Bonus Chapter 15

Probabilistic Time-Series Forecasting

In the previous chapters, we mostly emphasized **point forecasts** (making a single best guess for the future) without directly capturing any inherent uncertainty in predictions (with a few exceptions where we introduced probabilistic concepts). But practical real-world decisions require understanding uncertainty, such as understanding how confident our model is in the forecast produced. For example, just knowing the likely energy demand for tomorrow isn't enough; you also need to ask, "How confident are we in that number?" This is where probabilistic time-series forecasting comes in. It's not just about a single number; it's about understanding the entire range of possible outcomes so you can make truly informed decisions.

Traditional point-forecasting algorithms focus on minimizing a single loss function, such as the **Mean Absolute Error** (**MAE**) or **Root Mean Squared Error** (**RMSE**), where errors are assumed to be treated equally. This is where probabilistic time-series forecasting comes in. It's not just a mathematical (calculating uncertainty) or technical (changing modeling objective) focus, but rather includes a shift in thinking in how we interpret the forecast to make informed decisions. Probabilistic forecasting addresses the limitations of point forecasts by estimating a probability distribution for each future time step to construct **prediction intervals** that tell us how confident we are in our forecast.

Generally, there are two main approaches to generating probabilistic forecasts, each with different computational requirements and underlying assumptions:

- **In-model (intrinsic) approaches**: In this approach, we are modifying the model's training objective itself. Instead of training the model to predict a single point value, the model is trained to directly output multiple values of the predicted distribution (e.g., 5^{th}, 50^{th}, or 95^{th} percentile). Examples of this approach include the following:
    - **Quantile regression**: Models trained to directly predict specific quantile values of the target distribution, producing prediction intervals by design
    - **Bayesian models**: Models that use Bayesian inference, such as Bayesian structural time series
    - **Ensemble methods**: Multiple models are trained on the same data, each model providing their forecast, and the variability in the results provides an empirical measure of prediction uncertainty

- **Post hoc (post-training, model-agnostic) approaches:** In this approach, we start with a standard point-forecasting model and apply post-processing techniques to generate uncertainty estimates. These techniques don't require retraining the original model. Examples of this approach include the following:
    - **Monte Carlo (MC) dropout**: At inference, dropout is applied multiple times to obtain a different set of predictions that are used to estimate uncertainty. This is mostly applied to neural network-based models.
    - **Conformal Prediction (CP)**: After the model is trained, a calibration set is used to estimate residuals (errors) to construct prediction intervals, but what separates CP is that it guarantees coverage.
    - **Bootstrap models**: Generate intervals by fitting the model to multiple bootstrap-resampled versions of the training dataset and aggregating the resulting forecasts.

Generating a prediction interval is just the first step; one should evaluate its quality, reliability, coverage, and calibration. **Coverage** measures whether prediction intervals contain the actual future values at the expected rate (e.g., a 90% prediction interval should capture the true value roughly 90% of the time across many forecasts). Deviation from the expected rate can indicate a flaw in the uncertainty estimation: **under-coverage** (the intervals are too narrow and the model is overconfident) or **over-coverage** (the intervals are too wide and the model is under-confident). **Calibration** goes a bit deeper, examining whether the model's confidence levels match reality across different scenarios and forecast horizons.

An important point that needs clarification for this chapter is the difference between **confidence intervals** and **prediction intervals**:

- **Prediction intervals:** Provide a range where we expect a future data point to fall. For example, we are 95% confident that the energy consumption tomorrow is between 14,500 and 15,500 **Megawatts** (**MW**).
- **Confidence intervals**: Provide a range for an estimated model parameter. For example, we are 95% confident that the true coefficient for the daily seasonal component is between 0.8 and 1.2.

In this chapter, we will use the same libraries you were introduced to in previous chapters and explore the options provided to create prediction intervals. We will explore a variety of techniques, including **MC dropout**, **quantile regression**, **CP**, and **Conformalized Quantile Regression** (**CQR**). You will learn about the difference between the main types of uncertainty and how each library offers different implementations for each:

- **Epistemic:** Due to a lack of knowledge or data that can be reduced by collecting more or better data. Think of it as *what we don't know yet*.
- **Aleatoric**: Inherent randomness or noise present in the data that cannot be reduced. Think of it as *what we can't know* due to the unpredictable variability in the system itself.

Prediction intervals aim to capture both epistemic and aleatoric uncertainty. The width of the interval reflects the combined effect of what the model has learned (and what it hasn't) and the inherent randomness in the data. A good probabilistic forecast will produce intervals that are wide enough to cover the true values while still being narrow enough to be useful for decision-making.

In this chapter, you will explore the following recipes:

- Prediction intervals with statsmodels
- Probabilistic forecasting with Darts
- Probabilistic forecasting with NeuralForecast
- Probabilistic forecasting with NeuralProphet

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/Ch15`.

All of the 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 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/Ch15/')
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()
```

This should produce the time-series plot shown in *Figure 15.1* for the hourly energy consumption:

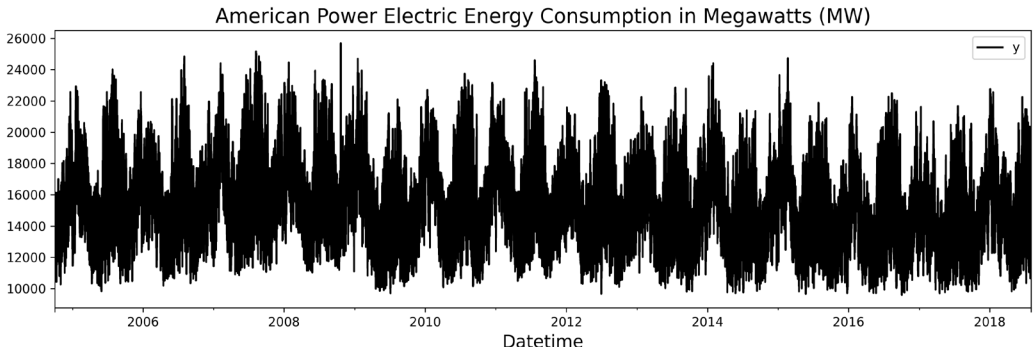

Figure 15.1: Plot of the hourly American Power Electric (AEP) energy consumption from October 2004 to August 2018

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 1*. The chapter covers two methods for creating Python virtual environments, via `conda` or `venv`.

Prediction intervals with statsmodels

In past chapters, you were introduced to `statsmodels'` statistical models, such as **ARIMA**, **SARIMAX**, **ExponentialSmoothing**, and **UnobservedComponents**. In *Chapter 10*, we briefly mentioned the `get_forecast()` method for out-of-sample forecasts that include prediction intervals. There's also a more general `get_prediction()` method, which can provide intervals for both in-sample fits and out-of-sample forecasts. In this recipe, we'll focus on `get_forecast()` since we're interested in predicting the future.

Both of these methods return a special `PredictionResults` object, which then gives access to the probabilistic forecast components, including the following:

- `predicted_mean`: This attribute provides the point forecast.
- `conf_int(alpha=0.05)`: This method returns forecast uncertainty bands for future observations, known as prediction intervals. The default returns the 95% prediction interval.
- `summary_frame()`: This method returns a pandas DataFrame containing the point forecast (`mean`) and prediction intervals (`mean_ci_lower` and `mean_ci_upper`), as well as `mean_se`.

Getting ready

You can install `statsmodels` using `pip`:

```
pip install statsmodels
```

You can also install it using `conda`:

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

How to do it...

Many of the statistical models in the `statsmodels` library have access to the `get_forecast` method, which we will be exploring in this recipe by developing a SARIMAX model with Fourier terms. This will mimic our approach in *Chapter 15* with the UCM model with frequency seasonal terms, where we specified the number of harmonics for each seasonal component in the form:

```
'freq_seasonal':[{'period': day, 'harmonics': 4},
                 {'period': week, 'harmonics': 6},
                 {'period': year, 'harmonics': 8}]
```

In this recipe, we use SARIMAX augmented with Fourier seasonal terms to capture complex periodic patterns in the data. Fourier terms model seasonality in the frequency domain, while SARIMAX captures the remaining autocorrelation in the time domain. The Fourier terms will be passed as exogenous variables (`exog`). Once we fit our model, the `get_forecast()` method produces point forecasts along with prediction intervals that incorporate model uncertainty.

To implement prediction intervals with SARIMAX and Fourier terms, follow along with these steps:

1. Start by importing the components needed from the `statsmodels` library for the recipe:

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

2. Define the Fourier terms for each seasonal component using the Fourier class, which takes a period and an order. We will be using similar parameter values as we did with the UCM, but feel free to explore different values:

```
day = 24
week = day*7
year = day*365
```

```
print(f'''
    day = {day} hours
    week = {week} hours
    year = {year} hours
''')
>>
day = 24 hours
week = 168 hours
year = 8760 hours

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

We use relatively small orders (4 daily, 6 weekly, and 8 yearly) to balance flexibility and overfitting. Though higher values would create more harmonics, it can risk capturing noise.

This will produce a total of 36 terms: 8 daily, 12 weekly, and 16 annual Fourier terms. Each harmonic order produces sine and cosine pairs, so `order=4` means 8 terms (4 sine and 4 cosine). Keep in mind that a higher order can lead to overfitting.

For the annual period, we used 365 days for simplicity. For even higher accuracy on very long datasets, some practitioners use an average of 365.25 to account for leap years. However, for most models, 365 is a perfectly effective approximation.

3. Each Fourier object we created (`fourier_daily`, `fourier_weekly`, and `fourier_annual`) has an `in_sample` method. This method generates the actual sine and cosine terms for each timestamp in the index. We'll generate these terms for each seasonality and concatenate them into a single pandas DataFrame. The DataFrame, `X`, will be used as our set of exogenous variables when we fit the SARIMAX model:

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

X = pd.concat(X_list, axis=1)
print(X.columns)
>>
Index(['sin(1,24)', 'cos(1,24)', 'sin(2,24)', 'cos(2,24)', 'sin(3,24)',
       'cos(3,24)', 'sin(4,24)', 'cos(4,24)', 'sin(1,168)', 'cos(1,168)',
```

```
       'sin(2,168)', 'cos(2,168)', 'sin(3,168)', 'cos(3,168)',
'sin(4,168)',
       'cos(4,168)', 'sin(5,168)', 'cos(5,168)', 'sin(6,168)',
'cos(6,168)',
       'sin(1,8760)', 'cos(1,8760)', 'sin(2,8760)', 'cos(2,8760)',
       'sin(3,8760)', 'cos(3,8760)', 'sin(4,8760)', 'cos(4,8760)',
       'sin(5,8760)', 'cos(5,8760)', 'sin(6,8760)', 'cos(6,8760)',
       'sin(7,8760)', 'cos(7,8760)', 'sin(8,8760)', 'cos(8,8760)'],
      dtype='object')
```

4. Let's split both DataFrames (`df` and `X`) into training and testing sets, holding out the last week (168 hours) for testing:

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

5. Define the SARIMAX model by passing the training set and exogenous variables. We will start with a simple model, without seasonal components, since our Fourier terms are designed to model these instead (frequency domain as opposed to time domain):

```
mod = SARIMAX(
    train['y'],
    exog=exog_train,
    order=(1, 0, 1)
)
```

   Since our Fourier terms already capture strong seasonal effects, the SARIMAX component is kept simple (just enough to capture residual autocorrelation). We will start with the ARMA(1,1) structure (`order=(1,0,1)`), keeping the model lightweight.

   Let's fit the model. We'll add `disp=False` to the `fit` method to prevent the model's convergence information from being printed:

```
res = mod.fit(disp=False)
```

6. Once training is complete, you can inspect the model using `res.summary()` and inspect the residuals using `res.plot_diagnostics()`. We will jump ahead and focus on creating our forecasts, but this time, instead of using the `forecast()` method, let's use `get_forecast()`:

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

7. The `fc` object is of the `PredictionResults` type, which gives access to the `predicted_mean` attribute (point forecast), the `conf_int()` method (upper and lower points), as well as a convenience method to get all forecast components as a DataFrame via `summary_frame()`.

Visualize the forecast against actual test data with prediction intervals using the `plot_statsmodels_forecast` utility function:

```
from utils import plot_statsmodels_forecast
plot_statsmodels_forecast(fc, test, ylabel='MW')
```

The function extracts the point forecast (`predicted_mean`) and 95% prediction intervals from the fc object, then plots them against the actual test values. The shaded region represents forecast uncertainty which is the range where future values are likely to fall with 95% probability.

In statsmodels, calling `conf_int()` on the forecast object (from `get_forecast()`) returns **prediction intervals**, which represent the uncertainty range for a future data point. However, calling `conf_int()` on the main fitted model object (`res.conf_int()`) returns the **confidence intervals** for the estimated model's parameters, not for the forecast.

This produces a plot showing the forecast against actual values with a shaded area representing the 95% prediction interval.

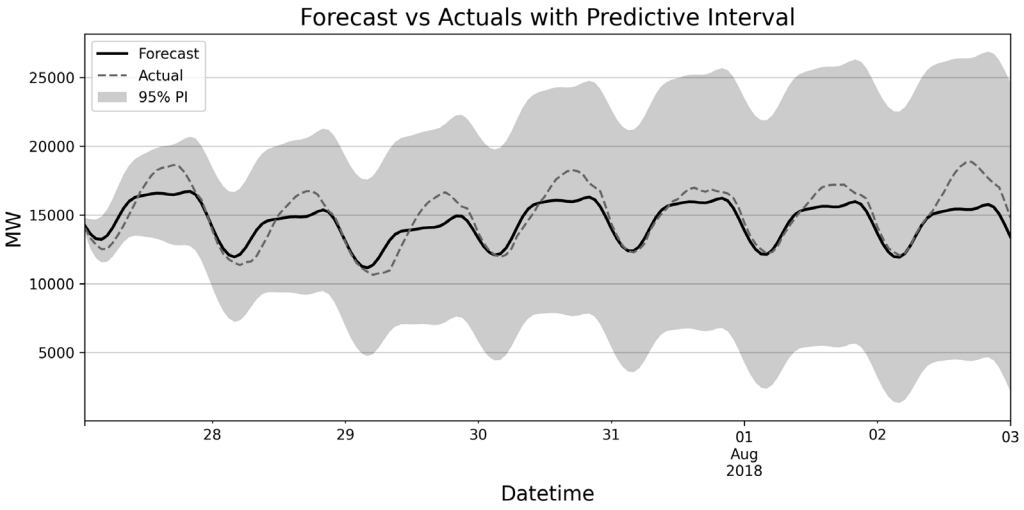

Figure 15.2: Plot of the forecast with prediction intervals for the SARIMAX with Fourier terms model

In *Figure 15.2*, you can observe that the prediction intervals are pretty wide, indicating a significant amount of uncertainty in the forecast. The interval bands get even wider over time, which is typical for time-series forecasts. This visualizes the concept that uncertainty accumulates as we go further into the future, which is why the model is more confident about the next few hours than it is for a full week.

How it works...

In the recipe, you generated the Fourier terms using the Fourier class from `statsmodels`, which generates a pair of sine and cosine terms for each order specified. The Fourier basis functions include the following:

$$sin\left(\frac{2\pi it}{p}\right), cos\left(\frac{2\pi it}{p}\right)$$

Here, we have the following:

- **p (period)**: This is the length of the season (e.g., daily=24 hours, weekly=168 hours, yearly=8760 hours).
- **i (harmonic number)**: This is the harmonic number, ranging from 1 to the specified order, which creates our sine and cosine pairs (terms). For example, an order of 2 will create a total of 4 terms (2 sine and 2 cosine).
- **t (time index)**: This is the time step of the observation.

The Fourier terms are deterministic; they are a fixed set of sine and cosine pairs at known frequencies. When used in SARIMAX, the model tackles the learning process in two stages:

- First, it performs a **regression**, using each of the Fourier terms (our exogenous variables) as a feature. The model estimates (learns) the coefficients for each of these terms to capture the complex seasonal patterns.
- Then, whatever is left over (the **residuals** from the regression) is modeled using the ARMA(1,1) structure to capture any remaining autocorrelation.

This hybrid approach allows us to model seasonality in the **frequency domain** (with Fourier terms) and short-term dynamics in the **time domain** (with ARMA).

Once training is completed, the `get_forecast` method returns the point forecast and allows us to generate **prediction intervals** using the `conf_int(alpha)` method, which has an `alpha` parameter to specify the significance level for the prediction interval. The default `alpha=0.05` corresponds to a 95% interval with coverage `1-alpha` (e.g., 1-0.05 = 0.95).

There's more...

In *Chapter 15*, we used StateForecast's **Multiple Seasonal-Trend decomposition using Loess** (**MSTL**) implementation, which not only decomposes a time series with multiple seasonal components but also enables us to create a forecast. We will use the same code from that recipe with one minor tweak; we'll exclude the yearly component to speed up the training process. The impact is very small.

Let's prepare our `df` DataFrame that we created in the *Technical requirements* section:

```
from statsforecast import StatsForecast
from statsforecast.models import MSTL, AutoARIMA
```

Since our original DataFrame had `Datetime` as the index, we reset it to turn `Datetime` back into a column before renaming it `ds`:

```
sf_df = df.reset_index().copy()
sf_df['unique_id'] = 'AEP_hourly'
sf_df = sf_df.rename(columns={'Datetime': 'ds'})

day = 24
week = day*7

sf_train = sf_df.iloc[:-week]
sf_test = sf_df.iloc[-week:]

Now, let's fit our model:

from statsforecast.models import MSTL, AutoARIMA
mstl = MSTL(
    season_length=[day, week],
    trend_forecaster=AutoARIMA()
)

sf = StatsForecast(
    models=[mstl],
    freq='H',
    n_jobs=-1
)
sf.fit(sf_train)
```

The preceding code should be exactly what we did in *Chapter 15* but without supplying the `year` component to speed up the training process.

Now, we are ready to generate our forecast with different interval levels, which is one of the nice features about StatsForecast (we explored this in *Chapter 12* as well):

```
levels = [80, 90, 95]
sf_forecast = sf.predict(h=168, level=levels)
sf_forecast.head()
```

The resulting DataFrame contains the MSTL point forecasts along with lower and upper bounds for each confidence level (e.g., MSTL-lo-80, MSTL-hi-80 for the 80% interval):

	unique_id	ds	MSTL	MSTL-lo-95	MSTL-lo-90	MSTL-lo-80	MSTL-hi-80	MSTL-hi-90	MSTL-hi-95
0	AEP_hourly	2018-07-27 01:00:00	14075.815222	13870.632277	13903.620266	13941.653305	14209.977138	14248.010177	14280.998166
1	AEP_hourly	2018-07-27 02:00:00	13356.375400	13012.647087	13067.909506	13131.623536	13581.127263	13644.841293	13700.103713
2	AEP_hourly	2018-07-27 03:00:00	12851.335911	12364.002414	12442.352748	12532.685665	13169.986157	13260.319074	13338.669408
3	AEP_hourly	2018-07-27 04:00:00	12644.714237	12021.343109	12121.564695	12237.113765	13052.314709	13167.863779	13268.085365
4	AEP_hourly	2018-07-27 05:00:00	12711.650625	11962.328521	12082.799692	12221.695236	13201.606013	13340.501557	13460.972728

Figure 15.3: StatsForecast prediction output including prediction intervals representing 80, 90, and 95% interval levels

Plot the forecast against actual values with 80% and 95% prediction intervals:

```
from utils import plot_statsforecast_prediction

# Plot and get forecast for metrics
sf_yhat = plot_statsforecast_prediction(
    sf_forecast,
    sf_test,
    model_name='MSTL',
    ylabel='MW'
)
```

The resulting plot shows the point forecast, actual test values, and shaded prediction intervals:

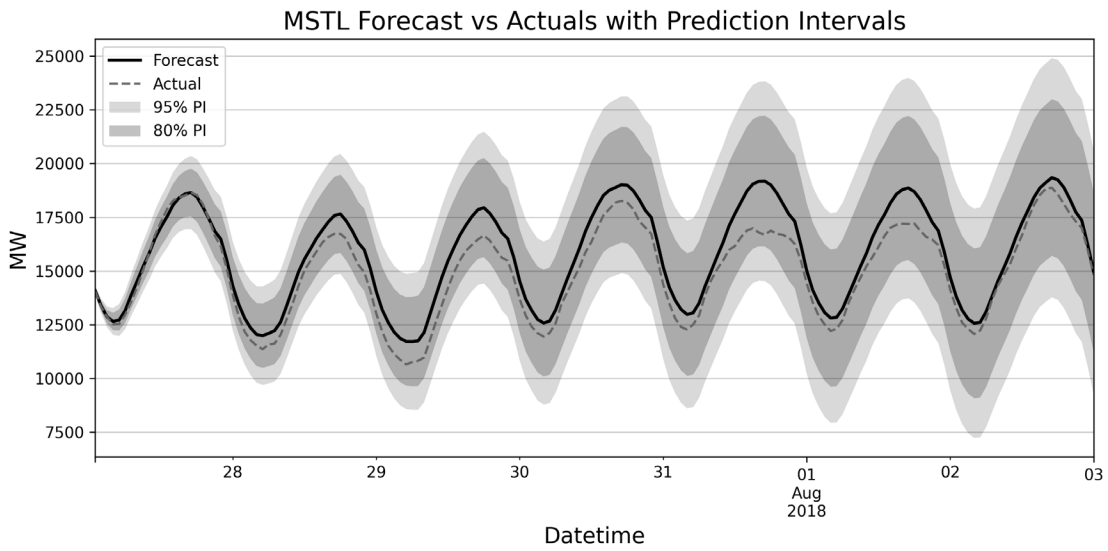

Figure 15.4: StatsForecast's MSTL forecast with 80 and 95% prediction intervals displayed

There is a stark difference between the outputs in *Figure 15.2* and *Figure 15.4*.

In *Figure 15.4*, we have narrowed down the prediction intervals (PI), with the 80% PI as the inside shaded area and the 95% PI as the outer area. The figure still confirms a similar pattern to what we observed in *Figure 15.2*: our model does great at short-term forecasts, but as we go further into the future, the uncertainty accumulates. Overall, the MSTL model did a better job, as confirmed by the lower MAE and RMSE scores:

```
from statsmodels.tools.eval_measures import rmse, meanabs as mae
mstl_mae = mae(sf_test['y'], sf_yhat)
mstl_rmse = rmse(sf_test['y'], sf_yhat)
sarimax_mae = mae(test['y'], yhat)
sarimax_rmse = rmse(test['y'], yhat)

print(f'MSTL MAE: {mstl_mae}')
print(f'MSTL RMSE: {mstl_rmse}')

print(f'SARIMAX MAE: {sarimax_mae}')
print(f'SARIMAX RMSE: {sarimax_rmse}')
>>
MSTL MAE: 756.7793753275537
MSTL RMSE: 886.4183741972955
SARIMAX MAE: 905.377821530133
SARIMAX RMSE: 1180.8093838302636
```

The MSTL model is specifically designed for multiple seasonal patterns (see *Chapter 15*), which explains why it produces tighter and more confident prediction intervals. The results also indicate that the MSTL produces a more accurate point forecast (lower MAE and RMSE).

See also

- To learn more about probabilistic forecasting in StatsForecast, you can check out their tutorial here: `https://nixtlaverse.nixtla.io/statsforecast/docs/tutorials/UncertaintyIntervals`

Probabilistic forecasting with Darts

In *Chapter 13,* you were introduced to Darts' capabilities using deep learning models for time-series forecasting, including **RNNModel**, **BlockRNNModel**, **TCNModel**, and **NHiTSModel**. In the **TCNModel** recipe, we touched on MC dropout to generate 200 samples (predictions) and discussed the use of the **quantile loss** (**pinball loss**) as opposed to point forecast measures such as the RMSE and MAE.

In this recipe, we will dive into other options that Darts offer for probabilistic forecasting and uncertainty quantification. But before we dive into the how, let's understand the two main types of uncertainties that we will explore:

- **Aleatoric uncertainty**: This uncertainty arises from the randomness inherent in the data-generating process. Even if we have a great model, there is still a level of uncertainty that we cannot ignore or eliminate due to the natural noise and variability that exists in the data-generating process. Techniques such as **quantile regression** are great for capturing this.

- **Epistemic uncertainty**: This represents the model's uncertainty about its own predictions due to limited training (the model only knows from the data it learned from with leading to limited exposure). Here, we focus on the model's uncertainty: what the model doesn't know and how its confidence changes. A classic example is the **MC dropout**; when we used the MC dropout (in *Chapter 13*), the model had already been trained (fixed weights) and we were mostly measuring its epistemic uncertainty.

In this recipe, we will use NHiTSModel and compare the different options for quantifying these different types of uncertainty.

Getting ready

You can install Darts using `pip`:

```
pip install darts
```

You can also install it using `conda`:

```
conda install -c conda-forge -c pytorch u8darts-all
```

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. You will be building an NHiTS model similar to the one developed in *Chapter 13*, with minor changes to speed up the training process given the size of this dataset (hourly, spanning 14 years) compared to the daily energy dataset used.

Point forecast

We will build a standard NHiTS model for the point forecast first, then expand on this to develop our probabilistic models:

1. Start by importing the necessary components from the Darts library and any additional libraries needed for the recipe:

```
from darts.models import NHiTSModel
from darts import TimeSeries
from darts.dataprocessing.transformers import Scaler
from darts.metrics import rmse, mae, mql
from darts.utils.likelihood_models import QuantileRegression
from pytorch_lightning.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
import torch
```

2. Convert the pandas DataFrame into a `TimeSeries` Darts object:

```
darts_df = df.reset_index().copy()

ts = TimeSeries.from_dataframe(darts_df,
```

```
                                    time_col='Datetime',
                                    value_cols='y',
                                    freq='h')
ts = ts.astype(np.float32)
```

3. Split the data into training, validation, and test sets. We will hold out the last two days (48 hours) as our test set and the preceding two weeks for validation:

```
day = 24
week = day*7

n_total = len(ts)
n_test = day*2
n_val = week*2

# Calculate split indices
test_start = n_total - n_test
val_start = test_start - n_val

# Create splits maintaining temporal order
d_train = ts[:val_start]
d_val = ts[val_start:test_start]
d_test = ts[test_start:]

print(f"""
    train_np : {len(d_train)}
    val_np : {len(d_val)}
    test_np: {len(d_test)}
""")
>>
train_np : 120912
val_np : 336
test_np: 48
```

From the previous recipe, using `statsmodels` and `StatsForecast`, we realized that the data is complex and long-term forecasting will accumulate higher uncertainties (the prediction intervals become wider). We were holding out 168 hours (one week) for testing. Based on these observations, we will focus on a shorter forecast, let's say 48 hours (two days ahead).

4. Deep learning models are sensitive to scaling, so we will need to scale our datasets:

```
d_scaler = Scaler(StandardScaler())

d_train_scaled = d_scaler.fit_transform(d_train)
```

```
d_val_scaled = d_scaler.transform(d_val)
d_test_scaled = d_scaler.transform(d_test)
d_series = d_scaler.transform(ts)
```

5. The following code is exactly the same setup and configuration as in *Chapter 13* (if you want to dive into each element in more detail, refer to that chapter). There are a few key changes made to speed up the training process, which are highlighted in the following code:

```
PATIENCE=10 # lowered the EarlyStopping threshold
early_stopper = EarlyStopping(
    monitor="val_loss",
    patience=PATIENCE,
    min_delta=0.001,
    mode='min',
    verbose=True)

trainer_kwargs = {
    "callbacks": [early_stopper],
    "enable_progress_bar": True,
    "accelerator":"auto"}

lr_scheduler_kwargs = {
        "mode": "min",
        "monitor": "val_loss",
        "factor": 0.2,
        "patience": PATIENCE//3,
        "min_lr": 1e-6
    }

nhits_point_model = NHiTSModel(
    input_chunk_length=day*5, # 120 hours in
    output_chunk_length=day*2, # 48 hours out
    n_epochs=100, # modified
    num_blocks=1,
    num_stacks=2, # modified (lowered)
    num_layers=1, # modified (lowered)
    pooling_kernel_sizes=None,
    n_freq_downsample=None,
    batch_size=168, # modified (increased from 32)
    optimizer_kwargs={"lr": 0.001},
    model_name="AEP_hourly",
    random_state=42,
```

```
    force_reset=True,
    save_checkpoints=True,
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs
)

# Fit the model with transformed target
nhits_point_model.fit(
    series=d_train_scaled,
    val_series=d_val_scaled,
    verbose=True)
```

We reduced `num_stacks` and `num_layers` compared to our *Chapter 13* setup to speed up training without sacrificing too much accuracy. This is a practical trade-off when working with such a large hourly dataset. Feel free to experiment with increasing these parameters if you're willing to wait longer for potentially better results.

6. Once training is complete, we are ready to make our forecast:

```
d_train_val_sc = d_train_scaled.append(d_val_scaled)

nhits_forecast_sc = nhits_point_model.predict(
    n=len(d_test),
    series=d_train_val_sc
)
# inverse transformation to get the forecast back to original scale
nhits_forecast_inv = d_scaler.inverse_transform(nhits_forecast_sc)
```

7. Plot the forecast against the actual for visual comparison:

```
plt.figure(figsize=(16, 5))
d_test.plot(label='Actual', linestyle='--', alpha=0.65)
nhits_forecast_inv.plot(label='Forecast', color='k',
                        linestyle='-', alpha=0.65)
plt.title('Darts NHITS vs Actuals)')
plt.xlabel('Hours')
plt.ylabel('Energy Consumption')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

This should produce a plot with a solid line for the forecast and a dashed line for the test set:

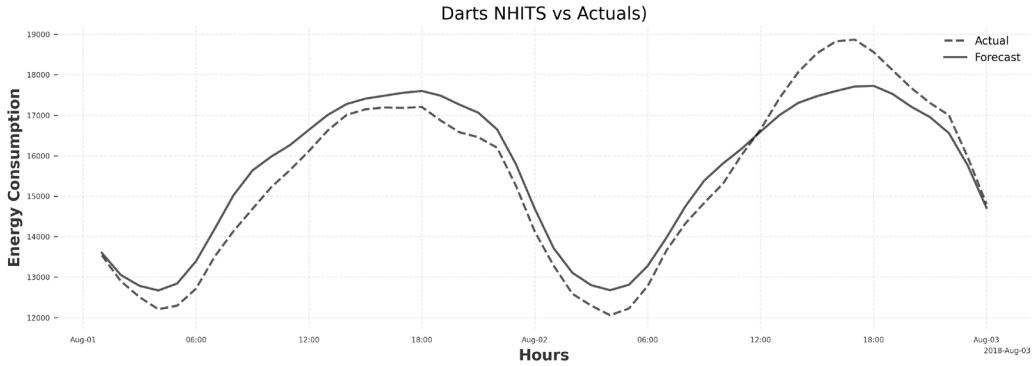

Figure 15.5: Darts NHiTS 48-hour forecast against actuals

8. Let's calculate both the RMSE and MAE for our point forecast:

```
rmse_val = rmse(d_test, nhits_forecast_inv)
mae_val = mae(d_test, nhits_forecast_inv)
print(f"Darts Test Set MAE:  {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE:  516.3734
Darts Test Set RMSE: 578.2925
```

MC dropout

MC dropout repurposes a feature that is originally designed for training: the dropout layer. During the training of neural networks, dropout randomly turns off some neurons in the network to prevent overfitting. Normally, at prediction time, the dropout is turned off and all neurons are used.

By setting `mc_dropout=True`, we are telling the model to keep the dropout active during forecasting. Each time we generate a forecast (a single sample), a different, random set of neurons is deactivated. This essentially creates multiple versions of our trained model, each giving slightly different predictions, allowing us to test the model's ability and measure its uncertainty.

By setting `num_samples=200`, we are essentially generating 200 *slightly* different forecasts, which would give us a distribution of possible outcomes. We use the word "*slightly*" here because the variations between the forecasts will generally be small since the model has already been trained and we have already established its weights (*fixed weights*). This will be reflected when plotting our uncertainty intervals, as you will notice that they will be much narrower. This issue is due to the model being overconfident because it has already learned and captured the main patterns, but it will still fail to account for the inherent, real-world randomness of energy consumption. This is because MC dropout primarily measures **epistemic** uncertainty (the model's uncertainty in its own parameters) and does not account for **aleatoric** uncertainty (inherent randomness in the data itself).

To implement MC dropout and generate uncertainty estimates, you'll use the same

trained model but activate dropout during prediction:

1. Use the same `nhits_d_model` but this time, during prediction, we will specify 200 samples and turn on the MC dropout:

```
d_train_val_sc = d_train_scaled.append(d_val_scaled)

nhits_forecast_mc = nhits_point_model.predict(
    n=len(d_test),
    series=d_train_val_sc,
    num_samples=200,
    mc_dropout=True
    )

nhits_forecast_mc_inv = d_scaler.inverse_transform(nhits_forecast_mc)
```

2. Visualize the probabilistic forecast using the plot_darts_forecast utility function, which handles Darts' TimeSeries format and prediction intervals:

```
from utils import plot_darts_forecast
plot_darts_forecast(
    d_test,
    nhits_forecast_mc_inv,
    'Darts NHITS Probabilistic Forecast (MC Dropout)'
)
```

This plots the test set against the MC dropout predictions, showing the forecast mean and quantile-based uncertainty intervals.

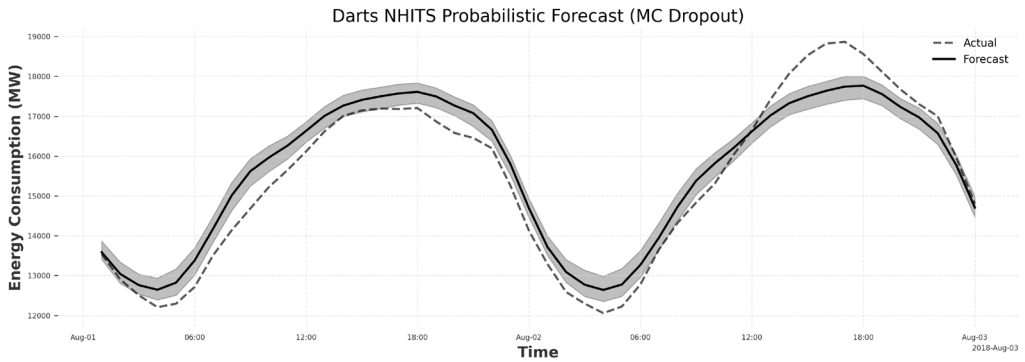

Figure 15.6: MC dropout forecast with uncertainty intervals

As expected, the uncertainty intervals are very narrow.

3. Let's calculate the quantile loss. We will use the mean quantile loss from Darts:

```
from darts.metrics import mql

# Calculate the loss for the lower bound (5th percentile)
loss_p05 =mql(d_test, nhits_forecast_mc_inv, q=0.05)

# Calculate the loss for the median (50th percentile)
# This value will be identical to your MAE score
loss_p50 = mql(d_test, nhits_forecast_mc_inv, q=0.50)

# Calculate the loss for the upper bound (95th percentile)
loss_p95 = mql(d_test, nhits_forecast_mc_inv, q=0.95)

print(f"MC Dropout Average Quantile Loss at 5th percentile: "
      f"{loss_p05:.4f}")
print(f"MC Dropout Average Quantile Loss at 50th percentile (MAE): "
      f"{loss_p50:.4f}")
print(f"MC Dropout Average Quantile Loss at 95th percentile: "
      f"{loss_p95:.4f}")
>>
MC Dropout Average Quantile Loss at 5th percentile:  297.9872
MC Dropout Average Quantile Loss at 50th percentile (MAE): 505.8007
MC Dropout Average Quantile Loss at 95th percentile: 241.0836
```

   Lower quantile loss values are better. The 5th and 95th percentile losses tell us how well our model captures the lower and upper bounds of the energy consumption distribution, while the 50th percentile loss (equivalent to the MAE) measures the accuracy of our median forecast. A good probabilistic model should have balanced performance across all quantiles.

   The MC dropout creates an ensemble of similar models at prediction time by randomly deactivating neurons. Since the base weights remain fixed (from training), this primarily captures **epistemic uncertainty** rather than the full range of possible outcomes.

Likelihood estimation

Darts provides a `likelihood` parameter for many of the Torch-based forecasting models, such as TCNModel, NHiTSModel, NBEATSModel, and TFTModel, to name a few. We pass a **likelihood** object at model creation time. Darts supports a variety of likelihoods, which you can view here: `https://unit8co.github.io/darts/generated_api/darts.utils.likelihood_models.torch.html`.

The `likelihood` parameter changes the model's behavior during both training and prediction. Instead of training the model to predict a single point value (typically the mean), the model's output layer is reconfigured to predict the parameters of the specified probability distribution. This shifts the entire objective of the neural network.

The choice of likelihood specifies which distribution the model is trying to fit. In this recipe, we will use `QuantileRegression` as our likelihood, which directly predicts specific quantiles of the future distribution. The model is trained using the **quantile loss** (also known as the **pinball loss**), which penalizes under-predictions more severely for higher quantiles and over-predictions more severely for lower quantiles. This approach directly tackles the inherent randomness in the data itself (the **aleatoric uncertainty**) by modeling the full distribution rather than just its central tendency.

To implement likelihood-based prediction intervals, you'll configure the NHiTS

model with QuantileRegression to predict the 5th, 50th (median), and 95th percentiles:

1. Use the same setup, but modify `nhits_d_model` to include the `likelihood` parameter:

```
nhits_d_model_qr = NHiTSModel(
    input_chunk_length=day*5, # 120 hours in
    output_chunk_length=day*2, # 48 hours out
    n_epochs=100, # modified
    num_blocks=1,
    num_stacks=2, # modified (lowered)
    num_layers=1, # modified (lowered)
    pooling_kernel_sizes=None,
    n_freq_downsample=None,
    batch_size=168, # modified (increased from 32)
    optimizer_kwargs={"lr": 0.001},
    model_name="AEP_hourly",
    random_state=42,
    force_reset=True,
    save_checkpoints=True,
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs,
    likelihood=QuantileRegression([0.05, 0.5, 0.95])
)
```

   When we use `likelihood=QuantileRegression([0.05, 0.5, 0.95])`, we're telling NHiTS to predict three distinct values (instead of just the mean value): the 5th percentile, the 50th percentile (the median), and the 95th percentile of the energy consumption distribution for each future time step.

2. We will continue with a similar process of fitting the model:

```
nhits_d_model_qr.fit(
    series=d_train_scaled,
    val_series=d_val_scaled,
    verbose=True
)
```

3. Once training is done, you can generate the forecast. Even though we are not using the MC dropout by keeping `mc_drouput` turned off (default), we still need `num_samples`. Our model no longer predicts a single point; instead, it's trained to output the parameters of a distribution at each future time step. Thus, we need `num_samples` to draw many samples from the predicted distribution to build our probabilistic forecast and understand its uncertainty:

```
d_train_val_sc = d_train_scaled.append(d_val_scaled)

nhits_forecast_qr = nhits_d_model_qr.predict(
    n=len(d_test),
    series=d_train_val_sc,
    num_samples=200
)

nhits_forecast_qr_inv = d_scaler.inverse_transform(nhits_forecast_qr)
```

4. Plot the generated forecast using our `plot_darts_forecast` function:

```
plot_darts_forecast(
    d_test,
    nhits_forecast_qr_inv,
    'Darts NHITS Probabilistic Forecast (Quantile Regression)'
)
```

The resulting plot displays the forecast with prediction intervals derived from the quantile regression likelihood:

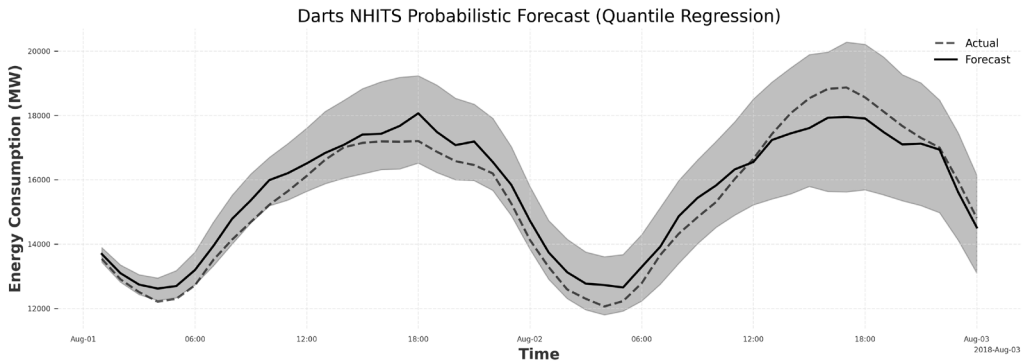

Figure 15.7: NHiTS with QuantileRegression likelihood estimator

5. Let's calculate our scores:

```
loss_p05 = mql(d_test, nhits_forecast_qr_inv, q=0.05)
loss_p50 = mql(d_test, nhits_forecast_qr_inv, q=0.50)
loss_p95 = mql(d_test, nhits_forecast_qr_inv, q=0.95)
```

```
print(f"QuantileRegression Average Quantile Loss at 5th percentile:"
      f"{loss_p05:.4f}")
print(f"QuantileRegression Average Quantile Loss at 50th percentile"
      f"(MAE): {loss_p50:.4f}")
print(f"QuantileRegression Average Quantile Loss at 95th percentile:"
      f"{loss_p95:.4f}")
>>
QuantileRegression Average Quantile Loss at 5th percentile:  100.8952
QuantileRegression Average Quantile Loss at 50th percentile (MAE):
469.3718
QuantileRegression Average Quantile Loss at 95th percentile: 148.7617
```

Likelihood estimation with MC dropout

6. Let's expand on our previous model by turning on `mc_dropout=True` and generating new samples at prediction time:

```
d_train_val_sc = d_train_scaled.append(d_val_scaled)

nhits_forecast_qr_mc = nhits_d_model_qr.predict(
    n=len(d_test),
    series=d_train_val_sc,
    num_samples=200,
    mc_dropout=True
)

nhits_forecast_qr_mc_inv = d_scaler.inverse_transform(nhits_forecast_qr_
mc)
```

7. Let's generate our plot for visual comparison:

```
plot_darts_forecast(
    d_test,
    nhits_forecast_qr_mc_inv,
    'Darts NHITS Probabilistic Forecast (QR + MC Dropout)'
)
```

The resulting plot displays the forecast with prediction intervals derived from the quantile regression likelihood and MC dropout:

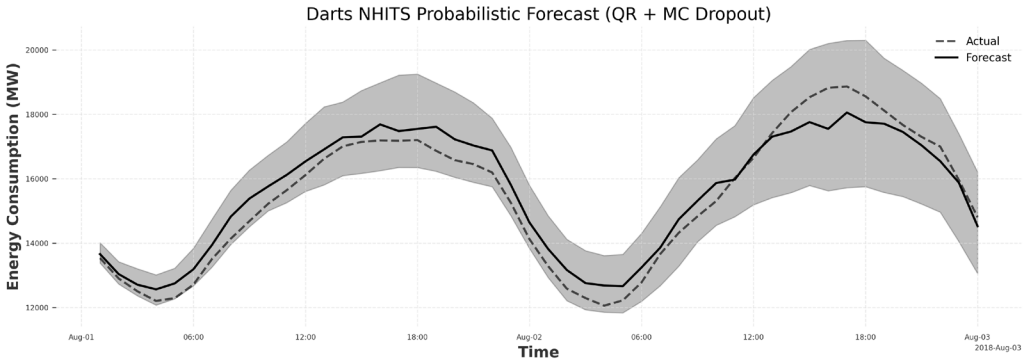

Figure 15.8: NHiTS with QuantileRegression likelihood estimator and MC dropout

8. Lastly, let's compute our scores:

```
loss_p05 = mql(d_test, nhits_forecast_qr_mc_inv, q=0.05)
loss_p50 = mql(d_test, nhits_forecast_qr_mc_inv, q=0.50)
loss_p95 = mql(d_test, nhits_forecast_qr_mc_inv, q=0.95)

print(f"QuantileRegression with MC Average Quantile Loss at "
      f"5th percentile:  {loss_p05:.4f}")
print(f"QuantileRegression with MC Average Quantile Loss at "
      f"50th percentile (MAE): {loss_p50:.4f}")
print(f"QuantileRegression with MC Average Quantile Loss at "
      f"95th percentile: {loss_p95:.4f}")
>>
QuantileRegression with MC Average Quantile Loss at 5th percentile:
100.0776
QuantileRegression with MC Average Quantile Loss at 50th percentile
(MAE): 450.6700
QuantileRegression with MC Average Quantile Loss at 95th percentile:
152.0868
```

How it works...

In this recipe, we explored three different ways to generate probabilistic forecasts with Darts, each giving a unique perspective to view uncertainty:

- **Point forecast**: Similar to what we did in *Chapter 13*, here we trained a standard `NHiTSModel` to get a single best-guess prediction. This gave us a baseline MAE of 451.67, but it offered no insights into its potential range of future values; in other words, we had no visibility into how confident the model is.

- **MC dropout**: By enabling `mc_dropout` at prediction time, we are essentially creating an ensemble of slightly different models on the fly. This technique is great for measuring epistemic uncertainty (the uncertainty of the model's own parameters). This resulted in a narrow prediction intervals, suggesting that the model is pretty confident in the patterns it learned. The median MAE was 445.71.
- **Likelihood estimation (quantile regression)**: Here, we shifted from training the model to produce a single value to being trained to directly predict the 5th, 50th, and 95th percentiles of the data's distribution. This approach addresses aleatoric uncertainty, which is the inherent, unavoidable randomness in the energy consumption data itself. The prediction intervals were much wider and probably more realistic. The median MAE was a bit higher at 496.26.
- **Combining likelihood estimation and MC dropout**: Here, we combined both `QuantileRegression` and `mc_dropout`. This is a more comprehensive approach to capture the model's uncertainty (epistemic) and the data's randomness (aleatoric). This gave us a forecast with a good median MAE of 476.61 and the best quantile loss scores.

The point of comparing different methods is to illustrate that there is no single "best" approach for all situations. The right choice will depend on your specific needs and goals. If you are concerned about the model's confidence, the MC dropout is a great tool to leverage. If you want to capture the inherent volatility of the process you are trying to forecast, then a likelihood-based model is your way to go.

There's more...

In the recipe, we used `QuantileRegression` as the likelihood, which is a powerful non-parametric approach. In this section, we will explore the use of `GaussianLikelihood` as a parametric approach to see how it affects the model output and the uncertainty intervals generated. Instead of directly predicting quantiles as in `QuantileRegression`, with `GaussianLikelihood`, the model will learn to predict the mean and standard deviation of the Gaussian (normal) distribution for each future time step:

```
from darts.utils.likelihood_models import GaussianLikelihood
nhits_d_model_gauss = NHiTSModel(
    input_chunk_length=day*5, # 120 hours in
    output_chunk_length=day*2, # 48 hours out
    n_epochs=100, # modified
    num_blocks=1,
    num_stacks=2, # modified (lowered)
    num_layers=1, # modified (lowered)
    pooling_kernel_sizes=None,
    n_freq_downsample=None,
    batch_size=168, # modified (increased from 32)
    optimizer_kwargs={"lr": 0.001},
    model_name="AEP_hourly",
    random_state=42,
    force_reset=True,
    save_checkpoints=True,
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
```

```
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs,
    likelihood=GaussianLikelihood()
)
nhits_d_model_gauss.fit(
    series=d_train_scaled,
    val_series=d_val_scaled,
    verbose=True
)
```

Once training is done, we will use `num_samples` to sample from the Gaussian predicted distributions to generate the forecast and its uncertainty intervals:

```
d_train_val_sc = d_train_scaled.append(d_val_scaled)
nhits_forecast_gauss = nhits_d_model_gauss.predict(
    n=len(d_test),
    series=d_train_val_sc,
    num_samples=200
)

nhits_forecast_gauss_inv = d_scaler.inverse_transform(nhits_forecast_gauss)
```

Plot the generated forecast using our `plot_darts_forecast` function:

```
plot_darts_forecast(
    d_test,
    nhits_forecast_gauss_inv,
    'Darts NHITS Probabilistic Forecast (Gaussian)'
)
```

The resulting plot displays the forecast with prediction intervals derived from the gaussian likelihood:

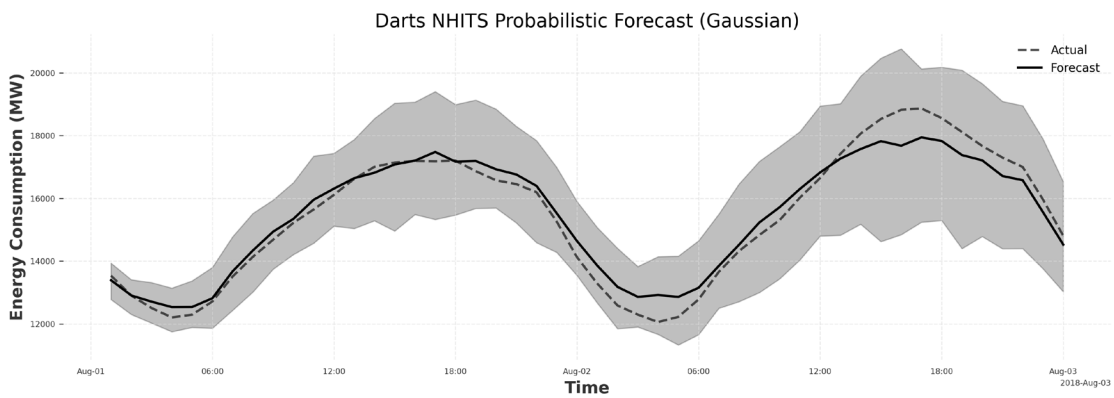

Figure 15.9: NHiTS with the GaussianLikelihood estimator

Finally, let's evaluate the overall performance:

```
loss_p05 = mql(d_test, nhits_forecast_gauss_inv, q=0.05)
loss_p50 = mql(d_test, nhits_forecast_gauss_inv, q=0.50)
loss_p95 = mql(d_test, nhits_forecast_gauss_inv, q=0.95)

print(f"Gaussian Likelihood Average Quantile Loss at 5th percentile: "
      f"{loss_p05:.4f}")
print(f"Gaussian Likelihood Average Quantile Loss at 50th percentile (MAE): "
      f"{loss_p50:.4f}")
print(f"Gaussian Likelihood Average Quantile Loss at 95th percentile: "
      f"{loss_p95:.4f}")
>>
Gaussian Likelihood Average Quantile Loss at 5th percentile:  162.3221
Gaussian Likelihood Average Quantile Loss at 50th percentile (MAE): 357.5733
Gaussian Likelihood Average Quantile Loss at 95th percentile: 168.2610
```

Notice the results: the median forecast accuracy (the 50th percentile loss, or MAE) is at 376.39, which is the best achieved thus far compared to other techniques used in this recipe. This suggests that the Gaussian likelihood assumption aligns well with this dataset's characteristics, leading to a more accurate central tendency.

When choosing between `GaussianLikelihood` and `QuantileRegression`, consider the nature of your forecast errors (residuals). If you believe the residuals of your model will be approximately normally distributed, `GaussianLikelihood` may provide better results and is more parameter-efficient. However, if your data has skewed distributions, heavy tails, or multiple modes, `QuantileRegression` is likely to capture these characteristics better since it makes no assumptions about the underlying distribution (non-parametric).

Lastly, if you are interested in getting the estimated distribution parameters, such as the mean and standard deviation of the GaussianLikelihood or the quantiles from `QuantileRegression`, you can use the `predict_likelihood_parameters=True` argument in the `predict` method. Here is an illustration:

```
# Params for GaussianLikelihood
forecast_params_gauss = nhits_d_model_gauss.predict(
    n=len(d_test),
    series=d_train_val_sc,
    predict_likelihood_parameters=True
)
print(forecast_params_gauss.to_dataframe().head())
>>
                         y_mu   y_sigma
Datetime
2018-08-01 01:00:00 -0.805166  0.140893
```

```
2018-08-01 02:00:00 -1.027776  0.140102
2018-08-01 03:00:00 -1.109318  0.152793
2018-08-01 04:00:00 -1.173990  0.161144
2018-08-01 05:00:00 -1.124817  0.188276


# Params for QuantileRegression
forecast_params_qr = nhits_d_model_qr.predict(
    n=len(d_test),
    series=d_train_val_sc,
    predict_likelihood_parameters=True
)
print(forecast_params_qr.to_dataframe().head())
>>
                     y_q0.05   y_q0.50   y_q0.95
Datetime
2018-08-01 01:00:00 -0.801715 -0.703966 -0.606822
2018-08-01 02:00:00 -1.037714 -0.919474 -0.807610
2018-08-01 03:00:00 -1.174309 -1.026826 -0.900245
2018-08-01 04:00:00 -1.234250 -1.069475 -0.898426
2018-08-01 05:00:00 -1.202631 -1.018375 -0.814162
```

To put it all together, the following table summarizes how different approaches compare on our test set:

Method	Median MAE (50th Quantile Loss)	5th Quantile Loss	95th Quantile Loss	Primary Uncertainty Captured
Point Forecast	516.37	N/A	N/A	None
MC Dropout	505.80	297.98	241.08	Epistemic (Model)
Quantile Regression (Likelihood)	469.37	100.89	148.76	Aleatoric (Data)
QR + MC Dropout	450.67	100.07	152.08	Both
Gaussian (Likelihood)	357.57	162.32	168.26	Aleatoric (Data)

Table 16.1: Summary of the different probabilistic forecasting approaches using Darts from the recipe

See also

- To learn more about Darts' probabilistic forecasting capabilities, you should start with this dedicated page in their documentation: `https://unit8co.github.io/darts/quickstart/00-quickstart.html#Probabilistic-forecasts`

Probabilistic forecasting with NeuralForecast

In previous chapters, you explored Nixtla's libraries: **StatsForecast** (*Chapter 11* and *Chapter 15*), for statistical models, and **NeuralForecast** (*Chapter 13*), for neural-based time-series forecasts, such as LSTM, NHiTS, and TCN.

In the first recipe of this chapter, you explored StatsForecast probabilistic modeling with prediction intervals. In this recipe, we will explore NeuralForecast's N-HiTS with the **Multi-Quantile Loss** (**MQLoss**) function.

Getting ready

You can install NeuralForecast using `pip`:

```
pip install neuralforecast
```

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.

To implement probabilistic forecasting with NeuralForecast, follow these steps:

1. Start by importing the necessary components from the library:

```
from neuralforecast import NeuralForecast
from neuralforecast.losses.pytorch import MQLoss
from neuralforecast.models import  NHITS
```

2. Prepare the DataFrame in the format that NeuralForecast expects. Rename the `Datetime` column to `ds`. The target column is already `y`. Add a `unique_id` columns to identify the time series:

```
nf_df = df.reset_index().copy()
nf_df = nf_df.rename(columns={'Datetime': 'ds'})
nf_df['unique_id'] = 'AEP_hourly'
```

3. We will use a similar split as we did in the Darts recipe, by holding out the last two days (48 hours) as our test set:

```
day = 24
week = day*7

n_test = day*2
n_val = week*2

nf_train = nf_df.iloc[:-n_test]
nf_test = nf_df.iloc[-n_test:]

print(nf_train.shape)
```

```
print(nf_test.shape)
>>
(121248, 3)
(48, 3)
```

4. In Darts, when using `QuantileRegression` as a likelihood estimator, we provided our quantiles as `[0.05, 0.5, 0.95]`. In NeuralForecast, we can provide **quantiles** or **levels**. Quantiles are the actual fractional values between 0 and 1 that represent the quantiles of the prediction distribution you want to estimate. Levels are expressed as percentages that represent the prediction interval coverage, for example, 90%, which corresponds to the quantiles 0.05 and 0.95 (for a two-sided 90% prediction interval). We will provide three prediction interval levels (80%, 90%, and 95%) for the model to learn. The 80% and 95% intervals will be used primarily for plotting visualizations, while the 90% interval will be used to calculate the quantile loss metric:

```
levels = [80, 90, 95]

nhits = NHITS(
    input_size=day*5,
    h=day*2,
    loss=MQLoss(level=levels),
    valid_loss=MQLoss(level=levels)
)
```

We provided `loss` and `valid_loss`, where the `loss` parameter specifies the training loss function, while `valid_loss` defines the validation loss function used when validation data is provided during training.

5. Create an instance of NeuralForecast and fit the model, providing a validation size of two weeks (`n_val`):

```
nf_model = NeuralForecast(models=[nhits], freq='h')
nf_model.fit(df=nf_train, val_size=n_val)
```

6. Once training is completed, you can create the forecast and then plot the forecast against the actual (the test set) with the 80th and 95th prediction intervals:

```
nf_forecast = nf_model.predict()
```

Visualize the NHITS forecast with prediction intervals using the `nf_plot_forecast` utility function:

```
from utils import nf_plot_forecast
nf_plot_forecast(nf_forecast,
                 actuals_df=nf_test,
                 point_forecast='NHITS-median',
                 title='NHITS Forecast vs Actuals with MQLoss Prediction
Intervals')
```

This should produce the following plot:

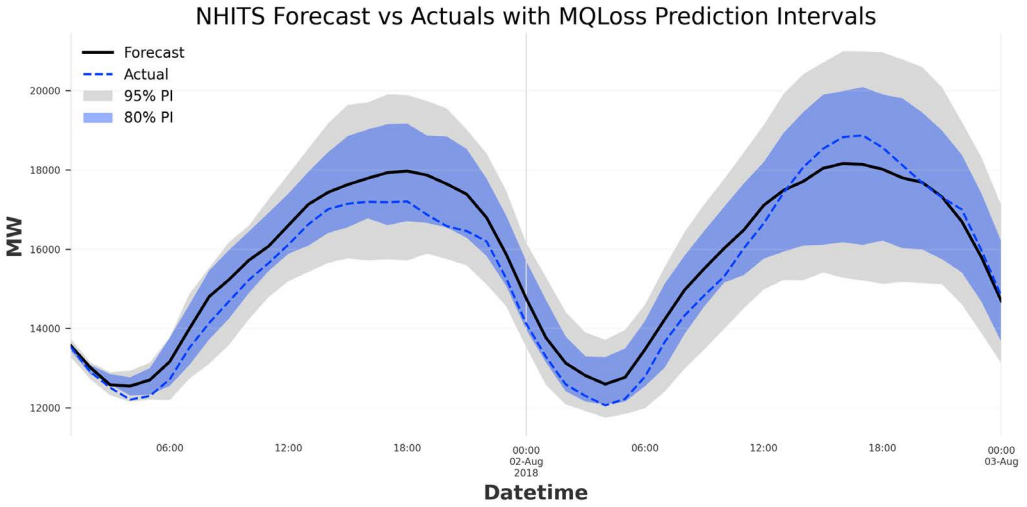

Figure 15.10: NeuralForecast N-HiTS with 80th and 95th prediction intervals using MQLoss

7. To fairly evaluate our probabilistic forecast and compare it to Darts, we'll calculate the quantile loss (also known as the pinball loss) in a similar way to how Darts have implemented it in their source code. We'll create our own `quantile_loss` function based on Darts' `mql` implementation, in which Darts multiplies the standard pinball loss by a factor of 2:

```
import numpy as np

# Define the quantile loss function
def quantile_loss(y_true, y_pred, q):
    """
    This implementation matches the Darts library, where the result is
scaled by 2
    so that the loss for the median (q=0.5) is equivalent to the MAE
    """
    errors = y_true - y_pred
    losses = 2.0 * np.maximum((q - 1) * errors, q * errors)
    return np.mean(losses)

results_df = pd.merge(nf_test, nf_forecast, on='ds')

# Extract the actual values
y_true = results_df['y']

# Extract the predicted quantile values
```

```
y_pred_p05 = results_df['NHITS-lo-90']
y_pred_p50 = results_df['NHITS-median']
y_pred_p95 = results_df['NHITS-hi-90']

# Calculate the loss for each quantile
loss_p05 = quantile_loss(y_true, y_pred_p05, q=0.05)
loss_p50 = quantile_loss(y_true, y_pred_p50, q=0.50)
loss_p95 = quantile_loss(y_true, y_pred_p95, q=0.95)

print(f"NeuralForecast Average Quantile Loss with MQLoss at "
      f"5th percentile:  {loss_p05:.4f}")
print(f"NeuralForecast Average Quantile Loss with MQLoss at "
      f"50th percentile (MAE): {loss_p50:.4f}")
print(f"NeuralForecast Average Quantile Loss with MQLoss at "
      f"95th percentile: {loss_p95:.4f}")
>>
NeuralForecast Average Quantile Loss with MQLoss at 5th percentile:
106.4772
NeuralForecast Average Quantile Loss with MQLoss at 50th percentile
(MAE): 489.1147
NeuralForecast Average Quantile Loss with MQLoss at 95th percentile:
167.7960
```

How it works...

In the recipe, we used quantile regression through MQLoss as our loss function for both training and validation. This changes the model's objective, and instead of training the NHITS model to predict the single best values (e.g., mean), we are training it to predict multiple values at once:

- The lower bound of the 80% prediction interval (10^{th} percentile)
- The upper bound of the 80% prediction interval (90^{th} percentile)
- The lower bound of the 90% prediction interval (5^{th} percentile)
- The upper bound of the 90% prediction interval (95^{th} percentile)
- The lower bound of the 95% prediction interval (2.5^{th} percentile)
- The upper bound of the 95% prediction interval (97.5^{th} percentile)
- The median (50^{th} percentile) as the point forecast

We can view this by inspecting the resulting DataFrame:

```
nf_forecast.columns
>>
Index(['ds', 'unique_id', 'NHITS-median', 'NHITS-lo-95', 'NHITS-lo-90',
       'NHITS-lo-80', 'NHITS-hi-80', 'NHITS-hi-90', 'NHITS-hi-95'],
      dtype='object')
```

This is similar to what we did in Darts when we used `QuantileRegression` as the likelihood to estimate the **aleatoric** uncertainty. The MQLoss function scores the model based on how well all of its quantile predictions match the actual historical data.

There's more...

In the recipe, we used MQLoss as our loss function, which changes the model's objective during training to directly predict specific quantiles of the future distribution (instead of a point forecast). NeuralForecast provides another approach to probabilistic modeling through CP, where the model is trained to produce a point forecast, followed by a separate post-processing step to calculate the prediction intervals based on the model's past errors on the validation set.

Let's start by importing the necessary modules:

```
from neuralforecast.utils import PredictionIntervals
from neuralforecast.losses.pytorch import MAE
prediction_intervals = PredictionIntervals(method='conformal_distribution')
```

Let's define our model, which will be trained to predict a point forecast using the MAE as the training and validation loss function:

```
nhits_cp = NHITS(
    input_size=day*5,
    h=day*2,
    loss=MAE(),
    valid_loss=MAE()
)
```

Now, let's fit the model:

```
nf_cp_model = NeuralForecast(models=[nhits_cp], freq='h')
nf_cp_model.fit(df=nf_train, val_size=n_val,
                prediction_intervals=prediction_intervals)
```

Unlike using the MQLoss and supplying our levels (quantiles), CP does not change the model; instead, it uses the residuals from the validation set to "calibrate" prediction intervals. CP does not make any assumptions about the distribution of the intervals; it doesn't assume Gaussian errors and doesn't require the model to learn quantiles. CP uses past errors to guarantee nominal coverage on average across forecasts. This means we don't need to retrain the model if we want to add new coverage levels since this is done at prediction time.

For example, assume you have a weather forecasting model trained to predict just one temperature, such as 70°F (point forecast). CP is like an editor who reviews the forecaster's past performance on the validation set. The editor collects the size of all the past errors (e.g., "it was off by 2," "it was off by 5," "it was off by 1"). In order to create a 95% prediction interval, the editor simply finds the 95th percentile of those error sizes. Let's say that value is 5 degrees.

This 5 degree value becomes the "margin of error." Now, when the forecaster predicts a new temperature of 75°F, the editor "calibrates" it by adding and subtracting that 5-degree margin, giving a final interval of 70°F to 80°F. The key point here is that we didn't need to retrain the original forecaster (the model); we just used its past mistakes to build a more reliable, data-driven margin of error.

Coverage here refers to the proportion of time that the actual future value (e.g., our test data value) falls inside the prediction interval. This is in contrast to the quantile regression (MQLoss), where coverage is not guaranteed.

Generate the forecast using the calibrated model, specifying the desired coverage

levels (80%, 90%, and 95%):

```
nf_cp_forecast = nf_cp_model.predict(level=[80, 90, 95])
nf_cp_forecast.columns
>>
Index(['unique_id', 'NHITS', 'NHITS-lo-95', 'NHITS-lo-90', 'NHITS-lo-80',
       'NHITS-hi-80', 'NHITS-hi-90', 'NHITS-hi-95'],
      dtype='object')
```

We will use the same function for plotting:

```
nf_plot_forecast(
    nf_cp_forecast,
    actuals_df=nf_test,
    point_forecast='NHITS',
    title='NHITS Forecast vs Actuals with Conformal Prediction Intervals'
)
```

This should produce the following plot:

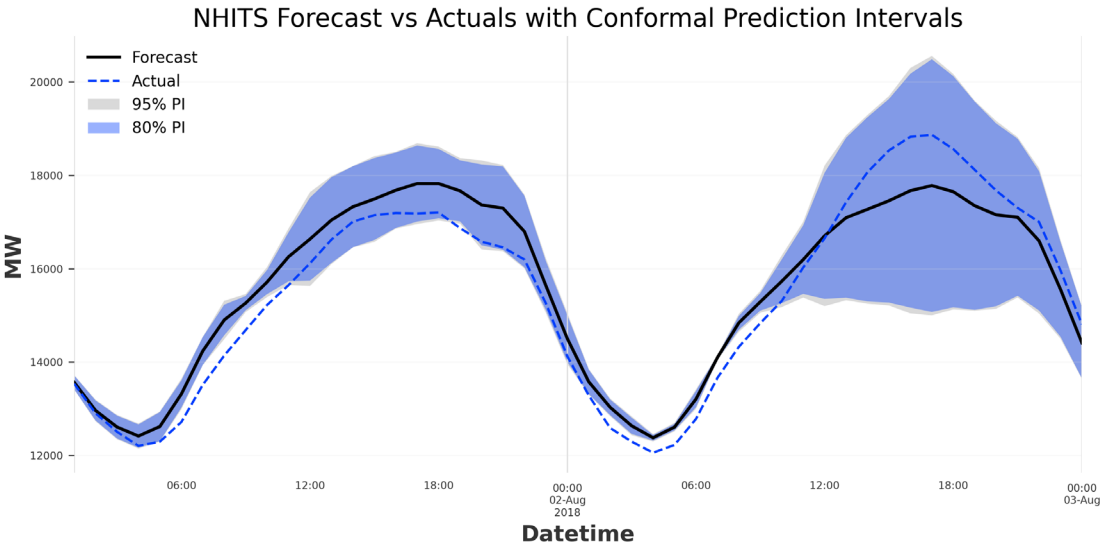

Figure 15.11: NeuralForecast N-HiTS with 80th and 95th prediction intervals using CP

Notice the intervals widen significantly during peaks (afternoon/evening). This widening reflects the fact that CP intervals depend on past residuals, so when the model struggles, the CP compensates by making the intervals widen (inflated).

Finally, let's evaluate our model's performance using the `quantile_loss` function:

```
results_cp_df = pd.merge(nf_test, nf_cp_forecast, on='ds')

# Extract the predicted quantile values
y_pred_cp_p05 = results_cp_df['NHITS-lo-90']
y_pred_cp_p50 = results_cp_df['NHITS']
y_pred_cp_p95 = results_cp_df['NHITS-hi-90']

# Extract the actual values
y_true = results_df['y']

# Calculate the loss for each quantile
loss_cp_p05 = quantile_loss(y_true, y_pred_cp_p05, q=0.05)
loss_cp_p50 = quantile_loss(y_true, y_pred_cp_p50, q=0.50)
loss_cp_p95 = quantile_loss(y_true, y_pred_cp_p95, q=0.95)

print(f"NeuralForecast Average Quantile Loss with CP at 5th percentile: "
      f"{loss_cp_p05:.4f}")
```

```
print(f"NeuralForecast Average Quantile Loss with CP at 50th percentile (MAE):"
      f"{loss_cp_p50:.4f}")
print(f"NeuralForecast Average Quantile Loss with CP at 95th percentile: "
      f"{loss_cp_p95:.4f}")
>>
NeuralForecast Average Quantile Loss with CP at 5th percentile:  235.2145
NeuralForecast Average Quantile Loss with CP at 50th percentile (MAE): 498.5925
NeuralForecast Average Quantile Loss with CP at 95th percentile: 101.9440
```

The key difference between the MQLoss and CP is that the MQLoss changes the model to learn quantiles during training. Instead of a single best guess (mean), the model is trained to produce multiple conditional quantiles. This means the model internally learns the distribution of possible futures, not just a point. In contrast, in CP (a post hoc technique), you train a standard point-forecasting model, and then CP uses the validation errors to build **calibrated intervals** around those predictions.

CP does not improve the model itself; it improves the intervals. If the point forecasts are biased, CP intervals will widen to compensate. Coverage in CP means that, across many forecasts, a 90% interval will contain the true value about 90% of the time. It does not guarantee inclusion for every single forecast (CP guarantees coverage on average across many forecasts, not per individual forecast). CP can be applied to any point-forecasting model and is especially valuable when error distributions are non-Gaussian or heteroscedastic.

See also

- To learn more about NeuralForecast's probabilistic forecasting, visit their documentation page here: `https://nixtlaverse.nixtla.io/neuralforecast/docs/tutorials/uncertainty_quantification.html`

Probabilistic forecasting with NeuralProphet

In *Chapter 15*, you explored `NeuralProphet` for handling multiple seasonalities. Now, you will explore `NeuralProphet` again, on the same dataset and a similar setup, to explore probabilistic modeling and uncertainty quantification. In previous recipes, you implemented `QuantileRegression` using Darts and `NeuralForecast`. In this recipe, you will learn how to generate prediction intervals using quantile regression in `NeuralProphet`.

Getting ready

You will need to install `NeuralProphet`:

```
pip install neuralprophet plotly-resampler ipywidgets
```

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.

To implement probabilistic forecasting with quantile regression in `NeuralProphet`,

follow these steps:

1. Start by importing the necessary components from the `NeuralProphet` library and setting a random seed for reproducibility:

```
from neuralprophet import NeuralProphet, set_random_seed
set_random_seed(seed=42)
```

2. Prepare the dataset for `NeuralProphet's` expected format. Rename the `Datetime` column to `ds`, and split the data into training, validation, and test sets. For consistency with previous recipes, we will hold the last two days (48 hours) for the test set and the preceding two weeks for validation:

```
df_np = df.copy()
df_np.reset_index(inplace=True)
df_np.rename(columns={'Datetime': 'ds'}, inplace=True)

day = 24
week = day*7

n_total = len(df_np)
n_test = day*2
n_val = week*2

# Calculate split indices
test_start = n_total - n_test
val_start = test_start - n_val

# Create splits maintaining temporal order
train_np = df_np[:val_start]
val_np= df_np[val_start:test_start]
test_np = df_np[test_start:]

print(f"""
    train_np : {len(train_np)}
    val_np : {len(val_np)}
    test_np: {len(test_np)}
```

```
""")
>>
train_np : 120912
val_np : 336
test_np: 48
```

3. Instantiate the `NeuralProphet` model. To enable quantile regression, pass a list of desired quantiles to the `quantiles` parameter. Here, we will request the 5th and 95th quantiles to construct 90% prediction intervals. `NeuralProphet` will return the median (50th) by default as `yhat1`:

```
quantile_list = [0.05, 0.95]
m = NeuralProphet(quantiles=quantile_list)
Fit the model on the training set and provide the validation set:
m.set_plotting_backend("plotly-static")
metrics = m.fit(train_np, validation_df=val_np, freq="H")
```

4. Once training is complete, use the `predict()` method on the trained model, `m`, to generate forecasts for the test period. `NeuralProphet` requires a DataFrame with a `ds` column containing the timestamps for the forecast horizon. It will use these timestamps to generate predictions and will ignore the `y` column if present:

```
forecast = m.predict(df=test_np)
forecast.columns
>>
Index(['ds', 'y', 'yhat1', 'yhat1 5.0%', 'yhat1 95.0%', 'trend',
       'season_yearly', 'season_weekly', 'season_daily'],
      dtype='object')
```

   The output now includes columns for the lower (`yhat1 5.0%`) and upper (`yhat1 95.0%`) bounds of our prediction interval. The `yhat1` column represents the median forecast (50th percentile).

5. You can visualize the forecast and its uncertainty interval using the `m.plot()` method:

```
m.plot(forecast)
```

The resulting visualization displays the forecast with prediction intervals:

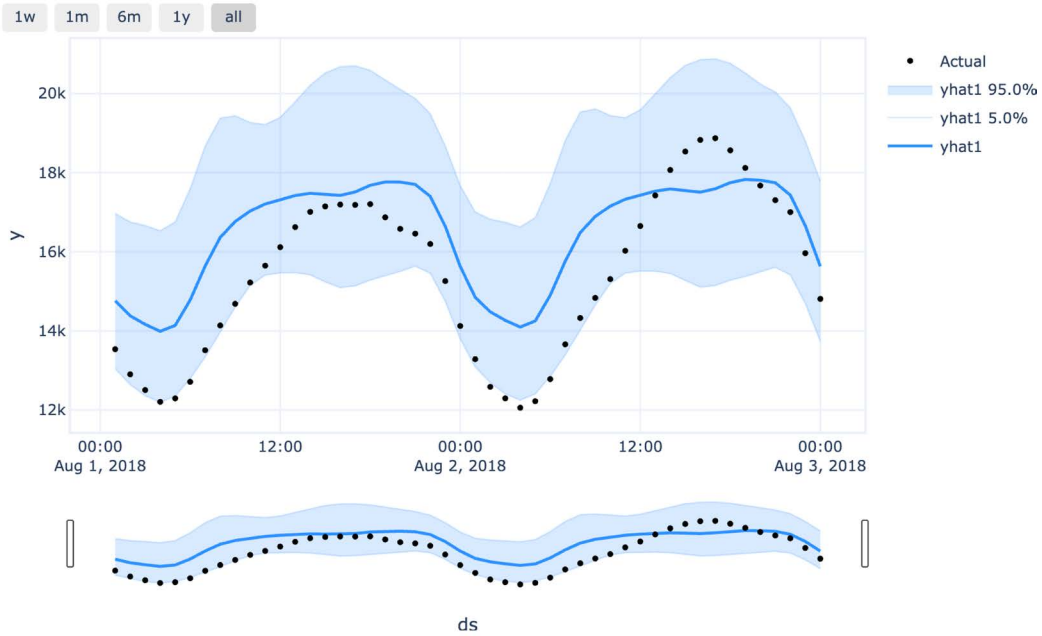

Figure 15.12: Predicted time series with NeuralProphet with quantile regression

The plot shows the 48-hour forecast. The solid line (`yhat1`) is the median prediction, while the shaded area represents the 90% prediction interval between the 5th and 95th percentiles. The black dots show the actual energy consumption values from the test set.

6. To fairly evaluate our probabilistic forecast and compare it to Darts and NeuralForecast, we'll calculate the quantile loss (also known as the pinball loss) in a similar way to how Darts have implemented it in their source code. We will reuse the `quantile_loss` function from the previous recipe:

```
def quantile_loss(y_true, y_pred, q):
    """
    This implementation matches the Darts library, where the result is
scaled by 2
    so that the loss for the median (q=0.5) is equivalent to the MAE
    """
    errors = y_true - y_pred
    losses = 2.0 * np.maximum((q - 1) * errors, q * errors)
    return np.mean(losses)

# Extract the predicted quantile values
y_pred_qr_p05 = forecast['yhat1 5.0%']
y_pred_qr_p50 = forecast['yhat1']
```

```
y_pred_qr_p95 = forecast['yhat1 95.0%']

# Extract the actual values
y_true = forecast['y']

# Calculate the loss for each quantile
loss_qr_p05 = quantile_loss(y_true, y_pred_qr_p05, q=0.05)
loss_qr_p50 = quantile_loss(y_true, y_pred_qr_p50, q=0.50)
loss_qr_p95 = quantile_loss(y_true, y_pred_qr_p95, q=0.95)

print(f"NeuralProphet Average Quantile Loss with QR at "
      f"5th percentile:  {loss_qr_p05:.4f}")
print(f"NeuralProphet Average Quantile Loss with QR at "
      f"50th percentile (MAE): {loss_qr_p50:.4f}")
print(f"NeuralProphet Average Quantile Loss with QR at "
      f"95th percentile: {loss_qr_p95:.4f}")
>>
NeuralProphet Average Quantile Loss with QR at 5th percentile:  135.3577
NeuralProphet Average Quantile Loss with QR at 50th percentile (MAE):
1265.6756
NeuralProphet Average Quantile Loss with QR at 95th percentile: 361.3481
```

The quantile loss values for `NeuralProphet` are higher than those from Darts and NeuralForecast in previous recipes. This is primarily because we used the model's default configuration. `NeuralProphet's` performance can be significantly improved by tuning hyperparameters such as the following:

- `n_lags`: Controls the autoregressive component
- `n_forecasts`: Affects how many steps ahead are predicted simultaneously
- `learning_rate`: Influences training convergence speed
- `epochs`: Determines how long the model trains

7. While the quantile loss tells us about the magnitude of the errors, it doesn't tell us how often our intervals actually captured the true value. For this, we can calculate the **empirical coverage**. Calculating the metric is straightforward; it's the percentage of time the actual value fell within our prediction interval. For a 90% prediction interval (between the 5th and 95th percentiles), we expect a coverage value close to 0.90:

```
coverage_90_qr = ((y_true >= y_pred_qr_p05) & (y_true <= y_pred_qr_p95)).
mean()
print(f"Neural Prophet QR Empirical Coverage (90%): "
      f"{coverage_90_qr:.3f}")
>>
Neural Prophet QR Empirical Coverage (90%): 0.854
```

A coverage of 0.854 means our intervals were a little too narrow, only capturing the true value 85% of the time, which is not quite the 90% we were aiming for, but pretty close. This shows that our model's uncertainty estimate can be further calibrated.

How it works...

`NeuralProphet` builds on the foundational structure of Prophet by modeling a time series as a sum of components such as trend, multiple seasonalities, and autoregressive terms (via the `n_lags` parameter). Unlike Prophet, it uses a neural network backbone to capture complex nonlinearities and interactions.

In this recipe, `NeuralProphet` generates probabilistic forecasts using quantile regression. Instead of training the model to predict a single future value by minimizing the mean squared error, the model's training objective changes from minimizing the mean squared error to minimizing the quantile loss (or pinball loss). By setting the `quantiles` parameter, the model is now trained to predict multiple conditional quantiles simultaneously (for example, the 5^{th}, 50^{th}, and 95^{th} percentiles). It does this by doing the following:

- Modifying the neural network architecture to have multiple output nodes (one for each quantile)
- Applying the appropriate quantile loss function during training for each output
- Optimizing the network parameters to minimize these quantile-specific losses

This approach produces a set of forecasts for each percentile of the predicted distribution, which collectively form prediction intervals to express the model's uncertainty about future values.

There's more...

In the previous recipe (using NeuralForecast), you explored how CP can generate intervals for a point forecast model. In this section, you will explore the CQR, which you can think of as a hybrid approach.

In the CQR, we start with the quantile regression by specifying the quantiles in the model definition. This influences how the model is trained to produce multiple outputs (forecasts) for each quantile. The CQR further adjusts the quantile intervals using residuals on a calibration (validation) set, yielding calibrated prediction intervals with formal coverage guarantees. This post-processing step improves uncertainty quantification by accounting for model mis-specification or distributional shifts without retraining.

Put simply, the CQR takes the prediction intervals generated from the quantile regression and uses a calibration set to calibrate them, ensuring they are reliable and statistically robust.

To use the CQR, we need to reserve a separate chunk of data for calibration. Our timeline will now be split into four sequential blocks: *training*, *validation*, *calibration*, and, finally, *testing*. The model is trained on the training set, its hyperparameters could be tuned on the validation set, its intervals are then calibrated on the calibration set, and its final performance is measured on the unseen test set.

We will split our data to include train, test, calibration, and validation sets:

```
day = 24
week = day*7

n_total = len(df_np)

# Define sizes for each set
n_test = day * 2      # 2 days for final testing
n_calib =  week * 3     # 1 week for calibration
n_val = week * 2       # 1 week for validation
n_train = n_total - n_test - n_calib - n_val

# Create the splits chronologically
train_df = df_np[:n_train]
val_df = df_np[n_train : n_train + n_val]
calib_df = df_np[n_train + n_val : n_train + n_val + n_calib]
test_df = df_np[n_train + n_val + n_calib:]


print(f"""
    train_df : {len(train_df)}
    val_df : {len(val_df)}
    test_df: {len(test_df)}
    calib_df {len(calib_df)}
""")
>>
train_df : 120408
val_df : 336
test_df: 48
calib_df 504
```

Now, we will use the `conformal_predict()` method:

```
method = "cqr"
cqr_forecast = m.conformal_predict(
    test_np,
    calibration_df=calib_df,
    alpha=0.1,
    method=method,
    plotting_backend="plotly-static"
)
```

Notice that with the CQR (and the CP in general), there is no need to retrain the model. The `alpha` parameter represents the significance level that you would specify for the prediction intervals. Setting `alpha=0.10` means the following:

- We want a 90% prediction interval (meaning 90% coverage)
- This implies that we accept that about 10% of the future observations might fall outside these intervals
- The 10% error is split equally (by default) between the lower and upper tails of the forecast distribution (5% error allowance below the lower bound and 5% above the upper bound)

Generally, a smaller `alpha` value would give us wider intervals with higher confidence, and a larger `alpha` value would give us narrower intervals but increase the risk of missing true values.

You can plot `cqr_foreacst` using the `m.plot()` method:

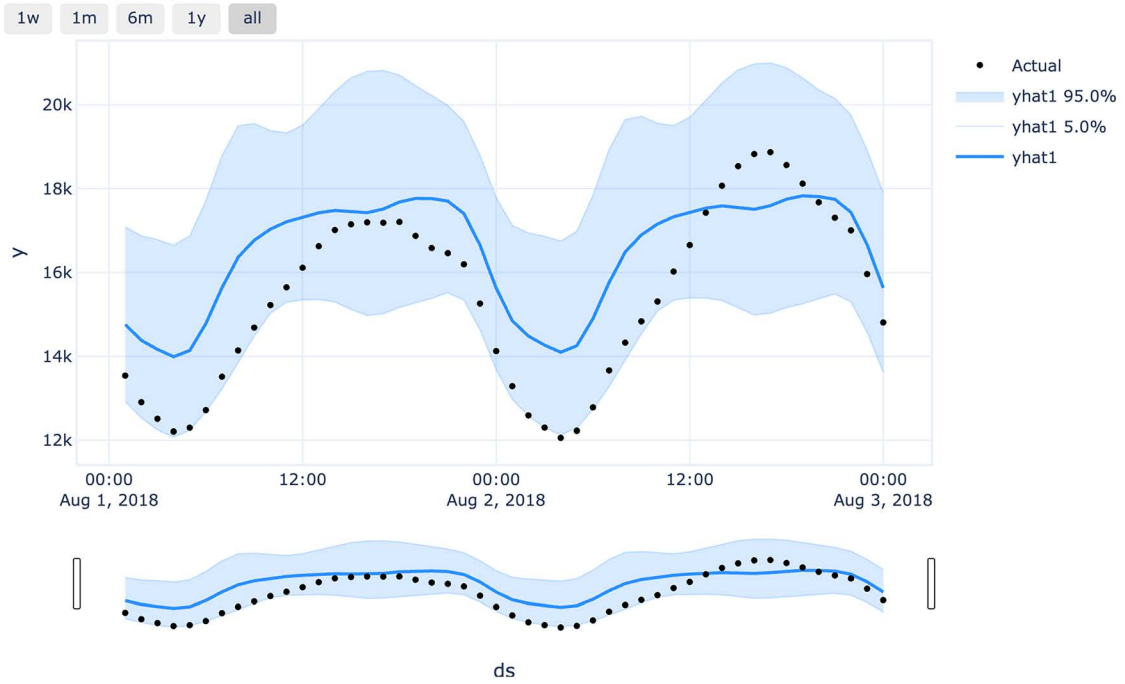

Figure 15.13: CQR corrected prediction intervals from NeuralProphet

Let's evaluate the resulting CQR forecast using our `quantile_loss` function:

```
# Extract the predicted quantile values
y_pred_cp_p05 = cqr_forecast['yhat1 5.0%']
y_pred_cp_p50 = cqr_forecast['yhat1']
y_pred_cp_p95 = cqr_forecast['yhat1 95.0%']

# Extract the actual values
```

```
y_true = cqr_forecast['y']

# Calculate the loss for each quantile
loss_cp_p05 = quantile_loss(y_true, y_pred_cp_p05, q=0.05)
loss_cp_p50 = quantile_loss(y_true, y_pred_cp_p50, q=0.50)
loss_cp_p95 = quantile_loss(y_true, y_pred_cp_p95, q=0.95)

print(f"NeuralProphet Average Quantile Loss with CQR at 5th percentile: "
      f"{loss_cp_p05:.4f}")
print(f"NeuralProphet Average Quantile Loss with CQR at 50th percentile (MAE):
"
      f"{loss_cp_p50:.4f}")
print(f"NeuralProphet Average Quantile Loss with CQR at 95th percentile: "
      f"{loss_cp_p95:.4f}")
>>
NeuralProphet Average Quantile Loss with CQR at 5th percentile:  119.7666
NeuralProphet Average Quantile Loss with CQR at 50th percentile (MAE):
1265.6756
NeuralProphet Average Quantile Loss with CQR at 95th percentile: 373.2079
```

The CQR forecast DataFrame (`cqr_forecast`) will have the same `yhat1` (median) values as the original forecast DataFrame. CP only adjusts the intervals (`yhat1 5.0%` and `yhat1 95.0%`), not the point forecast. This is why the `loss_cp_p50` value is identical to `loss_qr_p50`.

Finally, let's compute the empirical coverage:

```
coverage_90_cqr = ((y_true >= y_pred_cp_p05) &
                   (y_true <= y_pred_cp_p95)).mean()
print(f"Neural Prophet CQR Empirical Coverage (90%): "
      f"{coverage_90_cqr:.3f}")
>>
Neural Prophet CQR Empirical Coverage (90%): 0.958
```

Notice our coverage went from 0.854 to 0.958. This is the effect of the CQR. Our original quantile regression model was clearly **miscalibrated** and a bit overconfident (its 90% intervals were too narrow), only capturing the true value 85.4% of the time. The CQR automatically fixed this. It used the model's mistakes on the calibration set (`calib_df`) and calculated the adjustment needed for the intervals. This resulted in widening the original intervals. This result is a more reliable forecast that honors the 90% coverage (and even slightly exceeds it at 95.8%).

This brings up a crucial practical tip based on experiments (which you can explore by changing the calibration set's size): the size of your calibration set matters. If the calibration set is too small, the model gets a poor estimate of its own errors, and the final coverage can be less reliable. You want a set that's large enough to be representative of the kinds of errors your model typically makes. It's a trade-off, but sacrificing a bit more data for calibration often pays for itself in more trustworthy intervals.

Lastly, the **quantile loss** provides a standardized way of evaluating probabilistic forecasts. It's a metric that requires careful interpretation. `NeuralProphet's` performance depends strongly on model configuration (e.g., autoregressive lags, seasonalities). Also, conformal methods such as the CQR may increase the interval width for better uncertainty coverage, which can raise loss values without indicating worse forecast quality. Consider using multiple metrics and visual inspection alongside the quantile loss.

See also

- To learn more about `NeuralProphet's` probabilistic forecasting and uncertainty quantification, you can visit their official documentation here: `https://neuralprophet.com/how-to-guides/feature-guides/uncertainty_quantification.html`