In this chapter, we briefly discuss four more advanced forecasting methods that build on the models discussed in earlier chapters.

12.1 Complex seasonality

So far, we have mostly considered relatively simple seasonal patterns such as quarterly and monthly data. However, higher frequency time series often exhibit more complicated seasonal patterns. For example, daily data may have a weekly pattern as well as an annual pattern. Hourly data usually has three types of seasonality: a daily pattern, a weekly pattern, and an annual pattern. Even weekly data can be challenging to forecast as there are not a whole number of weeks in a year, so the annual pattern has a seasonal period of 365.25/7\approx 52.179 on average. Most of the methods we have considered so far are unable to deal with these seasonal complexities.

We don't necessarily want to include all of the possible seasonal periods in our models — just the ones that are likely to be present in the data. For example, if we have only 180 days of data, we may ignore the annual seasonality. If the data are measurements of a natural phenomenon (e.g., temperature), we can probably safely ignore any weekly seasonality.

Figure 12.1 shows the number of calls to a North American commercial bank per 5-minute interval between 7:00am and 9:05pm each weekday over a 33 week period. Figure 12.2 shows the first four weeks of the same time series. There is a strong daily seasonal pattern with period 169 (there are 169 5-minute intervals per day), and a weak weekly seasonal pattern with period 169 \times 5=845. (Call volumes on Mondays tend to be higher than the rest of the week.) If a longer series of data were available, we may also have observed an annual seasonal pattern.

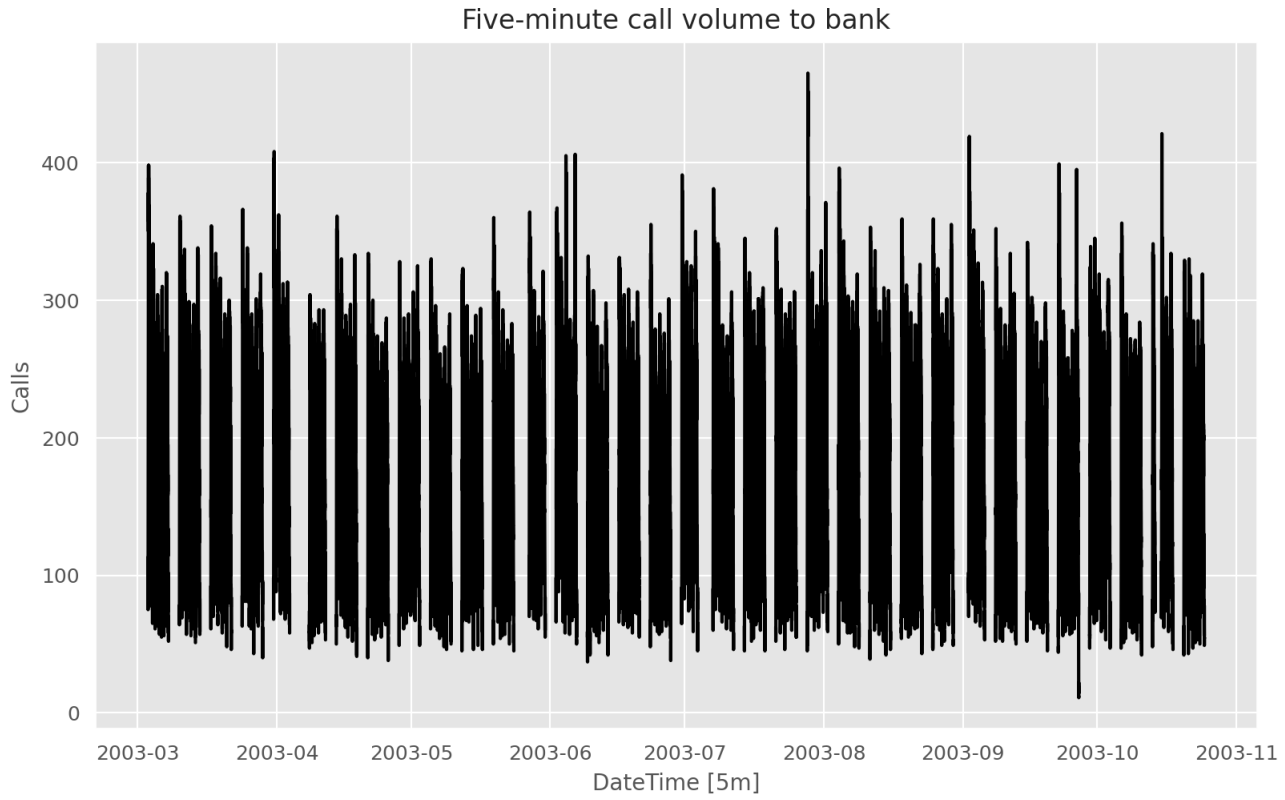

Figure 12.1: Five-minute call volume handled on weekdays between 7:00am and 9:05pm in a large North American commercial bank. Data from 3 March – 24 October 2003.

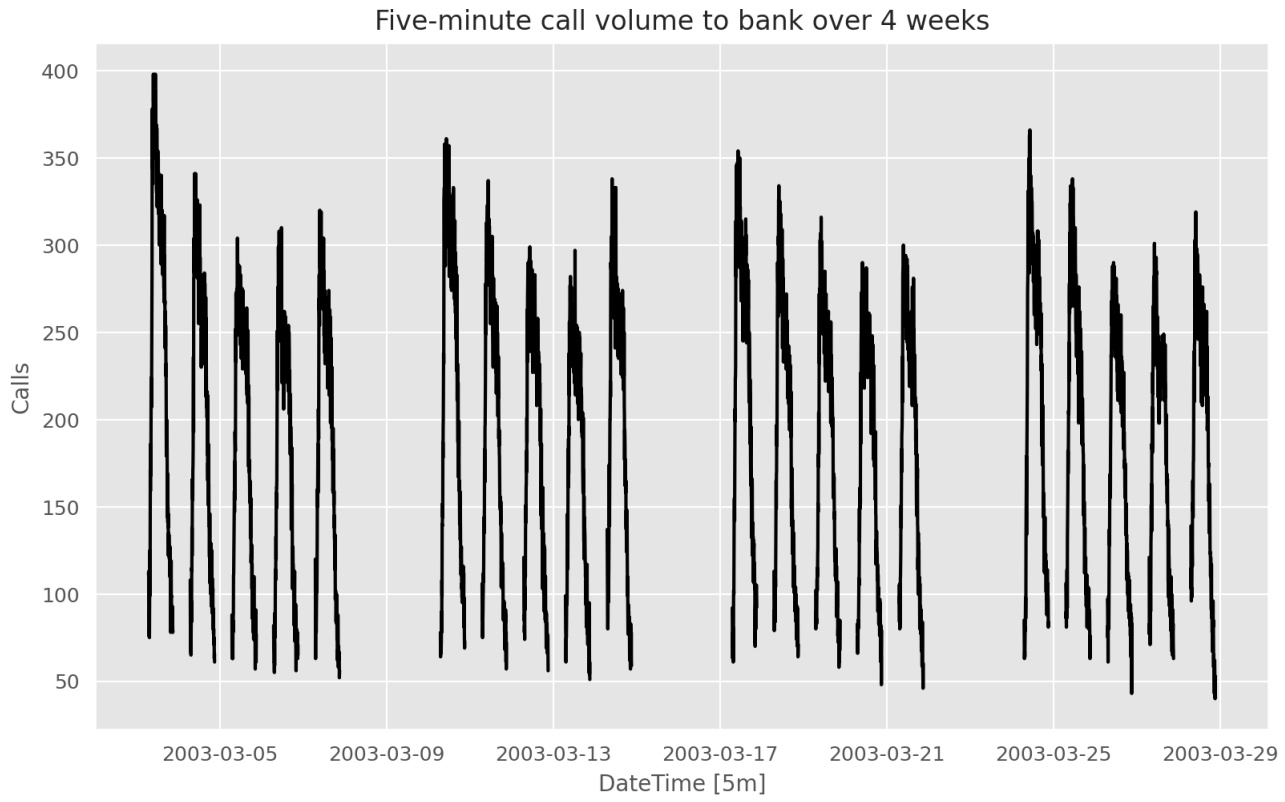

Figure 12.2: First four weeks of data from the North American commercial bank.

Apart from the multiple seasonal periods, this series has the additional complexity of missing values between the working periods.

MSTL with multiple seasonal periods

The `MSTL()` function is designed to deal with multiple seasonalities, and it can be used to decompose a series into its trend, multiple seasonal, and remainder components.

```
sf = StatsForecast(models=[MSTL(season_length=[169, 169 * 5])], freq="5min")

sf = sf.fit(bank_calls_or.assign(y = lambda df: np.sqrt(df["y"])))
dcmp = sf.fitted_[0, 0].model_

fig, axes = plt.subplots(nrows=5, ncols=1, sharex=True)
sns.lineplot(data=dcmp, x=dcmp.index, y="data", ax=axes[0], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="trend", ax=axes[1], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="seasonal169", ax=axes[2], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="seasonal845", ax=axes[3], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="remainder", ax=axes[4], color="black")
fig.suptitle("STL decomposition")
fig.text(0.5, 0.94, "sqrt(Calls) = trend + seasonal169 + seasonal845 + remainder",
         ha='center')
plt.xlabel("")
plt.show()
```

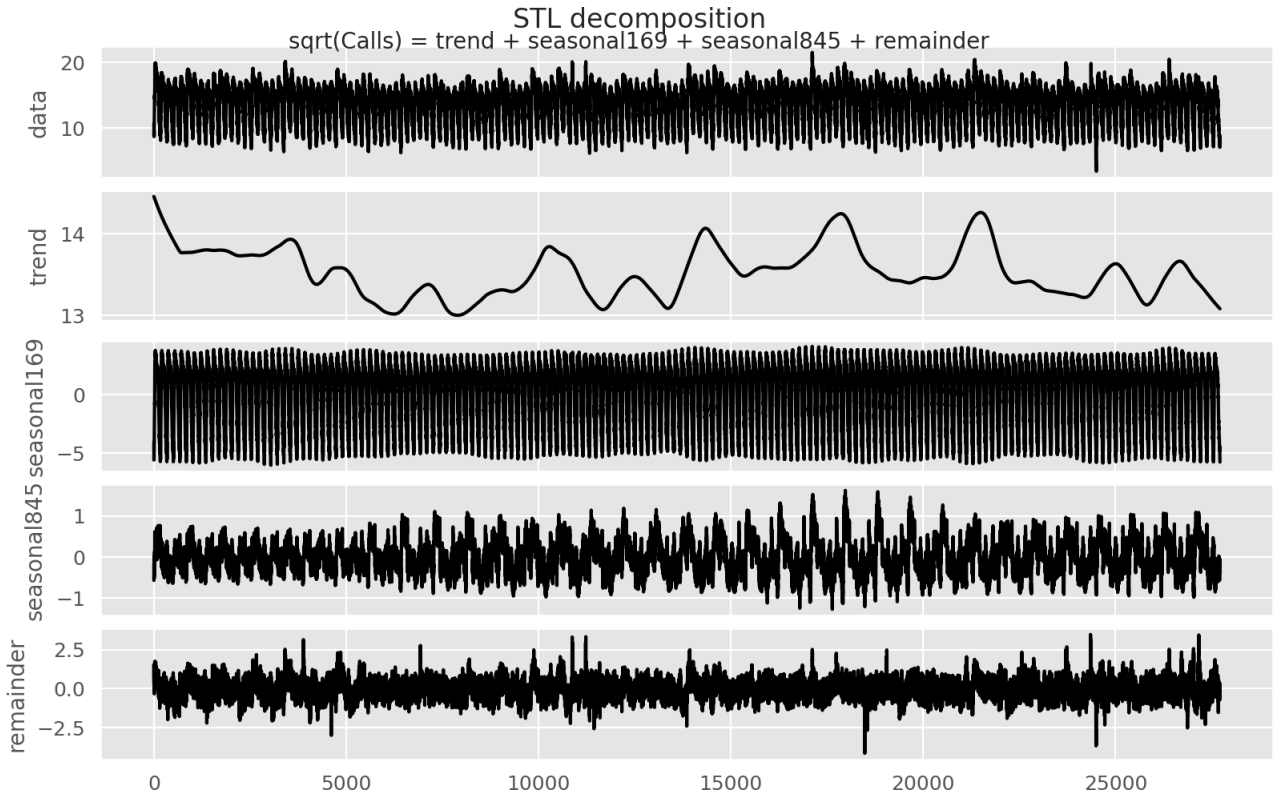

Figure 12.3: STL decomposition with multiple seasonality for the call volume data.

There are two seasonal patterns shown, one for the time of day (the third panel), and one for the time of week (the fourth panel). Note that in this case, there is little trend seen in the data, and the weekly seasonality is weak.

The decomposition can also be used in forecasting, with each of the seasonal components forecast using a seasonal naïve method, and the seasonally adjusted data forecast using `ETS`.

The `MSTL` method can also be used for forecasting, as shown below. Since `StatsForecast` always produces continuous dates, we need to account for the bank being closed before 7:00am, after 9:05pm, and on weekends. We will do this by generating the correct dates directly, using the knowledge that the dataset we used to generate the forecast ends on Friday the 24th at 9:05 PM.

```
fc = sf.predict(h=169 * 5, level=[80, 95])
fc[fc.select_dtypes(include=np.number).columns] = fc.select_dtypes(include=np.number).transform(la



dates = pd.date_range(start="2003-10-27 07:00:00", periods = 169 * 5, freq="5T")
fc["ds"] = dates
fc = fc[fc["ds"].dt.weekday <= 4]
fc = fill_gaps(fc, freq="5T")

plot_series(bank_calls, fc, level=[80, 95],
            max_insample_length=169 * (14 + 5),
            xlabel="DateTime [5m]", ylabel="Calls",
            title="Five-minute call volume to bank", rm_legend=False)
```

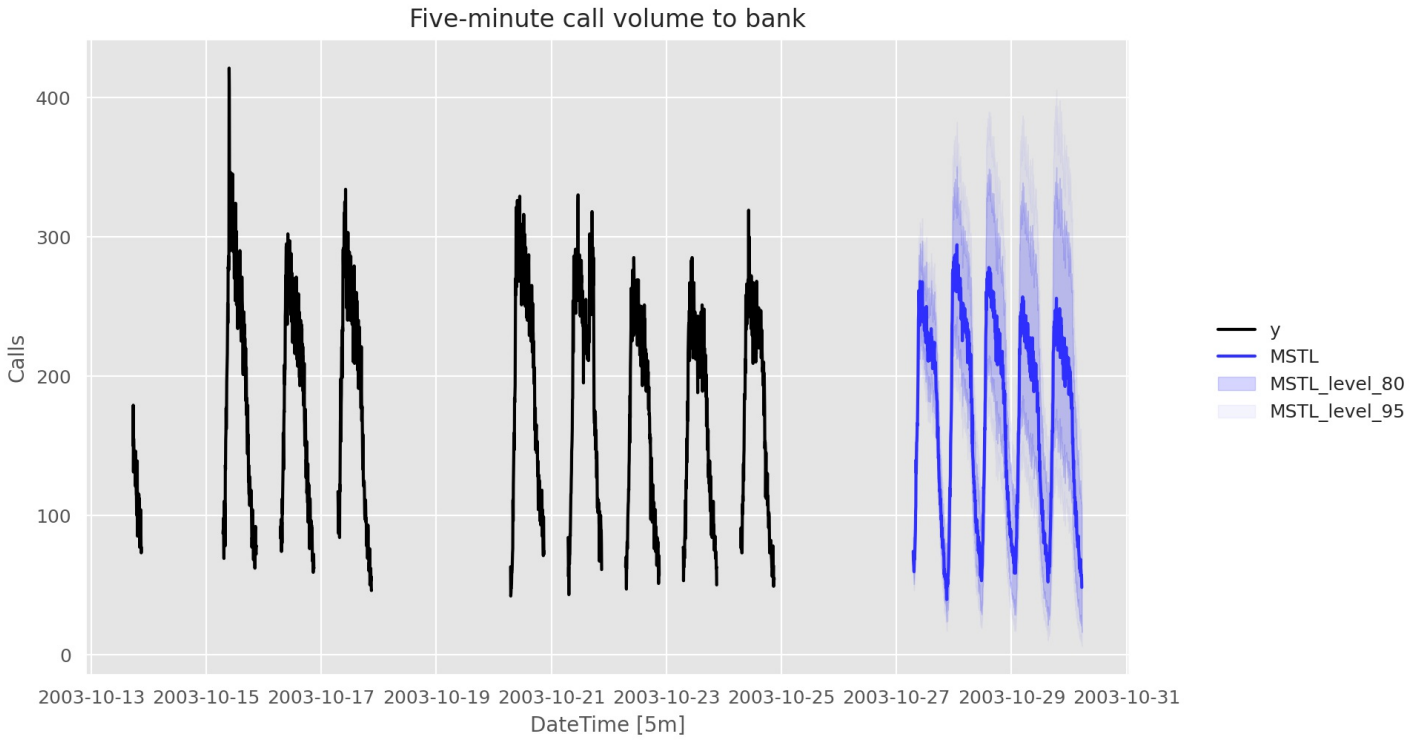

Figure 12.4: Forecasts of the call volume data using an STL decomposition with the seasonal components forecast using a seasonal naïve method, and the seasonally adjusted data forecast using ETS.

Dynamic harmonic regression with multiple seasonal periods

With multiple seasonalities, we can use Fourier terms as we did in earlier chapters (see Sections 7.4 and 10.5). Because there are multiple seasonalities, we need to add Fourier terms for each seasonal period. In this case, the seasonal periods are 169 and 845, so the Fourier terms are of the form $\sin\left(\frac{2\pi kt}{169}\right)$, $\cos\left(\frac{2\pi kt}{169}\right)$, $\sin\left(\frac{2\pi kt}{845}\right)$, and $\cos\left(\frac{2\pi kt}{845}\right)$, for $k=1,2,\dots$. As usual, the `fourier()` function can generate these for you.

We will fit a dynamic harmonic regression model with an ARIMA error structure. The total number of Fourier terms for each seasonal period could be selected to minimise the AICc. However, for high seasonal periods, this tends to over-estimate the number of terms required, so we will use a more subjective choice with 10 terms for the daily seasonality and 5 for the weekly seasonality. We set D=d=0 in order to handle the non-stationarity through the regression terms, and P=Q=0 in order to handle the seasonality through the regression terms (this is achieved by using `seasonal=False` while instantiating `AutoARIMA` as follows).

```
features = [
    partial(fourier, season_length=12, k=12),
    partial(fourier, season_length=5 * 169, k=5),
]
bank_calls_fourier, bank_calls_futr_fourier = pipeline(
    bank_calls_or,
    features=features,
    freq='5min',
    h=169 * 5
)
sf = StatsForecast(
    models=[AutoARIMA(max_d=0, seasonal=False, nmodels=5)],
    freq='5min'
)
sf = sf.fit(bank_calls_fourier)
fc_fourier = sf.predict(h=169 *  5, X_df=bank_calls_futr_fourier, level=[80, 95])

dates = pd.date_range(start="2003-10-27 07:00:00", end="2003-10-31 21:00:00", freq="5min")
dates = dates[
    (dates.time >= pd.Timestamp("07:00").time())
    & (dates.time <= pd.Timestamp("21:00").time())
]
fc_fourier["ds"] = dates

plot_series(bank_calls, fc_fourier, level=[80, 95],
            max_insample_length=169 * 15,
            xlabel="DateTime [5m]", ylabel="Calls",
            title="Five-minute call volume to bank",
            rm_legend=False)
```

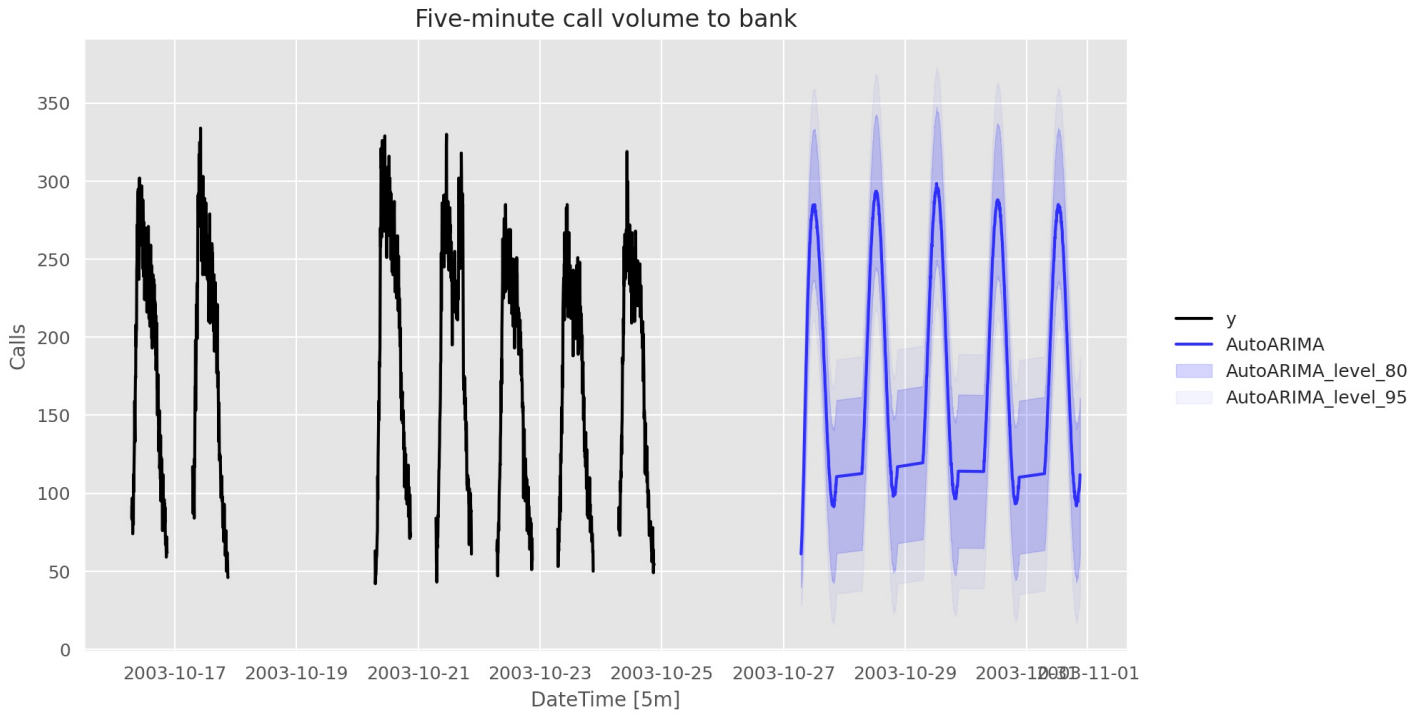

Figure 12.5: Forecasts from a dynamic harmonic regression applied to the call volume data.

This is a large model, containing 38 parameters: 4 ARMA coefficients, 24 Fourier coefficients for period 169, and 10 Fourier coefficients for period 845. Not all of the Fourier terms for period 845 are used because there is some overlap with the terms of period 169 (since 845=5\times169).

Example: Electricity demand

One common application of such models is electricity demand modelling. Figure 12.6 shows half-hourly electricity demand (MWh) in Victoria, Australia, during 2012–2014, along with temperatures (degrees Celsius) for the same period for Melbourne (the largest city in Victoria).

```
vic_elec = pd.read_csv('../data/vic_elec.csv', parse_dates=['ds'])
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series_utils(vic_elec, ax=axes)
for ax in fig.axes:
    ax.tick_params(axis='both')
    if ax.get_legend():
        ax.get_legend().remove()
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.legends = []
fig
```

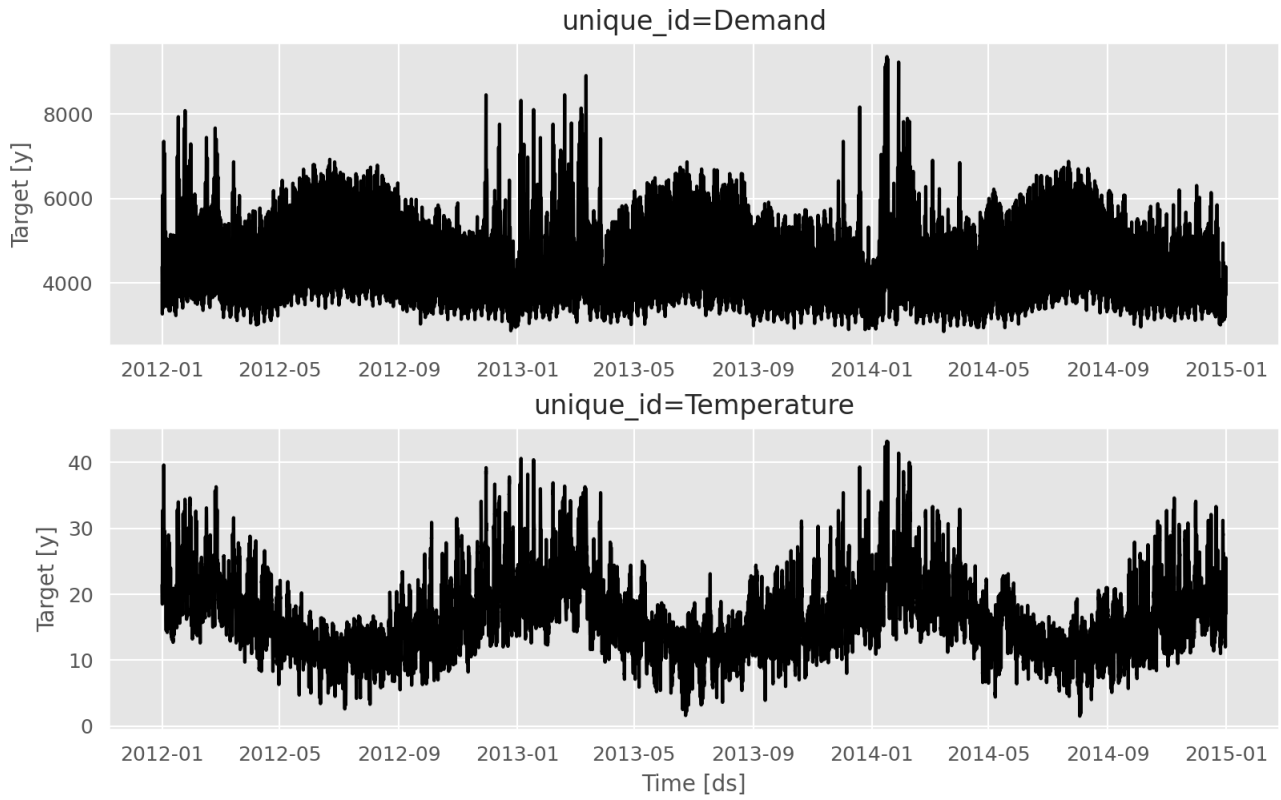

Figure 12.6: Half-hourly electricity demand and corresponding temperatures in 2012–2014, Victoria, Australia.

Plotting electricity demand against temperature (Figure 12.7) shows that there is a nonlinear relationship between the two, with demand increasing for low temperatures (due to heating) and increasing for high temperatures (due to cooling).

```
vic_elec_df = vic_elec.pivot(index = 'ds', columns = 'unique_id', values = 'y').reset_index()
vic_elec_df = vic_elec_df.merge(vic_elec[['ds', 'Holiday']].drop_duplicates(), on = 'ds')
#vic_elec_df.set_index('ds', inplace=True)


vic_elec_df['Demand'] = vic_elec_df['Demand']/1e3
vic_elec_df.insert(0, 'unique_id', 'VIC')
vic_elec_df = vic_elec_df.query('ds >= "2014-01-01"')
vic_elec_df['Day_Week'] = vic_elec_df['ds'].dt.dayofweek
vic_elec_df["Working_Day"] = vic_elec_df['Day_Week'].isin([5, 6])
vic_elec_df["Cooling"] = np.maximum(vic_elec_df["Temperature"], 18)

fig = plt.figure()
sns.scatterplot(data=vic_elec_df, x='Temperature', y='Demand', hue='Working_Day')
plt.xlabel('Maximum daily temperature')
plt.ylabel('Electricity demand (GW)')
plt.show()
```

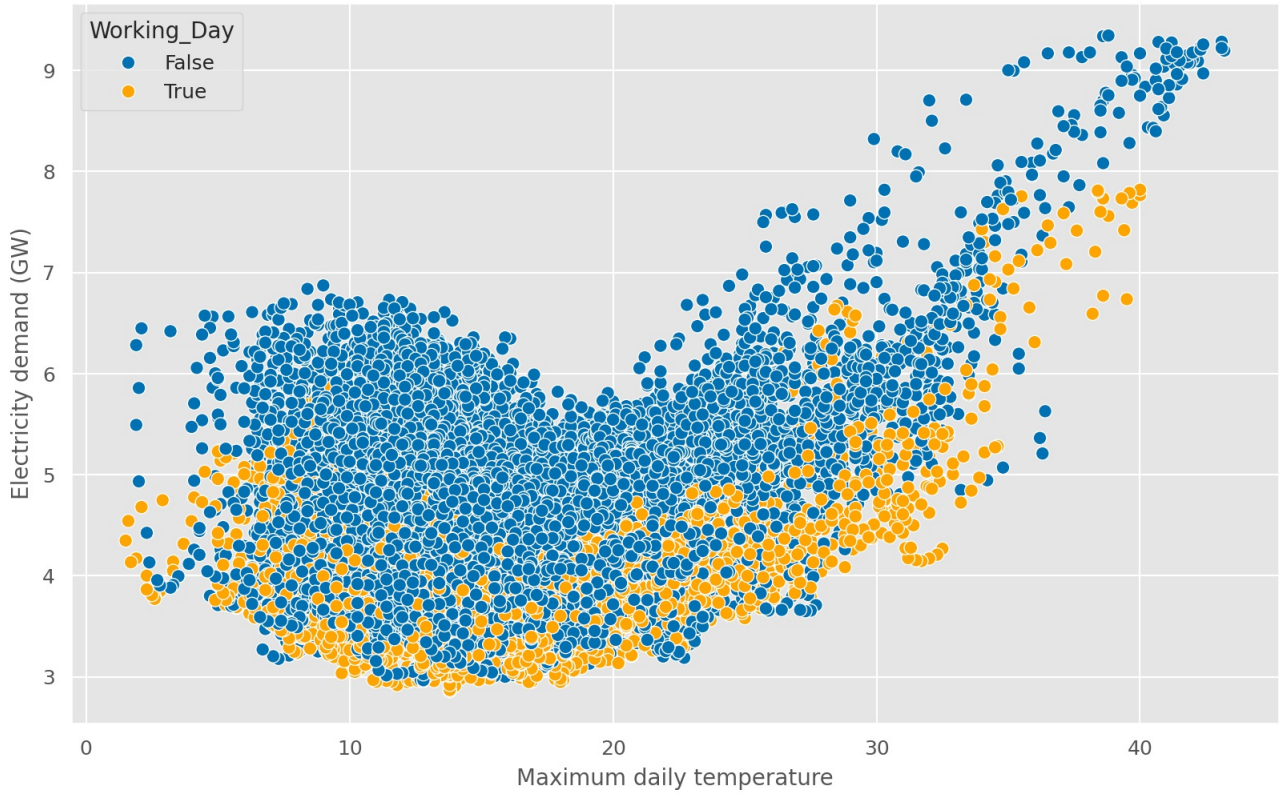

Figure 12.7: Half-hourly electricity demand for Victoria, plotted against temperatures for the same times in Melbourne, the largest city in Victoria.

We will fit a regression model with a piecewise linear function of temperature (containing a knot at 18 degrees), and harmonic regression terms to allow for the daily seasonal pattern. Again, we set the orders of the Fourier terms subjectively, while using the AICc to select the order of the ARIMA errors.

```
vic_elec_df = vic_elec_df.drop(columns=["Holiday", "Day_Week"])
vic_elec_df = vic_elec_df.rename(columns={"Demand": "y"})

features = [
    partial(fourier, season_length=2 * 24, k=10),
    partial(fourier, season_length=2 * 24 * 7, k=5),
    partial(fourier, season_length=2 * 24 * 7 * 365, k=3),
]
vic_elec_fourier, vic_elec_futr_fourier = pipeline(
    vic_elec_df,
    features=features,
    freq='30min',
    h=2 * 48
)
sf = StatsForecast(
    models=[AutoARIMA(max_d=0, seasonal=False, nmodels=20)],
    freq='30min'
)
sf = sf.fit(vic_elec_fourier)
```

Forecasting with such models is difficult because we require future values of the predictor variables. Future values of the Fourier terms are easy to compute, but future temperatures are, of course, unknown. If we are only interested in forecasting up to a week ahead, we could use temperature forecasts obtained from a meteorological model. Alternatively, we could use scenario forecasting (Section 6.5) and plug in possible temperature patterns. In the following example, we have used a repeat of the last two days of temperatures to generate future possible demand values.

```
for col in ["Temperature", "Cooling", "Working_Day"]:
    vic_elec_futr_fourier[col] = vic_elec_df[col].tail(2 * 48).values
fc_fourier = sf.predict(h=2 * 48, X_df=vic_elec_futr_fourier[vic_elec_fourier.columns.drop("y")],

plot_series(
    vic_elec_fourier, fc_fourier, level=[80, 95], max_insample_length=10 * 48,
    xlabel="Time [30m]",
    ylabel="Demand (MWh)",
    title="Half hourly electricity demand: Victoria",
    rm_legend=False)
```

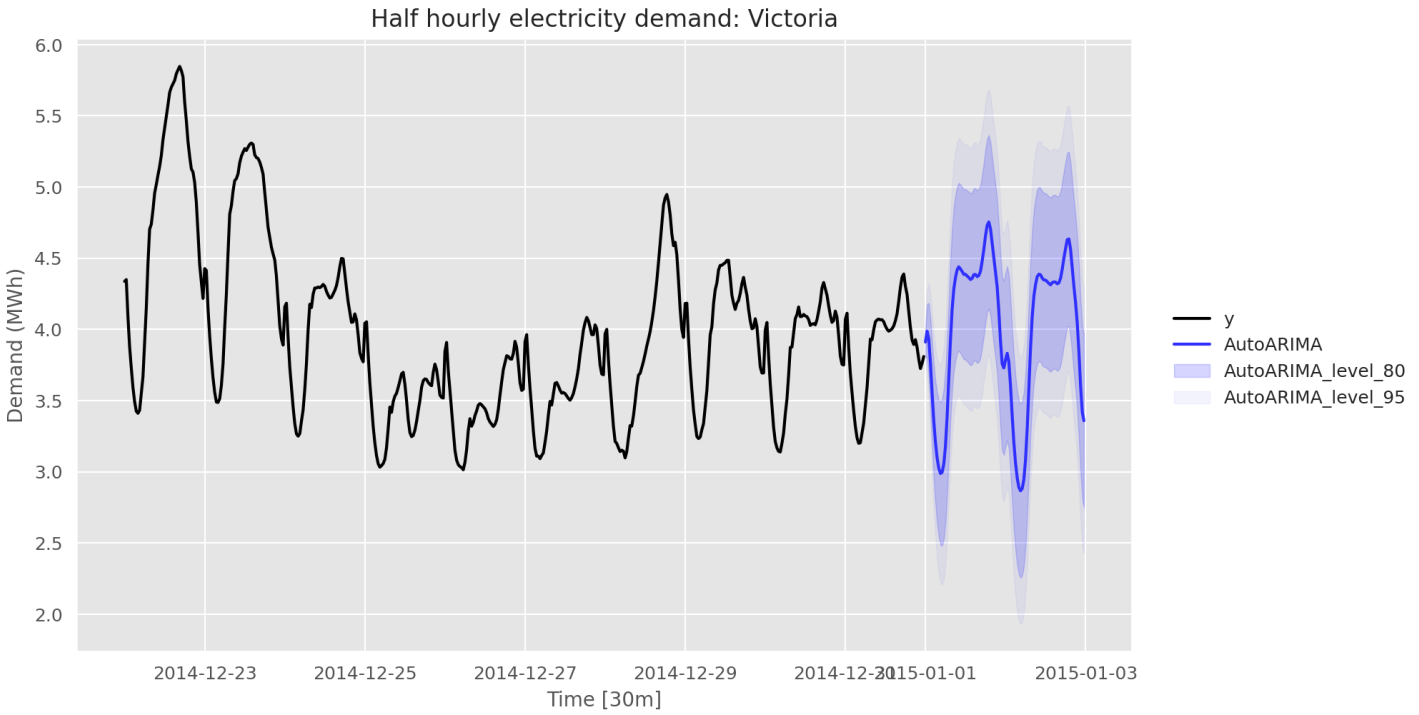

Figure 12.8: Forecasts from a dynamic harmonic regression model applied to half-hourly electricity demand data.

```
residuals = sf.fitted_[0, 0].model_["residuals"]

fig = plt.figure()

gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, :])
ax2 = fig.add_subplot(gs[1, 0])
ax3 = fig.add_subplot(gs[1, 1])

ax1.plot(vic_elec_fourier["ds"], residuals, marker="o")
ax1.set_ylabel("Innovation Residuals")
ax1.set_xlabel("Year")

plot_acf(residuals, ax2,
         zero=False,
         bartlett_confint=False,
         auto_ylims=True)
ax2.set_ylabel("acf")
ax2.set_xlabel("lag")

ax3.hist(residuals, bins=20)
ax3.set_ylabel("count")
ax3.set_xlabel("residuals")

plt.show()
```

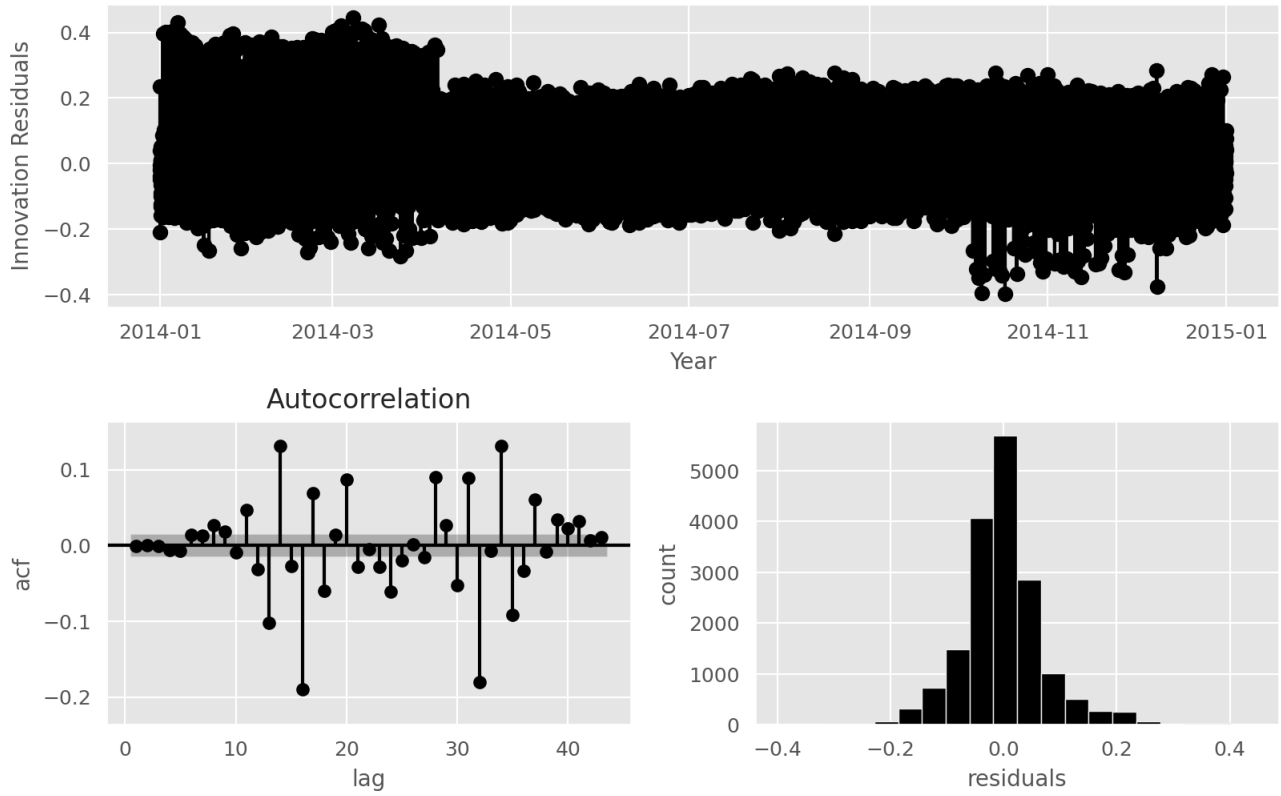

Figure 12.9: Residual diagnostics for the dynamic harmonic regression model.

Although the short-term forecasts look reasonable, this is a crude model for a complicated process. The residuals, plotted in Figure 12.9, demonstrate that there is a lot of information that has not been captured with this model.

More sophisticated versions of this model which provide much better forecasts are described in Hyndman and Fan (2010) and Fan and Hyndman (2012).

12.2 Prophet model

A recent proposal is the Prophet model, available via PyPI. This model was introduced by Facebook (Taylor and Letham 2018), originally for forecasting daily data with weekly and yearly seasonality, plus holiday effects. It was later extended to cover more types of seasonal data. It works best with time series that have strong seasonality and several seasons of historical data.

Prophet can be considered a nonlinear regression model (Chapter 7), of the form y_t = g(t) + s(t) + h(t) + \varepsilon_t, where g(t) describes a piecewise-linear trend (or “growth term”), s(t) describes the various seasonal patterns, h(t) captures the holiday effects, and \varepsilon_t is a white noise error term.

- The knots (or changepoints) for the piecewise-linear trend are automatically selected if not explicitly specified. Optionally, a logistic function can be used to set an upper bound on the trend.
- The seasonal component consists of Fourier terms of the relevant periods. By default, order 10 is used for annual seasonality and order 3 is used for weekly seasonality.
- Holiday effects are added as simple dummy variables.
- The model is estimated using a Bayesian approach to allow for automatic selection of the changepoints and other model characteristics.

We illustrate the approach using two data sets: a simple quarterly example, and then the electricity demand data described in the previous section.

Example: Quarterly cement production

For the simple quarterly example, we will repeat the analysis from Section 9.10 in which we compared an ARIMA and ETS model, but we will add in a Prophet model for comparison.. By default, Prophet fits weekly and yearly seasonalities. Since we have quarterly data, we need to specify this directly using `add_seasonality`.

```
aus_production = pd.read_csv("../data/aus_production.csv", parse_dates=["ds"])
cement = (
    aus_production[["ds", "Cement"]]
    .query('ds >= "1988-01-01"')
    .reset_index(drop=True)
    .assign(unique_id="aus_production")
)
cement.rename(columns={"Cement": "y"}, inplace=True)

train = cement.query('ds < "2008-01-01"')
test = cement.query('ds >= "2008-01-01"')

m = Prophet(weekly_seasonality=False)
m.add_seasonality(name="quarterly", period=4, fourier_order=2)
m.fit(train)

future = m.make_future_dataframe(periods=10, freq="QS")
fc_prophet = m.predict(future)
fc_prophet = fc_prophet[["ds", "yhat", "yhat_lower", "yhat_upper"]]
fc_prophet.rename(
    columns={
        "yhat": "Prophet",
        "yhat_lower": "Prophet-lo-80",
        "yhat_upper": "Prophet-hi-80",
    },
    inplace=True,
)
```

```
sf = StatsForecast(
    models=[AutoARIMA(season_length=4), AutoETS(season_length=4)], freq="QS"
)
fc_stats = sf.forecast(df=train, h=10, level=[80])

fc = fc_stats.merge(fc_prophet, on=["ds"])
plot_series(cement, fc, level=[80],
            xlabel="Quarter", ylabel="Cement",
            rm_legend=False,
            title="Cement production: Australia")
```

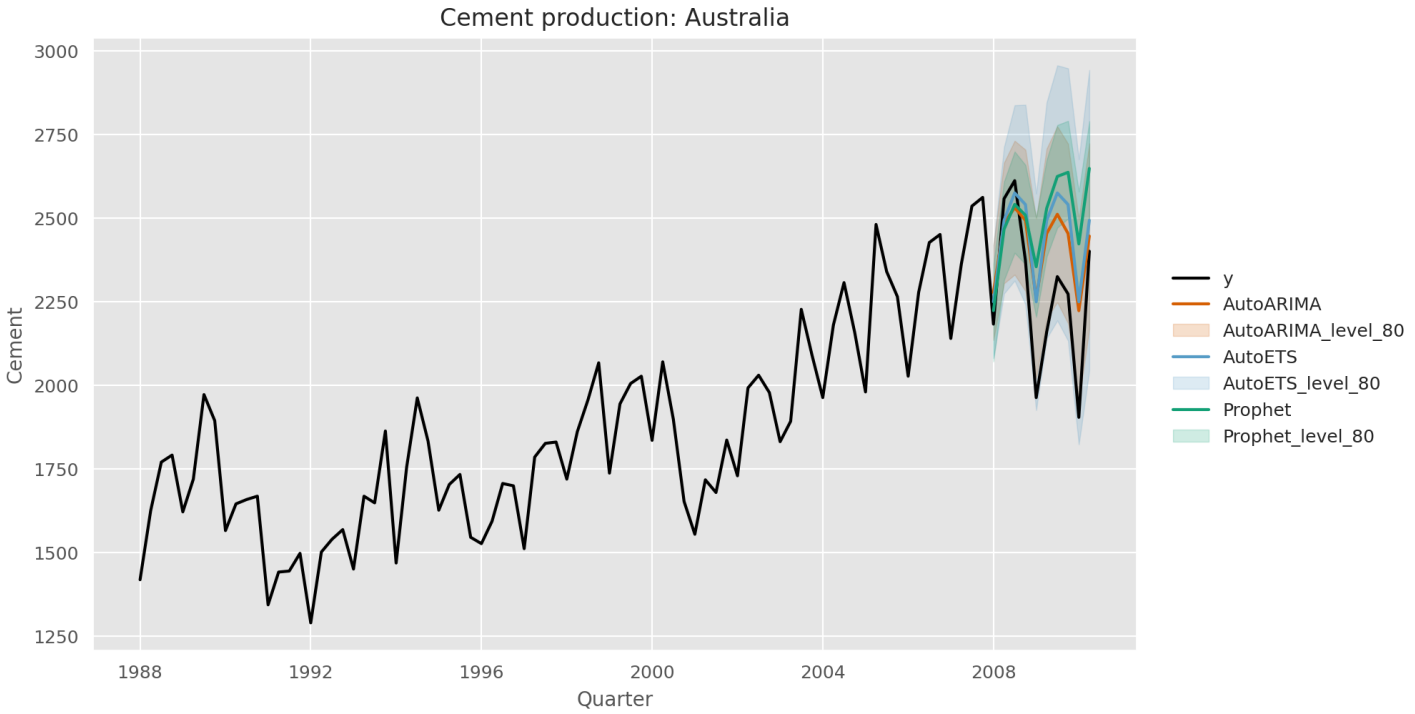

Figure 12.10: Forecasts of the Australian cement production data using Prophet and StatsForecast.

In this example, the Prophet forecasts are worse than either the ETS or ARIMA forecasts.

```
evaluate(
    fc.merge(test, on=["unique_id", "ds"]),
    metrics=[rmse, mae, mape, partial(mase, seasonality=4)],
    train_df=cement,
)
```

	unique_id	metric	AutoARIMA	AutoETS	Prophet
0	aus_production	rmse	195.204	222.180	296.015
1	aus_production	mae	169.078	191.152	252.953
2	aus_production	mape	0.079	0.089	0.117
3	aus_production	mase	1.124	1.271	1.682

Example: Half-hourly electricity demand

We will now fit a prophet model to half-hourly electricity demand data, and compare its results with the MSTL model from the previous section.

```
elec = pd.read_csv("../data/vic_elec.csv", parse_dates=["ds"])
vic_demand = elec.query('unique_id == "Demand"').drop("Holiday", axis=1)[
    ["unique_id", "ds", "y"]
]

train = vic_demand.iloc[:-48]
test = vic_demand.iloc[-48:]

m = Prophet()
m.add_seasonality(name="hourly", period=48, fourier_order=5)
m.add_seasonality(name="subhourly", period=48 * 7, fourier_order=5)
m.fit(train)

future = m.make_future_dataframe(periods=48 * 2, freq="30T")
fc_prophet = m.predict(future)
fc_prophet.rename(
    columns={
        "yhat": "Prophet",
        "yhat_lower": "Prophet-lo-80",
        "yhat_upper": "Prophet-hi-80",
    },
    inplace=True,
)
fc_prophet["resid"] = train["y"] - fc_prophet["Prophet"]
fig, axes = plt.subplots(nrows=5, ncols=1, sharex=True, figsize=(8, 7))
sns.lineplot(data=train, x=fc_prophet["ds"], y="y", ax=axes[0], color="black")
sns.lineplot(data=fc_prophet, x=fc_prophet["ds"], y="trend", ax=axes[1], color="black")
sns.lineplot(data=fc_prophet, x=fc_prophet["ds"], y="multiplicative_terms", ax=axes[2], color="bla

axes[2].set_ylabel("multiplicative_term")
sns.lineplot(data=fc_prophet, x=fc_prophet["ds"], y="additive_terms", ax=axes[3], color="black")
axes[3].set_ylabel("additive_term")
sns.lineplot(data=fc_prophet, x=fc_prophet["ds"], y="resid", ax=axes[4], color="black")
fig.suptitle("Prophet decomposition")
fig.text(0.5, 0.955,
         "Demand = trend * (1 + multiplicative_terms) + additive_terms + resid",
         ha='center')
plt.xlabel("")
plt.show()
```

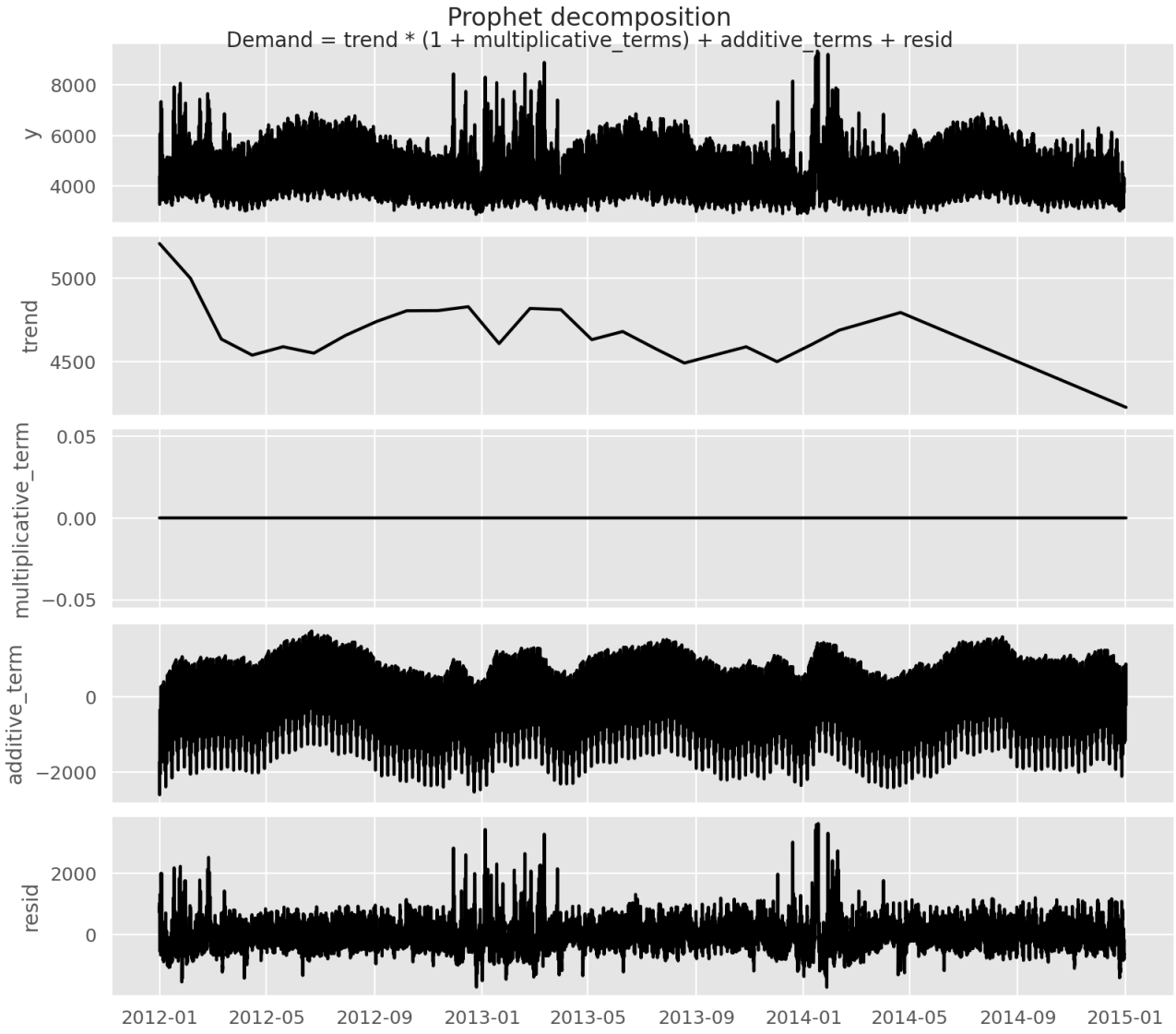

Figure 12.11: Components of a Prophet model fitted to the Victorian electricity demand

Figure 12.11 shows the trend and seasonal components of the fitted model.

The model specification is very similar to the DHR model in the previous section, although the result is different in several important ways. The Prophet model adds a piecewise linear time trend which is not really appropriate here as we don't expect the long term forecasts to continue to follow the downward linear trend at the end of the series.

There is also substantial remaining autocorrelation in the residuals,

```
residuals = fc_prophet["resid"]

fig = plt.figure()

gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, :])
ax2 = fig.add_subplot(gs[1, 0])
ax3 = fig.add_subplot(gs[1, 1])

ax1.plot(fc_prophet["ds"], residuals, marker="o")
ax1.set_ylabel("Innovation Residuals")
ax1.set_xlabel("Year")

plot_acf(residuals.dropna(), ax2,
         zero=False,
         bartlett_confint=False,
         auto_ylims=True)
ax2.set_ylabel("acf")
ax2.set_xlabel("lag")

ax3.hist(residuals, bins=20)
ax3.set_ylabel("count")
ax3.set_xlabel("residuals")

plt.show()
```

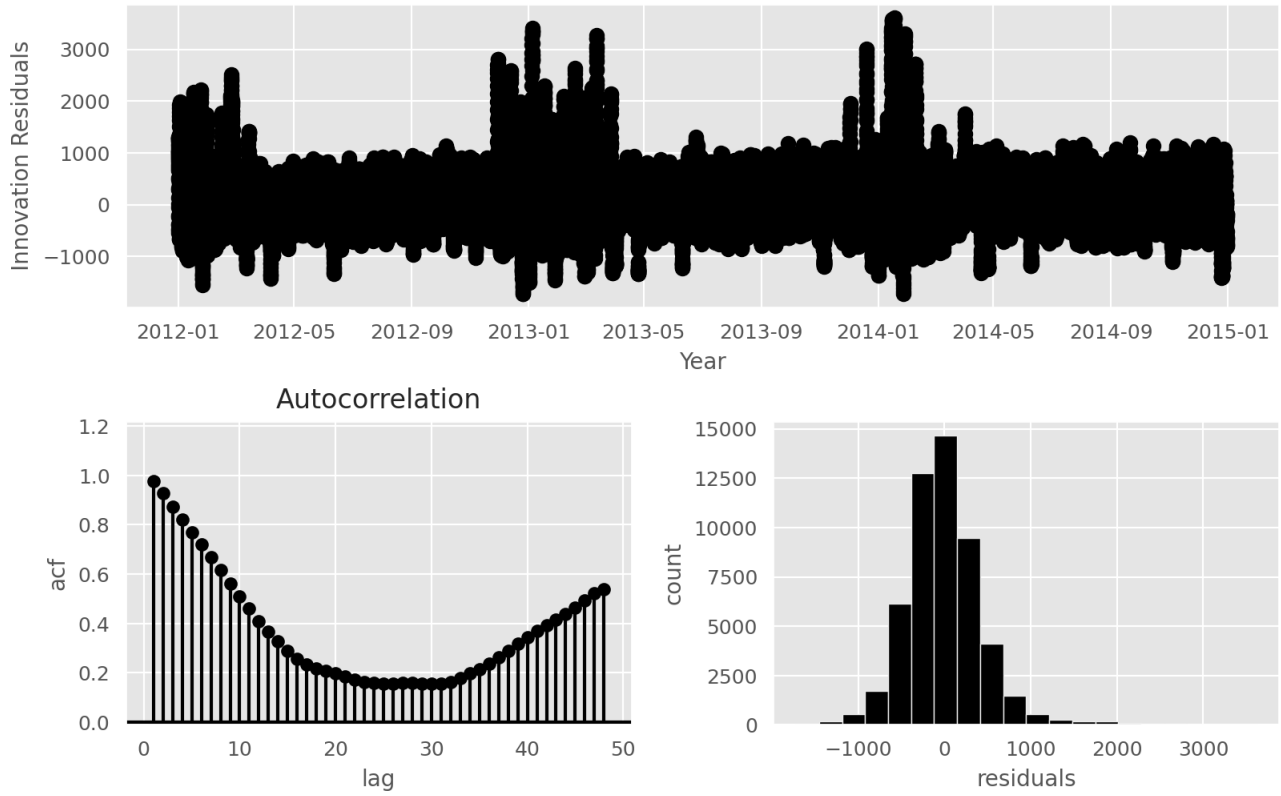

Figure 12.12: Residuals from the Prophet model for Victorian electricity demand.

The prediction intervals would be narrower if the autocorrelations were taken into account. Now let's compare these forecasts against MSTL.

```
sf = StatsForecast(models=[MSTL(season_length=[48, 48 * 7])], freq="30T")
fc_stats = sf.forecast(df=train, h=48, level=[80])

fc = fc_stats.merge(fc_prophet[["ds", "Prophet", "Prophet-lo-80", "Prophet-hi-80"]], on=["ds"])
plot_series(vic_demand, fc, level=[80],
            max_insample_length=48 * 7,
            xlabel="Date", ylabel="Demand (MWh)",
            title = "Demand (MWh)",
            rm_legend=False)
```

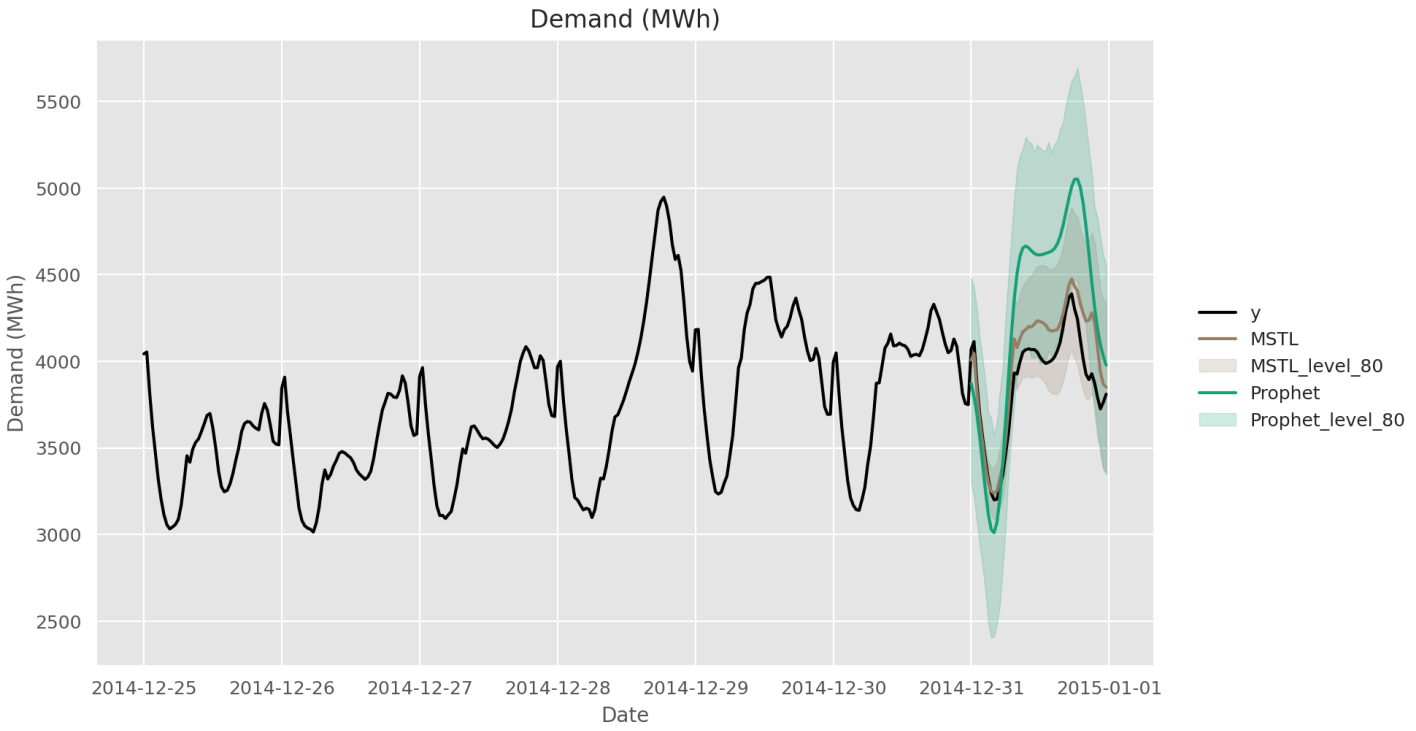

Figure 12.13: Two day forecasts from the Prophet and the MSTL model for Victorian electricity demand.

```
evaluate(
    fc.merge(test, on=["unique_id", "ds"]),
    metrics=[rmse, mae, mape, partial(mase, seasonality=4)],
    train_df=vic_demand,
)
```

	unique_id	metric	MSTL	Prophet
0	Demand	rmse	165.452	519.633
1	Demand	mae	138.730	466.853
2	Demand	mape	0.035	0.117
3	Demand	mase	0.364	1.226

Prophet rarely gives better forecast accuracy than the alternative approaches, as these two examples have illustrated. However, it is a popular model for forecasting, with thousands of stars on GitHub and used as a dependency in hundreds of repositories. Its popularity can partly be explained by its open-source nature and ease of use. However, several studies, such as Garza et al. (2022) and Menculini et al. (2021), have shown that it does not perform well in terms of time and accuracy.

StatsForecast offers an adapter to Prophet with improved performance, as shown by Garza et al. (2022). It returns the best ARIMA model using Prophet's feature preprocessing. To use it, simply replace `Prophet` with `AutoProphet`, as shown below using the previous example.

```
m = AutoARIMAProphet()
m.add_seasonality(name="hourly", period=48, fourier_order=5)
m.add_seasonality(name="subhourly", period=48 * 7, fourier_order=5)
m.fit(train)

future = m.make_future_dataframe(periods=48 * 2, freq="30T")
fc_ap = m.predict(future)
```

There is a more advanced model called NeuralProphet that combines neural networks with traditional time series algorithms and was inspired by Prophet. This model is available through PyPI and can be set up in a way similar to Prophet.

12.3 Vector autoregressions

One limitation of the models that we have considered so far is that they impose a unidirectional relationship — the forecast variable is influenced by the predictor variables, but not vice versa. However, there are many cases where the reverse should also be allowed for — where all variables affect each other. In previous chapters, the changes in personal consumption expenditure (C_t) were forecast based on the changes in personal disposable income (I_t). However, in this case a bi-directional relationship may be more suitable: an increase in I_t will lead to an increase in C_t and vice versa.

An example of such a situation occurred in Australia during the Global Financial Crisis of 2008–2009. The Australian government issued stimulus packages that included cash payments in December 2008, just in time for Christmas spending. As a result, retailers reported strong sales and the economy was stimulated. Consequently, incomes increased.

Such feedback relationships are allowed for in the vector autoregressive (VAR) framework. In this framework, all variables are treated symmetrically. They are all modelled as if they all influence each other equally. In more formal terminology, all variables are now treated as “endogenous”. To signify this, we now change the notation and write all variables as ys: y_{1,t} denotes the tth observation of variable y_1, y_{2,t} denotes the tth observation of variable y_2, and so on.

A VAR model is a generalisation of the univariate autoregressive model for forecasting a vector of time series.[1] It comprises one equation per variable in the system. The right hand side of each equation includes a constant and lags of all of the variables in the system. To keep it simple, we will consider a two variable VAR with one lag. We write a 2-dimensional VAR(1) model as \begin{align} y_{1,t} &= c_1+\phi _{11,1}y_{1,t-1}+\phi _{12,1}y_{2,t-1}+\varepsilon_{1,t} \\ y_{2,t} &= c_2+\phi _{21,1}y_{1,t-1}+\phi _{22,1}y_{2,t-1}+\varepsilon_{2,t}, \end{align} \tag{12.1} where \varepsilon_{1,t} and \varepsilon_{2,t} are white noise processes that may be contemporaneously correlated. The coefficient \phi_{ii,\ell} captures the influence of the \ellth lag of variable y_i on itself, while the coefficient \phi_{ij,\ell} captures the influence of the \ellth lag of variable y_j on y_i.

If the series are stationary, we forecast them by fitting a VAR to the data directly (known as a “VAR in levels”). If the series are non-stationary, we take differences of the data in order to make them stationary, then fit a VAR model (known as a “VAR in differences”). In both cases, the models are estimated equation by equation using the principle of least squares. For each equation, the parameters are estimated by minimising the sum of squared \varepsilon_{i,t} values.

The other possibility, which is beyond the scope of this book and therefore we do not explore here, is that the series may be non-stationary but cointegrated, which means that there exists a linear combination of them that is stationary. In this case, a VAR specification that includes an error correction mechanism (usually referred to as a vector error correction model) should be included, and alternative estimation methods to least squares estimation should be used.[2]

Forecasts are generated from a VAR in a recursive manner. The VAR generates forecasts for *each* variable included in the system. To illustrate the process, assume that we have fitted the 2-dimensional VAR(1) model described in Equations 12.1, for all observations up to time T. Then the one-step-ahead forecasts are generated by \begin{align*} \hat y_{1,T+1|T} &=\hat{c}_1+\hat\phi_{11,1}y_{1,T}+\hat\phi_{12,1}y_{2,T} \\ \hat y_{2,T+1|T} &=\hat{c}_2+\hat\phi _{21,1}y_{1,T}+\hat\phi_{22,1}y_{2,T}. \end{align*} This is the same form as Equations 12.1, except that the errors have been set to zero and parameters have been replaced with their estimates. For h=2, the forecasts are given by \begin{align*} \hat y_{1,T+2|T} &=\hat{c}_1+\hat\phi_{11,1}\hat y_{1,T+1|T}+\hat\phi_{12,1}\hat y_{2,T+1|T}\\ \hat y_{2,T+2|T}&=\hat{c}_2+\hat\phi_{21,1}\hat y_{1,T+1|T}+\hat\phi_{22,1}\hat y_{2,T+1|T}. \end{align*} Again, this is the same form as Equations 12.1, except that the errors have been set to zero, the parameters have been replaced with their estimates, and the unknown values of y_1 and y_2 have been replaced with their forecasts. The process can be iterated in this manner for all future time periods.

There are two decisions one has to make when using a VAR to forecast, namely how many variables (denoted by K) and how many lags (denoted by p) should be included in the system. The number of coefficients to be estimated in a VAR is equal to K+pK^2 (or 1+pK per equation). For example, for a VAR with K=5 variables and p=3 lags, there are 16 coefficients per equation, giving a total of 80 coefficients to be estimated. The more coefficients that need to be estimated, the larger the estimation error entering the forecast.

In practice, it is usual to keep K small and include only variables that are correlated with each other, and therefore useful in

forecasting each other. Information criteria are commonly used to select the number of lags to be included. Care should be taken when using the AICc as it tends to choose large numbers of lags; instead, for VAR models, we often use the BIC instead. A more sophisticated version of the model is a “sparse VAR” (where many coefficients are set to zero); another approach is to use “shrinkage estimation” (where coefficients are smaller).

A criticism that VARs face is that they are atheoretical; that is, they are not built on some economic theory that imposes a theoretical structure on the equations. Every variable is assumed to influence every other variable in the system, which makes a direct interpretation of the estimated coefficients difficult. Despite this, VARs are useful in several contexts:

1. forecasting a collection of related variables where no explicit interpretation is required;
2. testing whether one variable is useful in forecasting another (the basis of Granger causality tests);
3. impulse response analysis, where the response of one variable to a sudden but temporary change in another variable is analysed;
4. forecast error variance decomposition, where the proportion of the forecast variance of each variable is attributed to the effects of the other variables.

Example: A VAR model for forecasting US consumption

You can use a VAR model in Python via the `statsmodels.tsa.api` module. Note that you need to set the dates as the index of the DataFrame before fitting the model. We will fit two models: one using the AIC and another using the BIC.

```
us_change = pd.read_csv(
    "../data/US_change.csv",
    parse_dates=["ds"],
).rename(columns={"y": "Consumption"}).drop(columns=["unique_id"])
us_change.set_index("ds", inplace=True)
us_change.index = pd.DatetimeIndex(us_change.index.values, freq="QS", name="ds")

model = VAR(us_change)
fit_aic = model.fit(ic="aic")
print("Lags selected by AIC:", fit_aic.k_ar)

fit_bic = model.fit(ic="bic")
print("Lags selected by BIC:", fit_bic.k_ar)
```

```
Lags selected by AIC: 5
Lags selected by BIC: 1
```

A VAR(5) model is selected using the AICc (the default), while a VAR(1) model is selected using the BIC. This is not unusual — the BIC will always select a model that has fewer parameters than the AICc model as it imposes a stronger penalty for the number of parameters.

```
fig, axes = plt.subplots(nrows=4, ncols=1)
plot_acf(
    fit_aic.resid["Consumption"],
    ax=axes[0],
    title="ACF of VAR(5) Model Residuals - Consumption",
    zero=False,
    auto_ylims=True,
    bartlett_confint=False,
)
plot_acf(
    fit_aic.resid["Income"],
    ax=axes[1],
    title="ACF of VAR(5) Model Residuals - Income",
    zero=False,
    auto_ylims=True,
    bartlett_confint=False,
)
plot_acf(
    fit_bic.resid["Consumption"],
    ax=axes[2],
    title="ACF of VAR(1) Model Residuals - Consumption",
    zero=False,
    auto_ylims=True,
    bartlett_confint=False,
)
plot_acf(
    fit_bic.resid["Income"],
    ax=axes[3],
    title="ACF of VAR(1) Model Residuals - Income",
    zero=False,
    auto_ylims=True,
    bartlett_confint=False,
)
plt.show()
```

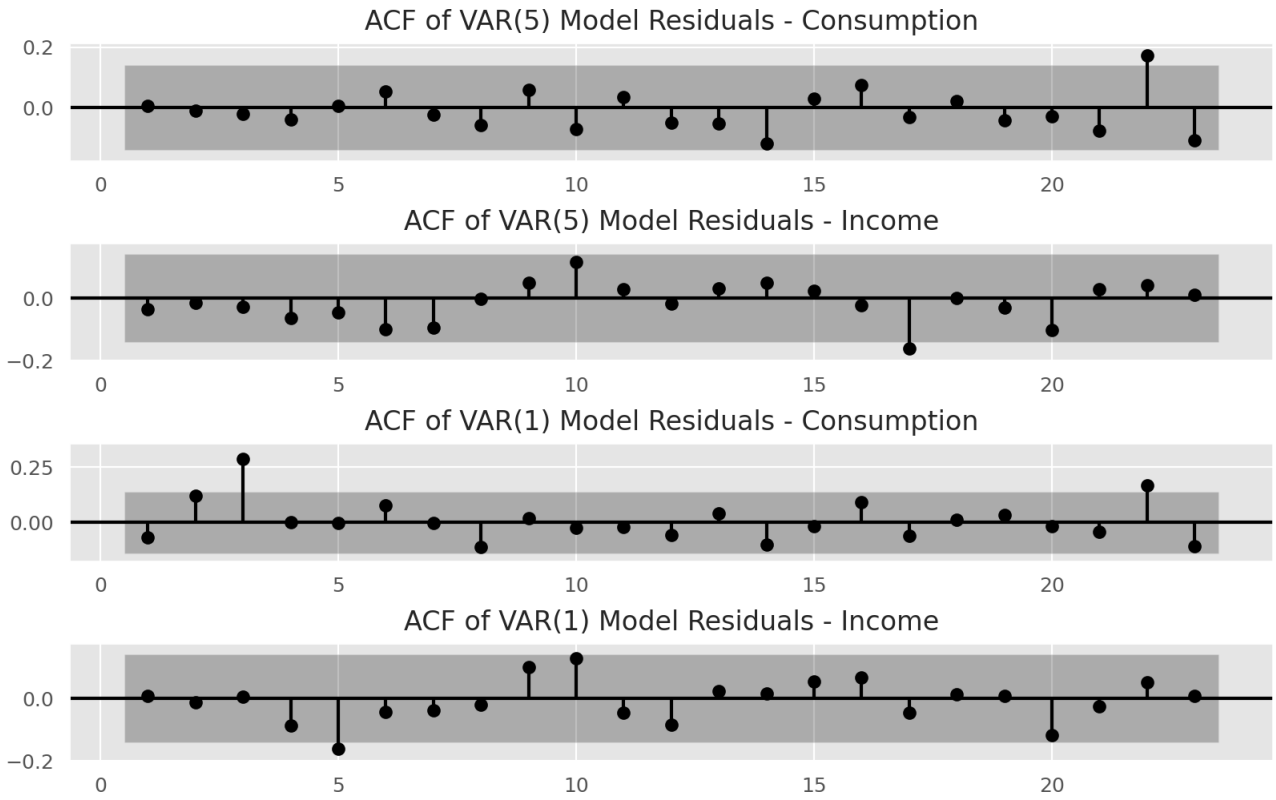

Figure 12.14: ACF of the residuals from the two VAR models. A VAR(5) model is selected by the AICc, while a VAR(1) model is selected using the BIC.

We see that the residuals from the VAR(1) model (BIC) have significant autocorrelation for Consumption, while the VAR(5) model has effectively captured all the information in the data.

The forecasts generated by the VAR(5) model are plotted in Figure 12.15 and Figure 12.16. Note that `forecast_interval` generates both the mean forecast and the (1-\alpha)\% prediction interval.

```
lag_order = fit_aic.k_ar
fc80 = fit_aic.forecast_interval(us_change.values[-lag_order:], steps=8, alpha=0.2)
fc95 = fit_aic.forecast_interval(us_change.values[-lag_order:], steps=8, alpha=0.05)

fc_consumption = pd.DataFrame(
    {
        "unique_id": "Consumption",
        "ds": pd.date_range(start="2019-07-01", periods=8, freq="QS"),
        "Consumption": fc80[0][:, 0],
        "Consumption-lo-80": fc80[1][:, 0],
        "Consumption-hi-80": fc80[2][:, 0],
        "Consumption-lo-95": fc95[1][:, 0],
        "Consumption-hi-95": fc95[2][:, 0],
    }
)

fc_income = pd.DataFrame(
    {
        "unique_id": "Income",
        "ds": pd.date_range(start="2019-07-01", periods=8, freq="QS"),
        "Income": fc80[0][:, 1],
        "Income-lo-80": fc80[1][:, 1],
        "Income-hi-80": fc80[2][:, 1],
        "Income-lo-95": fc95[1][:, 1],
        "Income-hi-95": fc95[2][:, 1],
    }
)

us_change.reset_index(inplace=True)
us_change.rename(columns={"index": "ds"}, inplace=True)
```

```
consumption_df = us_change[["ds", "Consumption"]].copy()
consumption_df.rename(columns={"Consumption": "y"}, inplace=True)
consumption_df.insert(0, "unique_id", "Consumption")

plot_series(consumption_df, fc_consumption, level=[80, 95],
            rm_legend=False, title="Consumption")
```

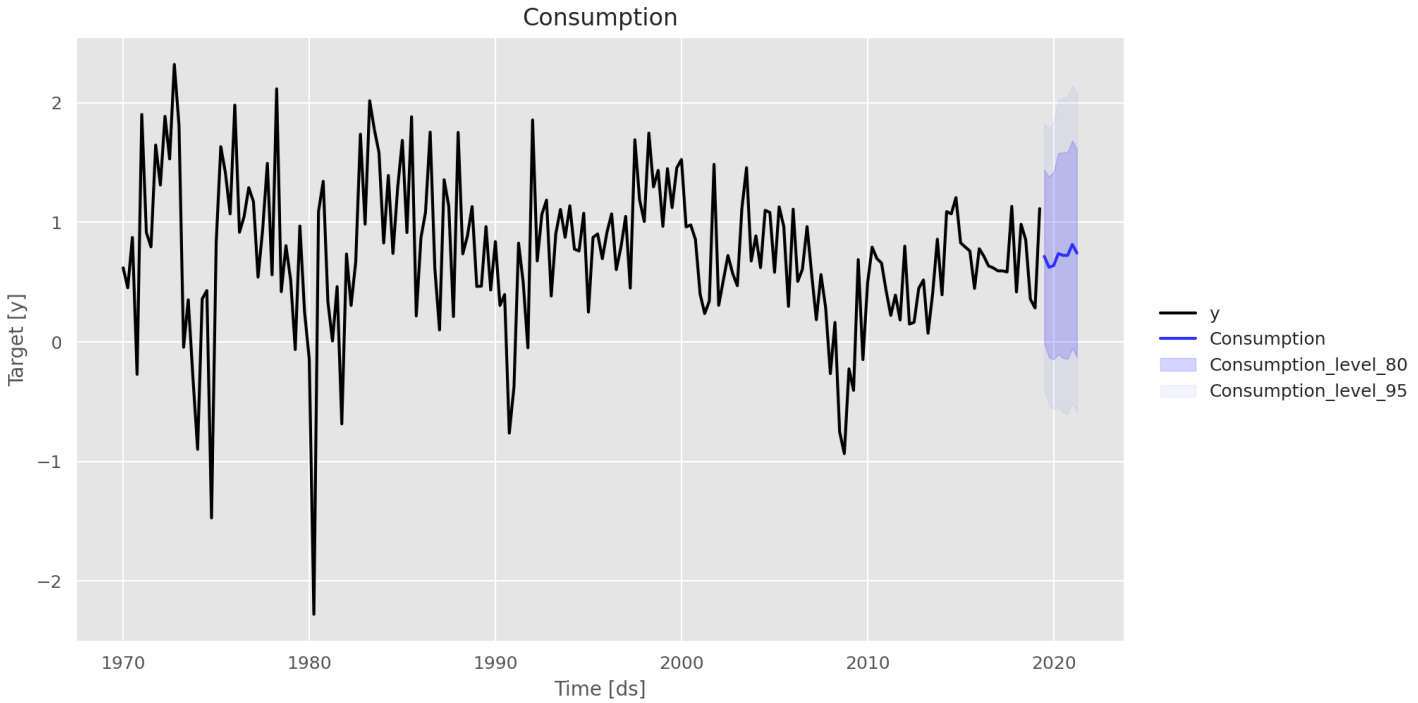

Figure 12.15: Forecasts for US consumption from a VAR(5) model.

```
income_df = us_change[["ds", "Income"]].copy()
income_df.rename(columns={"Income": "y"}, inplace=True)
income_df.insert(0, "unique_id", "Income")

plot_series(income_df, fc_income, level=[80, 95],
            rm_legend=False, title="Income")
```

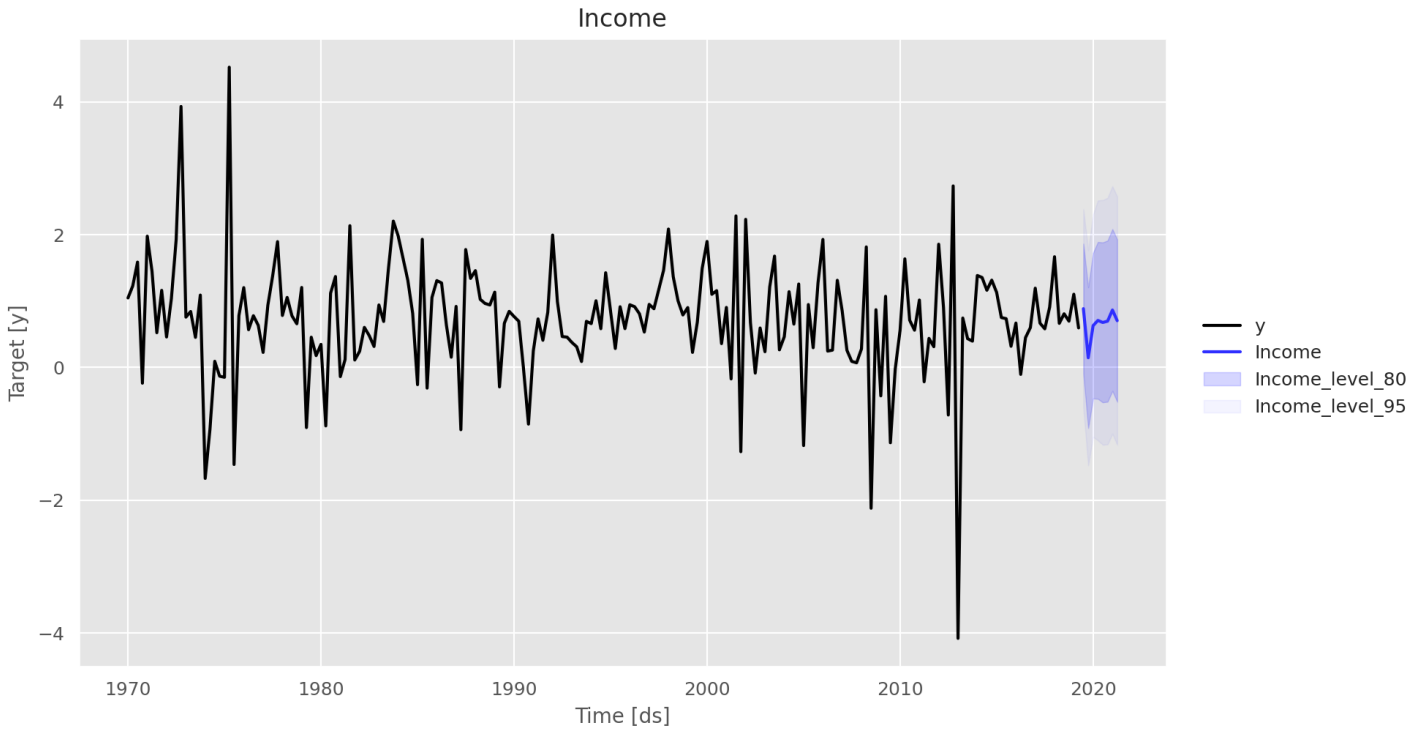

Figure 12.16: Forecasts for US income generated from a VAR(5) model.

12.4 Bootstrapping and bagging

Bootstrapping time series

In the preceding section, and in Section 5.5, we bootstrap the residuals of a time series in order to simulate future values of a series using a model.

More generally, we can generate new time series that are similar to our observed series, using another type of bootstrap.

First, the time series is transformed if necessary, and then decomposed into trend, seasonal and remainder components using STL. Then we obtain shuffled versions of the remainder component to get bootstrapped remainder series. Because there may be autocorrelation present in an STL remainder series, we cannot simply use the re-draw procedure that was described in Section 5.5. Instead, we use a "blocked bootstrap", where contiguous sections of the time series are selected at random and joined together. These bootstrapped remainder series are added to the trend and seasonal components, and the transformation is reversed to give variations on the original time series.

Consider the quarterly cement production in Australia from 1988 Q1 to 2010 Q2. First we check, see Figure 12.17 that the decomposition has adequately captured the trend and seasonality, and that there is no obvious remaining signal in the remainder series. (In `StatsForecast`, an STL decomposition is defined by specifying a single seasonality in `MSTL()`).

```
sf = StatsForecast(models=[MSTL(season_length=4)], freq="QS")

sf = sf.fit(cement)
dcmp = sf.fitted_[0, 0].model_

fig, axes = plt.subplots(nrows=4, ncols=1, sharex=True)
sns.lineplot(data=dcmp, x=dcmp.index, y="data", ax=axes[0], color="black")
axes[0].set_ylabel("Cement")
sns.lineplot(data=dcmp, x=dcmp.index, y="trend", ax=axes[1], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="seasonal", ax=axes[2], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="remainder", ax=axes[3], color="black")
plt.xlabel("")
fig.suptitle("STL decomposition")
fig.text(0.5, 0.94, "Cement = trend + seasonal + remainder", ha='center')
plt.show()
```

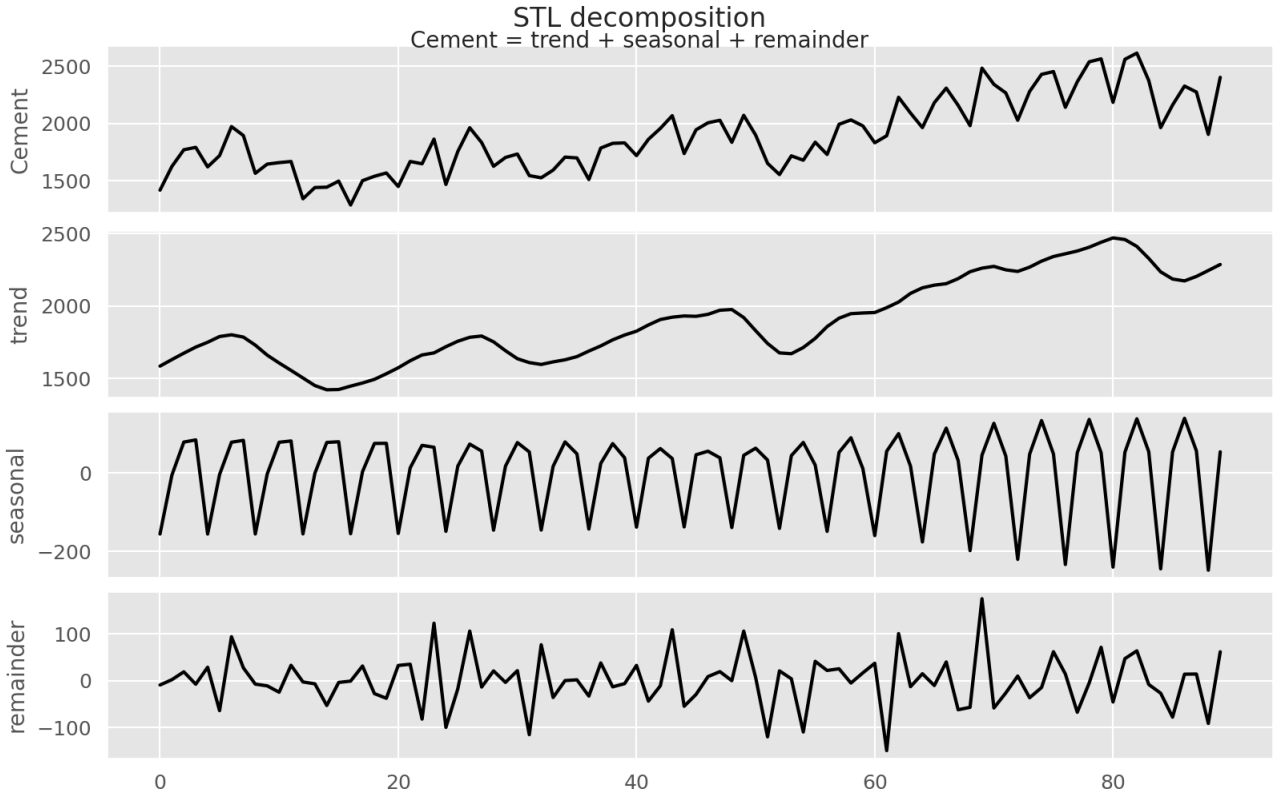

Figure 12.17: STL decomposition of quarterly Australian cement production.

Now we can generate several bootstrapped versions of the data. We will use a block size of 8 to cover two years of data and the `blocked_bootstrap` function to generate the simulated data from the STL decomposition we have just obtained.

```
def blocked_bootstrap(dcmp, block_size, simulations):
    sim = []
    num_blocks = np.ceil(len(dcmp) / block_size).astype(int)
    for _ in range(simulations):
        blocks = []
        for _ in range(num_blocks):
            start = np.random.randint(0, len(dcmp) - block_size + 1)
            end = start + block_size
            blocks.append(dcmp["remainder"].iloc[start:end])
        series = (
            dcmp["trend"]
            + dcmp["seasonal"]
            + pd.concat(blocks, ignore_index=True)[0 : len(dcmp)]
        )
        sim.append(series)

    sim_df = pd.DataFrame(sim).T

    return sim_df


sim_df = blocked_bootstrap(dcmp, 8, 10)
sim_df.insert(0, "unique_id", "aus_production")
sim_df.insert(1, "ds", cement["ds"])
sim_df.rename(columns=str, inplace=True)
plot_series(cement, sim_df,
            xlabel="Quarter [1Q]", ylabel="Tonnes ('000)",
            title="Cement production: Bootstrapped series")
```

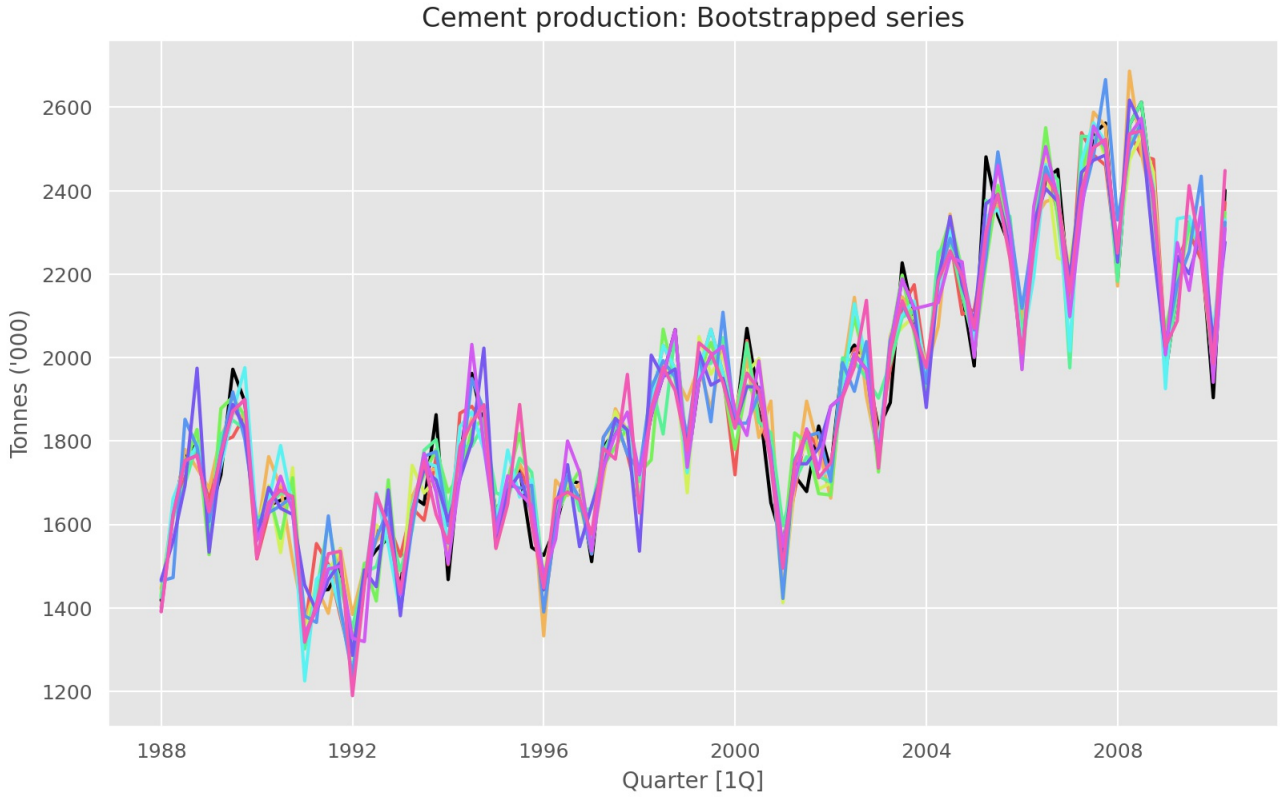

Figure 12.18: Ten bootstrapped versions of quarterly Australian cement production (coloured), along with the original data (black).

Bagged forecasts

One use for these bootstrapped time series is to improve forecast accuracy. If we produce forecasts from each of the additional time series, and average the resulting forecasts, we get better forecasts than if we simply forecast the original time series directly. This is called “bagging” which stands for “**b**ootstrap **agg**regating**g**”.

We demonstrate the idea using the `cement` data. First, we simulate many time series that are similar to the original data using boostrapping.

```
sim_df = blocked_bootstrap(dcmp, 8, 100)
sim_df.insert(0, "unique_id", "aus_production")
sim_df.insert(1, "ds", cement["ds"])
```

For each of these series, we fit an ETS model. A different ETS model is selected for each case, although all models will be similar because the series themselves are similar. However, since the estimated parameters will be slightly different, the forecasts will also be different. This can be a time-consuming process due to the large number of series involved.

```
from matplotlib.dates import YearLocator, DateFormatter

sf = StatsForecast(models=[AutoETS(season_length=4)], freq="QS")

forecasts = []
for i in range(2, len(sim_df.columns)):
    train = sim_df.iloc[:, [0, 1, i]].copy()
    train.rename(columns={sim_df.columns[i]: "y"}, inplace=True)
    fc = sf.forecast(df=train, h=10)
    fc.rename(columns={"AutoETS": f"AutoETS_{i-1}"}, inplace=True)
    forecasts.append(fc)

forecasts_df = reduce(
    lambda left, right: pd.merge(left, right, on=["unique_id", "ds"]), forecasts
)
plot_series(cement, forecasts_df,
            xlabel="Quarter [1Q]", ylabel="Tonnes ('000)",
            title="Cement production: bootstrapped forecasts")
```

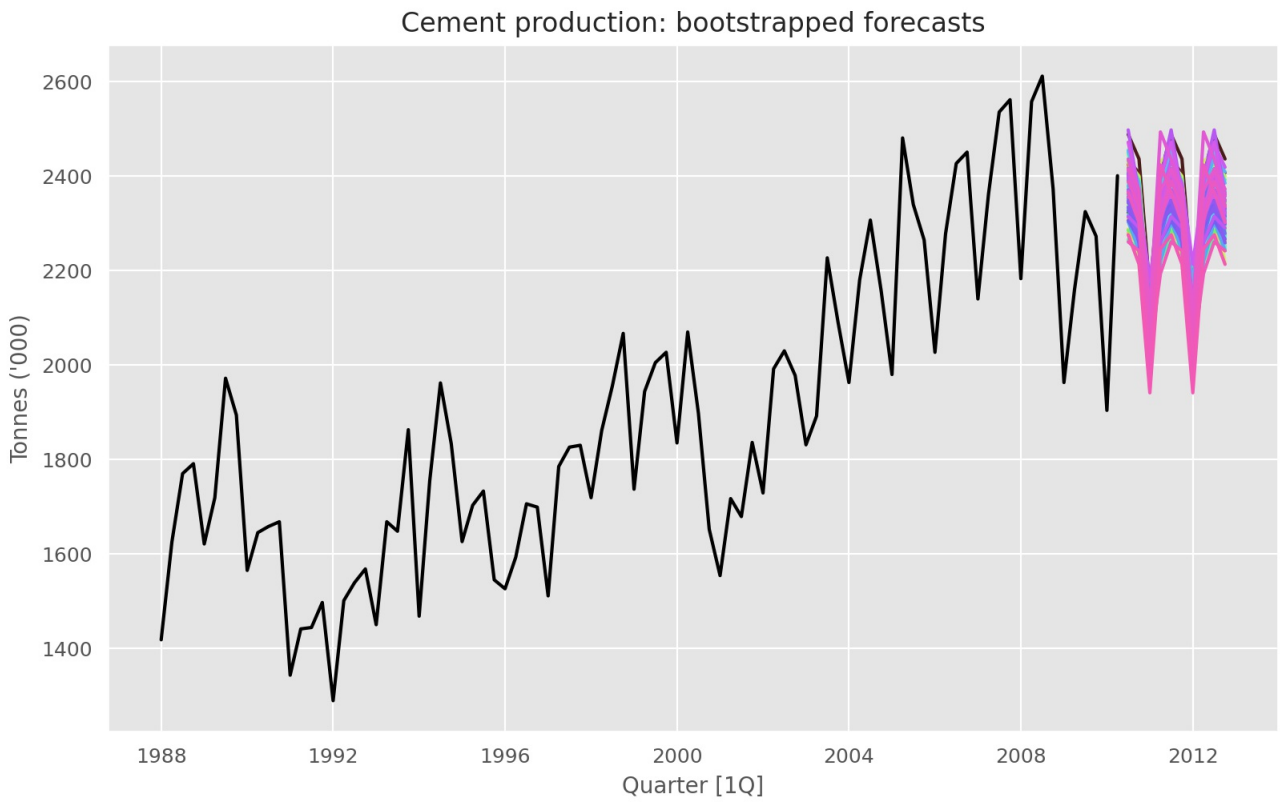

Figure 12.19: Forecasts of 100 bootstrapped series obtained using AutoETS models.

Finally, we average these forecasts for each time period to obtain the “bagged forecasts” for the original data. We also generate an ETS model applied directly to the data for comparison.

```
fc_ets = sf.forecast(df=cement, h=10)
forecasts_df["AvgAutoETS"] = forecasts_df[
    [col for col in forecasts_df.columns if col.startswith("AutoETS")]
].mean(axis=1)
plot_series(
    cement,
    fc_ets.merge(
        forecasts_df[["unique_id", "ds", "AvgAutoETS"]], on=["unique_id", "ds"]
    ),
    xlabel="Quarter [1Q]", ylabel="Tonnes ('000)",
    title="Cement production in Australia", rm_legend=False
)
```

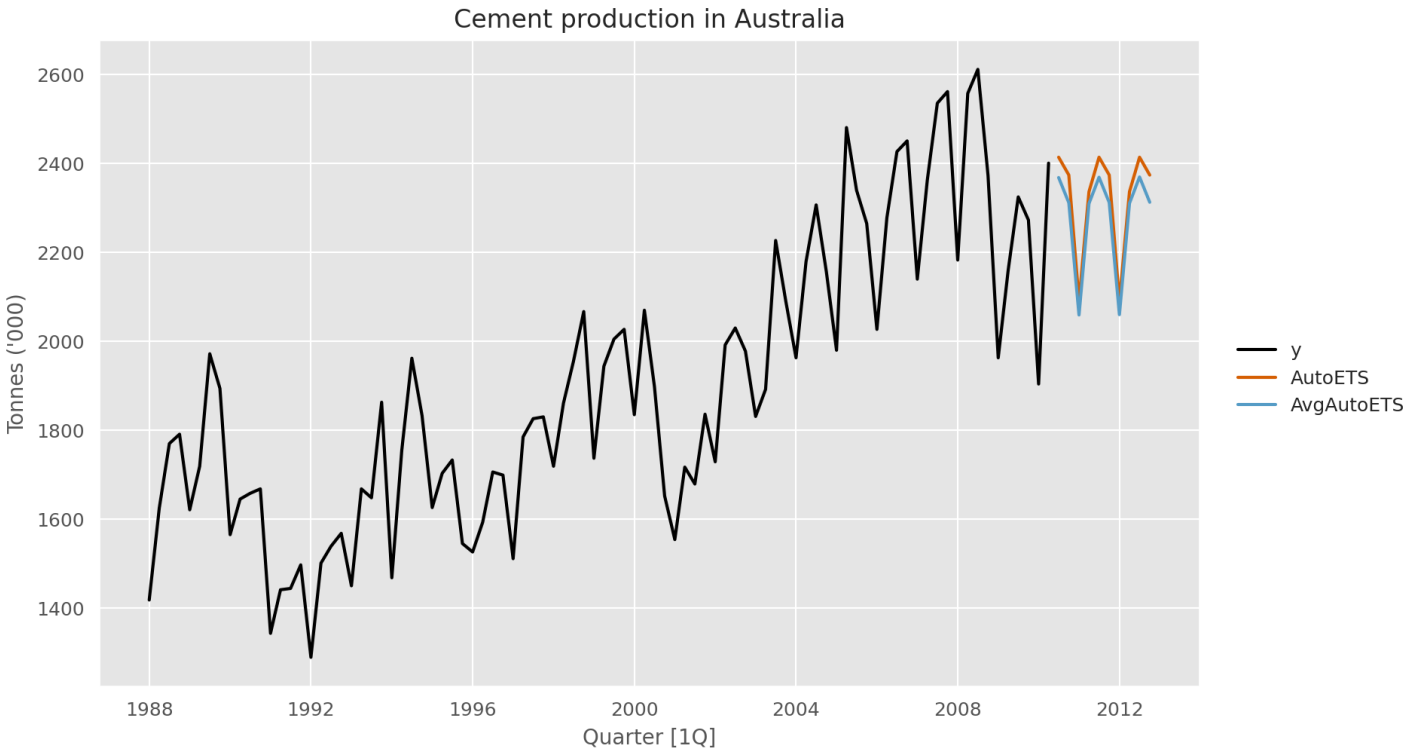

Figure 12.20: Comparing bagged MSTL forecasts (the average of 100 bootstrapped forecasts) and MSTL applied directly to the data.

Bergmeir, Hyndman, and Benítez (2016a) show that, on average, bagging gives better forecasts than just applying `AutoETS()` directly. Of course, it is slower because a lot more computation is required.

12.5 Exercises

1. Compare STL and Dynamic Harmonic Regression forecasts for one of the series in the `pedestrian` data set.
    a. Try modifying the order of the Fourier terms to minimize the AICc value.
    b. Check the residuals for each model. Do they capture the available information in the data?
    c. Which of the two sets of forecasts are best? Explain.
2. Consider the weekly data on US finished motor gasoline products supplied (millions of barrels per day) (series `us_gasoline`).
    a. Use the Prophet model to forecast the next 13 weeks of data.
    b. Repeat the previous exercise using the `AutoARIMA()` and the `AutoETS()` classes from StatsForecast.
    c. Compare the accuracy of the models you generated.

12.6 Further reading

- The Prophet model is described in Taylor and Letham (2018).
- Pfaff (2008) provides a book-length overview of VAR modelling and other multivariate time series models.
- A current survey of the use of recurrent neural networks for forecasting is provided by Hewamalage, Bergmeir, and Bandara (2021).
- Bootstrapping for time series is discussed in Lahiri (2003).
- Bagging for time series forecasting is relatively new. Bergmeir, Hyndman, and Benítez (2016b) is one of the few papers which addresses this topic.

12.7 Used modules and classes

StatsForecast

- `StatsForecast` class- Core forecasting engine
- `MSTL` model - For multiple seasonal decomposition
- `AutoARIMA` model - For automatic ARIMA modeling
- `AutoETS` model - For automatic exponential smoothing
- `AutoARIMAProphet` adapter - Prophet adapter with improved performance

UtilsForecast

- `plot_series` utility - For creating time series visualizations
- `evaluate` utility - For model evaluation metrics

---

1. A more flexible generalisation would be a Vector ARMA process. However, the relative simplicity of VARs has led to their dominance in forecasting. Interested readers may refer to Athanasopoulos, Poskitt, and Vahid (2012). □□

2. Interested readers should refer to Hamilton (1994) and Lütkepohl (2007). □□

← Chapter 11 Forecasting hierarchical and grouped time series

Chapter 13 Some practical forecasting issues →