	1.064659	0.510292
13142	2015-10-01 20:00:00	0.557611	0.510292
13144	2015-10-01 21:00:00	0.580036	0.510292
13146	2015-10-01 22:00:00	0.850192	0.510292
13148	2015-10-01 23:00:00	0.891540	0.510292

In [1254]:
```
# Calculating the Mean Absolute Error
print("The Mean Absolute Error for Naive Forecasting is : %.4f "%mean_absolute_error(Y_true, Y_pred))
```

```
The Mean Absolute Error for Naive Forecasting is : 0.1734
```

In [ ]:

In [ ]:

For Home Type F

In [1255]:
```
# Getting input date from the user
naive_dataframe_F = df_F.copy()

print('Enter Date between ', min(naive_dataframe_F['Date & Time']),' and ', max(naive_dataframe_F['Date & Time'
Year, Month, Day, Hour, Minute = input("Enter Timestamp YYYY,MM,DD,HH,MM: ").split()

input_date_F = datetime(int(Year), int(Month), int(Day), int(Hour), int(Minute) , 0)
print("You entered date = ",str(input_date_F))
```

```
Enter Date between  2016-01-01 00:00:00  and  2016-12-15 22:30:00
Enter Timestamp YYYY,MM,DD,HH,MM: 2016 10 20 00 00
You entered date =  2016-10-20 00:00:00
```

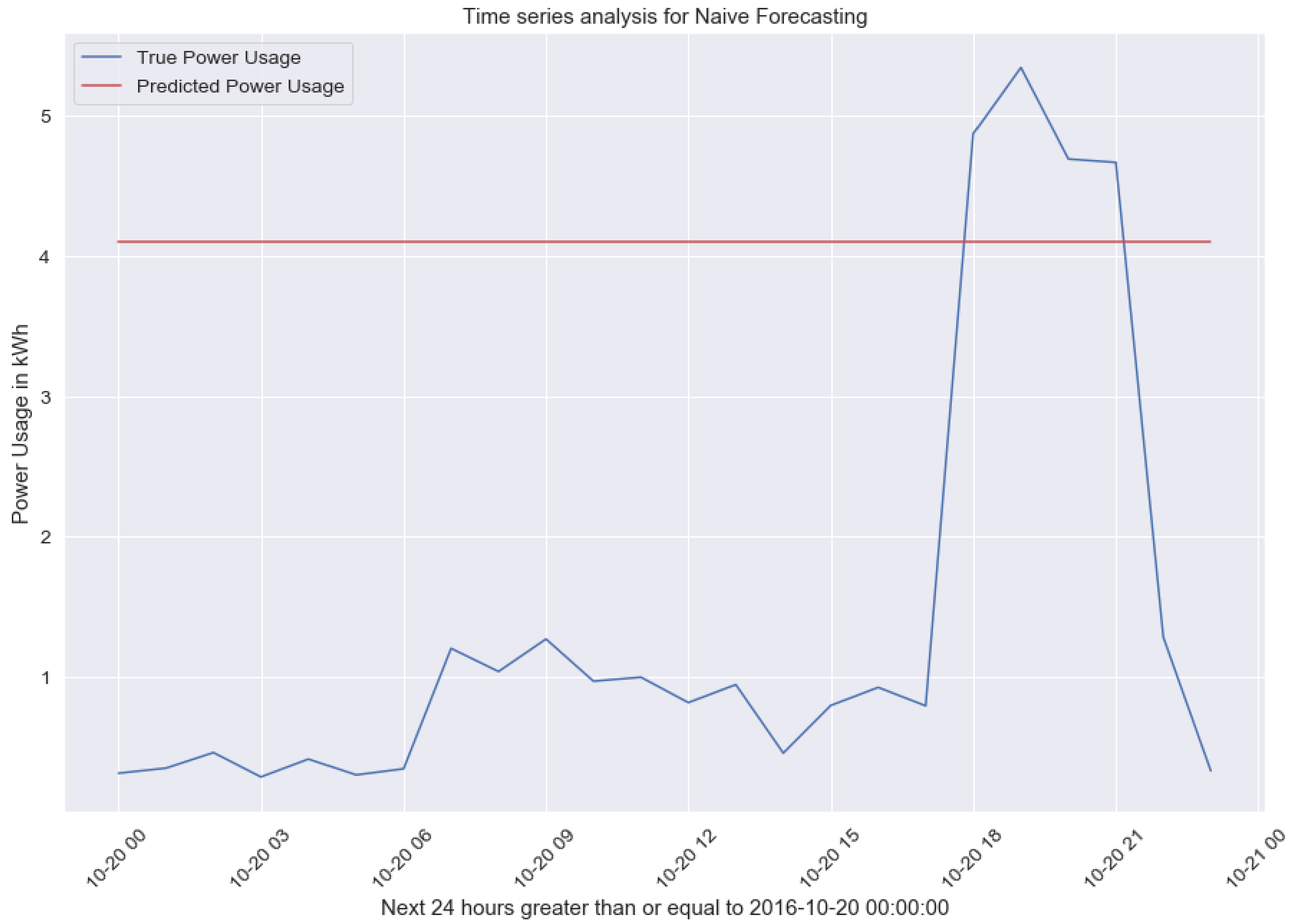
In [1256]:
```
# Adding the predicted usage column based on Naive Prediction
naive_dataframe_F['predicted_usage'] = naive_dataframe_F['use [kW]']

def add_naive_prediction(df):
    df['predicted_usage'] = df['use [kW]']
    df['predicted_usage'][0] = 0

    predicted_date = df['Date & Time'][0]
    for i in range(1, df['predicted_usage'].shape[0]):
        if pd.to_datetime(df['Date & Time'][i]) < input_date_F:
            df['predicted_usage'][i] = df['use [kW]'][i-1]
            predicted_date = df['use [kW]'][i]
        else:
            df['predicted_usage'][i] = predicted_date

add_naive_prediction(naive_dataframe_F)
naive_dataframe_F = naive_dataframe_F.filter(['Date & Time','use [kW]','predicted_usage'], axis=1)
naive_dataframe_F['Date & Time'] = pd.to_datetime(naive_dataframe_F['Date & Time'])
```

In [1257]:
```
# Training and Testing

# Training data is the portion before the input date
# Testing data is the one after the input date

datapoints = naive_dataframe_F.copy()

training_data = datapoints[datapoints['Date & Time'] < input_date_F]
predicted_testing_data = datapoints[datapoints['Date & Time'] >= input_date_F]

print("Dataframe of size = ",len(datapoints)," has training size = ",len(training_data)," ",int(len(training_da
```

```
Dataframe of size =  16802  has training size =  14062   83 %
```

In [1261]:
```
# Getting the next 24 hours data
# next_24h_data = predicted_testing_data.iloc[:48:2, :]

def getNext24HourData(df):
    count = 1
    indices = []
    for i in range(0, len(df), 2):
        if count <=24:
           indices.append(i)
        count = count + 1
    next_24h_data = df.iloc[indices]
    return next_24h_data

next_24h_data_F = getNext24HourData(predicted_testing_data)

print('The Naive Forecast Data for input date = ',input_date_F)
x_val = next_24h_data_F['Date & Time']
Y_true = next_24h_data_F['use [kW]']
Y_pred = next_24h_data_F['predicted_usage']

def plotGraph(x_val,Y_true, Y_pred):
    x_val = np.array(x_val.values)
    Y_true = np.array(Y_true.values)
    Y_pred = np.array(Y_pred.values)
    plt.title('Time series analysis for Naive Forecasting')
    plt.xlabel('Next 24 hours greater than or equal to ' + str(input_date_F))
    plt.ylabel('Power Usage in kWh')
    plt.plot(x_val, Y_true,'b', label='True Power Usage')
    plt.plot(x_val, Y_pred,'r', label='Predicted Power Usage')
    plt.xticks(rotation=45)
    plt.legend(loc='upper left')
    plt.show()

plotGraph(x_val, Y_true, Y_pred)
```

```
The Naive Forecast Data for input date =  2016-10-20 00:00:00
```



In [1259]:
```
print('The Predicted Data dimensions = ',next_24h_data_F.shape)
next_24h_data_F
```

```
The Predicted Data dimensions =  (24, 3)
```

Out[1259]:

	Date & Time	use [kW]	predicted_usage
14062	2016-10-20 00:00:00	0.314400	4.102083
14064	2016-10-20 01:00:00	0.350450	4.102083
14066	2016-10-20 02:00:00	0.461433	4.102083
14068	2016-10-20 03:00:00	0.287800	4.102083
14070	2016-10-20 04:00:00	0.414950	4.102083
14072	2016-10-20 05:00:00	0.302533	4.102083
14074	2016-10-20 06:00:00	0.345783	4.102083
14076	2016-10-20 07:00:00	1.204567	4.102083
14078	2016-10-20 08:00:00	1.039650	4.102083
14080	2016-10-20 09:00:00	1.271350	4.102083
14082	2016-10-20 10:00:00	0.970683	4.102083
14084	2016-10-20 11:00:00	0.998817	4.102083
14086	2016-10-20 12:00:00	0.818033	4.102083
14088	2016-10-20 13:00:00	0.946167	4.102083
14090	2016-10-20 14:00:00	0.457883	4.102083
14092	2016-10-20 15:00:00	0.797533	4.102083
14094	2016-10-20 16:00:00	0.926400	4.102083
14096	2016-10-20 17:00:00	0.794933	4.102083
14098	2016-10-20 18:00:00	4.872667	4.102083
14100	2016-10-20 19:00:00	5.344967	4.102083
14102	2016-10-20 20:00:00	4.692300	4.102083
14104	2016-10-20 21:00:00	4.668400	4.102083
14106	2016-10-20 22:00:00	1.284683	4.102083
14108	2016-10-20 23:00:00	0.331617	4.102083

In [1260]:
```
# Calculating the Mean Absolute Error
print("The Mean Absolute Error for Naive Forecasting is : %.4f "%mean_absolute_error(Y_true, Y_pred))
```

```
The Mean Absolute Error for Naive Forecasting is : 2.9538
```

In [ ]:

In [ ]:

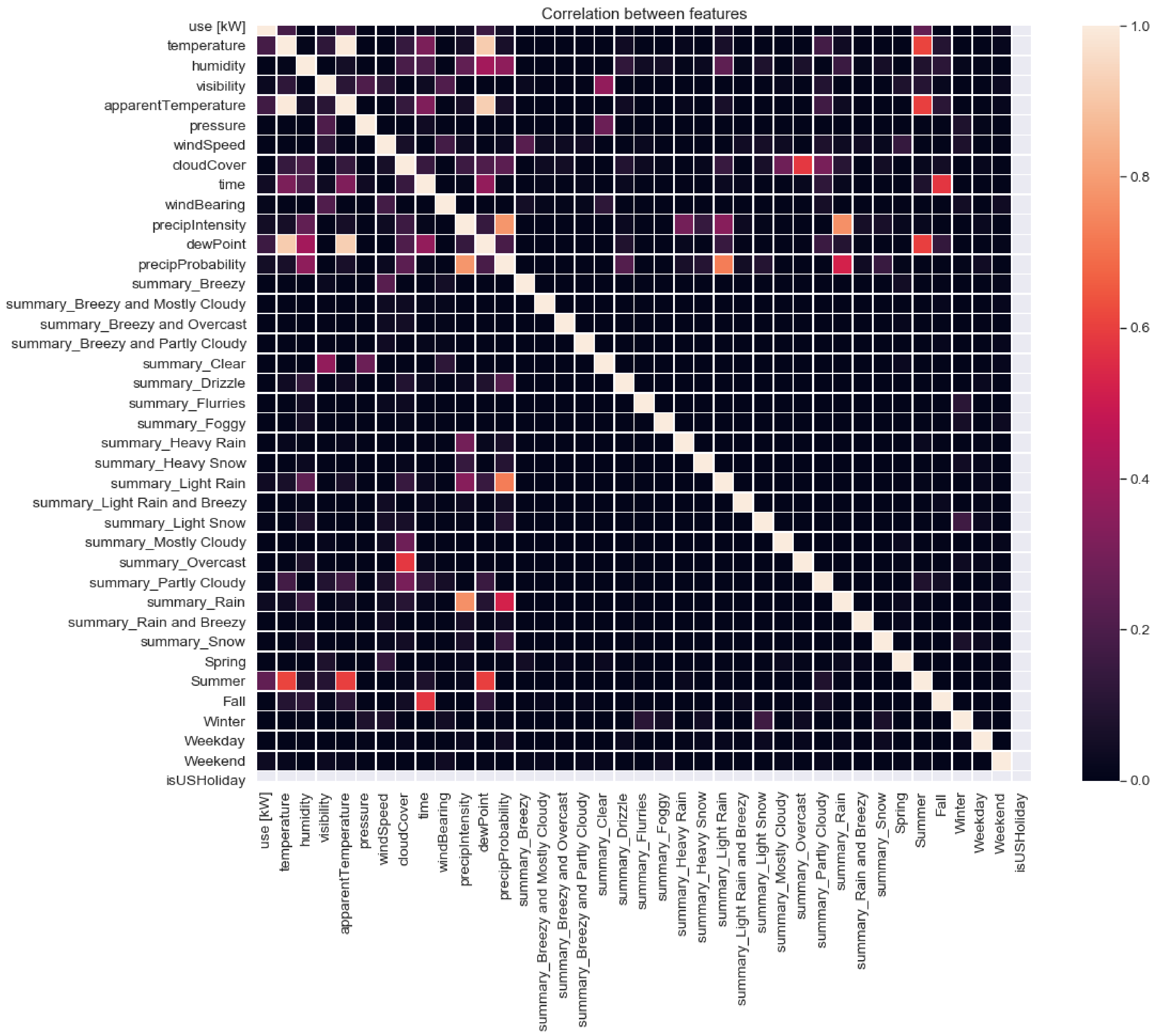
In [ ]:

Finding Correlation

For Home Type B

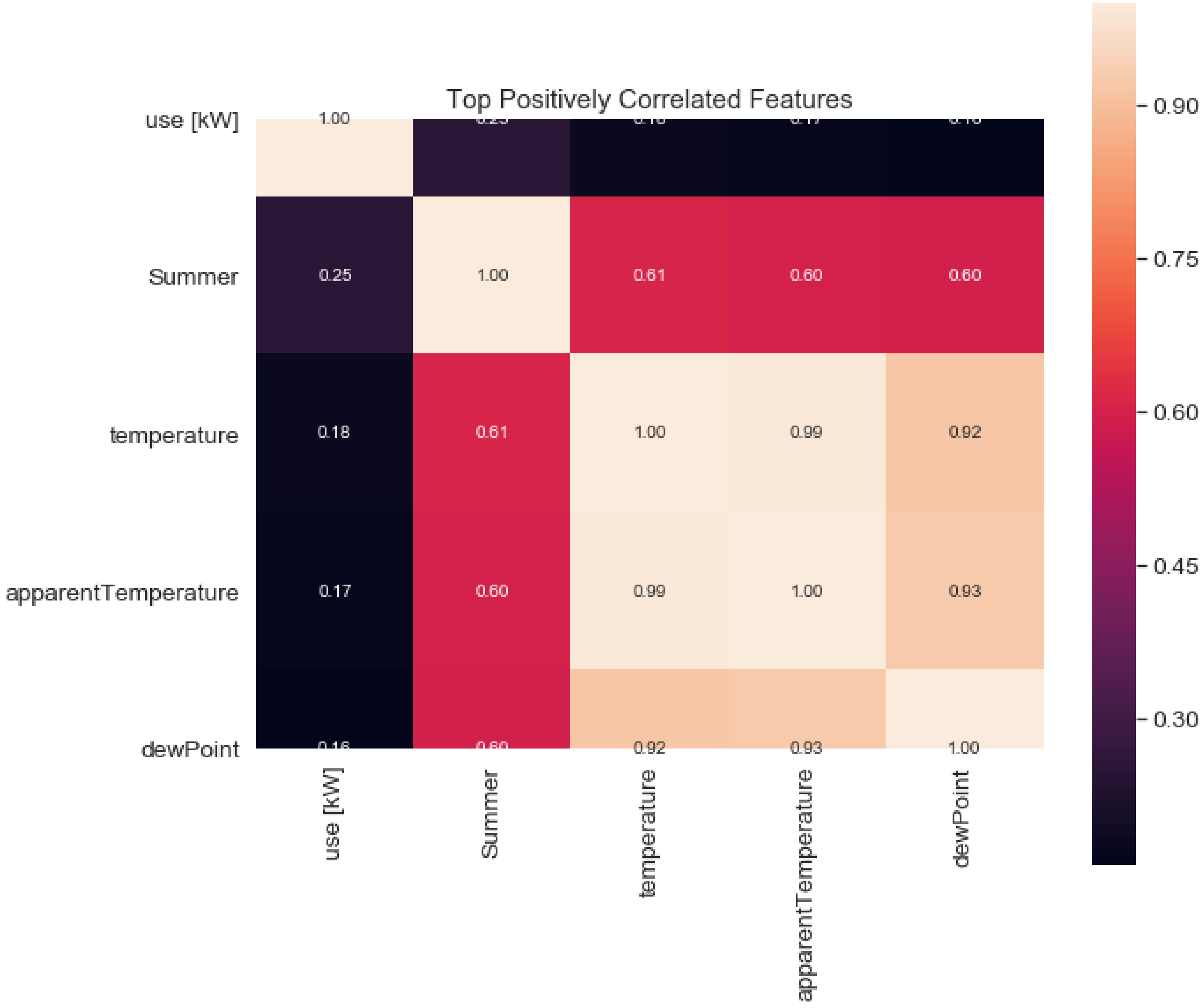
In [1123]:
```python
# Finding correlation between features
dataframe_B = df_B.copy()
correlation = dataframe_B.corr(method='pearson')
plt.subplots(figsize=(17, 13))
plt.title("Correlation between features")
sns.heatmap(correlation, linewidths=.5, vmin=0, vmax=1, square=True)
```

Out[1123]: `<matplotlib.axes._subplots.AxesSubplot at 0x1c414b5f50>`



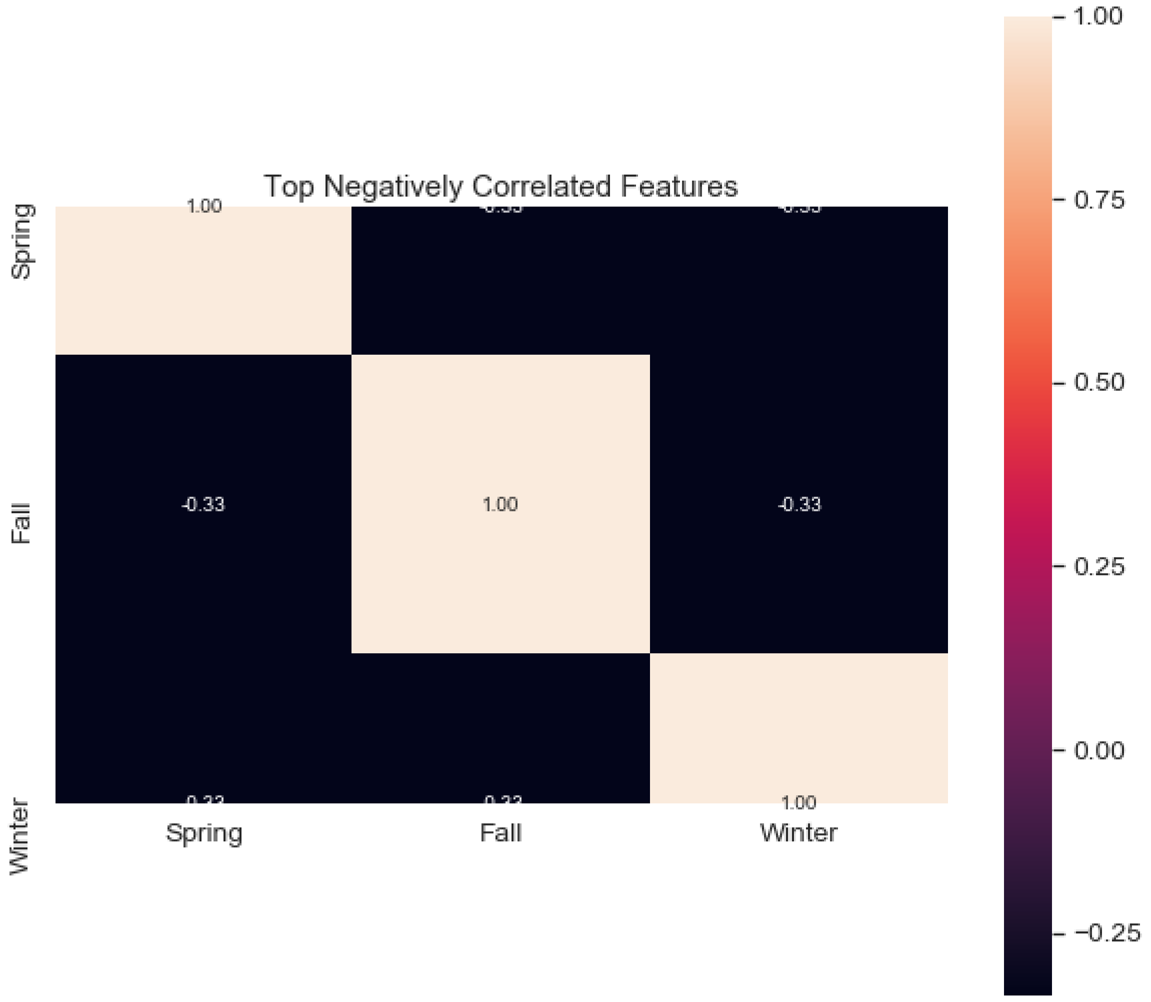
In [1124]:
```
'''Using the nlargest function to get the highly POSITIVE correlated values'''
numberOfVariablesToBeSelected = 5
columnsPositive_B = correlation.nlargest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

print(columnsPositive_B)
cm = np.corrcoef(dataframe[columnsPositive_B].values.T)
sns.set(font_scale=1.25)
plt.subplots(figsize=(10, 9))
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columnsI
plt.title("Top Positively Correlated Features")
plt.show()
```

```
Index(['use [kW]', 'Summer', 'temperature', 'apparentTemperature', 'dewPoint'], dtype='object')
```



In [1125]:
```
Using the nsmallest function to get the highly NEGATIVE correlated values'''
berOfVariablesToBeSelected = 3
umnsNegative_B = correlation.nsmallest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

nt(columnsNegative_B)
= np.corrcoef(dataframe[columnsNegative_B].values.T)
.set(font_scale=1.25)
.subplots(figsize=(10, 9))
.title("Top Negatively Correlated Features")
= sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columnsNega
.show()
```

```
Index(['Spring', 'Fall', 'Winter'], dtype='object')
```



In [ ]:

In [ ]:

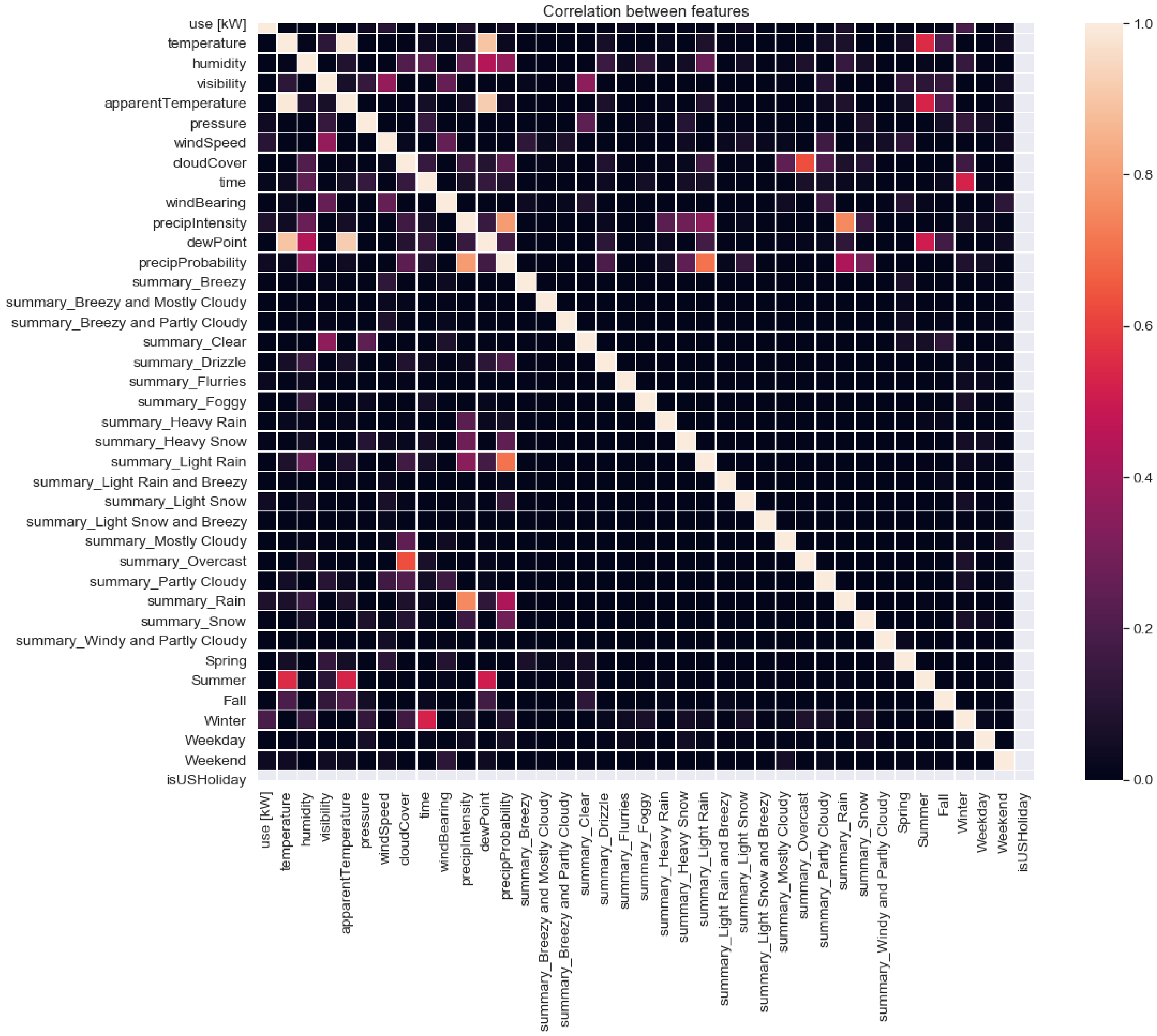
In [ ]:

In [ ]:

For Home Type C

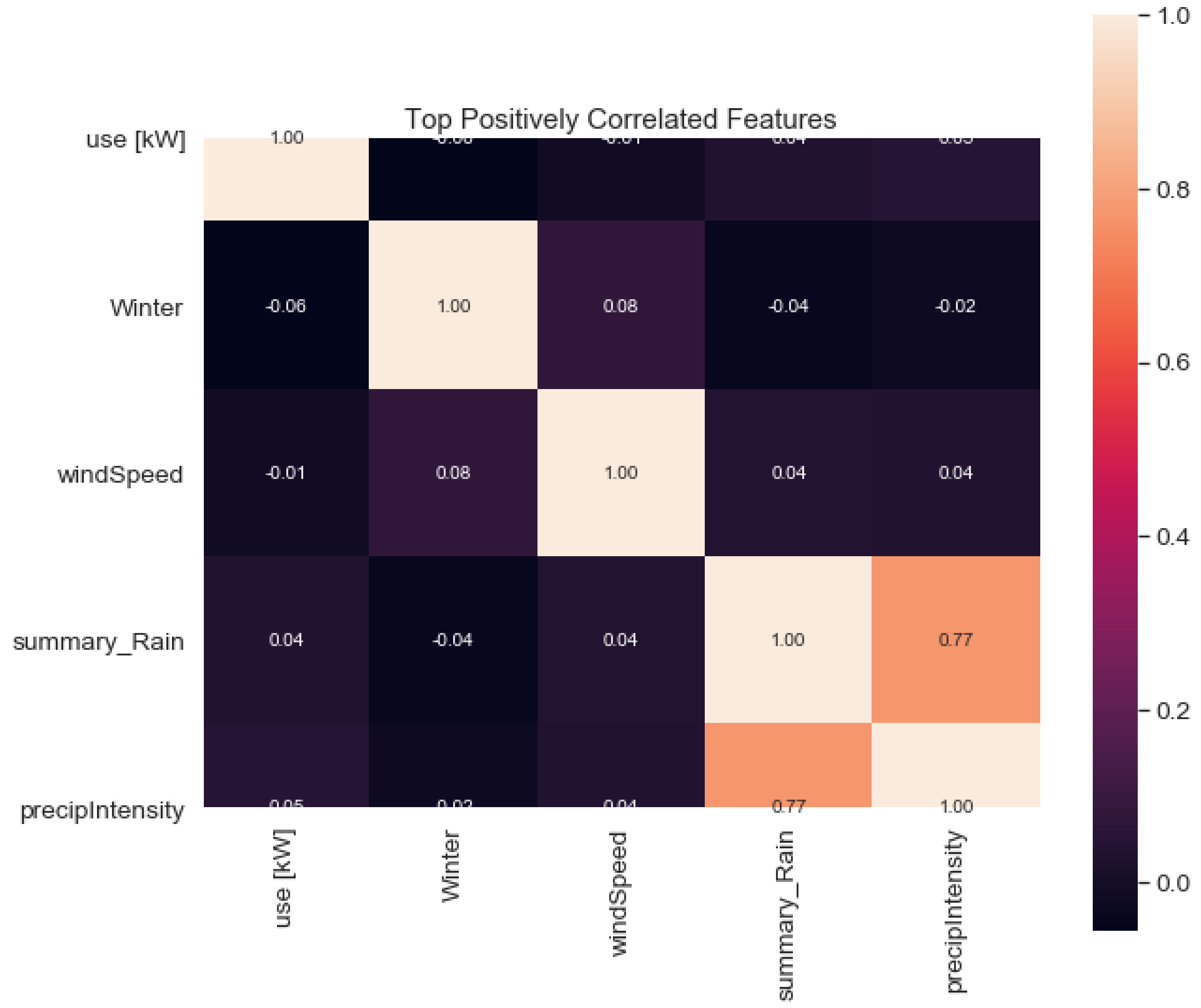
In [1126]:
```
# Finding correlation between features
dataframe_C = df_C.copy()
correlation = dataframe_C.corr(method='pearson')
plt.subplots(figsize=(17, 13))
plt.title("Correlation between features")
sns.heatmap(correlation, linewidths=.5, vmin=0, vmax=1, square=True)
```

Out[1126]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5c7602d0>



In [1127]:
```
'''Using the nlargest function to get the highly POSITIVE correlated values'''
numberOfVariablesToBeSelected = 5
columnsPositive_C = correlation.nlargest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

print(columnsPositive_C)
cm = np.corrcoef(dataframe[columnsPositive_C].values.T)
sns.set(font_scale=1.25)
plt.subplots(figsize=(10, 9))
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columnsI
plt.title("Top Positively Correlated Features")
plt.show()
```

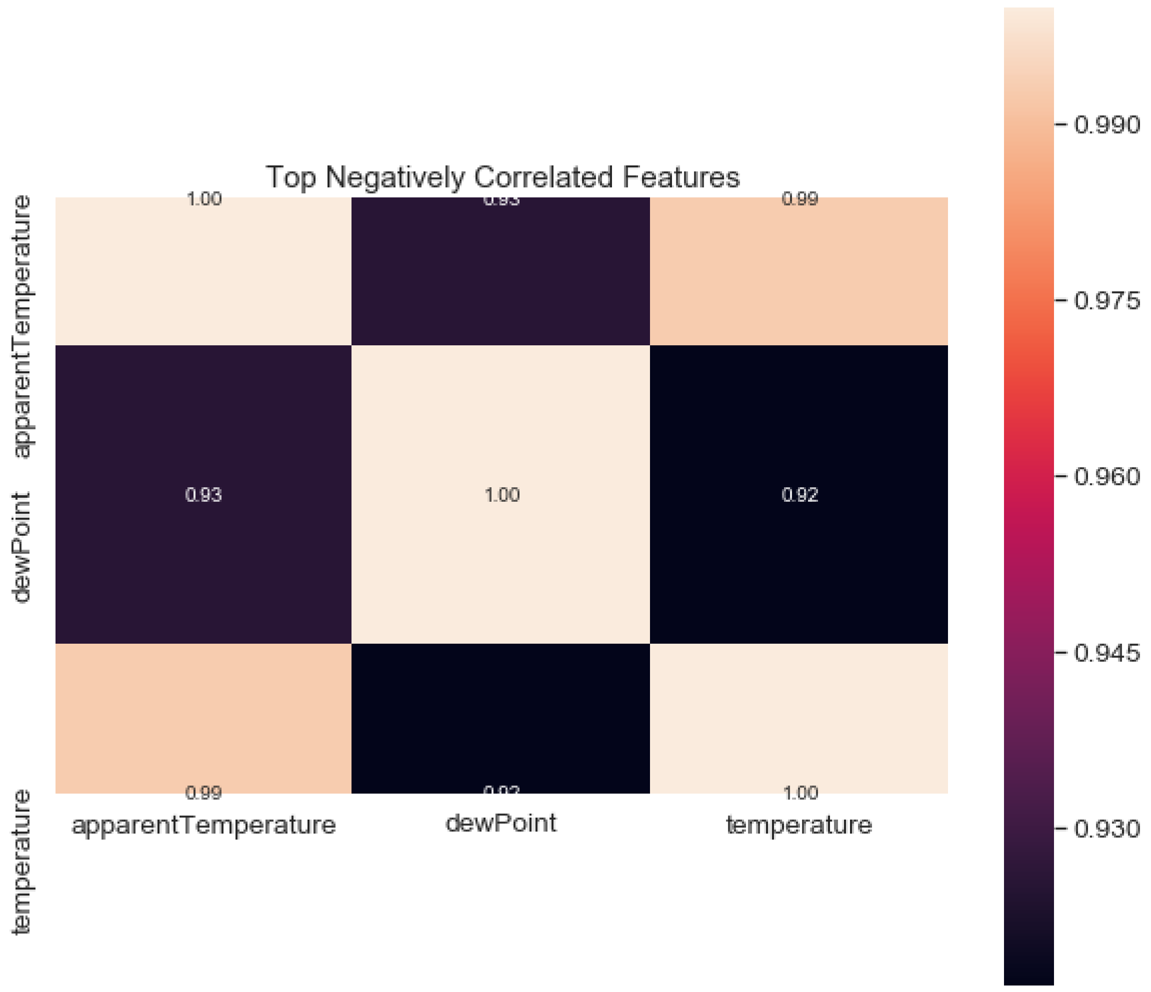
```
Index(['use [kW]', 'Winter', 'windSpeed', 'summary_Rain', 'precipIntensity'], dtype='object')
```



In [1128]:

```
'''Using the nsmallest function to get the highly NEGATIVE correlated values'''
numberOfVariablesToBeSelected = 3
columnsNegative_C = correlation.nsmallest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

print(columnsNegative_C)
cm = np.corrcoef(dataframe[columnsNegative_C].values.T)
sns.set(font_scale=1.25)
plt.subplots(figsize=(10, 9))
plt.title("Top Negatively Correlated Features")
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columns
plt.show()
```

```
Index(['apparentTemperature', 'dewPoint', 'temperature'], dtype='object')
```



In [ ]:

In [ ]:

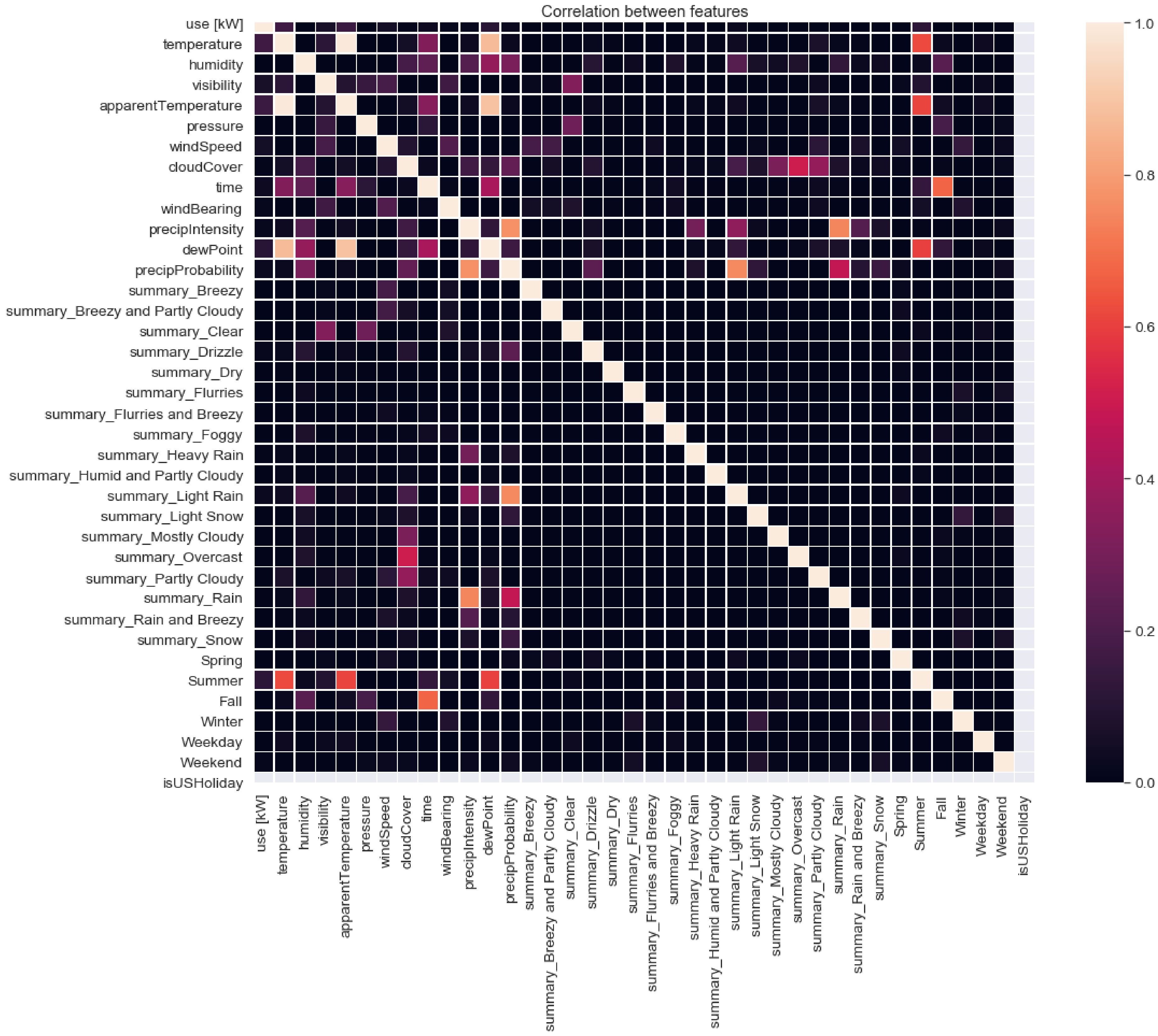
In [ ]:

In [ ]:

For Home Type F

In [1129]:
```
# Finding correlation between features
dataframe_F = df_F.copy()
correlation = dataframe_F.corr(method='pearson')
plt.subplots(figsize=(17, 13))
plt.title("Correlation between features")
sns.heatmap(correlation, linewidths=.5, vmin=0, vmax=1, square=True)
```

Out[1129]: <matplotlib.axes._subplots.AxesSubplot at 0x1c41e3a410>

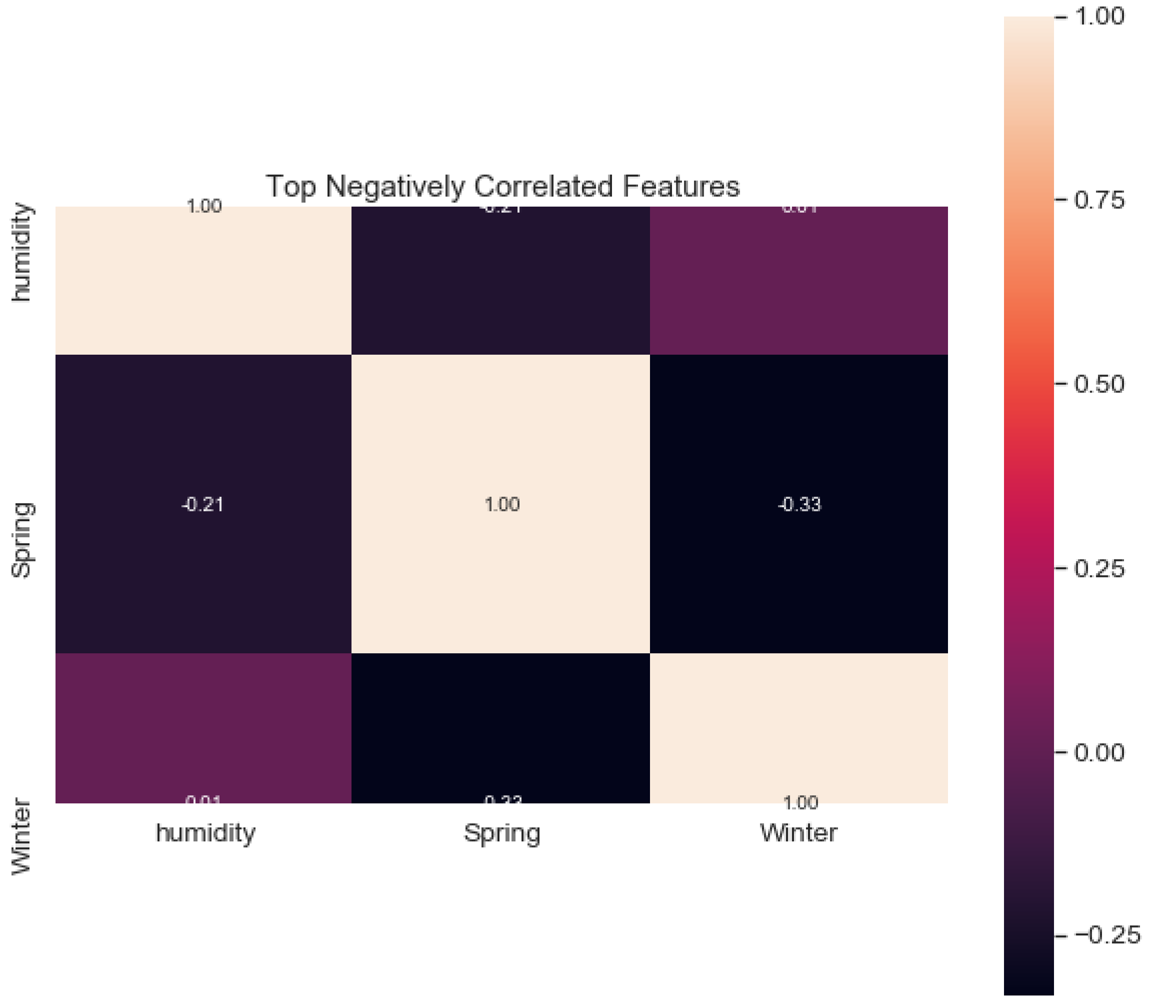


In [1130]:
```
'''Using the nlargest function to get the highly POSITIVE correlated values'''
numberOfVariablesToBeSelected = 5
columnsPositive_F = correlation.nlargest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

print(columnsPositive_F)
cm = np.corrcoef(dataframe[columnsPositive_F].values.T)
sns.set(font_scale=1.25)
plt.subplots(figsize=(10, 9))
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columns
plt.title("Top Positively Correlated Features")
plt.show()
```

```
Index(['use [kW]', 'temperature', 'apparentTemperature', 'Summer', 'dewPoint'], dtype='object')
```

In [1131]:
```
'''Using the nsmallest function to get the highly NEGATIVE correlated values'''
numberOfVariablesToBeSelected = 3
columnsNegative_F = correlation.nsmallest(numberOfVariablesToBeSelected, 'use [kW]')['use [kW]'].index

print(columnsNegative_F)
cm = np.corrcoef(dataframe[columnsNegative_F].values.T)
sns.set(font_scale=1.25)
plt.subplots(figsize=(10, 9))
plt.title("Top Negatively Correlated Features")
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10}, yticklabels=columns
plt.show()
```

```
Index(['humidity', 'Spring', 'Winter'], dtype='object')
```



In [ ]:

In [ ]:

Performing Linear Regression on combined data

In [1285]:
```
lr_df_B = dataframe.copy()
columns = columnsPositive_B.append(columnsNegative_B)

def separate_to_train_test(col):
    x_col = []
    y_col = []
    global x_train_lr
    global y_train_lr
    global x_test_lr
    global y_test_lr
    global y_test_duration


    for name in col:
        if name not in ['use [kW]']:
            x_col.append(name)

    y_col.append('use [kW]')
    x_col.append('Date & Time')
    x_train_lr = lr_df_B.filter(x_col)
    y_train_lr = lr_df_B.filter(y_col)

    x_test_lr  = x_train_lr[pd.to_datetime(x_train_lr['Date & Time']) >= input_date_B]
    x_train_lr = x_train_lr[pd.to_datetime(x_train_lr['Date & Time']) < input_date_B]

    y_train_lr = y_train_lr[pd.to_datetime(lr_df['Date & Time']) < input_date_B]
    y_test_lr = lr_df_B[pd.to_datetime(lr_df_B['Date & Time']) >= input_date_B]['use [kW]']


separate_to_train_test(columns)
x_train_lr = x_train_lr.drop('Date & Time', axis=1)

prediction_date_range = x_test_lr['Date & Time']
x_test_lr = x_test_lr.drop('Date & Time', axis=1)
```

In [1286]:
```
# Scaling dataframe using Standard Scaler

sc = StandardScaler()
x_train_lr = sc.fit_transform(x_train_lr)
x_test_lr = sc.transform(x_test_lr)
```

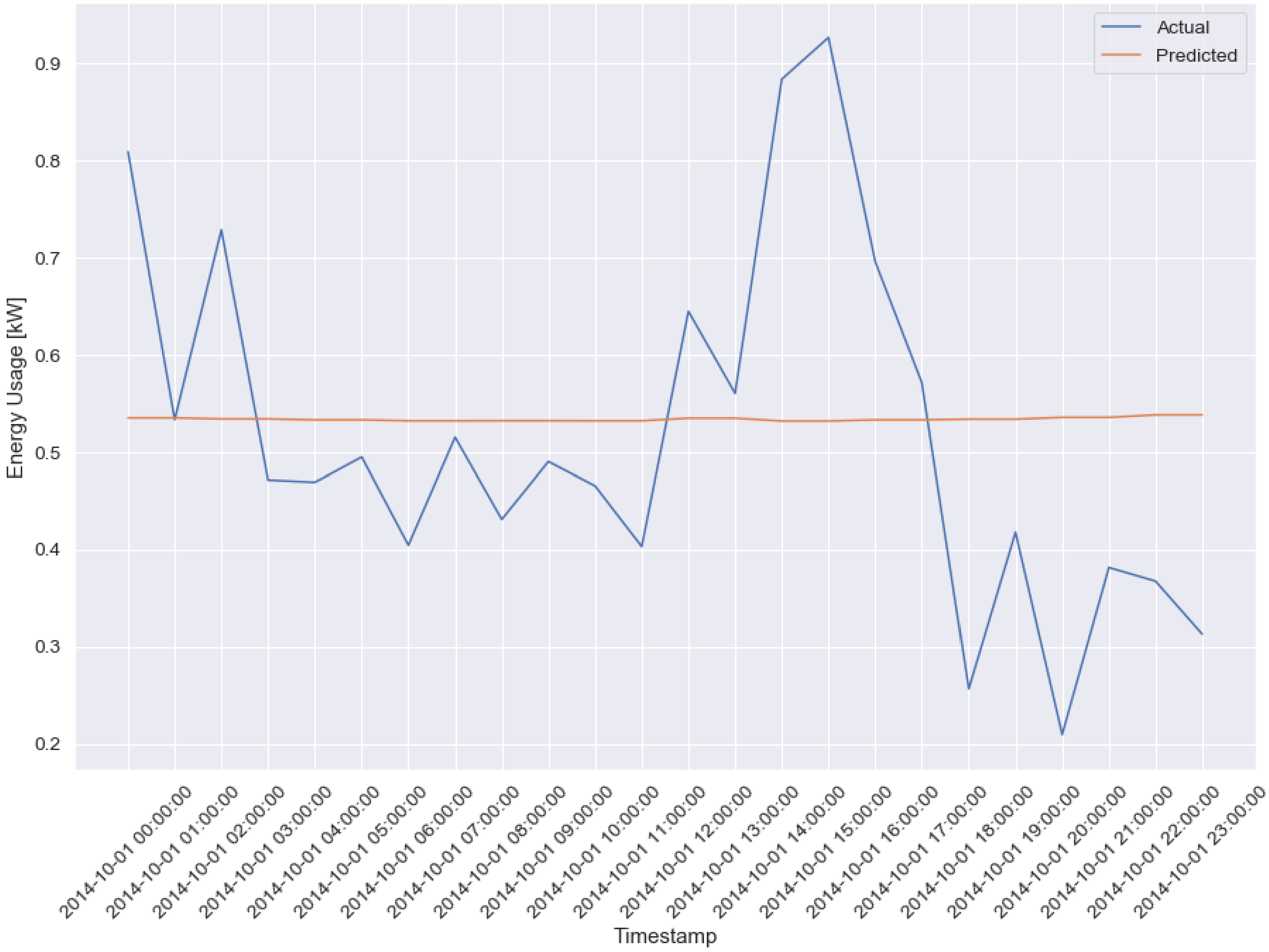
In [1287]:
```
# Model training and prediction

regression_lr = linear_model.LinearRegression()
regression_lr.fit(x_train_lr, y_train_lr)
y_pred_lr = regr.predict(x_test_lr)
```

In [1288]:
```

# Get 24 hours prediction
x_duration = pd.DataFrame(columns=['DateTime'])

for index in range(prediction_date_range.index[0],prediction_date_range.index[len(prediction_date_range)-1],2):
    x_duration = np.append(x_duration,prediction_date_range[index])

x_duration = x_duration[0:24]
```

In [1289]:
```
print("Mean absolute error by Linear Regression: %.9f"
      % mean_absolute_error(y_test_lr,y_pred_lr))
```

```
Mean absolute error by Linear Regression: 0.241135423
```

In [1290]:
```
y_test_lr = y_test_lr[0:24]
y_pred_lr = y_pred_lr[0:24]

plt.plot(x_duration, y_test_lr, label="Actual")
plt.plot(x_duration, y_pred_lr, label="Predicted")
plt.xlabel('Timestamp')
plt.ylabel('Energy Usage [kW]')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [ ]:
```

```

Implementing ARIMA Model on combined data

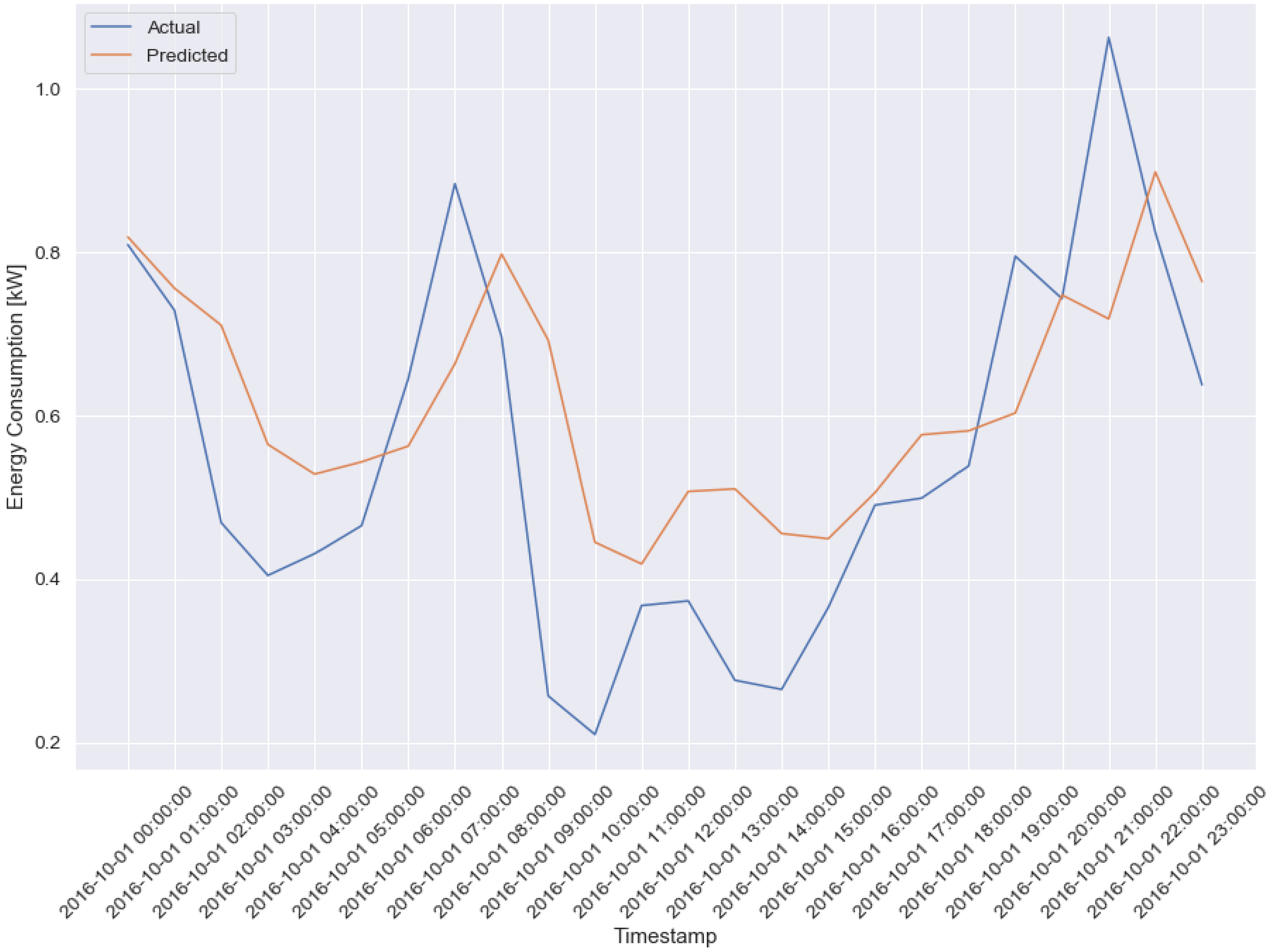
In [1322]:
```
adf_B = dataframe.copy()
adf_B = adf_B.filter(['Date & Time','use [kW]'], axis=1)


def separate_to_train_test_arima(adf):

    global train_arima
    global test_arima

    adf_temp =  pd.DataFrame(columns=adf.columns)

    # Take only hours
    for index in range(0,len(adf),2):
       adf_temp = adf_temp.append(adf.iloc[index])

    train_arima  = adf_temp[pd.to_datetime(adf_temp['Date & Time']) < input_date_B]
    test_arima = adf_temp[pd.to_datetime(adf_temp['Date & Time']) >= input_date_B]
    train_arima = train_arima.set_index('Date & Time')
    test_arima = test_arima.set_index('Date & Time')


separate_to_train_test_arima(adf_B)
```

In [1324]:
```
history = train_arima.to_numpy()
predictions_arima = list()
for t in range(len(test_arima)):
    model = ARIMA(history, order=(1,0,0))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    yhat = output[0]
    predictions_arima.append(yhat)
    obs = test_arima.iloc[t]
    history = np.append(history,obs)
```

In [1325]:
```

error = mean_absolute_error(test_arima, predictions_arima)

print('Mean absolute error by ARIMA modeling: %.9f'%error)
```

```
Mean absolute error by ARIMA modeling: 0.203353026
```

In [1160]:
```
test_arima = test_arima[0:24]
predictions= predictions_arima[0:24]

plt.plot(x_duration, test_arima, label="Actual")
plt.plot(x_duration, predictions_arima, label="Predicted")
plt.xlabel('Timestamp')
plt.ylabel('Energy Consumption [kW]')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [ ]:

In [ ]:

Implementing Random Forest Model on combined data

In [1169]:
```
randF_df_B = dataframe.copy()
columns = columnsPositive_B.append(columnsNegative_B)

def separate_to_train_test_randF(col):
    x_col = []
    y_col = []
    global x_train_randF
    global y_train_randF
    global x_test_randF
    global y_test_randF

    for name in col:
        if name not in ['use [kW]']:
            x_col.append(name)

    y_col.append('use [kW]')
    x_col.append('Date & Time')
    x_train_randF = randF_df_B.filter(x_col)
    y_train_randF = randF_df_B.filter(y_col)

    x_test_randF  = x_train_randF[pd.to_datetime(x_train_randF['Date & Time']) >= input_date_B]
    x_train_randF = x_train_randF[pd.to_datetime(x_train_randF['Date & Time']) < input_date_B]

    y_test_randF = randF_df_B[pd.to_datetime(randF_df_B['Date & Time']) >= input_date_B]['use [kW]']
    y_train_randF = randF_df_B[pd.to_datetime(randF_df_B['Date & Time']) < input_date_B]['use [kW]']



separate_to_train_test_randF(columns)
x_train_randF = x_train_randF.drop('Date & Time', axis=1)
x_test_randF = x_test_randF.drop('Date & Time', axis=1)
```

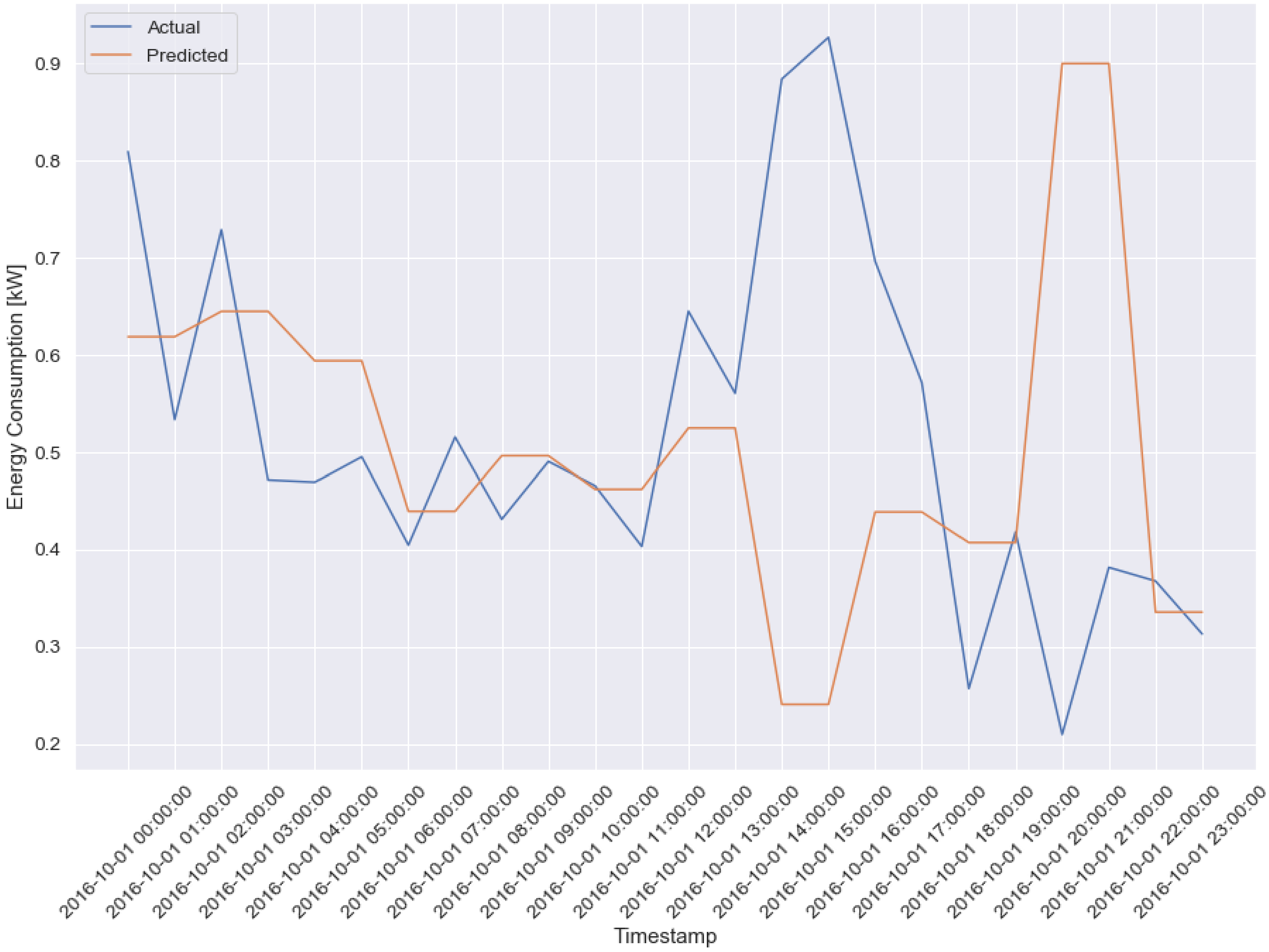
In [1170]:
```
# Scaling the dataframe using standard scaler

sc = StandardScaler()
x_train_randF = sc.fit_transform(x_train_randF)
x_test_randF = sc.transform(x_test_randF)
```

In [1171]:
```
# Training the algorithm
regressor = RandomForestRegressor(n_estimators=20, random_state=0)
regressor.fit(x_train_randF, y_train_randF)
y_pred = regressor.predict(x_test_randF)
```

In [1172]:
```
print('Mean absolute error by Random Forest modeling: %.9f'%mean_absolute_error(y_test_randF, y_pred))
```

```
Mean absolute error by Random Forest modeling: 0.301436240
```

In [1173]:
```
y_test_randF = y_test_randF[0:24]
y_pred = y_pred[0:24]

plt.plot(x_duration, y_test_randF, label="Actual")
plt.plot(x_duration, y_pred, label="Predicted")
plt.xlabel('Timestamp')
plt.ylabel('Energy Consumption [kW]')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [ ]:

In [ ]:

In [ ]:

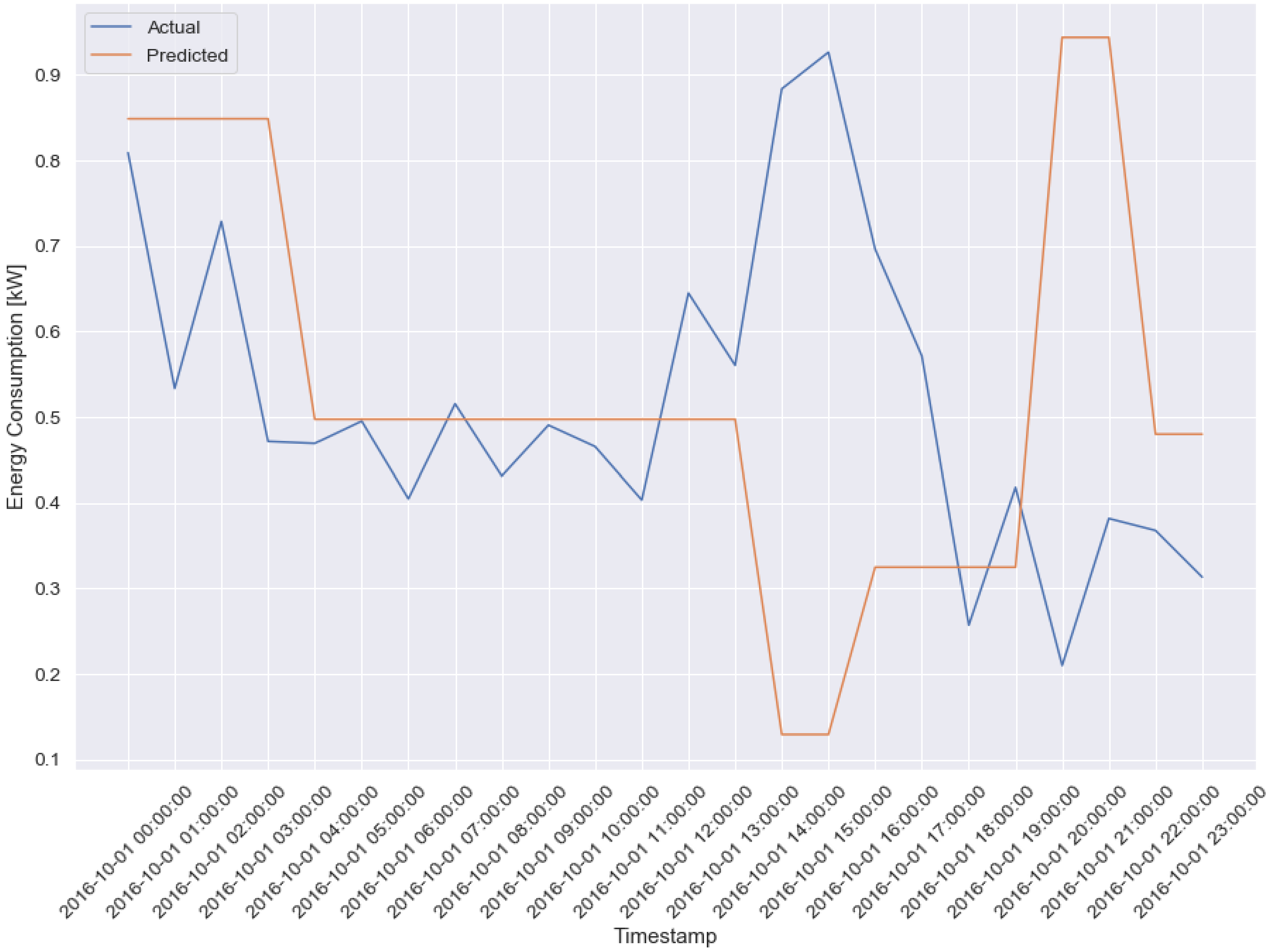
Implementing K-Means Clustering on combined data

In [1183]:
```
kMeans_df_B = dataframe.copy()
columns = columnsPositive_B.append(columnsNegative_B)

def separate_to_train_test_kMeans(col):
    x_col = []
    y_col = []
    global x_train_kMeans
    global y_train_kMeans
    global x_test_kMeans
    global y_test_kMeans

    for name in col:
        if name not in ['use [kW]']:
            x_col.append(name)

    y_col.append('use [kW]')
    x_col.append('Date & Time')
    x_train_kMeans = kMeans_df_B.filter(x_col)
    y_train_kMeans = kMeans_df_B.filter(y_col)

    x_test_kMeans  = x_train_kMeans[pd.to_datetime(x_train_kMeans['Date & Time']) >= input_date_B]
    x_train_kMeans = x_train_kMeans[pd.to_datetime(x_train_kMeans['Date & Time']) < input_date_B]

    y_test_kMeans = kMeans_df_B[pd.to_datetime(kMeans_df_B['Date & Time']) >= input_date_B]['use [kW]']
    y_train_kMeans = kMeans_df_B[pd.to_datetime(kMeans_df_B['Date & Time']) < input_date_B]['use [kW]']



separate_to_train_test_kMeans(columns)
x_train_kMeans = x_train_kMeans.drop('Date & Time', axis=1)
x_test_kMeans = x_test_kMeans.drop('Date & Time', axis=1)
```

In [1184]:
```
regression = KNeighborsRegressor(n_neighbors=1)
regression.fit(x_train_kMeans, y_train_kMeans)
y_pred_kMeans = regression.predict(x_test_kMeans)


print("Mean absolute error by KMeans Regression: %.9f"
      % mean_absolute_error(y_test_kMeans,y_pred_kMeans))
```

```
Mean absolute error by KMeans Regression: 0.377660248
```

In [1185]:
```
y_pred_kMeans = y_pred_kMeans[0:24]
y_test_kMeans = y_test_kMeans[0:24]

plt.plot(x_duration, y_test_kMeans, label="Actual")
plt.plot(x_duration, y_pred_kMeans, label="Predicted")
plt.xlabel('Timestamp')
plt.ylabel('Energy Consumption [kW]')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [ ]:

In [ ]:

In [ ]:

Implementing AdaBoost Regression on combined data

In [1192]:

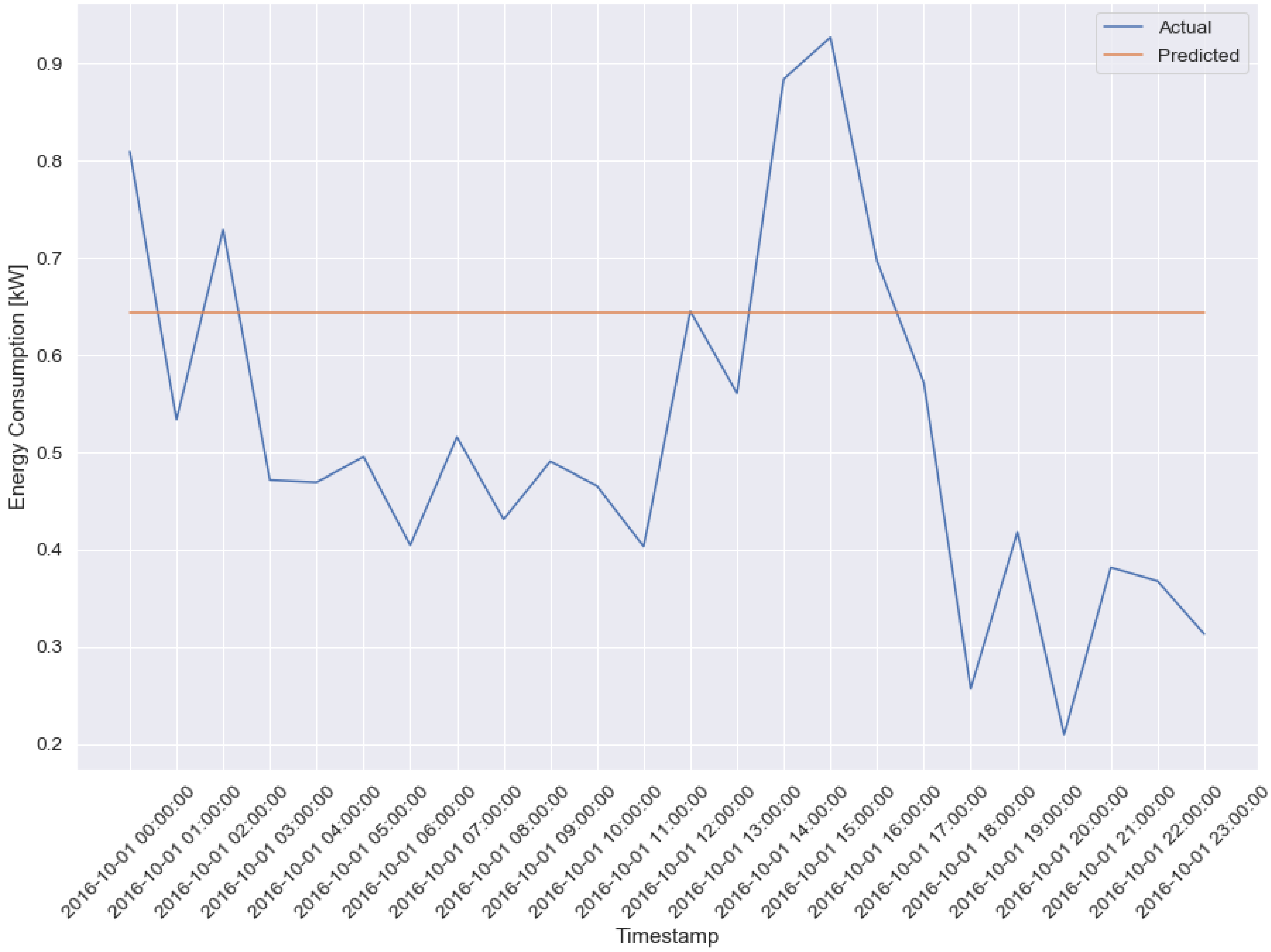
```
adaBoost_df_B = dataframe.copy()
columns = columnsPositive_B.append(columnsNegative_B)

def separate_to_train_test_adaBoost(col):
    x_col = []
    y_col = []
    global x_train_adaBoost
    global y_train_adaBoost
    global x_test_adaBoost
    global y_test_adaBoost

    for name in col:
        if name not in ['use [kW]']:
            x_col.append(name)

    y_col.append('use [kW]')
    x_col.append('Date & Time')
    x_train_adaBoost = adaBoost_df_B.filter(x_col)
    y_train_adaBoost = adaBoost_df_B.filter(y_col)

    x_test_adaBoost = x_train_adaBoost[pd.to_datetime(x_train_adaBoost['Date & Time']) >= input_date_B]
    x_train_adaBoost = x_train_adaBoost[pd.to_datetime(x_train_adaBoost['Date & Time']) < input_date_B]

    y_test_adaBoost = adaBoost_df_B[pd.to_datetime(adaBoost_df_B['Date & Time']) >= input_date_B]['use [kW]']
    y_train_adaBoost = adaBoost_df_B[pd.to_datetime(adaBoost_df_B['Date & Time']) < input_date_B]['use [kW]']



separate_to_train_test_adaBoost(columns)
x_train_adaBoost = x_train_adaBoost.drop('Date & Time', axis=1)
x_test_adaBoost = x_test_adaBoost.drop('Date & Time', axis=1)
```

In [1193]:

```
regression_adaBoost = AdaBoostRegressor(base_estimator=None, n_estimators=50, learning_rate=1.0, loss='linear',
regression_adaBoost.fit(x_train_adaBoost, y_train_adaBoost)
y_pred_adaBoost = regression_adaBoost.predict(x_test_adaBoost)

print("Mean absolute error by AdaBoost Regression: %.9f"% mean_absolute_error(y_test_adaBoost,y_pred_adaBoost))
```

```
Mean absolute error by AdaBoost Regression: 0.343649979
```

In [1194]:
```
y_pred_adaBoost = y_pred_adaBoost[0:24]
y_test_adaBoost = y_test_adaBoost[0:24]

plt.plot(x_duration, y_test_adaBoost, label="Actual")
plt.plot(x_duration, y_pred_adaBoost, label="Predicted")
plt.xlabel('Timestamp')
plt.ylabel('Energy Consumption [kW]')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



In [ ]:

Selecting the Best Model

In [1334]:
```
print("The ARIMA model has the least Mean Absolute Error of - 0.203353026 lesser than the Naive MAE - 0.4023")
```

```
The ARIMA model has the least Mean Absolute Error of - 0.203353026 lesser than the Naive MAE - 0.4023
```

The next 24 hours predictions based on the ARIMA model are -

In [1332]:
```
pd.DataFrame(predictions_arima,columns=["Predicted_Usage[kW]"])[0:24]
```

Out[1332]:

	Predicted_Usage[kW]
0	0.818559
1	0.755739
2	0.710548
3	0.564630
4	0.528269
5	0.543194
6	0.562483
7	0.663348
8	0.797642
9	0.692252
10	0.445240
11	0.418669
12	0.507402
13	0.510593
14	0.455921
15	0.449569
16	0.505920
17	0.576444
18	0.581204
19	0.603110
20	0.747600
21	0.718338
22	0.898264
23	0.763992

In [ ]: