MSDS692 Data Science Practicum Project

Topic: Forecasting Electricity Demand with Time Series

Date: 6JUNE2022

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

Time	Demand	Temperature	Holiday
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

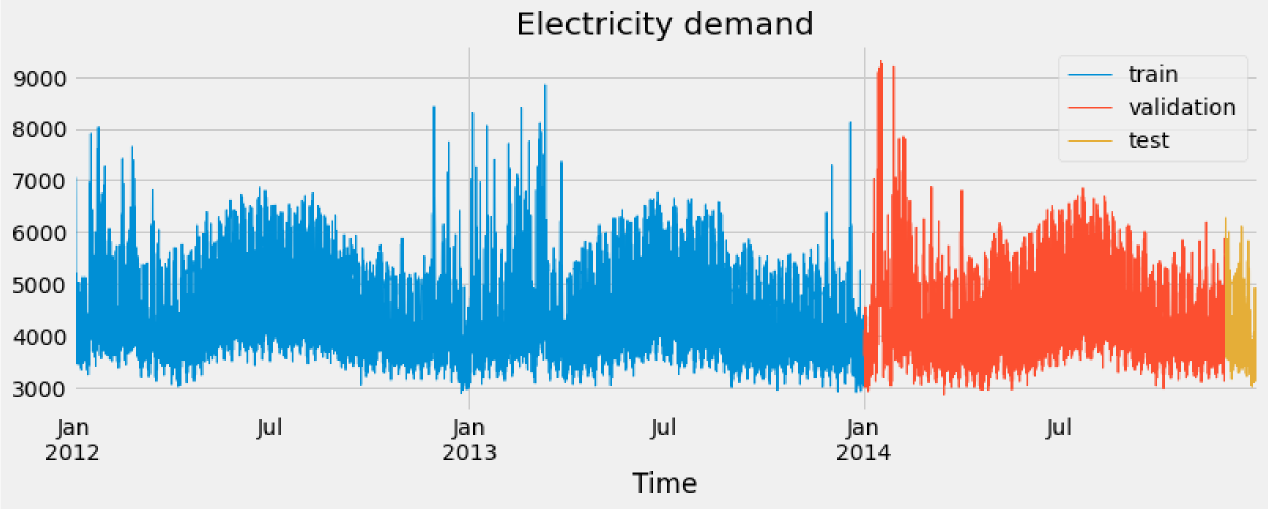
When it is necessary to generate a forecasting model, plotting the time series values could be useful. This allows identifying patterns such as trends and seasonality.

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();
```

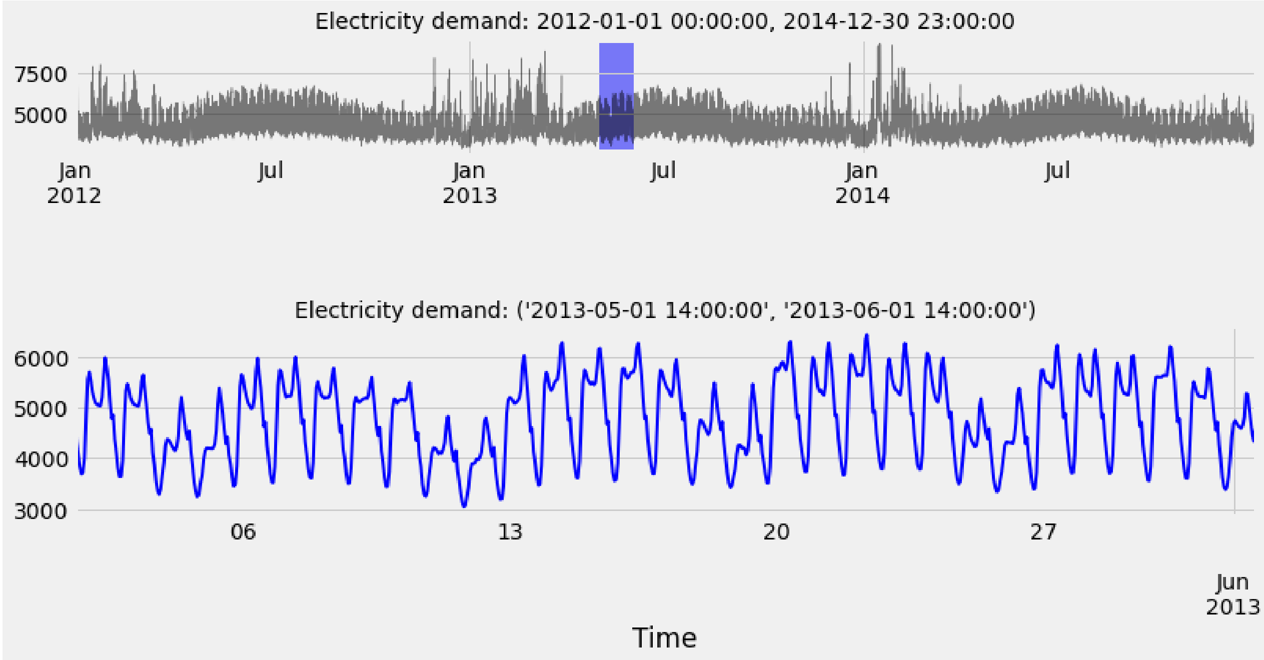


The above graph shows that electricity demand has annual seasonality. There is an increase centered on July and very accentuated demand peaks between January and March.

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)
```

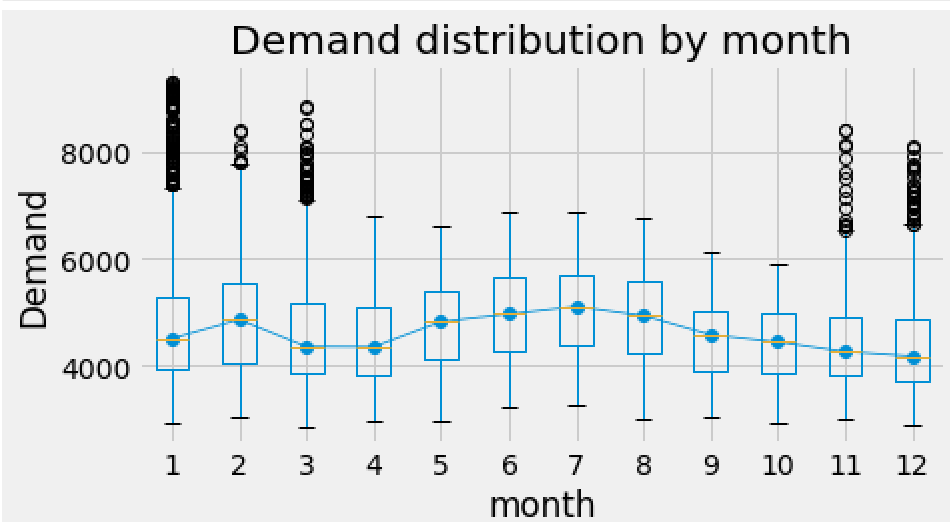


When zooming in on the time series, a clear weekly seasonality is evident, with higher consumption during the work week (Monday to Friday) and lower consumption on weekends. It is also observed that there is a clear correlation between the consumption of one day and that of the previous day.

Annual, weekly and daily seasonality

In [12]:

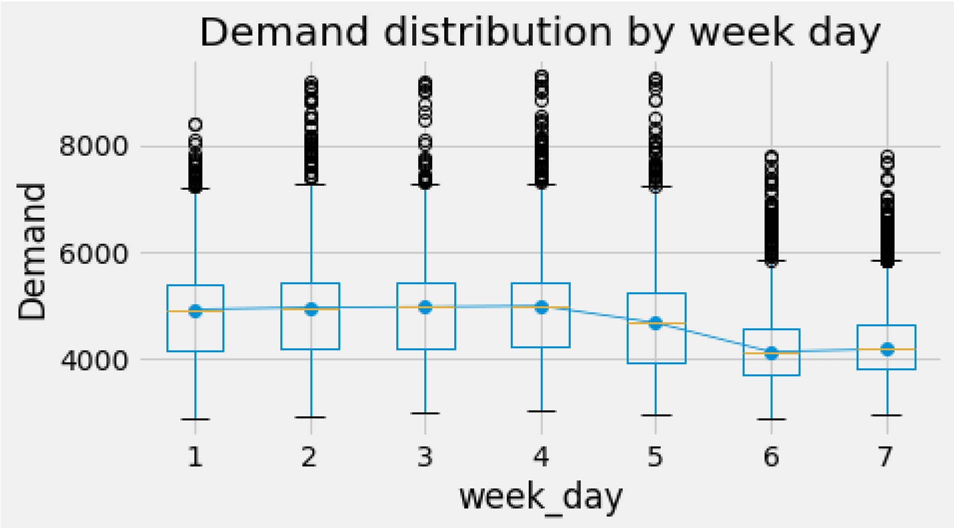
```
# Boxplot for annual seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3.5))
data['month'] = data.index.month
data.boxplot(column='Demand', by='month', ax=ax,)
data.groupby('month')['Demand'].median().plot(style='o-', linewidth=0.8,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

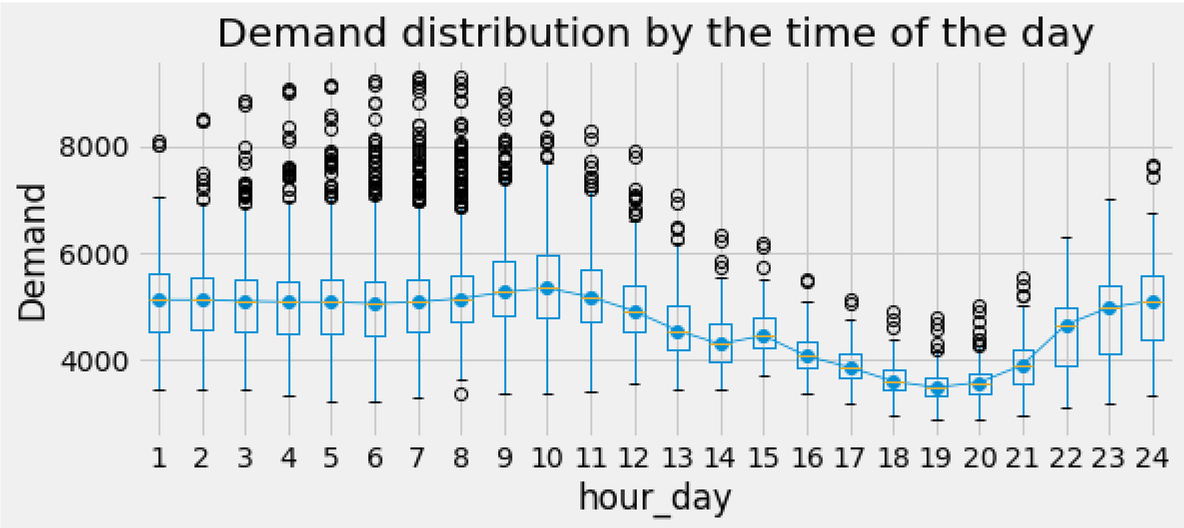
```
# Boxplot for weekly seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3.5))
data['week_day'] = data.index.day_of_week + 1
data.boxplot(column='Demand', by='week_day', ax=ax)
data.groupby('week_day')['Demand'].median().plot(style='o-', 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('');
```

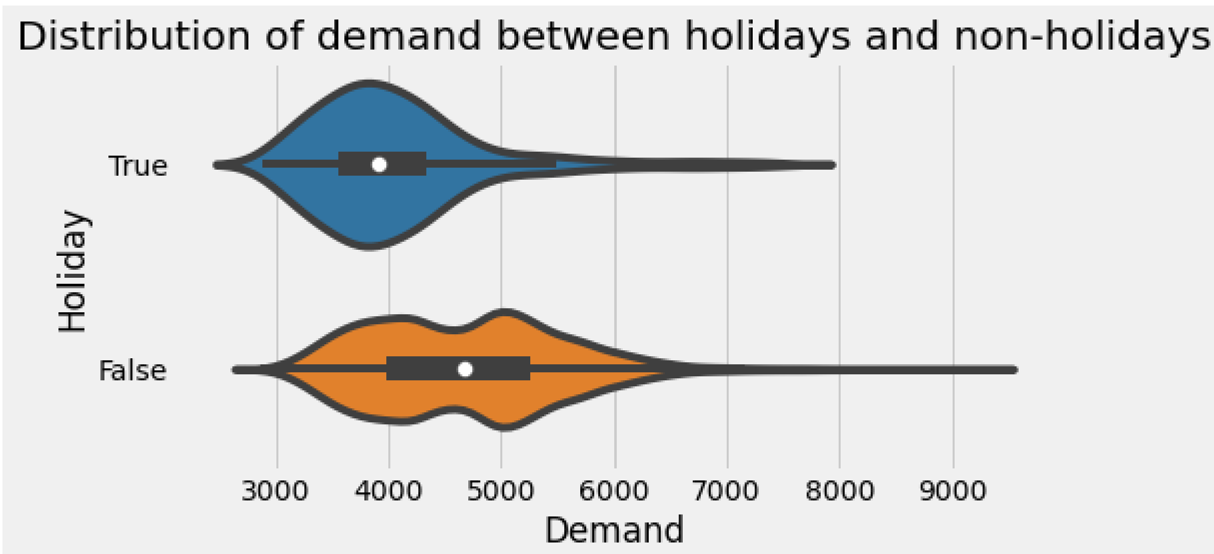


There is also a daily seasonality, with demand decreasing between 16:00 and 21:00 hours.

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');
```



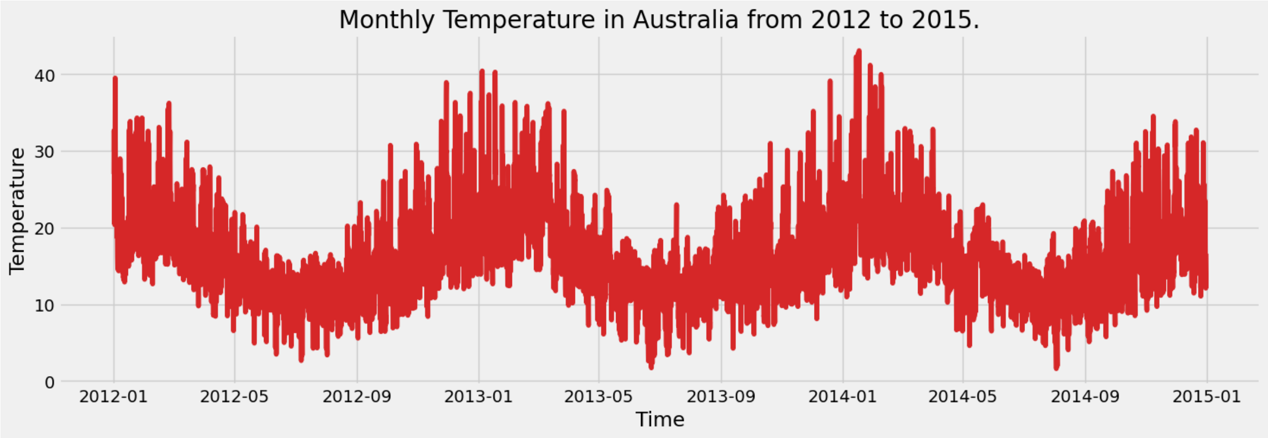
Holidays tend to have lower consumption.

EDA - Temperature

Monthly Temperature in Australia from 2012 to 2015

In [16]:

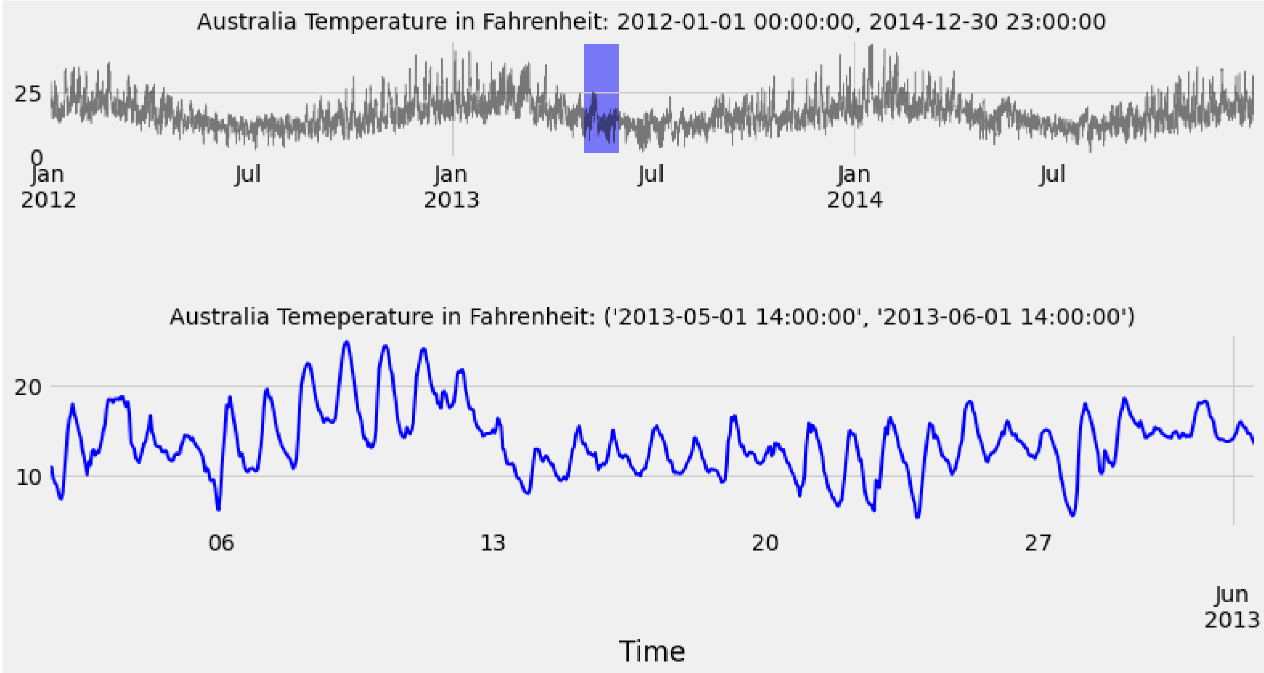
```
# Draw Plot
def plot_data(data, x, y, title="", xlabel='Time', ylabel='Temperature', dpi=100):
    plt.figure(figsize=(16,5), dpi=dpi)
    plt.plot(x, y, color='tab:red')
    plt.gca().set(title=title, xlabel=xlabel, ylabel=ylabel)
    plt.show()

plot_data(data, x=data.index, y=data.Temperature, title='Monthly Temperature in Austral
```



The above graph shows that electricity temperature has annual seasonality. There is an increase centered on October/November and very accentuated demand peaks between January and February.

In [17]:

```
#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.Temperature.plot(ax=main_ax, c='black', alpha=0.5, linewidth=0.5)
min_y = min(data.Temperature)
max_y = max(data.Temperature)
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]].Temperature.plot(ax=zoom_ax, color='blue', linewidth=2)

main_ax.set_title(f'Australia Temperature in Fahrenheit: {data.index.min()}, {data.inde
zoom_ax.set_title(f'Australia Temeperature in Fahrenheit: {zoom}', fontsize=14)
plt.subplots_adjust(hspace=1)
```

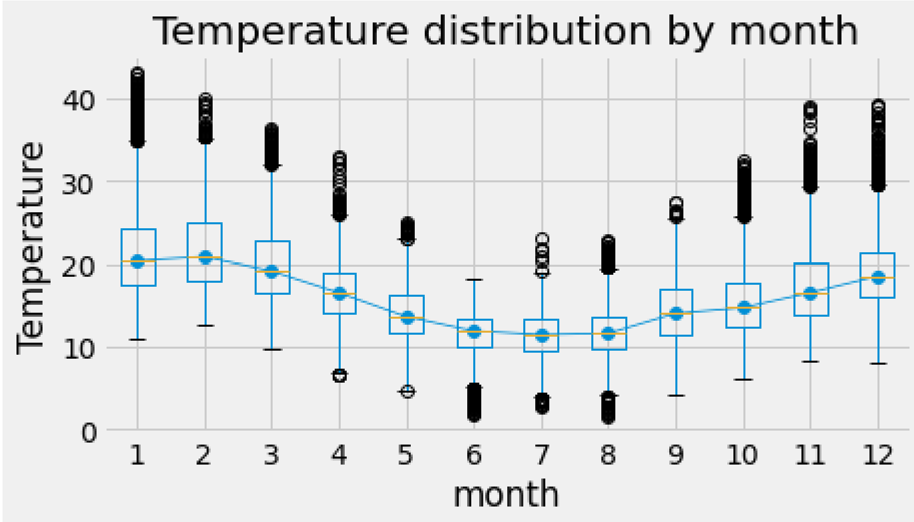


When zooming in on the time series, a weekly seasonality of temperature is not consistent it varies depending on the week, with higher consumption during the second work week (Monday to Friday) and lower consumption on fourth week. It is also observed that there is no clear correlation between the consumption of one day and that of the previous day.

Annual, weekly and daily temperature in Australia

In [18]:

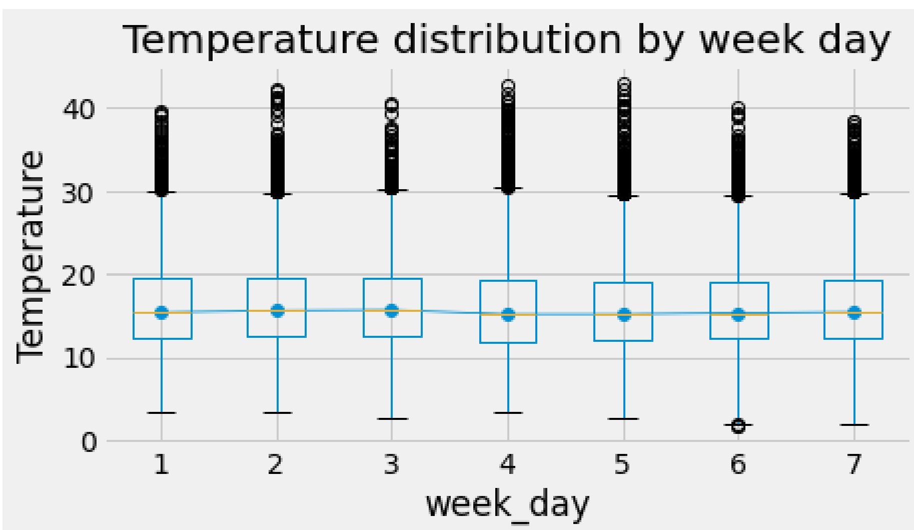
```
# Boxplot for annual seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(7, 3.5))
data['month'] = data.index.month
data.boxplot(column='Temperature', by='month', ax=ax,)
data.groupby('month')['Temperature'].median().plot(style='o-',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



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

In [19]:

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

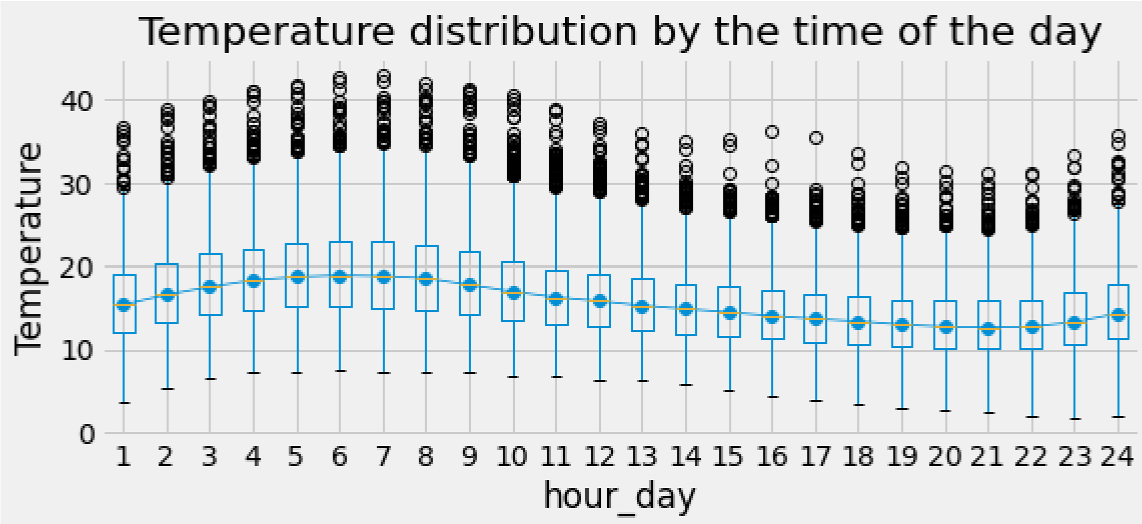


Weekly seasonality shows lower temperature during the week days and weekend.

In [20]:

```
# Boxplot for daily seasonality
# ==============================================================================
fig, ax = plt.subplots(figsize=(9, 3.5))
data['hour_day'] = data.index.hour + 1
data.boxplot(column='Temperature', by='hour_day', ax=ax)
data.groupby('hour_day')['Temperature'].median().plot(style='o-', linewidth=0.8, ax=ax)
ax.set_ylabel('Temperature')
```

```
ax.set_title('Temperature distribution by the time of the day')
fig.suptitle('');
```



There is also a daily seasonality, with temperature slightly decreasing between 19:00 and 21:00 hours.

In [21]:

```
print(data[['Demand', 'Temperature']].corr(method='spearman'))

# Print the correlation between Demand and Temperature columns
```

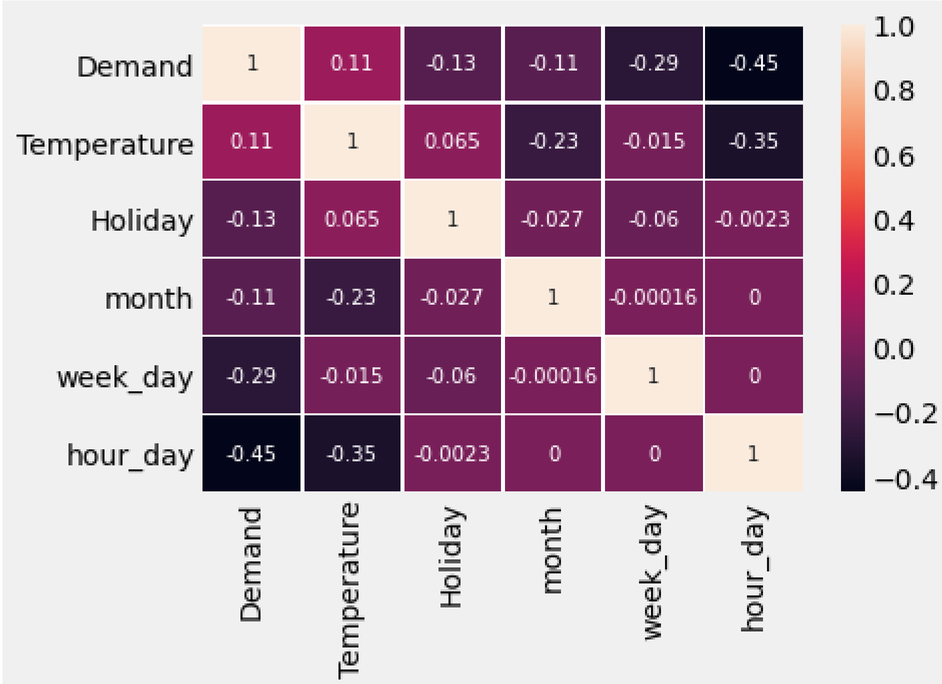
```
               Demand  Temperature
Demand       1.000000     0.113958
Temperature  0.113958     1.000000
```

The correlation matrix above indicates No relationship between Elecctricity Demand and the Temperature in Victoria Australia.

Verify Correlation via an Heatmap

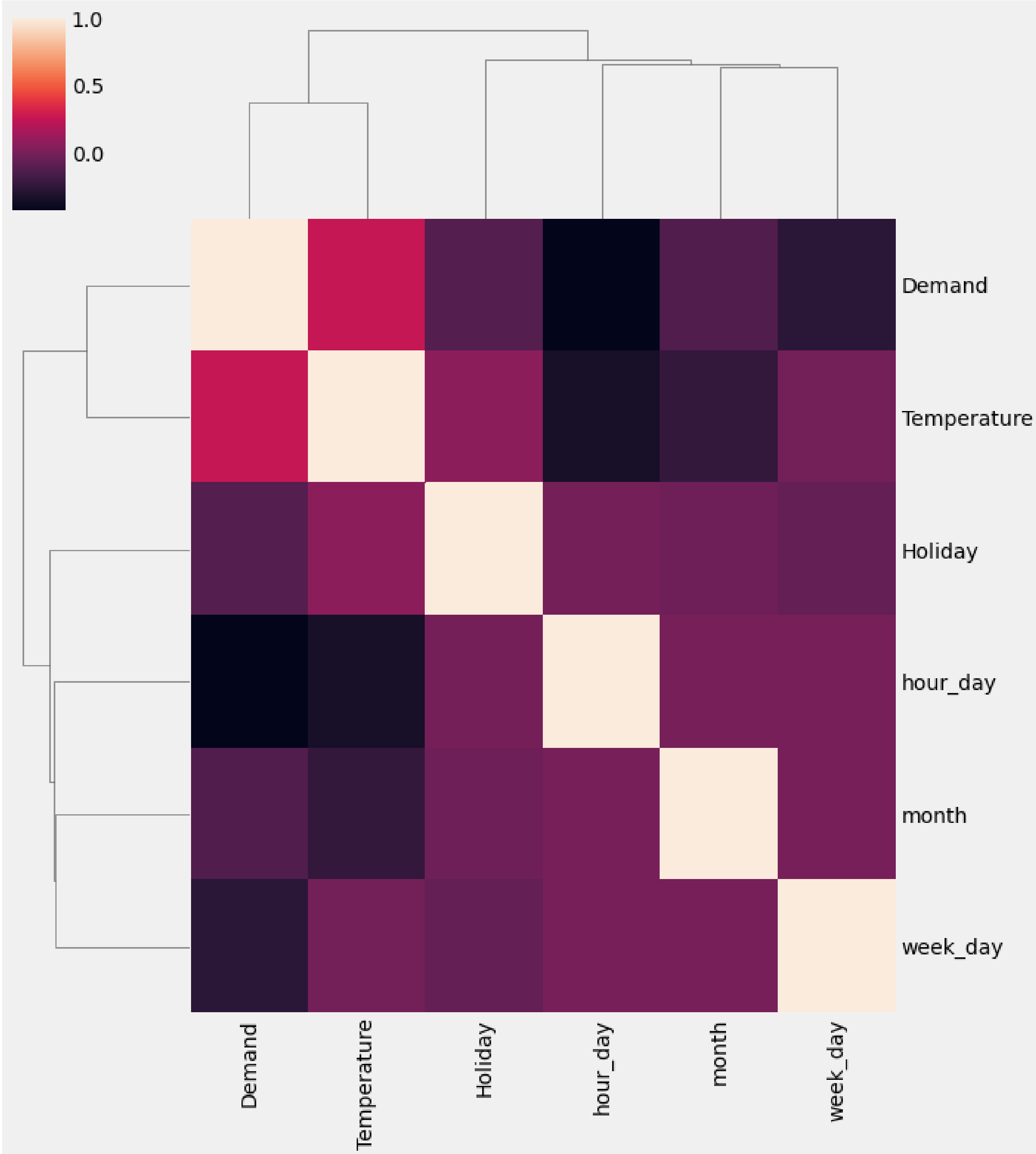
In [22]:

```
corr_data = data.corr(method='spearman')

# Customize the heatmap of the corr_meat correlation matrix
sns.heatmap(corr_data,
            annot=True,
            linewidths=0.4,
            annot_kws={'size': 10});

plt.xticks(rotation=90);
plt.yticks(rotation=0);
```



In [23]:

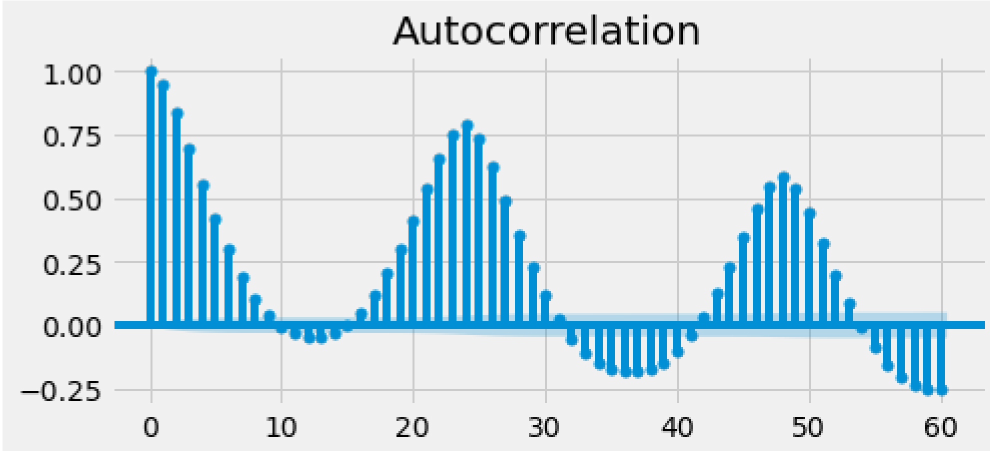
```
corr_data = data.corr(method='pearson')

# Customize the heatmap of the corr_meat correlation matrix
fig = sns.clustermap(corr_data,
               row_cluster=True,
               col_cluster=True,
               figsize=(10, 10));

plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90);
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0);
```



Autocorrelation plots

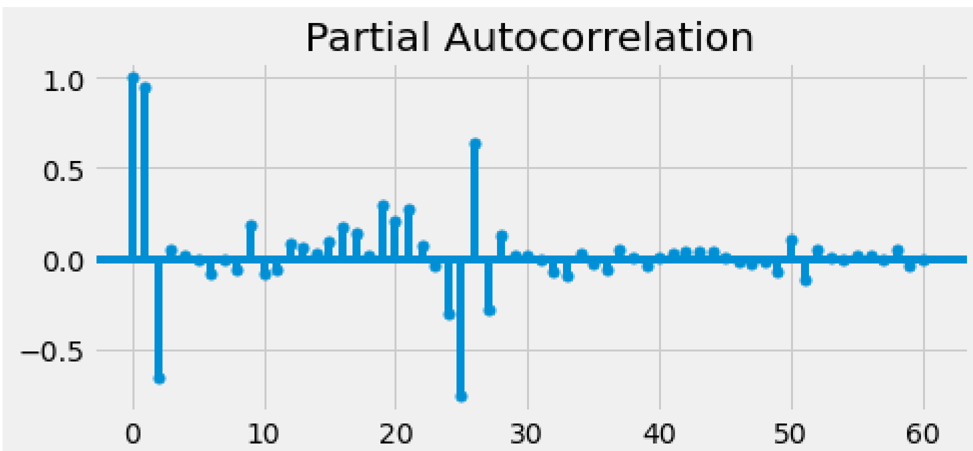
In [24]:

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



In [25]:

```
# 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 [26]:

```
# Create and train forecaster
# ==============================================================================
forecaster = ForecasterAutoreg(
                regressor = make_pipeline(StandardScaler(), Ridge()),
                lags      = 24
```

```
             )

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

Out[26]:

```
=================
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: 2022-06-06 00:07:30
Last fit date: 2022-06-06 00:07:31
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 [27]:

```
# 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
                      )
```

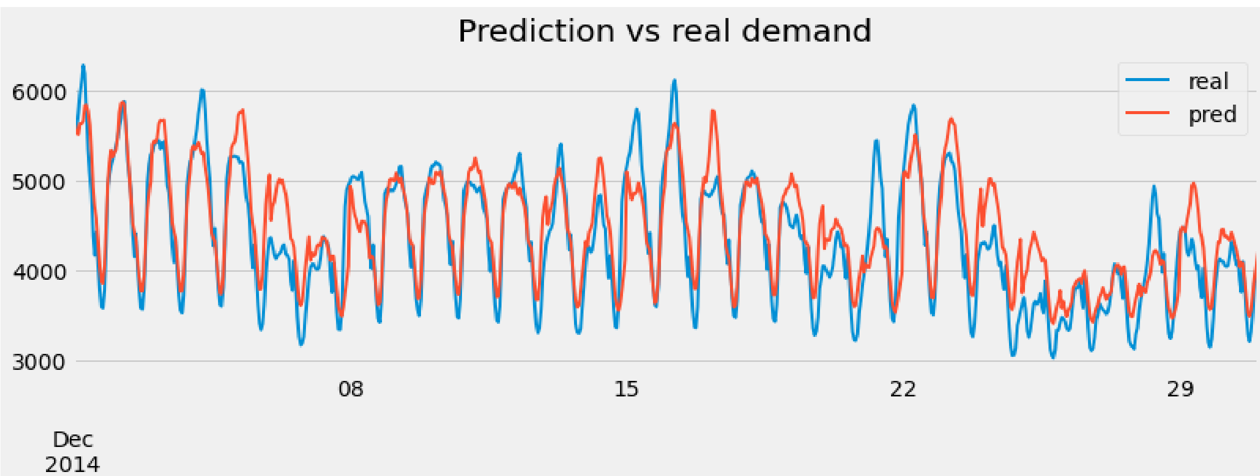
```
Information of backtesting process
----------------------------------
Number of observations used for initial training or as initial window: 25560
Number of observations used for backtesting: 720
    Number of folds: 30
    Number of steps per fold: 24

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 [28]:

```
# Plot
# ==============================================================================
fig, ax = plt.subplots(figsize=(12, 3.5))
data.loc[predictions.index, 'Demand'].plot(ax=ax, linewidth=2, label='real')
predictions.plot(linewidth=2, label='prediction', ax=ax)
ax.set_title('Prediction vs real demand')
ax.legend();
```



In [29]:

```
# Backtest error
# ==============================================================================
print(f'Backtest error: {metric}')
```

```
Backtest error: 289.5191503038139
```

Hyperparameter tuning

In the trained ForecasterAutoreg object, the first 24 lags and a Ridge model with the default hyperparameters have been utilized. However, there is no basis why these values are the most appropriate.

To recognize the best combination of lags and hyperparameters, a Grid Search with validation by Backtesting is employed. This process contains training a model with numerous combinations of hyperparameters and lags and evaluating its predictive capacity. In the search process, it is significant to analyze the models using only the validation data and not to include the test data, which are used only to evaluate the final model.

In [30]:

```
# Hyperparameter Grid search
# ==============================================================================
forecaster = ForecasterAutoreg(
                    regressor = make_pipeline(StandardScaler(), Ridge()),
                    lags      = 24 # This value will be replaced in the grid search
             )

# Lags used as predictors
lags_grid = [5, 24, [1, 2, 3, 23, 24, 25, 47, 48, 49]]

# Regressor's hyperparameters
param_grid = {'ridge__alpha': np.logspace(-3, 5, 10)}

results_grid = grid_search_forecaster(
                        forecaster  = forecaster,
                        y           = data.loc[:end_validation, 'Demand'],
                        param_grid  = param_grid,
                        lags_grid   = lags_grid,
                        steps       = 24,
                        metric      = 'mean_absolute_error',
                        refit       = False,
                        initial_train_size = len(data[:end_train]),
                        fixed_train_size   = False,
                        return_best = True,
                        verbose     = False
                )
```

```
loop lags_grid:   0%|                                          | 0/3 [00:00<?, ?it/
s]
loop param_grid:   0%|                                         | 0/10 [00:00<?, ?it/
s]
Number of models compared: 30
loop param_grid:  10%|██                              | 1/10 [00:01<00:16,  1.79s/
it]
loop param_grid:  20%|████                            | 2/10 [00:03<00:12,  1.55s/
it]
loop param_grid:  30%|██████                          | 3/10 [00:04<00:10,  1.51s/i
t]
loop param_grid:  40%|████████                        | 4/10 [00:06<00:08,  1.50s/
it]
loop param_grid:  50%|██████████                      | 5/10 [00:07<00:07,  1.48s/i
t]
loop param_grid:  60%|████████████                    | 6/10 [00:09<00:06,  1.51s/
it]
loop param_grid:  70%|██████████████                  | 7/10 [00:11<00:05,  1.94s/
it]
loop param_grid:  80%|████████████████                | 8/10 [00:13<00:03,  1.86s/i
t]
loop param_grid:  90%|██████████████████              | 9/10 [00:15<00:01,  1.90s/
it]
loop param_grid: 100%|████████████████████| 10/10 [00:17<00:00,  1.79s/i
```

```
t]
loop lags_grid:  33%|██████▌             | 1/3 [00:17<00:34, 17.20s/i
t]
loop param_grid:   0%|                   | 0/10 [00:00<?, ?it/
s]
loop param_grid:  10%|█▊                 | 1/10 [00:01<00:14,  1.62s/
it]
loop param_grid:  20%|███▋               | 2/10 [00:03<00:16,  2.03s/
it]
loop param_grid:  30%|█████▍             | 3/10 [00:05<00:13,  1.99s/i
t]
loop param_grid:  40%|███████▏           | 4/10 [00:07<00:10,  1.77s/
it]
loop param_grid:  50%|█████████          | 5/10 [00:08<00:08,  1.64s/i
t]
loop param_grid:  60%|██████████▊        | 6/10 [00:10<00:06,  1.56s/
it]
loop param_grid:  70%|████████████▋      | 7/10 [00:11<00:04,  1.58s/
it]
loop param_grid:  80%|██████████████▍    | 8/10 [00:13<00:03,  1.56s/i
t]
loop param_grid:  90%|████████████████▏  | 9/10 [00:14<00:01,  1.57s/
it]
loop param_grid: 100%|██████████████████| 10/10 [00:16<00:00,  1.57s/i
t]
loop lags_grid:  67%|█████████████       | 2/3 [00:33<00:16, 16.75s/i
t]
loop param_grid:   0%|                   | 0/10 [00:00<?, ?it/
s]
loop param_grid:  10%|█▊                 | 1/10 [00:01<00:14,  1.61s/
it]
loop param_grid:  20%|███▋               | 2/10 [00:03<00:12,  1.61s/
it]
loop param_grid:  30%|█████▍             | 3/10 [00:04<00:10,  1.53s/i
t]
loop param_grid:  40%|███████▏           | 4/10 [00:06<00:09,  1.53s/
it]
loop param_grid:  50%|█████████          | 5/10 [00:07<00:07,  1.46s/i
t]
loop param_grid:  60%|██████████▊        | 6/10 [00:08<00:05,  1.43s/
it]
loop param_grid:  70%|████████████▋      | 7/10 [00:10<00:04,  1.41s/
it]
loop param_grid:  80%|██████████████▍    | 8/10 [00:11<00:02,  1.41s/i
t]
loop param_grid:  90%|████████████████▏  | 9/10 [00:13<00:01,  1.43s/
it]
loop param_grid: 100%|██████████████████| 10/10 [00:15<00:00,  1.58s/i
t]
loop lags_grid: 100%|███████████████████| 3/3 [00:48<00:00, 16.23s/i
t]
`Forecaster` refitted using the best-found lags and parameters, and the whole data set:
  Lags: [ 1  2  3 23 24 25 47 48 49]
  Parameters: {'ridge__alpha': 215.44346900318823}
  Backtesting metric: 257.8431725056719
```

In [31]:

```
# Grid Search results
# ==============================================================================
results_grid
```

Out[31]:

	lags	params	metric	ridge__alpha
26	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 215.44346900318823}	257.843173	215.443469
25	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 27.825594022071257}	290.555205	27.825594
24	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 3.593813663804626}	306.631981	3.593814
23	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 0.46415888336127775}	309.393349	0.464159
22	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 0.05994842503189409}	309.776084	0.059948
21	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 0.007742636826811269}	309.825962	0.007743
20	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 0.001}	309.832410	0.001000
10	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 0.001}	325.041129	0.001000
11	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 0.007742636826811269}	325.043579	0.007743
12	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 0.05994842503189409}	325.062536	0.059948
13	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 0.46415888336127775}	325.208755	0.464159
14	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 3.593813663804626}	326.307375	3.593814
15	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 27.825594022071257}	333.395125	27.825594
27	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 1668.1005372000557}	356.547658	1668.100537
16	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 215.44346900318823}	360.841496	215.443469
17	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 1668.1005372000557}	396.342247	1668.100537
18	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 12915.496650148827}	421.002019	12915.496650
28	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 12915.496650148827}	443.551888	12915.496650
19	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14...	{'ridge__alpha': 100000.0}	540.659659	100000.000000
29	[1, 2, 3, 23, 24, 25, 47, 48, 49]	{'ridge__alpha': 100000.0}	545.502052	100000.000000
7	[1, 2, 3, 4, 5]	{'ridge__alpha': 1668.1005372000557}	611.236033	1668.100537
0	[1, 2, 3, 4, 5]	{'ridge__alpha': 0.001}	612.352191	0.001000
1	[1, 2, 3, 4, 5]	{'ridge__alpha': 0.007742636826811269}	612.352531	0.007743
2	[1, 2, 3, 4, 5]	{'ridge__alpha': 0.05994842503189409}	612.355162	0.059948

	lags	params	metric	ridge__alpha
3	[1, 2, 3, 4, 5]	{'ridge__alpha': 0.46415888336127775}	612.375445	0.464159
4	[1, 2, 3, 4, 5]	{'ridge__alpha': 3.593813663804626}	612.528081	3.593814
5	[1, 2, 3, 4, 5]	{'ridge__alpha': 27.825594022071257}	613.477722	27.825594
6	[1, 2, 3, 4, 5]	{'ridge__alpha': 215.44346900318823}	615.109317	215.443469
8	[1, 2, 3, 4, 5]	{'ridge__alpha': 12915.496650148827}	625.105850	12915.496650
9	[1, 2, 3, 4, 5]	{'ridge__alpha': 100000.0}	681.830571	100000.000000

The best results are obtained by using the lags [1, 2, 3, 23, 24, 25, 47, 48, 49] and a Ridge configuration {'alpha': 215.44}. By specifying return_best = True in the grid_search_forecaster() function, at the end of the process, the forecaster object is automatically retrained with the best configuration found and the complete dataset (train + validation).

In [32]:

```
forecaster
```

Out[32]:

```
=================
ForecasterAutoreg
=================
Regressor: Pipeline(steps=[('standardscaler', StandardScaler()),
                ('ridge', Ridge(alpha=215.44346900318823))])
Lags: [ 1  2  3 23 24 25 47 48 49]
Window size: 49
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': 215.44346900318823, 'ridge__copy_X': T
rue, 'ridge__fit_intercept': True, 'ridge__max_iter': None, 'ridge__normalize': 'depreca
ted', 'ridge__positive': False, 'ridge__random_state': None, 'ridge__solver': 'auto', 'r
idge__tol': 0.001}
Creation date: 2022-06-06 00:07:32
Last fit date: 2022-06-06 00:08:21
Skforecast version: 0.4.3
```

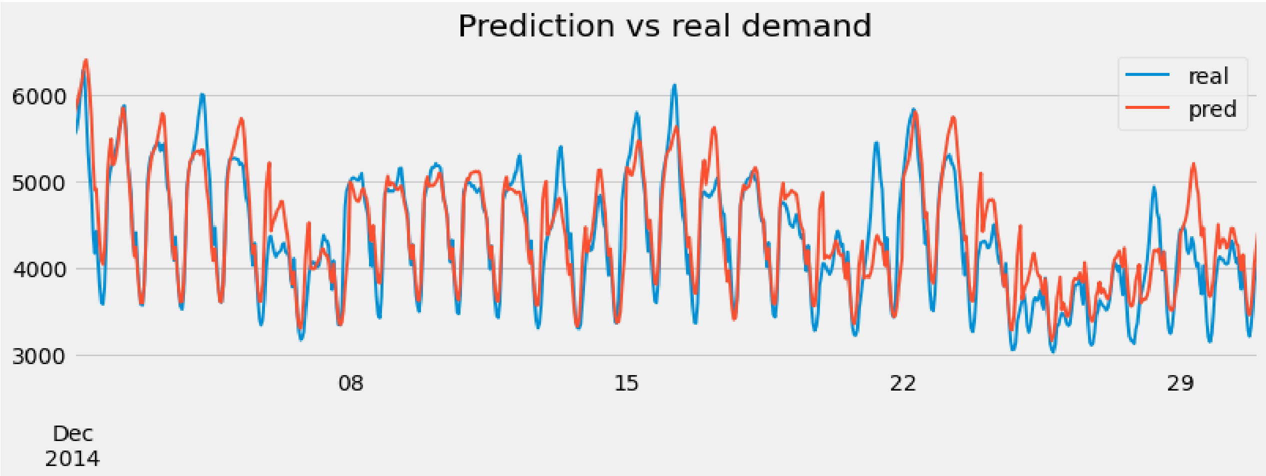
Backtest with test data

Once the best model has been identified and trained, its error in predicting the test data is calculated.

In [33]:

```
# Backtest final model
# ==============================================================================
metric, predictions = backtesting_forecaster(
                            forecaster = forecaster,
                            y          = data.Demand,
                            initial_train_size = len(data[:end_validation]),
                            fixed_train_size   = False,
                            steps      = 24,
```

```
                            metric      = 'mean_absolute_error',
                            refit       = False,
                            verbose     = False
                      )

fig, ax = plt.subplots(figsize=(12, 3.5))
data.loc[predictions.index, 'Demand'].plot(linewidth=2, label='real', ax=ax)
predictions.plot(linewidth=2, label='prediction', ax=ax)
ax.set_title('Prediction vs real demand')
ax.legend();
```



In [34]:

```
# Error backtest
# ==============================================================================
print(f'Backtest error: {metric}')
```

```
Backtest error: 251.93996461684006
```

After optimizing lags and hyperparameters, I observed that the prediction error was reduced from 289.5 to 251.9.

Prediction intervals

A prediction interval defines the interval within which the true value of "y" can be expected to be found with a given probability. Data Scientist list multiple ways to estimate prediction intervals, most of which require that the residuals (errors) of the model are normally distributed. When this property cannot be assumed, bootstrapping can be resorted to, which only assumes that the residuals are uncorrelated. This is the method used in the Skforecast library for the ForecasterAutoreg and ForecasterAutoregCustom type models.

In [35]:

```
# Backtest with test data and prediction intervals
# ==============================================================================
metric, predictions = backtesting_forecaster(
                            forecaster = forecaster,
                            y          = data.Demand,
                            initial_train_size = len(data.Demand[:end_validation]),
                            fixed_train_size   = False,
                            steps      = 24,
                            metric     = 'mean_absolute_error',
```

```
                                interval            = [10, 90],
                                n_boot              = 500,
                                in_sample_residuals = True,
                                verbose             = False
                           )

print('Backtesting metric:', metric)
predictions.head(10)
```

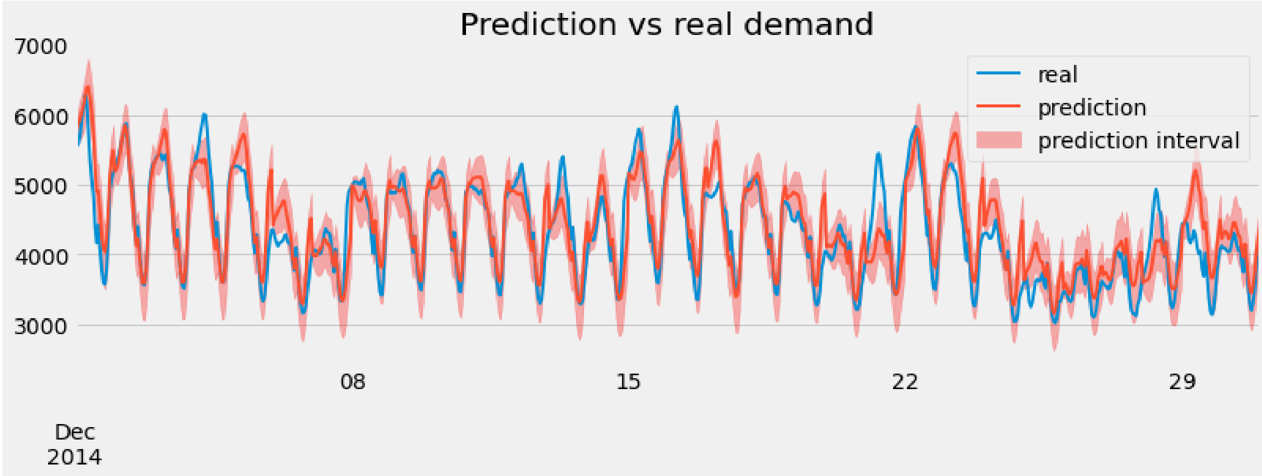
```
Backtesting metric: 251.93996461684006
```

Out[35]:

	pred	lower_bound	upper_bound
2014-12-01 00:00:00	5727.876561	5598.911947	5849.678211
2014-12-01 01:00:00	5802.876139	5599.270390	5974.888764
2014-12-01 02:00:00	5880.047528	5619.841591	6113.916977
2014-12-01 03:00:00	5953.541637	5657.437015	6240.035968
2014-12-01 04:00:00	6048.740321	5697.793533	6343.068708
2014-12-01 05:00:00	6137.445368	5765.494199	6452.822013
2014-12-01 06:00:00	6261.838234	5883.973887	6573.446149
2014-12-01 07:00:00	6386.619471	6007.654524	6724.388773
2014-12-01 08:00:00	6402.610840	5993.715349	6803.544231
2014-12-01 09:00:00	6244.943528	5829.013106	6631.480809

In [36]:

```
# Plot
# ==============================================================================
fig, ax = plt.subplots(figsize=(12, 3.5))
data.loc[predictions.index, 'Demand'].plot(linewidth=2, label='real', ax=ax)
predictions.iloc[:, 0].plot(linewidth=2, label='prediction', ax=ax)
ax.set_title('Prediction vs real demand')
ax.fill_between(
    predictions.index,
    predictions.iloc[:, 1],
    predictions.iloc[:, 2],
    alpha = 0.3,
    color = 'red',
    label = 'prediction interval'
)
ax.legend();
```



In [37]:

```
# Predicted interval coverage
# ==============================================================================
inside_interval = np.where(
                     (data.loc[end_validation:, 'Demand'] >= predictions["lower_bound"]
                     (data.loc[end_validation:, 'Demand'] <= predictions["upper_bound"]
                     True,
                     False
                  )

coverage = inside_interval.mean()
print(f"Predicted interval coverage: {round(100*coverage, 2)} %")
```

```
Predicted interval coverage: 79.03 %
```

The predicted interval has a lower coverage than expected (80%). It may be due to the marked high error made by the model for days 21, 24, and 25. These days are within the Christmas holiday period, usually characterized by a different consumption behavior than the rest of the month.

In [ ]: