Bonus Chapter 16

Analyzing Time Series in the Frequency Domain

In the previous chapters, you engineered cyclical features for **machine learning** (**ML**) (*Chapter 11*) and modeled seasonality using **harmonics** in **UCM** with `freq_seasonal` and in **TBATS** with `seasonal_length` (*Chapter 14*). You also used calendar-based **Fourier terms** as powerful exogenous variables in SARIMAX (*Chapter 15*). In all of these cases, you were leveraging principles from the **frequency domain**.

These techniques are part of a broader field called **signal processing**. You encountered some of its principles before in *Chapter 6*. When handling missing data using interpolation methods such as spline and polynomial, you were using signal-processing-related interpolation techniques that run behind the scenes via the SciPy library.

In this chapter, you will explore analyzing time series not only in the familiar **time domain** (values indexed by time) but also in the **frequency domain**, where a series is represented by its signal strength across different frequencies. You will learn how to extract these data-driven signal features to improve the performance of both statistical models, such as **SARIMAX**, and more advanced ML algorithms, such as **LGBMRegressor**.

In this chapter, the following recipes will be covered:

- Visualizing cycles with a periodogram
- Forecasting by extrapolating cycles with FFT
- Boosting ML models with signal features

Technical requirements

In this chapter, you will be using the *Hourly Energy Consumption* data from Kaggle (`https://www.kaggle.com/datasets/robikscube/hourly-energy-consumption`). Throughout the chapter, you will be working with the same energy dataset to observe how the different techniques compare.

The ZIP folder contains 13 CSV files, which are available in the GitHub repository for this book and can be found here: `https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch16`.

All recipes in this chapter will use one of these files (`AEP_hourly.csv`), but feel free to explore the other ones. The files represent hourly energy consumption measured in **megawatts** (**MW**).

Start by loading the data as a pandas DataFrame and preparing the data for the recipes. You should also load the shared libraries used throughout the chapter:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

plt.style.use('grayscale')
Read the AEP_hourly.csv file:
folder = Path('../../datasets/Ch17/')
file = folder.joinpath('AEP_hourly.csv')
df = pd.read_csv(file, index_col='Datetime', parse_dates=True)
df.shape
>>
(121273, 1)
```

The raw data needs to be cleaned: the index is not in order and has both duplicate timestamps and gaps in the hourly readings. Let's sort the index, then use `resample('H')` to enforce a complete hourly timeline. This step fills any gaps, which is why the record count increases slightly. We'll use `.last()` to aggregate any duplicates within an hour and, finally, use `.ffill()` to populate the newly created gaps:

```
df.sort_index(inplace=True)
df = df.resample('H').last()
df.columns = ['y']
df.ffill(inplace=True)
print(f"Final dataset shape: {df.shape}")
print(f"Data frequency: {df.index.freq}")
print(f"Any missing values: {df.isnull().any().any()}")
>>
Final dataset shape: (121296, 1)
Data frequency: <Hour>
Any missing values: False
```

You should have a DataFrame with `121296` records of hourly energy consumption from October 2004 to August 2018 (around 14 years).

Plot the data for visual inspection:

```
df.plot()
```

The resulting plot displays the hourly energy consumption trends, seasonality, and variations across the entire dataset:

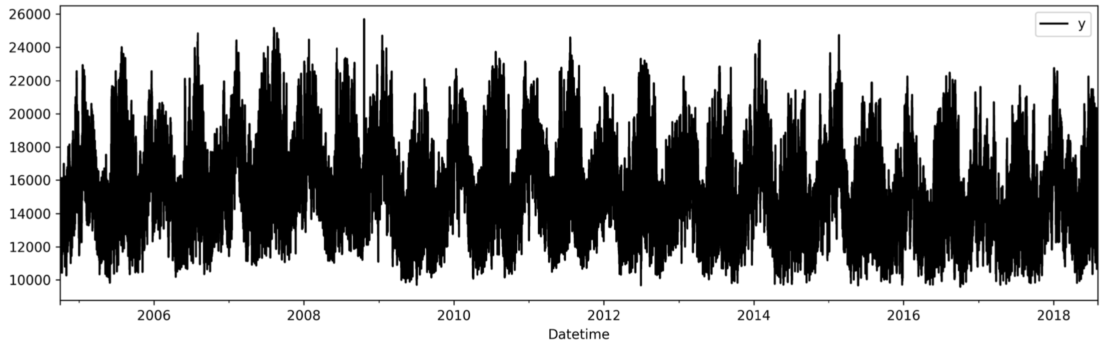

Figure 16.1: Plot of the hourly American Electric Power (AEP) energy consumption from October 2004 to August 2018

In each recipe, we will be using a different library, and to avoid version and dependency conflicts and mismatches, it is highly recommended that you create a separate Python environment for each recipe, especially when installing new libraries. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup* recipe from *Chapter 0*. The chapter covers two methods for creating Python virtual environments: via `conda` or `venv`.

Visualizing cycles with a periodogram

In this recipe, we will convert our series from the **time domain** (values ordered by datetime) into the **frequency domain**, where we measure signal strength versus frequency. The first step is to estimate the spectrum using SciPy's **periodogram**, which applies optional detrending and windowing, computes a discrete Fourier transform via **Fast Fourier Transform** (**FFT**), and returns arrays of frequencies and their corresponding spectral values. In essence, this is like decomposing our time-series dataset in the frequency domain, and the result is a collection of frequencies and their associated magnitude (the signal's spectral content). We then detect prominent peaks in this spectrum to identify the most significant repeating cycles (patterns). In time-series terms, these peaks correspond to seasonal patterns that can inform which Fourier terms to include in forecasting models.

SciPy's **periodogram** supports optional detrending (e.g., mean removal with `detrend='constant'`), a tapering window (e.g., `window='hann'`) to reduce leakage, FFT-based transformation, and scaling either as **power spectral density** (`scaling='density'`, `units^2/Hz`) or as **power spectrum** (`scaling='spectrum'`, `units^2`). In this recipe, we'll use `'spectrum'` since relative peak heights are sufficient for finding dominant cycles; the peak locations are unchanged by this choice. The function returns two arrays: frequencies and their spectral values.

To extract the key cycles, we use SciPy's `find_peaks`, which selects local maxima in the spectrum based on defined constraints. A peak can be filtered by an absolute height (minimum amplitude) or by prominence, which measures how much a peak stands out above its surrounding valleys. We will use a minimum prominence and optionally a minimum distance between the peaks to avoid counting near-duplicates around the same seasonal frequency.

This approach will help us confirm known periodicities (*daily*, *weekly*, and *yearly*) and uncover additional cycles present in the data, providing concrete targets when building Fourier terms for forecasting (e.g., in *Chapter 15*, where we relied on domain knowledge and intuition to construct Fourier terms for the SARIMAX model).

Getting ready

You can install statsmodels using `pip`:

```
pip install statsmodels scipy
```

You can also install it using `conda`:

```
conda install -c conda-forge statsmodels scipy
```

How to do it...

Make sure you run the code in the *Technical requirements* section to load the dataset. You should have a `df` DataFrame for the hourly energy dataset:

1. Start by importing the functions needed for the recipe:

```
from scipy.signal import periodogram, find_peaks
```

2. Prepare the target series:

```
y = df['y'].astype(float)
```

3. Compute the periodogram, which will return the frequencies and their associated magnitudes. We will perform mean removal, specify Hann as the window (ideal for reducing leakage), and use power spectrum scaling:

```
fs = 1.0 # 1 sample per hour
freqs, power = periodogram(
    y,
    fs=fs,
    window='hann',
    detrend='constant',
    scaling='spectrum',
    return_onesided=True
)
```

Since our data is sampled every hour, we set our sampling frequency to `fs=1.0` (one sample per hour). This establishes our frequency unit, where the relationship between a cycle's **period** (how long it takes) and its **frequency** (how often it happens) is as follows:

$$\text{Period} = \frac{1}{\text{Frequency}}$$

For example, a frequency of 1.0 Hz corresponds to a 1-hour period ($1/1.0 = 1$), while 0.1 Hz corresponds to a 10-hour period ($1/0.1 = 10$). For our daily cycles, we expect to see a peak around frequency ≈ 0.042 Hz ($1/24 \approx 0.042$).

If you are working with data at a different frequency (e.g., every minute), you will need to adjust `fs` accordingly.

4. We want to only work with positive frequencies since negative frequencies are just mirror images of real-valued signals:

```
valid = freqs > 0
f_pos = freqs[valid]
P_pos = power[valid]
```

5. Convert the frequency to a period:

```
period_hours = 1.0 / f_pos
```

Low frequencies correspond to long cycles and high frequencies to short cycles.

6. The `find_peaks` method allows you to specify a constraint based on the height or prominence of the peaks. We'll use the `prominence` argument, instead of absolute height, because it is more robust to varying signal amplitudes. We'll specify a threshold (20% of the maximum peak) to ensure we capture significant peaks (while filtering out noise) and a small minimum distance:

```
# Use a prominence threshold to find significant peaks
peaks, props = find_peaks(
    P_pos,
    prominence=np.max(P_pos) * 0.20,
    distance=2
)
```

7. Create a DataFrame for easy sorting and inspection:

```
spec_df = pd.DataFrame({
    'period_hours': period_hours[peaks],
    'power': P_pos[peaks],
    'prominence': props['prominences']
}).sort_values('power', ascending=False)
```

8. Find the most prominent cycles:

```
print("Most prominent cycles found:")
top_peaks = spec_df.head(5).copy()
top_peaks['days'] = top_peaks['period_hours'] / 24.0
print(top_peaks)
>>
Most prominent cycles found:
   period_hours         power    prominence   days
2          24.0  1.418589e+06  1.418589e+06    1.0
0        4332.0  1.257695e+06  1.256908e+06  180.5
1         168.0  4.092836e+05  4.092819e+05    7.0
3          12.0  3.071967e+05  3.071967e+05    0.5
```

   You can observe that the strongest cycle is daily (24 hours), followed by semi-annual (4,332 hours or 180.5 days), weekly (168 hours), and semi-daily (12 hours) cycles.

9. Define a helper function to assign descriptive labels to detected cycles based on their period length:

```
def get_period_label(period_hours):
    """Assigns a descriptive label to a period in hours."""
    if np.isclose(period_hours, 24, atol=1):
        return 'Daily (24h)'
    if np.isclose(period_hours, 12, atol=1):
        return 'Semi-daily (12h)'
    if np.isclose(period_hours, 168, atol=5):
        return 'Weekly (168h)'
    if np.isclose(period_hours, 4380, atol=360): # ~6 months
        return 'Semi-annual (~6mo)'
    return f'{period_hours:.0f}h' # Default label for other peaks

top_peaks['labels'] = top_peaks['period_hours'].apply(get_period_label)
```

   You'll notice that our detected peak was at 4,332 hours, while our function checks for 4,380 (the average hours in 6months). The large tolerance (atol=360) easily handles this difference.

10. Create a periodogram visualization with annotated peaks to highlight the dominant cycles in the energy load data:

```
color_map = {
    'Daily (24h)': 'red',
    'Weekly (168h)': 'green',
    'Semi-annual (~6mo)': 'purple',
    'Semi-daily (12h)': 'orange'
}

top_peaks['color'] = top_peaks['labels'].map(color_map)

plt.figure(figsize=(10,5))
plt.plot(period_hours, P_pos, lw=2)
plt.xscale('log')
plt.xlabel('Period (hours)')
plt.ylabel('Power (spectrum)')
plt.title('Periodogram of Hourly Energy Load')

for index, row in top_peaks.iterrows():
    # Draw the vertical line
    plt.axvline(
        x=row['period_hours'],
        color=row['color'],
        ls='--',
        linewidth=2,
        alpha=0.7
    )
    # Add the text label
    plt.text(
        x=row['period_hours'],
        y=plt.ylim()[1] * 0.9,
        s=row['labels'],
        rotation=90,
        va='top',
        ha='right',
        fontsize=10
    )

plt.tight_layout()
plt.show()
```

This should produce the following plot:

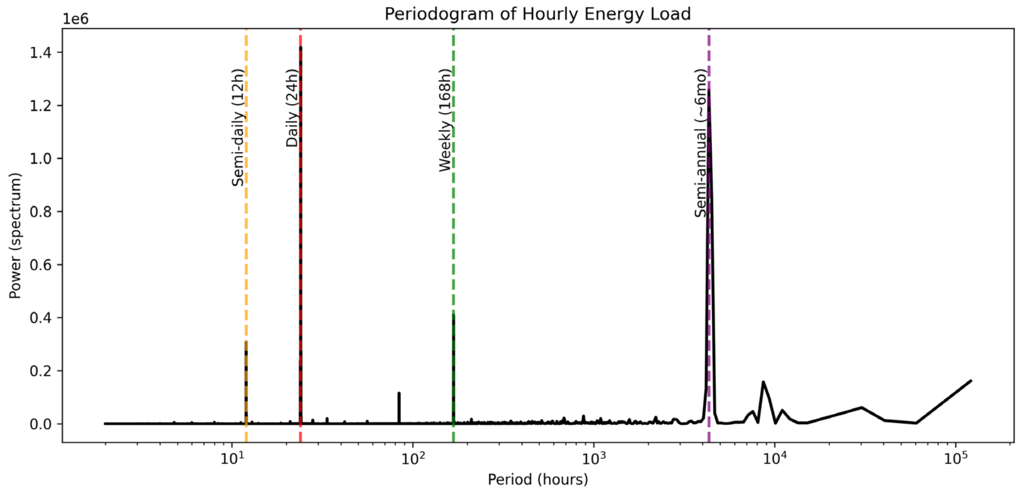

Figure 16.2: Periodogram with annotations highlighting critical peaks (semi-daily, daily, weekly, and semi-annual)

How it works...

In *Chapter 15*, we trained a SARIMAX model that used exogenous variables based on Fourier terms. The Fourier terms were based on general knowledge of the hourly energy consumption dataset. In that recipe, we created the following:

```
day = 24
week = day*7
year = day*365
fourier_daily  = Fourier(period=day, order=4)
fourier_weekly = Fourier(period=week, order=6)
fourier_annual = Fourier(period=year, order=8)
```

Let's use the same code, but this time we will create more informed Fourier terms based on the dominant frequencies identified with a periodogram (daily, weekly, and semi-annual). **Data-driven seasonal periods** (spectral peaks) reveal actual periodicities, so the Fourier terms target what the data exhibits rather than only assumed calendars. In other words, our model isn't guessing cycles based on calendars; instead, it is actually tuned to the rhythms present in the data itself:

```
from statsmodels.tsa.deterministic import Fourier
from statsmodels.tsa.statespace.sarimax import SARIMAX

day = 24
week = 168
semi_annual = 4332 # As detected by periodogram
```

```
fourier_daily  = Fourier(period=day, order=4)
fourier_weekly = Fourier(period=week, order=6)
fourier_semi_annual = Fourier(period=semi_annual, order=8)
```

We're keeping the `order` for each Fourier term the same as in *Chapter 15* to make our comparison fair. Note, though, that the order is another hyperparameter you can tune to control how complex each seasonal pattern can be:

```
idx = df.index
X_list = [
    fourier_daily.in_sample(idx),
    fourier_weekly.in_sample(idx),
    fourier_semi_annual.in_sample(idx),
]

X = pd.concat(X_list, axis=1)

train = df.iloc[:-week]
test = df.iloc[-week:]
exog_train = X.iloc[:-week]
exog_test = X.iloc[-week:]

mod = SARIMAX(
    train['y'],
    exog=exog_train,
    order=(1, 0, 1)
)
res = mod.fit(disp=False)
```

Overall, we're keeping the model structure from *Chapter 15*, but with one key upgrade. Instead of using a calendar-based annual cycle, we are swapping in the semi-annual cycle (4,332 hours) that the periodogram identified. This way, our data, not the calendar, guides our model's seasonality.

You might have noticed that we are generating Fourier terms over the entire dataset's index (`idx = df.index`) before splitting it into train and test sets. This approach is safe because Fourier terms are **deterministic**; they depend only on the timestamp, not on the target variable's values.

However, be careful. For other common features, such as lags or rolling averages, which depend on past values of the target, this shortcut can *introduce data leakage*. For those features, the golden rule is to *always split your data first*, then generate features separately for your training set (e.g., using `.in_sample(train.index)`) and your test set (e.g., using `.out_of_sample(test.index)`).

Let's generate our forecast:

```
fc = res.get_forecast(steps=len(test), exog=exog_test)
```

Now, we can calculate our MAE and RMSE scores and compare them with our previous attempt:

```
from statsmodels.tools.eval_measures import rmse, meanabs as mae

yhat = fc.predicted_mean

sarimax_mae = mae(test['y'], yhat)
sarimax_rmse = rmse(test['y'], yhat)
print(f'SARIMAX MAE: {sarimax_mae}')
print(f'SARIMAX RMSE: {sarimax_rmse}')
>>
SARIMAX MAE: 875.3165512597519
SARIMAX RMSE: 1142.8519523542466
```

We got an improvement in both the MAE (875.32 versus 905.38) and the RMSE (1142.85 versus 1180.81) scores with this small adjustment that we made (Fourier terms).

There's more…

In addition to SciPy's periodogram method, the **Welch method** can be used for smoother spectrum estimates, which can be great when working with noisy data. The periodogram computes one windowed FFT over the entire set, which can create a high-variance power estimate. Welch splits the data into overlapping segments, applies a window to each, computes their periodograms, and averages them, reducing variance at the cost of frequency resolution. Both the Welch and periodogram methods support windows, detrending, and scaling (density or spectrum).

We will use a similar setup to what we did in the recipe *Visualizing cycles with a periodogram*:

```
from scipy.signal import welch, find_peaks
fs = 1.0  # samples per hour
nperseg = 24 * 30 # 720 hours (30 days)

# Welch (use same preprocessing concepts as before)
f_welch, power = welch(
    y.values,  # Convert pandas Series to NumPy array
    fs=fs,
    window='hann',
    nperseg=nperseg,
    noverlap=nperseg // 2,
    detrend='constant',
    return_onesided=True,
    scaling='spectrum',
    average='mean'
)

# Positive frequencies only
valid = f_welch > 0
f_w = f_welch[valid]
P_w = power[valid]
period_w = 1.0 / f_w

# Peak picking in PSD (frequency domain)
# Start with 10% of max prominence and a small distance; tune per dataset
peaks, props = find_peaks(
    P_w,
    prominence=np.max(P_w) * 0.10,
    distance=2
)
# Build table of detected cycles
spec_w = pd.DataFrame({
    'frequency_hz': f_w[peaks],
    'period_hours': period_w[peaks],
    'power': P_w[peaks],
    'prominence': props['prominences']
}).sort_values('power', ascending=False)

print("Welch: most prominent cycles")
top_w = spec_w.head(5).copy()
top_w['days'] = top_w['period_hours'] / 24.0
print(top_w)
>>
Welch: most prominent cycles
   frequency_hz  period_hours         power    prominence  days
1      0.041667          24.0  2.047082e+06  2.040968e+06   1.0
0      0.005556         180.0  5.228443e+05  2.419046e+05   7.5
2      0.083333          12.0  3.678194e+05  3.671850e+05   0.5
```

Notice `nperseg=24*30`, which means that with our hourly data, each segment spans about one month. If you use longer segments, you'll get more precise estimates of each cycle's frequency, but you'll have fewer segments to average, which means more variability in the spectrum. Shorter segments make the spectrum smoother, but you lose some detail about individual cycles. With shorter segments, it means FFT looks at a smaller time window, and we end up with more batches to average, but eventually this would reduce variance and make the curve smoother.

`noverlap` specifies the number of samples by which consecutive segments overlap. Setting `noverlap=nperseg // 2` creates 50% overlap between consecutive segments. This increases the effective number of segments for averaging, which can reduce variance for improved reliability.

Visualize Welch's PSD with the same peak annotation style. Since we've already established this plotting pattern, use the `plot_spectrum_with_peaks` utility function:

```
from utils import plot_spectrum_with_peaks

plot_spectrum_with_peaks(
    period_w,
    P_w,
    top_w,
    title='Welch PSD with Detected Cycles',
    xlabel='Period (hours)',
    ylabel='Power (spectrum)'
)
```

This should produce the following:

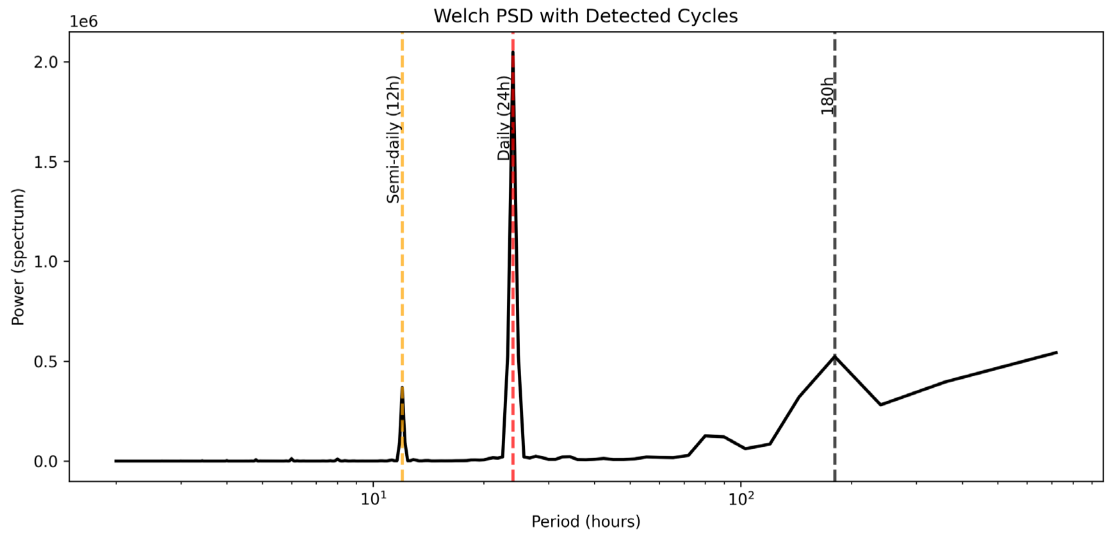

Figure 16.3: Welch with annotations highlighting critical peaks (semi-daily, daily, and 180 hours (~7.5 days))

Notice how Welch's method found a peak at 180 hours (~7.5 days) instead of a 168-hour weekly cycle. This can happen due to noisy data or when cycles aren't perfectly regular. This illustrates how different spectral estimation methods can reveal slightly different features in our data. For example, we can use a periodogram to explore fine spectral structure, and then use Welch to stabilize spectra for noisy data.

See also

- SciPy offers a lot of options for signal processing. For a comprehensive list of capabilities, you can visit their main documentation page at `https://docs.scipy.org/doc/scipy/reference/signal.html`.

Forecasting by extrapolating cycles with FFT

In this recipe, you will leverage Darts' FFT implementation for forecasting. Since FFTs are great tools for finding repeating cycles or patterns in time series (as we explored in the first recipe), Darts' FFT is ideal for highly **seasonal datasets** with multiple seasonalities. The FFT algorithm on its own cannot capture trends, hence why in the previous recipe we had to detrend our data (e.g., mean removal). Fortunately, Darts provides a `trend` parameter (which allows you to specify `"poly"` for polynomial or `"exp"` for exponential to model more complex trends).

At its core, the Darts FFT implementation follows similar steps to what we performed in the first recipe using SciPy's periodogram:

- **Detrending:** The model first fits a **polynomial** or **exponential** function to model the trend. It then removes the trend component by subtracting the fitted values (the trend predictions from the polynomial/exponential function) from the original series.
- **FFT transformation:** The detrended series is converted from the time domain to the frequency domain using NumPy's `fft`. The resulting algorithm then identifies the frequencies with the largest amplitudes (the strongest periodic signals).
- **Frequency selection**: The `nr_freqs_to_keep` parameter determines the number of top frequencies to keep; all others are discarded. For example, `nr_freqs_to_keep=20` will keep the top 20 frequencies by amplitude.
- **Forecasting**: The model forecasts the trend (using the fitted polynomial/exponential function) and the seasonal components (by extrapolating the selected frequencies into the future). Finally, it combines the forecasts (trend and season) to produce the final prediction.

FFT forecasting is particularly powerful for time series with multiple overlapping seasonal patterns. Unlike traditional forecasting methods that model trend and seasonality separately, FFT decomposes your data into frequency components and extrapolates the dominant patterns forward. This approach excels with highly seasonal datasets such as energy consumption, where daily, weekly, and longer-term cycles interact.

Getting ready

You can install Darts using `pip`:

```
pip install darts
```

How to do it...

Make sure you run the code in the *Technical requirements* section to load the dataset. You should have a `df` DataFrame for the hourly energy dataset:

1. Start by importing the additional classes and functions required for this recipe:

```
from darts.models import FFT
from darts import TimeSeries
from darts.metrics import mae, rmse
```

2. We have used Darts on numerous occasions, and by now we should be familiar with the format that Darts expects in order to convert our DataFrame into a Darts `TimeSeries` object:

```
darts_df = df.reset_index().copy()
ts = TimeSeries.from_dataframe(darts_df,
                               time_col='Datetime',
                               value_cols='y',
                               freq='h')
```

3. Split the data into training and test sets:

```
forecast_horizon=168

d_train = ts[:-forecast_horizon]
d_test = ts[-forecast_horizon:]
```

4. Instantiate and configure the FFT model:

```
fft_model = FFT(required_matches=None,
                nr_freqs_to_keep=20,
                trend='poly',
                trend_poly_degree=3)
```

Let's unpack this:

- `required_matches` is used for cropping (trimming) the training set and takes a set of strings. Each string needs to be a valid attribute of a `pandas.Timestamp` object. Common values include `"month"`, `"day"`, `"weekday"`, `"hour"`, and `"minute"`. For example, you can specify `required_matches={"weekday"}` and the model will crop (trim) the training data so the first timestamp matches the forecast start's weekday (e.g., Monday). If you set it to `None` (the default), then Darts will decide; it will try to find the best alignment based on the series' seasonality, and if it doesn't find a strong pattern, it simply uses the entire training series.
- `nr_freqs_to_keep` determines the top frequencies to keep by amplitude.
- `trend` specifies how the trend should be modeled, whether as a polynomial or exponential function. If `"poly"` is specified, you will need to provide `trend_poly_degree` (the polynomial degree), which defaults to `3`.

5. Fit the model and create the forecast. You will observe that the fitting process is pretty fast with FFT:

```
fft_model.fit(d_train)
>>
No matching timestamp could be found, returning original TimeSeries.
```

If you see a message such as the one displayed, it indicates that no cropping was performed. This simply means the model used the full training series to fit. Cropping is only done if a strong seasonal alignment is detected, which isn't always possible depending on your train/test split:

```
fft_pred = fft_model.predict(len(d_test))
```

6. Plot the forecast against the test set and compute the MAE and RMSE scores:

```
d_test.plot(label="actual", ls='--')
fft_pred.plot(label='forecast')
plt.title('Darts Fast Fourier Transform Forecasting Model')
plt.show()
```

This should produce the following plot:

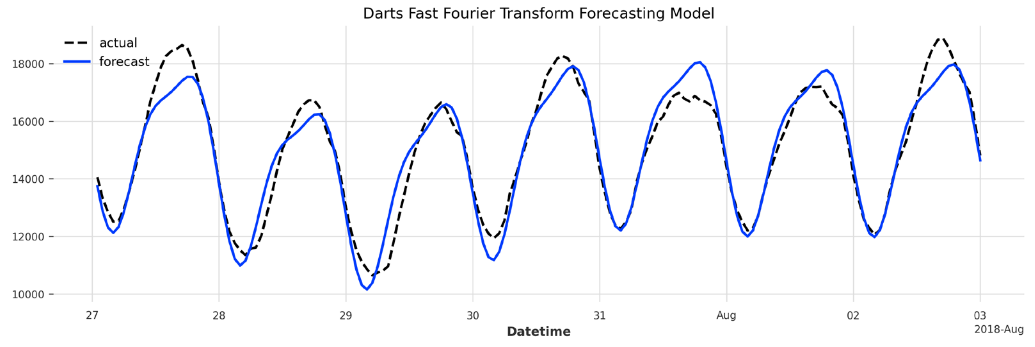

Figure 16.4: Darts' FFT forecasting versus actual

The plot shows the actual energy consumption (dashed line) versus the FFT forecast (solid line) over the 168-hour test period. Look for how well the forecast captures the daily and weekly cycles (FFT should closely follow the ups and downs of the actual data if the seasonal patterns are strong).

Compute the evaluation metrics:

```
print("Darts FFT MAE:", mae(fft_pred, d_test))
print("Darts FFT RMSE:", rmse(fft_pred, d_test))
>>
Darts FFT MAE: 442.4184797364066
Darts FFT RMSE: 588.4922154503801
```

Try varying `nr_freqs_to_keep` (e.g., `5`, `10`, `30`) to observe how the forecast changes. FFT-based methods perform best on datasets with strong, stable seasonal cycles. Choosing the right number of frequencies is a balance: too few and you'll smooth out noise but miss important seasonal patterns, and too many and you'll capture more details but risk overfitting and capturing noise. Start with 10–30 frequencies for hourly data and adjust based on your validation results.

How it works...

FFT requires the input to be stationary (no trend or changing mean), so the model first fits a cubic polynomial function (`trend_poly_degree=3`) to the trend and removes it. Once detrended, FFT is applied to transform the remaining data into the frequency domain and identifies all possible seasonal frequencies. Since we specified `nr_freqs_to_keep=20`, the model retains the strongest 20 seasonal frequencies to build the seasonal model. Note that the number of frequencies can influence the model's output, which is something you should experiment with. The final forecast combines the extrapolated trend and the seasonal cycles.

There's more...

The FFT model is fast and transparent for strongly periodic signals but doesn't expose frequencies or handle complex exogenous effects. An alternate approach is to use Fourier covariates with any of Darts' covariate-capable models, such as AutoARIMA, AutoETS, AutoTheta, XGBModel, TCNModel, NHiTSModel, and TFTModel, to name a few.

Let's explore **AutoETS** (which stands for **automatic exponential smoothing forecaster**) first without covariates and again with Fourier terms as future covariates. To learn more about future, past, and static covariates, refer to *Chapter 12*.

You may need to install the StatsForecast library, which is used under the hood in Darts. You can install it using `pip`:

```
pip install statsforecast
```

First, let's train our AutoETS and evaluate our forecast:

```
from darts.models import AutoETS
ets_model = AutoETS()
ets_model.fit(d_train)
ets_pred = ets_model.predict(len(d_test))

print("AutoETS MAE:", mae(ets_pred, d_test))
print("AutoETS RMSE:", rmse(ets_pred, d_test))
>>
AutoETS MAE: 1870.717835048086
AutoETS RMSE: 2170.773033209197
```

AutoETS is pretty fast, and this makes it a great model to explore by adding our Fourier as covariates. Through periodogram analysis (covered in the previous recipe), we identified three dominant seasonal patterns in our energy data: daily (24 hours), weekly (168 hours), and semi-annual (4,332 hours) cycles. You will use the same process to create the Fourier terms from the first recipe of this chapter and then pass them as future covariates to AutoETS.

First, let's create our Fourier terms, combine them into a single DataFrame, and then split the resulting DataFrame into train and test sets:

```
from statsmodels.tsa.deterministic import Fourier

day = 24
week=168
semi_annual = 4332
forecast_horizon=168

fourier_daily  = Fourier(period=day, order=4)
fourier_weekly = Fourier(period=week, order=6)
fourier_semi_annual = Fourier(period=semi_annual, order=8)

idx = df.index
X_list = [
    fourier_daily.in_sample(idx),
    fourier_weekly.in_sample(idx),
    fourier_semi_annual.in_sample(idx)
]

X = pd.concat(X_list, axis=1)

ts_ft = TimeSeries.from_dataframe(X.reset_index(),
                                  time_col='Datetime',
                                  value_cols=X.columns.tolist(),
                                  freq='h')

ft_cov_train = ts_ft[:-forecast_horizon]
ft_cov_test = ts_ft[-forecast_horizon:]
```

Instantiate the AutoETS, fit the model by passing `ft_cov_train` as future covariates, and evaluate the forecast:

```
ft_ets_model = AutoETS()
ft_ets_model.fit(d_train, future_covariates=ft_cov_train)
```

```
ft_ets_pred = ft_ets_model.predict(len(d_test), future_covariates=ts_ft)
print("AutoETS with Fourier Covariates MAE:", mae(ft_ets_pred, d_test))
print("AutoETS with Fourier Covariates RMSE:", rmse(ft_ets_pred, d_test))
>>
AutoETS with Fourier Covariates MAE: 891.1680381441104
AutoETS with Fourier Covariates RMSE: 1058.6425007597963
```

When using **future covariates** for forecasting in Darts, make sure your covariate `TimeSeries` covers all timestamps in your forecast window. If you get alignment errors (`future_covariates=ts_ft`), just pass the full covariate (`ts_ft`) series and Darts will grab what it needs automatically. This avoids common pitfalls with test/train splits and keeps your code running smoothly.

You will notice a few things from this experiment:

- AutoETS is pretty fast even with the Fourier covariates.
- The results show significant improvement by incorporating Fourier terms.
- The main FFT model is convenient because it finds the frequencies for you. The covariate approach gives you more control but requires you to identify the key seasonal periods first (as we did in the first recipe).

See also

- To learn more about FFT in Darts, you can check out the official documentation at `https://unit8co.github.io/darts/examples/03-FFT-examples.html` and `https://unit8co.github.io/darts/generated_api/darts.models.forecasting.fft.html`

Boosting ML models with signal features

In the previous recipes, we demonstrated the power of signal features by adding Fourier terms to traditional forecasting models such as SARIMAX and AutoETS, leading to significant accuracy gains. We saw how creating features that explicitly describe the dominant cycles in a time series helps these models make more accurate predictions.

In this recipe, we will extend this concept and apply the same Fourier feature engineering technique to a new class of models: gradient boosting machines. The goal is to demonstrate the universal utility of these signal features and show how they can boost the performance of even sophisticated ML algorithms. This approach bridges the gap between classical signal analysis and modern ML-based forecasting.

For this task, you will be introduced to **Light Gradient Boosting Machine** (**LightGBM**). In *Chapter 11*, you learned about gradient boosting models such as XGBoost. LightGBM is another high-performance gradient boosting framework renowned for its speed, efficiency, and low memory usage, making it a popular choice for large datasets. We will use its regressor, **LGBMRegressor**, within the familiar **sktime** framework (we used sktime extensively in *Chapter 11*).

You will build two models: a baseline LightGBM forecaster that relies only on past values and an enhanced model boosted with our Fourier terms. By comparing their performance, you will see firsthand how signal features can unlock a new level of accuracy in ML forecasting.

Getting ready

You can install LightGBM and sktime using `pip`:

```
pip install "lightgbm[scikit-learn]" sktime
```

For a refresher on sktime and the different forecasting strategies (*recursive*, *multioutput*, and *direct*), refer to *Chapter 11*.

How to do it...

Make sure you run the code in the *Technical requirements* section to load the dataset. You should have a `df` DataFrame for the hourly energy dataset.

Baseline LGBMRegressor

1. Start by importing the additional classes and functions that are necessary for the recipe:

```
from sktime.forecasting.compose import make_reduction
from sktime.forecasting.model_selection import temporal_train_test_split
from lightgbm import LGBMRegressor
from sktime.performance_metrics.forecasting import (
    mean_absolute_error as mae,
    mean_squared_error as mse
)
from sktime.forecasting.base import ForecastingHorizon
```

2. Split the data into train and test sets:

```
y = df['y']
y_train, y_test = temporal_train_test_split(y, test_size=168)
```

3. Instantiate LGBMRegressor. We will set `random_state=42` for reproducibility:

```
lgbm_regressor = LGBMRegressor(random_state=42)
```

4. Use `make_reduction` to convert our time series to be suitable for ML (tabular format):

```
# window_length is the number of past observations used to forecast
forecaster = make_reduction(lgbm_regressor,
                            strategy="recursive",
                            window_length=168)
```

5. Fit the model, make a forecast, and evaluate the results:

```
fh = ForecastingHorizon(y_test.index, is_relative=False)
forecaster.fit(y_train)
y_pred = forecaster.predict(fh=fh)

lgbm_mae = mae(y_true=y_test, y_pred=y_pred)
lgbm_rmse = np.sqrt(mse(y_true=y_test, y_pred=y_pred))

print("LightGBM MAE:", lgbm_mae)
print("LightGBM RMSE:", lgbm_rmse)
>>
LightGBM MAE: 884.0687825098901
LightGBM RMSE: 1113.528153864195
```

6. Plot the forecast versus actual values for visual evaluation:

```
y_test.plot(label='actual', style='--')
y_pred.plot(label='forecast')
plt.title("LightGBM Forecast vs Actual")
plt.legend()
plt.show()
```

The visualization below compares the model's predictions against the actual values for the 168-hour test period:

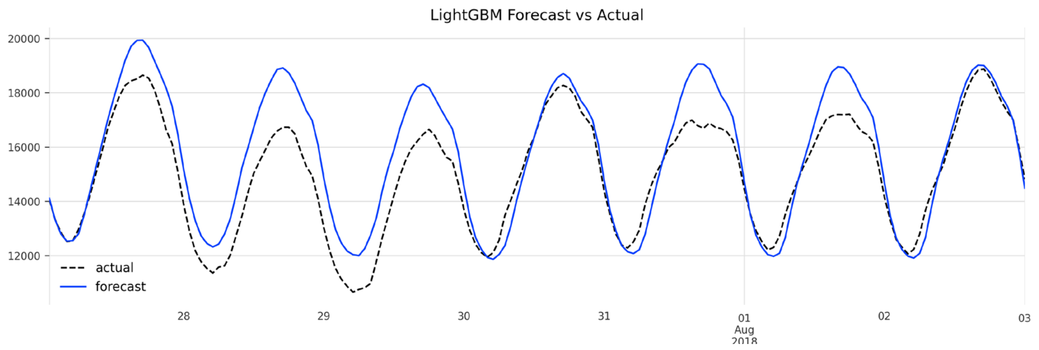

Figure 16.5: LightGBM forecast versus actual values

LGBMRegressor with Fourier terms

1. You will use the same Fourier terms established in the first recipe based on periodograms:

```
from statsmodels.tsa.deterministic import Fourier

day = 24
week=168
```

```
semi_annual = 4332

fourier_daily  = Fourier(period=day, order=4)
fourier_weekly = Fourier(period=week, order=6)
fourier_semi_annual = Fourier(period=semi_annual, order=8)

idx = df.index
X_list = [
    fourier_daily.in_sample(idx),
    fourier_weekly.in_sample(idx),
    fourier_semi_annual.in_sample(idx)
]

X = pd.concat(X_list, axis=1)
```

2. Split the X DataFrame (Fourier terms) into train and test sets:

```
X_train = X.loc[y_train.index]
X_test = X.loc[y_test.index]
```

3. Instantiate LGBMRegressor and fit the model with exogenous variables (Fourier terms):

```
# Create LightGBM regressor instance
ft_lgbm_regressor = LGBMRegressor(random_state=42)
# window_length is the number of past observations used to forecast
ft_forecaster = make_reduction(ft_lgbm_regressor,
                               strategy="recursive",
                               window_length=168)
ft_forecaster.fit(y=y_train, X=X_train)
```

4. Use the model to make a forecast and evaluate the results:

```
ft_y_pred = ft_forecaster.predict(fh=fh, X=X_test)
ft_lgbm_mae = mae(y_true=y_test, y_pred=ft_y_pred)
ft_lgbm_rmse = np.sqrt(mse(y_true=y_test, y_pred=ft_y_pred))

print("LightGBM with Fourier Terms MAE:", ft_lgbm_mae)
print("LightGBM with Fourier Terms RMSE:", ft_lgbm_rmse)
>>
LightGBM with Fourier Terms MAE: 539.5367880246538
LightGBM with Fourier Terms RMSE: 759.8291763728683
```

We notice a significant improvement just by incorporating Fourier terms without doing any hyperparameter tuning.

5. Plot the forecast versus actual values for visual inspection:

```
y_test.plot(label='actual', style='--')
ft_y_pred.plot(label='forecast')
plt.title("LightGBM with Fourier Terms Forecast vs Actual")
plt.legend()
plt.show()
```

The visual comparison below highlights the improved forecast accuracy with the addition of Fourier features:

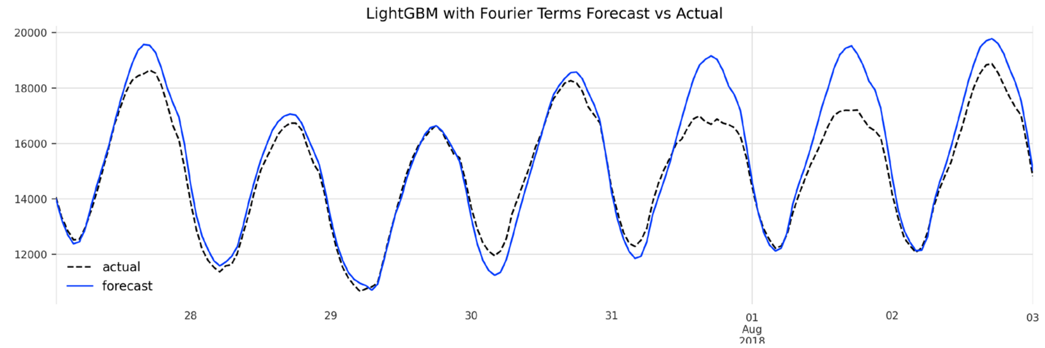

Figure 16.6: LightGBM with Fourier terms forecast versus actual values

How it works...

Gradient boosting models such as LightGBM are powerful, but their default approach relies on lagged values. By injecting explicit Fourier features, the model directly learns the cycles (daily, weekly, and semi-annual).

LightGBM is a tree-based ML model that learns patterns by iteratively building decision trees, each correcting the errors of the previous ones. In our baseline model, we used sktime's `make_reduction` with a recursive strategy, transforming the time series into a tabular format where each forecast uses the past 168 hours (one week) of energy values as features. This approach captures temporal patterns but relies solely on historical data, which might miss explicit seasonal signals.

In the *Visualizing cycles with a periodogram* recipe, we used a periodogram to identify dominant cycles in our data—daily (24 hours), weekly (168 hours), and semi-annual (4,332 hours). These cycles were converted into sine and cosine features using `statsmodels.tsa.deterministic.Fourier`, creating a set of exogenous variables that explicitly encode the rhythmic ups and downs of energy consumption. By feeding these into LightGBM as exogenous features (via `X=X_train`), we explicitly inject Fourier features so the model no longer has to "guess" the seasonal patterns from raw data alone. The Fourier terms act like a map, guiding LightGBM to focus on the most important cycles.

In this recipe, we used the **recursive** forecasting strategy. This means a single model is trained and used iteratively: it predicts one step ahead and then uses that prediction as an input feature to predict the next step, and so on. This is computationally efficient. However, as you may recall from *Chapter 11*, other strategies exist. A **direct** or **multioutput** strategy, for instance, trains a separate model for each step in the forecast horizon. While more computationally expensive, this can sometimes prevent the error accumulation that recursive strategies are prone to, especially over longer forecast horizons, and produce better results. Here is an example for a multioutput strategy:

The following code can take a long time to run (potentially over two hours) depending on factors such as your processor speed, available memory, and dataset size. It is provided primarily for illustration and learning purposes.

```python
from sklearn.multioutput import MultiOutputRegressor

fh = ForecastingHorizon(y_test.index, is_relative=False)

# Create LightGBM regressor instance
ft_mo_lgbm_regressor = LGBMRegressor(random_state=42)
# window_length is the number of past observations used to forecast
ft_mo_forecaster = make_reduction(MultiOutputRegressor(ft_mo_lgbm_regressor),
               strategy="multioutput",
               window_length=168)
ft_mo_forecaster.fit(y=y_train, X=X_train, fh=fh)
```

Recall from *Chapter 11* that some models do not support multioutput natively. So, instead, we wrap the model with the `MultiOutputRegressor` class from scikit-learn.

You can also leverage sktime's **ForecastingGridSearchCV** for hyperparameter tuning or **AutoEnsembleForecaster**, allowing you to combine the power of multiple forecasters while incorporating Fourier features (for more details, refer to *Chapter 11*). Here is an example of how you can use AutoEnsembleForecaster with Fourier terms:

```python
from sktime.forecasting.compose import AutoEnsembleForecaster
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

# 1. Create a list of forecasters
tree_based_forecasters = [
    ("lgbm", make_reduction(
        LGBMRegressor(n_estimators=10, random_state=42),
        strategy="recursive",
        window_length=168
    )),
    ("rf", make_reduction(
        RandomForestRegressor(n_estimators=10, random_state=42),
        strategy="recursive",
        window_length=168
    )),

    ("gb", make_reduction(
        GradientBoostingRegressor(n_estimators=10, random_state=42),
        strategy="recursive",
        window_length=168
    ))
]

# 2. Instantiate the AutoEnsembleForecaster with your custom list
ensemble_forecaster = AutoEnsembleForecaster(forecasters=tree_based_
forecasters)

# 3. Fit the ensemble, passing the Fourier terms in `X`
print("Fitting the tree-based ensemble...")
ensemble_forecaster.fit(y=y_train, X=X_train, fh=fh)
```

There's more...

The **Time Series Feature Extraction Library** (**TSFEL**) can automate feature extraction from your time series and offers features spanning multiple domains including spectral, statistical, temporal, and fractal analysis.

You will need to install the library:

```
pip install tsfel
```

You can simply run the following to get an idea of how the tool works:

```
import tsfel
cfg = tsfel.get_features_by_domain()

X = tsfel.time_series_features_extractor(cfg, df['y'])
print("TSFEL Features shape:", X.shape)
>>
TSFEL Features shape: (1, 156)
```

These are 156 features within the four main domains:

```
cfg.keys()
>>
dict_keys(['spectral', 'statistical', 'temporal', 'fractal'])
```

If you want only features in the signal domain, you can do so by specifying the domain, for example, statistical, temporal, spectral, or fractal:

```
cfg = tsfel.get_features_by_domain('spectral')
X = tsfel.time_series_features_extractor(cfg, df['y'])
print("TSFEL Features shape:", X.shape)
>>
TSFEL Features shape: (1, 111)
```

This shows that you can get 111 features extracted in the spectral domain.

The `cfg` object is a dictionary of different techniques that includes their name, complexity, function used, and a description:

```
cfg['spectral']
```

The following is a truncated example of the output:

```
{'Spectrogram mean coefficient': {'complexity': 'constant',
  'description': 'Calculates the average value for each frequency in the
spectrogram over the entire duration of the signal.',
  'function': 'tsfel.spectrogram_mean_coeff',
  'parameters': {'fs': 100, 'bins': 32},
  'n_features': 'bins',
  'use': 'yes'},

 'Fundamental frequency': {'complexity': 'log',
  'description': 'Computes the fundamental frequency.',
  'function': 'tsfel.fundamental_frequency',
  'parameters': {'fs': 100},
  'n_features': 1,
  'use': 'yes'},
```

To narrow down to specific frequency domain features in TSFEL, you can create a custom dictionary based on a list:

```
spectral_cfg = tsfel.get_features_by_domain('spectral')

selected_keys = [
    'Spectrogram mean coefficient',
    'Fundamental frequency',
    'Max power spectrum',
    'Maximum frequency',
    'Spectral centroid',
    'Spectral entropy',
    'Spectral roll-off',
    'Spectral slope',
    'Wavelet energy',
    'Wavelet entropy'
]

# Filter the config dictionary
filtered_features = {k: spectral_cfg['spectral'].get(k) for k in selected_keys
if k in spectral_cfg['spectral']}

custom_cfg = {'spectral': filtered_features}
```

TSFEL typically extracts one set of features for an entire time series. To use it for forecasting, we need to generate features for sequential segments of our data. We can do this using a **sliding window** approach. In the following code, we will be creating a window of 24 hours to capture daily cycles and **short-term** frequency characteristics. We will then slide this window forward by six hours at a time. For each 24-hour window, TSFEL will extract a set of signal-related features:

```
window_length = 24
# sliding step
step = 6
```

For computational efficiency, we'll use the **joblib** library to run feature extraction on all available CPU cores. This parallel approach can dramatically speed up the process, especially on large datasets:

```
from joblib import Parallel, delayed
def extract_features(i):
    segment = df['y'].iloc[i - window_length:i].values
    features = tsfel.time_series_features_extractor(
        custom_cfg, segment).values.flatten()
    return features

# Generate indices with sliding step to reduce total computations
indices = range(window_length, len(df), step)

# Parallelize feature extraction utilizing all CPU cores
features_list = Parallel(n_jobs=-1)(
    delayed(extract_features)(i) for i in indices)

# Convert to numpy array for downstream modeling
X = np.array(features_list)
```

The `Parallel` class is used to create the parallel jobs, while the `delayed` function decorates our `extract_features` function call, deferring its execution until it is submitted to a `Parallel` instance. `n_jobs=-1` specifies that all available CPU cores are to be used.

Align our exogenous features matrix X with our target series:

```
feature_index = df.index[window_length::step]
X_df = pd.DataFrame(X, index=feature_index)
y_aligned = df['y'].loc[feature_index]
```

Split the datasets into train and test sets, fit the model with our newly extracted features, and compare the results:

```
# Split into train and test sets
# Since our features are now generated every 6 hours
# we need 28 samples to cover the same 168-hour period (168 / 6 = 28)
y_train, y_test = temporal_train_test_split(y_aligned, test_size=28)
X_train, X_test = X_df.loc[y_train.index], X_df.loc[y_test.index]
```

We changed `test_size` from 168 to 28 to ensure that we are evaluating the TSFEL-based model on the same seven-day forecasting horizon as the other models in the chapter.

```
# Set up forecasting with exogenous TSFEL features
forecaster = make_reduction(
    LGBMRegressor(random_state=42),
    strategy="recursive",
    window_length=window_length
)

fh = ForecastingHorizon(y_test.index, is_relative=False)
forecaster.fit(y_train, X=X_train)
y_pred = forecaster.predict(fh=fh, X=X_test)

# Evaluate
lgbm_mae = mae(y_true=y_test, y_pred=y_pred)
lgbm_rmse = np.sqrt(mse(y_true=y_test, y_pred=y_pred))
print("LightGBM + TSFEL MAE:", lgbm_mae)
print("LightGBM + TSFEL RMSE:", lgbm_rmse)
>>
LightGBM + TSFEL MAE: 474.85128701749284
LightGBM + TSFEL RMSE: 665.8699842313326
```

The results are very promising and show a great improvement by incorporating the additional spectral features using TSFEL.

If you want to further improve the model, you are encouraged to consider tuning the LightGBM **hyperparameters** such as learning rate, number of leaves, and boosting rounds. Further, when working with TSFEL, you should explore other feature domains beyond spectral, such as statistical, temporal, or fractal domains. Depending on your dataset, these may enhance model accuracy or introduce noise, so careful experimentation is key.

Finally, note that the TSFEL sliding window approach generates features every six hours in this example, meaning evaluation is spaced accordingly. *Compare metrics cautiously* with hourly models since the forecasting frequency differs and direct equivalence may not hold. You should also explore different window sizes and steps and their effect on performance.

See also

- To learn more about TSFEL, you can read their documentation here: `https://tsfel.readthedocs.io/en/latest/`