MSDS692 Data Science Practicum Project

Topic: Forecasting Electricity Demand with Time Series

Date: 22MAY2022

Author: Olumide Aluko

Purpose: The project aims to generate a forecasting model capable of predicting the next day's energy demand at the hourly level by accurately predicting monthly electricity demand.

Dataset Source: https://raw.githubusercontent.com/JoaquinAmatRodrigo/skforecast/master/'+'data/vic_elec.csv'

Problem Statement:

A time series with electricity demand (Mega Watts) for the state of Victoria (Australia) from 2011-12-31 to 2014-12-31 is available. Demand for electricity in Australia has been in the spotlight for the general population due to the recently increasing price. Still, forecasts of the electricity demand have been expected to decrease due to various factors. The project aims to generate a forecasting model capable of predicting the next day's energy demand at the hourly level by accurately predicting monthly electricity demand. The proposed project design will be achieved using a time series forecasting with scikit-learn regressors

Data Source

The data used in this document were obtained from the R tsibbledata package but i download it from GitGub for this project. The dataset contains 5 columns and 52,608 complete records. The information in each column is:

Time: date and time of the record.

Date: date of the record.

Demand: electricity demand (MW).

Temperature: temperature in Melbourne, capital of the state of Victoria.

Holiday: indicator if the day is a public holiday.

Import Libraries:

In [1]:

```
# Data manipulation
# ==============================================================================
```

```
import numpy as np
import pandas as pd

# Plots
# ==============================================================================
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
plt.style.use('fivethirtyeight')

# Modelling and Forecasting
# ==============================================================================
from sklearn.linear_model import Ridge
from lightgbm import LGBMRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error
from skforecast.ForecasterAutoreg import ForecasterAutoreg
from skforecast.ForecasterAutoregMultiOutput import ForecasterAutoregMultiOutput
from skforecast.model_selection import grid_search_forecaster
from skforecast.model_selection import backtesting_forecaster

# Warnings configuration
# ==============================================================================
import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
# Data download
# ==============================================================================
data = pd.read_csv('victoria_electricity.csv', sep=',')
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 52608 entries, 0 to 52607
Data columns (total 5 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Time         52608 non-null  object 
 1   Demand       52608 non-null  float64
 2   Temperature  52608 non-null  float64
 3   Date         52608 non-null  object 
 4   Holiday      52608 non-null  bool   
dtypes: bool(1), float64(2), object(2)
memory usage: 1.7+ MB
```

In [3]:

```
len(pd.date_range(start="2011-12-31", end="2014-12-31"))
```

Out[3]: 1097

No missing values, and 3 years of data to enjoy :)

Let's compute some date features and start the intereating part of the analysis

In [4]:

```
# Data preparation
# ==============================================================================
```

```
data = data.copy()
data['Time'] = pd.to_datetime(data['Time'], format='%Y-%m-%dT%H:%M:%SZ')
data = data.set_index('Time')
data = data.asfreq('30min')
data = data.sort_index()
data.head(5)
```

Out[4]:

Time	Demand	Temperature	Date	Holiday
2011-12-31 13:00:00	4382.825174	21.40	2012-01-01	True
2011-12-31 13:30:00	4263.365526	21.05	2012-01-01	True
2011-12-31 14:00:00	4048.966046	20.70	2012-01-01	True
2011-12-31 14:30:00	3877.563330	20.55	2012-01-01	True
2011-12-31 15:00:00	4036.229746	20.40	2012-01-01	True

In [5]:

```
# Verify that a temporary index is complete
# ==============================================================================
(data.index == pd.date_range(start=data.index.min(),
                             end=data.index.max(),
                             freq=data.index.freq)).all()
```

Out[5]: True

In [6]:

```
print(f"Number of rows with missing values: {data.isnull().any(axis=1).mean()}")
```

```
Number of rows with missing values: 0.0
```

In [7]:

```
# Fill gaps in a temporary index
# ==============================================================================
# data.asfreq(freq='30min', fill_value=np.nan)
```

For the 11:00 average value, the 11:00 point value is not included because, in reality, the value is not yet available at that exact time.

In [8]:

```
# Aggregating in 1H intervals
# ==============================================================================
# The Date column is eliminated so that it does not generate an error when aggregating.
# The Holiday column does not generate an error since it is Boolean and is treated as 0
data = data.drop(columns='Date')
data = data.resample(rule='H', closed='left', label ='right').mean()
data
```

Out[8]:

Time	Demand	Temperature	Holiday
2011-12-31 14:00:00	4323.095350	21.225	True
2011-12-31 15:00:00	3963.264688	20.625	True

	Demand	Temperature	Holiday
Time			
2011-12-31 16:00:00	3950.913495	20.325	True
2011-12-31 17:00:00	3627.860675	19.850	True
2011-12-31 18:00:00	3396.251676	19.025	True
...	...	...	...
2014-12-31 09:00:00	4069.625550	21.600	False
2014-12-31 10:00:00	3909.230704	20.300	False
2014-12-31 11:00:00	3900.600901	19.650	False
2014-12-31 12:00:00	3758.236494	18.100	False
2014-12-31 13:00:00	3785.650720	17.200	False

26304 rows × 3 columns

The dataset starts on 2011-12-31 14:00:00 and ends on 2014-12-31 13:00:00. The first 10 and the last 13 records are discarded so that it starts on 2012-01-01 00:00:00 and ends on 2014-12-30 23:00:00. In addition, to optimize the hyperparameters of the model and evaluate its predictive capability, the data are divided into 3 sets, training, validation, and test.

In [9]:

```
# Split data into train-val-test
# ==============================================================================
data = data.loc['2012-01-01 00:00:00': '2014-12-30 23:00:00']
end_train = '2013-12-31 23:59:00'
end_validation = '2014-11-30 23:59:00'
data_train = data.loc[: end_train, :]
data_val   = data.loc[end_train:end_validation, :]
data_test  = data.loc[end_validation:, :]

print(f"Train dates      : {data_train.index.min()} --- {data_train.index.max()}")
print(f"Validation dates : {data_val.index.min()} --- {data_val.index.max()}")
print(f"Test dates       : {data_test.index.min()} --- {data_test.index.max()}")
```

```
Train dates      : 2012-01-01 00:00:00 --- 2013-12-31 23:00:00
Validation dates : 2014-01-01 00:00:00 --- 2014-11-30 23:00:00
Test dates       : 2014-12-01 00:00:00 --- 2014-12-30 23:00:00
```

Data Exploration

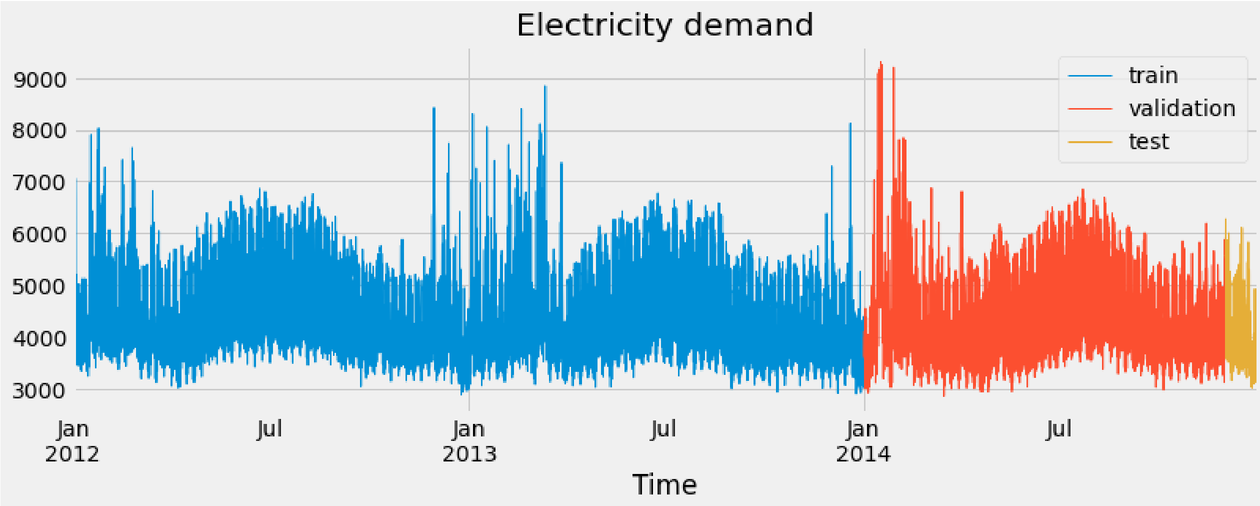
When it is necessary to generate a forecasting model, plotting the time series values could be useful. This allows identifying patterns such as trends and seasonality.

Full time series:

In [10]:

```
# Time series plot
# ==============================================================================
fig, ax = plt.subplots(figsize=(12, 4))
```

```
data_train.Demand.plot(ax=ax, label='train', linewidth=1)
data_val.Demand.plot(ax=ax, label='validation', linewidth=1)
data_test.Demand.plot(ax=ax, label='test', linewidth=1)
ax.set_title('Electricity demand')
ax.legend();
```



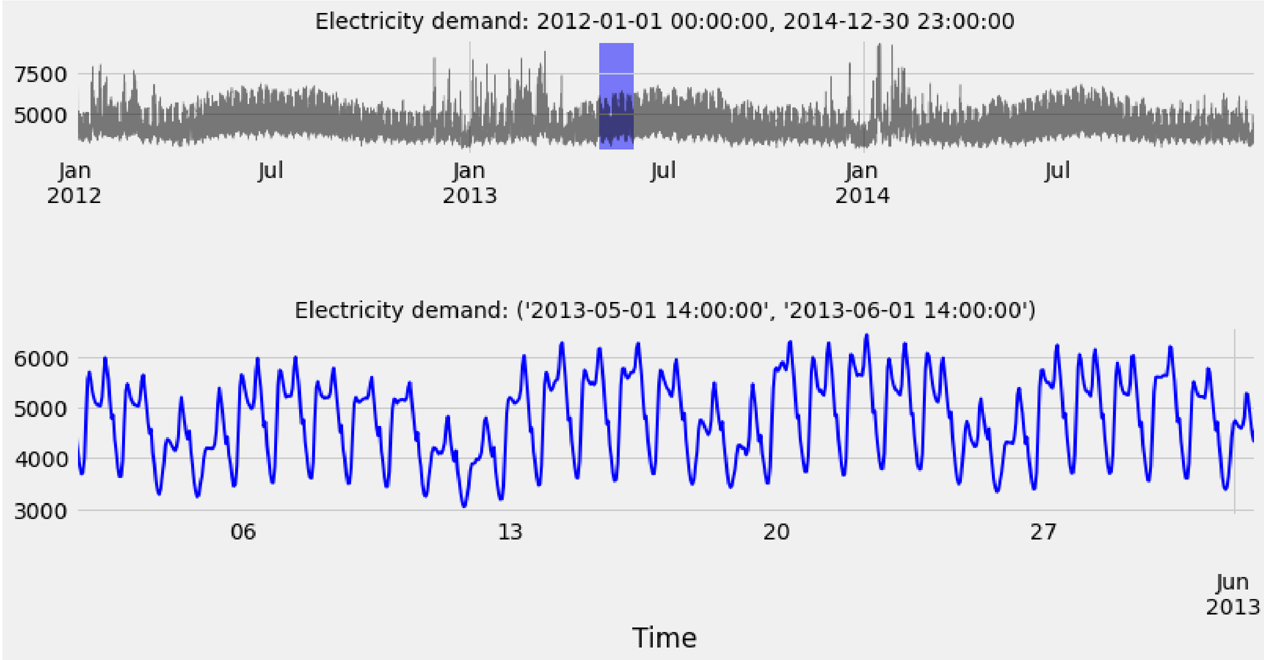
The above graph shows that electricity demand has annual seasonality. There is an increase centered on July and very accentuated demand peaks between January and March.

Section of the time series

Due to the variance of the time series, it is not possible to appreciate with a single chart the possible intraday pattern.

In [11]:

```
#Zooming time series chart
# ==============================================================================
zoom = ('2013-05-01 14:00:00','2013-06-01 14:00:00')

fig = plt.figure(figsize=(12, 6))
grid = plt.GridSpec(nrows=8, ncols=1, hspace=0.6, wspace=0)

main_ax = fig.add_subplot(grid[1:3, :])
zoom_ax = fig.add_subplot(grid[5:, :])

data.Demand.plot(ax=main_ax, c='black', alpha=0.5, linewidth=0.5)
min_y = min(data.Demand)
max_y = max(data.Demand)
main_ax.fill_between(zoom, min_y, max_y, facecolor='blue', alpha=0.5, zorder=0)
main_ax.set_xlabel('')

data.loc[zoom[0]: zoom[1]].Demand.plot(ax=zoom_ax, color='blue', linewidth=2)

main_ax.set_title(f'Electricity demand: {data.index.min()}, {data.index.max()}', fontsi
zoom_ax.set_title(f'Electricity demand: {zoom}', fontsize=14)
plt.subplots_adjust(hspace=1)
```

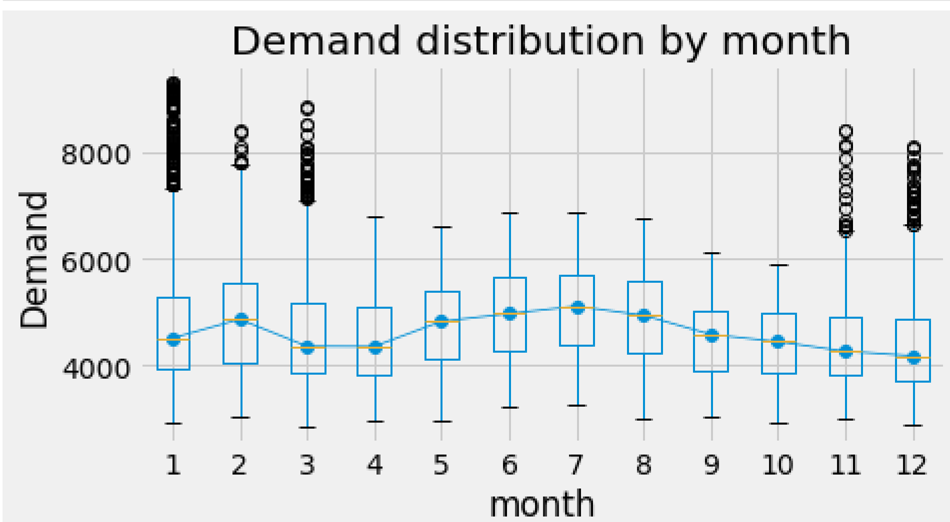


When zooming in on the time series, a clear weekly seasonality is evident, with higher consumption during the work week (Monday to Friday) and lower consumption on weekends. It is also observed that there is a clear correlation between the consumption of one day and that of the previous day.

Annual, weekly and daily seasonality

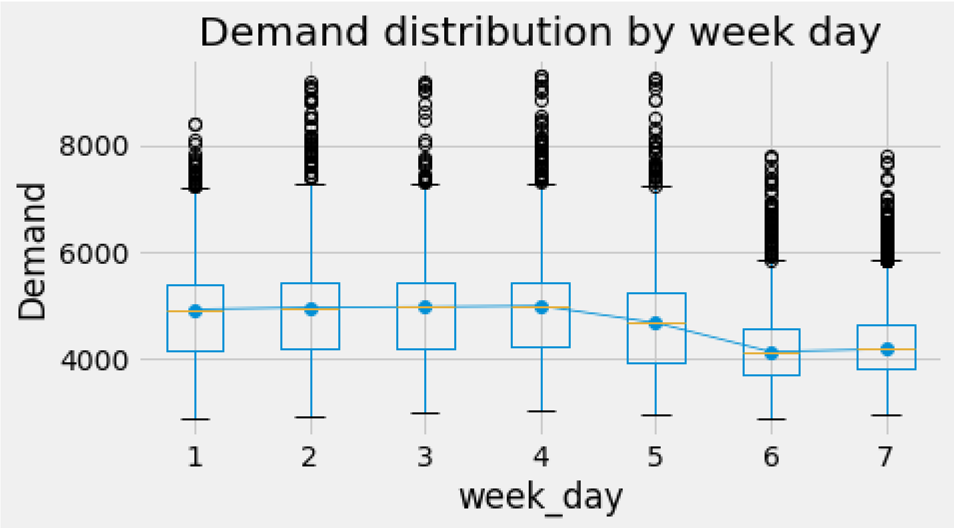
In [12]:

```
# Boxplot for annual seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3.5))
data['month'] = data.index.month
data.boxplot(column='Demand', by='month', ax=ax,)
data.groupby('month')['Demand'].median().plot(style='o-', linewidth=0.8, ax=ax)
ax.set_ylabel('Demand')
ax.set_title('Demand distribution by month')
fig.suptitle('');
```



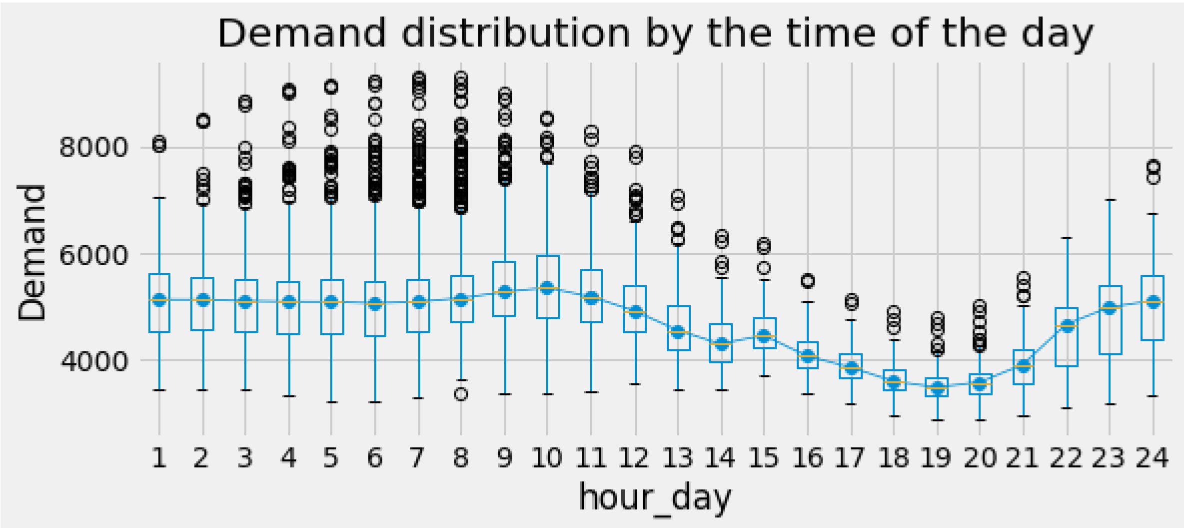
It is observed that there is an annual seasonality, with higher (median) demand values in June, July, and August, and with high demand peaks in November, December, January, February, and March.

In [13]:
```
# Boxplot for weekly seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3.5))
data['week_day'] = data.index.day_of_week + 1
data.boxplot(column='Demand', by='week_day', ax=ax)
data.groupby('week_day')['Demand'].median().plot(style='o-', linewidth=0.8, ax=ax)
ax.set_ylabel('Demand')
ax.set_title('Demand distribution by week day')
fig.suptitle('');
```



Weekly seasonality shows lower demand values during the weekend.

In [14]:
```
# Boxplot for daily seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(9, 3.5))
data['hour_day'] = data.index.hour + 1
data.boxplot(column='Demand', by='hour_day', ax=ax)
data.groupby('hour_day')['Demand'].median().plot(style='o-', linewidth=0.8, ax=ax)
ax.set_ylabel('Demand')
ax.set_title('Demand distribution by the time of the day')
fig.suptitle('');
```

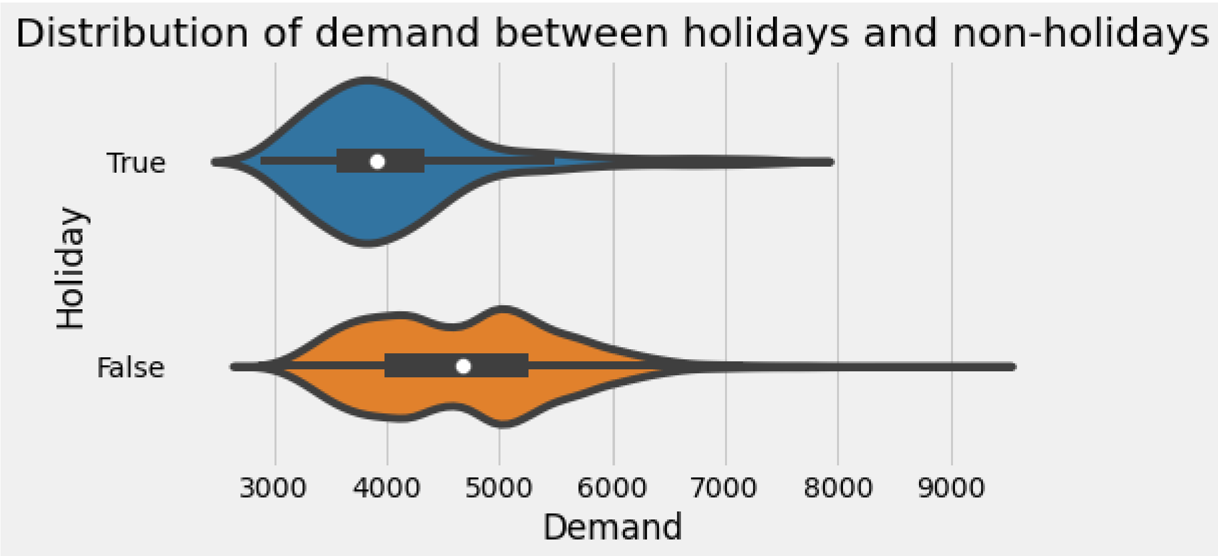


There is also a daily seasonality, with demand decreasing between 16:00 and 21:00 hours.

Holidays and non-holiday days

In [15]:

```
# Violinplot
# ==============================================================================
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(7, 3.5))
sns.violinplot(
    x       = 'Demand',
    y       = 'Holiday',
    data    = data.assign(Holiday = data.Holiday.astype(str)),
    palette = 'tab10',
    ax      = ax
)
ax.set_title('Distribution of demand between holidays and non-holidays')
ax.set_xlabel('Demand')
ax.set_ylabel('Holiday');
```

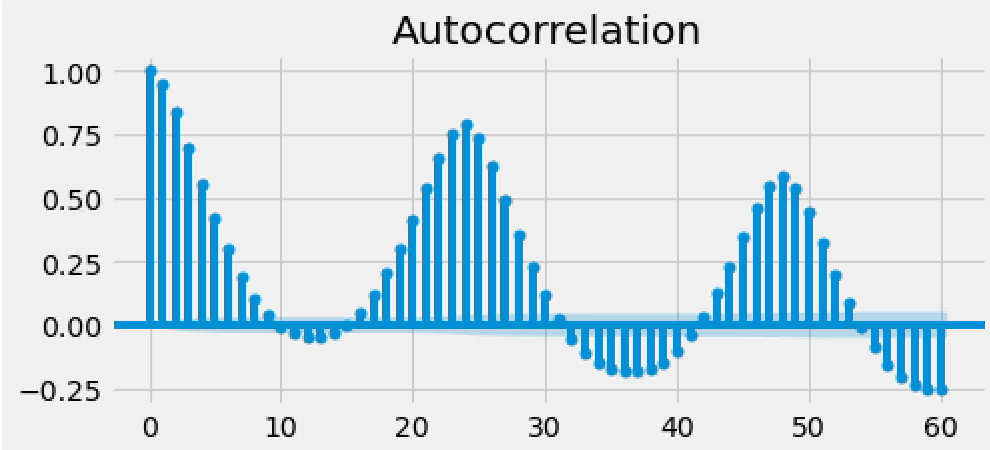


Holidays tend to have lower consumption.

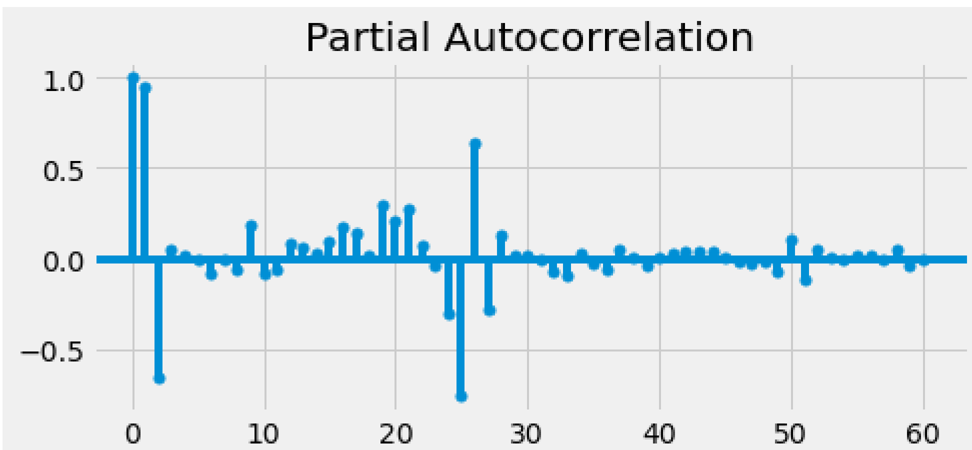
Autocorrelation plots

In [16]:

```
# Autocorrelation plot
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3))
plot_acf(data.Demand, ax=ax, lags=60)
plt.show()
```



In [17]:
```
# Partial autocorrelation plot
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3))
plot_pacf(data.Demand, ax=ax, lags=60)
plt.show()
```



The autocorrelation and partial autocorrelation plots show a clear association between one hour's demand and previous hours, as well as between one hour's demand and the same hour's demand on previous days. This type of correlation is an indication that autoregressive models can work well.

Recursive autoregressive forecasting

A recursive autoregressive model (ForecasterAutoreg) is created and trained from a linear regression model with a Ridge penalty and a time window of 24 lags. The latter means that, for each prediction, the demand values of the previous 24 hours are used as predictors.

Forecaster training

In [18]:
```
# Create and train forecaster
# ==============================================================================
forecaster = ForecasterAutoreg(
                regressor = make_pipeline(StandardScaler(), Ridge()),
                lags      = 24
```

```
            )

forecaster.fit(y=data.loc[:end_validation, 'Demand'])
forecaster
```

Out[18]:

```
=================
ForecasterAutoreg
=================
Regressor: Pipeline(steps=[('standardscaler', StandardScaler()), ('ridge', Ridge())])
Lags: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
Window size: 24
Included exogenous: False
Type of exogenous variable: None
Exogenous variables names: None
Training range: [Timestamp('2012-01-01 00:00:00'), Timestamp('2014-11-30 23:00:00')]
Training index type: DatetimeIndex
Training index frequency: H
Regressor parameters: {'standardscaler__copy': True, 'standardscaler__with_mean': True,
'standardscaler__with_std': True, 'ridge__alpha': 1.0, 'ridge__copy_X': True, 'ridge__fi
t_intercept': True, 'ridge__max_iter': None, 'ridge__normalize': 'deprecated', 'ridge__p
ositive': False, 'ridge__random_state': None, 'ridge__solver': 'auto', 'ridge__tol': 0.0
01}
Creation date: 2022-05-22 17:13:26
Last fit date: 2022-05-22 17:13:26
Skforecast version: 0.4.3
```

Backtest

How the model would have behaved if it had been trained with the data from 2012-01-01 00:00 to 2014-11-30 23:59 and then, at 23:59 each day, the following 24 hours were predicted is evaluated. This type of evaluation, known as Backtesting, can be easily implemented with the function backtesting_forecaster(). This function returns, in addition to the predictions, an error metric.

In [19]:

```
# Backtest
# ==============================================================================
metric, predictions = backtesting_forecaster(
                            forecaster = forecaster,
                            y          = data.Demand,
                            initial_train_size = len(data.loc[:end_validation]),
                            fixed_train_size   = False,
                            steps      = 24,
                            metric     = 'mean_absolute_error',
                            refit      = False,
                            verbose    = True
                        )
```

```
Information of backtesting process
----------------------------------
Number of observations used for initial training or as initial window: 25560
Number of observations used for backtesting: 720
    Number of folds: 30
    Number of steps per fold: 24

Data partition in fold: 0
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-01 00:00:00 -- 2014-12-01 23:00:00
Data partition in fold: 1
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
```

```
    Validation: 2014-12-02 00:00:00 -- 2014-12-02 23:00:00
Data partition in fold: 2
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-03 00:00:00 -- 2014-12-03 23:00:00
Data partition in fold: 3
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-04 00:00:00 -- 2014-12-04 23:00:00
Data partition in fold: 4
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-05 00:00:00 -- 2014-12-05 23:00:00
Data partition in fold: 5
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-06 00:00:00 -- 2014-12-06 23:00:00
Data partition in fold: 6
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-07 00:00:00 -- 2014-12-07 23:00:00
Data partition in fold: 7
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-08 00:00:00 -- 2014-12-08 23:00:00
Data partition in fold: 8
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-09 00:00:00 -- 2014-12-09 23:00:00
Data partition in fold: 9
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-10 00:00:00 -- 2014-12-10 23:00:00
Data partition in fold: 10
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-11 00:00:00 -- 2014-12-11 23:00:00
Data partition in fold: 11
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-12 00:00:00 -- 2014-12-12 23:00:00
Data partition in fold: 12
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-13 00:00:00 -- 2014-12-13 23:00:00
Data partition in fold: 13
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-14 00:00:00 -- 2014-12-14 23:00:00
Data partition in fold: 14
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-15 00:00:00 -- 2014-12-15 23:00:00
Data partition in fold: 15
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-16 00:00:00 -- 2014-12-16 23:00:00
Data partition in fold: 16
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-17 00:00:00 -- 2014-12-17 23:00:00
Data partition in fold: 17
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-18 00:00:00 -- 2014-12-18 23:00:00
Data partition in fold: 18
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-19 00:00:00 -- 2014-12-19 23:00:00
Data partition in fold: 19
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-20 00:00:00 -- 2014-12-20 23:00:00
Data partition in fold: 20
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-21 00:00:00 -- 2014-12-21 23:00:00
Data partition in fold: 21
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-22 00:00:00 -- 2014-12-22 23:00:00
Data partition in fold: 22
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-23 00:00:00 -- 2014-12-23 23:00:00
Data partition in fold: 23
```

```
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-24 00:00:00 -- 2014-12-24 23:00:00
Data partition in fold: 24
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-25 00:00:00 -- 2014-12-25 23:00:00
Data partition in fold: 25
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-26 00:00:00 -- 2014-12-26 23:00:00
Data partition in fold: 26
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-27 00:00:00 -- 2014-12-27 23:00:00
Data partition in fold: 27
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-28 00:00:00 -- 2014-12-28 23:00:00
Data partition in fold: 28
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-29 00:00:00 -- 2014-12-29 23:00:00
Data partition in fold: 29
    Training:   2012-01-01 00:00:00 -- 2014-11-30 23:00:00
    Validation: 2014-12-30 00:00:00 -- 2014-12-30 23:00:00
```

In [ ]: