Chapter 12 Advanced forecasting methods

Ankit Gupta

10/04/2023'

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

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.

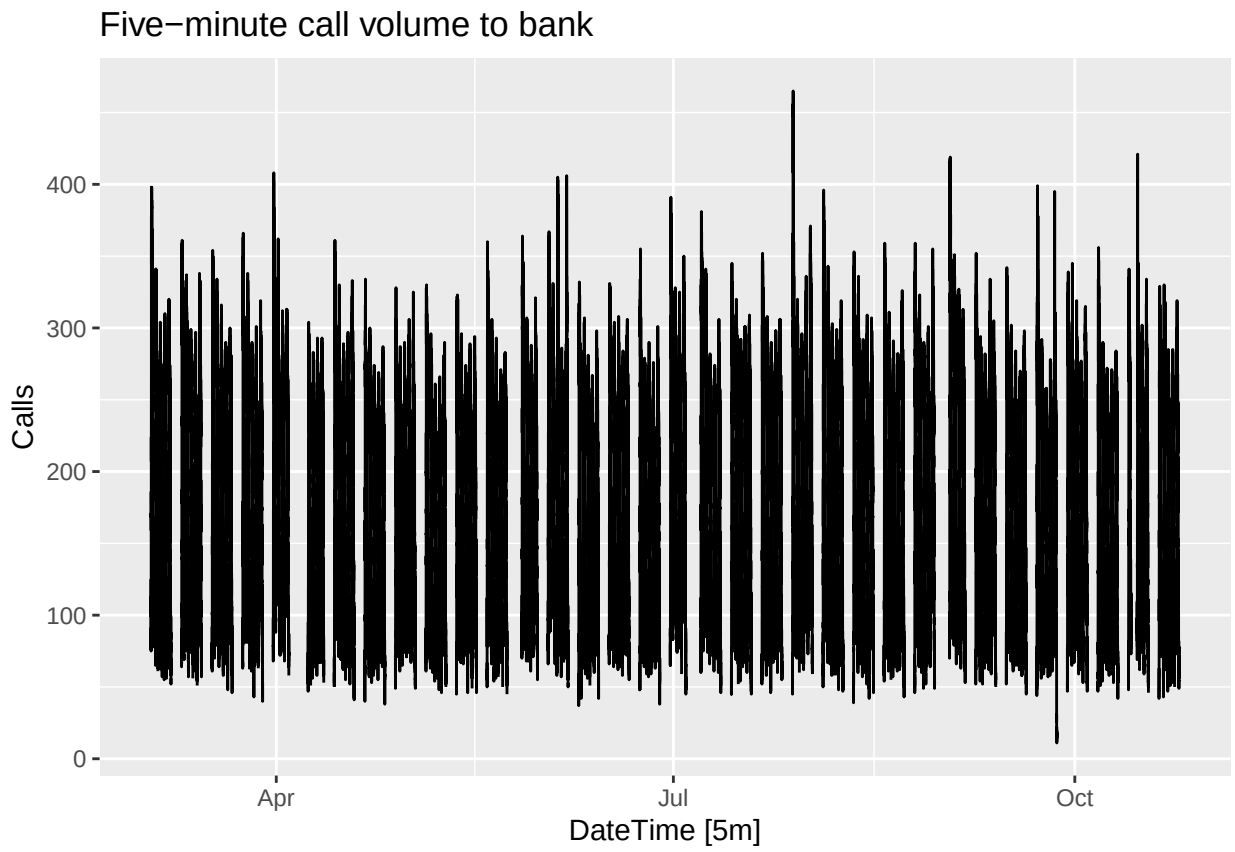
Figure 12.1 shows the number of calls to a North American commercial bank per 5-minute interval between 7:00am and 9:05pm each weekday over a 33 week period. The lower panel shows the first four weeks of the same time series. There is a strong daily seasonal pattern with period 169 (there are 169 5-minute 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.

```
library(fpp3)
```

```
## -- Attaching packages ------------------------------------------------ fpp3 0.5 --
## v tibble      3.2.1     v tsibble     1.1.3
## v dplyr       1.1.3     v tsibbledata 0.4.1
## v tidyr       1.3.0     v feasts      0.3.1
## v lubridate   1.9.3     v fable       0.3.3
## v ggplot2     3.4.4     v fabletools  0.3.4
## -- Conflicts --------------------------------------------------- fpp3_conflicts --
## x lubridate::date()    masks base::date()
## x dplyr::filter()      masks stats::filter()
## x tsibble::intersect() masks base::intersect()
## x tsibble::interval()  masks lubridate::interval()
## x dplyr::lag()         masks stats::lag()
## x tsibble::setdiff()   masks base::setdiff()
## x tsibble::union()     masks base::union()
```

```
bank_calls |>
  fill_gaps() |>
  autoplot(Calls) +
```

```
  labs(y = "Calls",
       title = "Five-minute call volume to bank")
```

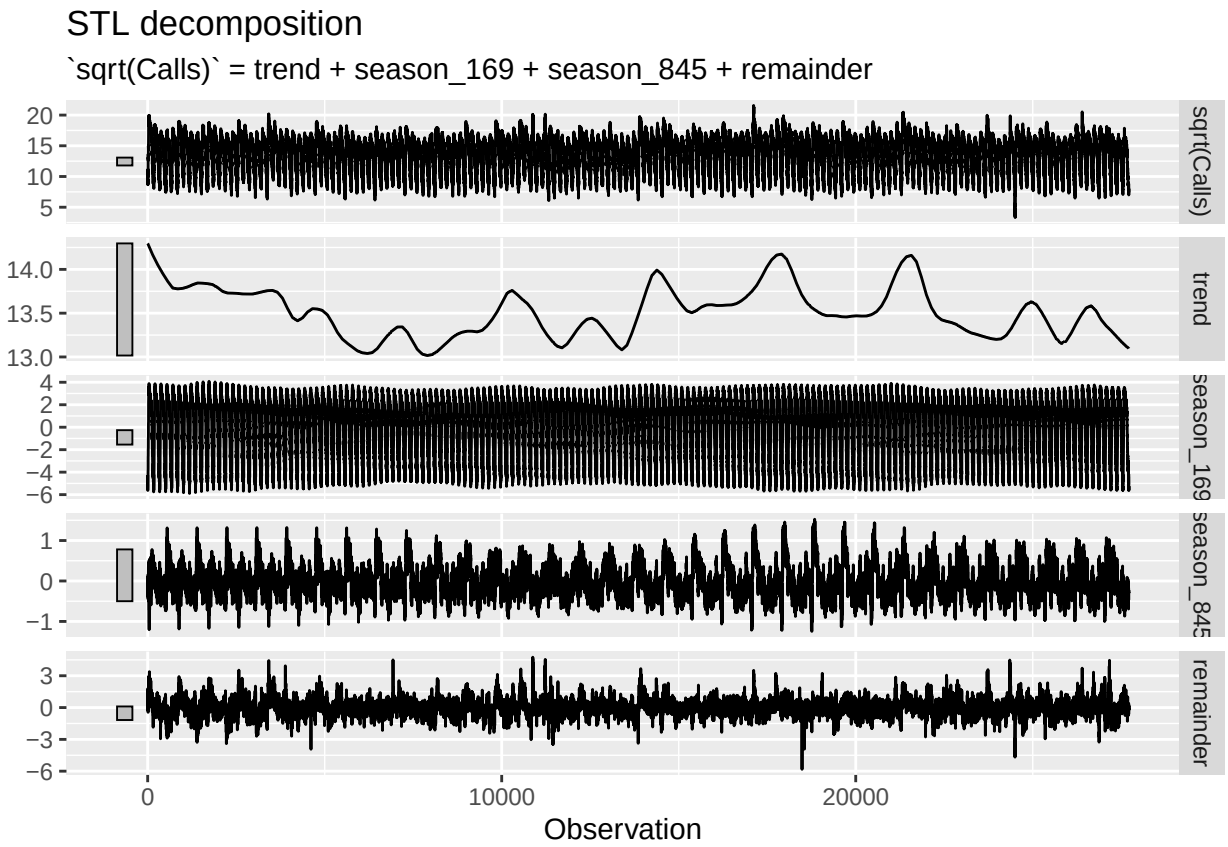


Apart from the multiple seasonal periods, this series has the additional complexity of missing values between the working periods.

STL with multiple seasonal periods

The STL() function is designed to deal with multiple seasonality. It will return multiple seasonal components, as well as a trend and remainder component. In this case, we need to re-index the tsibble to avoid the missing values, and then explicitly give the seasonal periods.

```
calls <- bank_calls |>
  mutate(t = row_number()) |>
  update_tsibble(index = t, regular = TRUE)
calls |>
  model(
    STL(sqrt(Calls) ~ season(period = 169) +
                      season(period = 5*169),
        robust = TRUE)
  ) |>
  components() |>
  autoplot() + labs(x = "Observation")
```



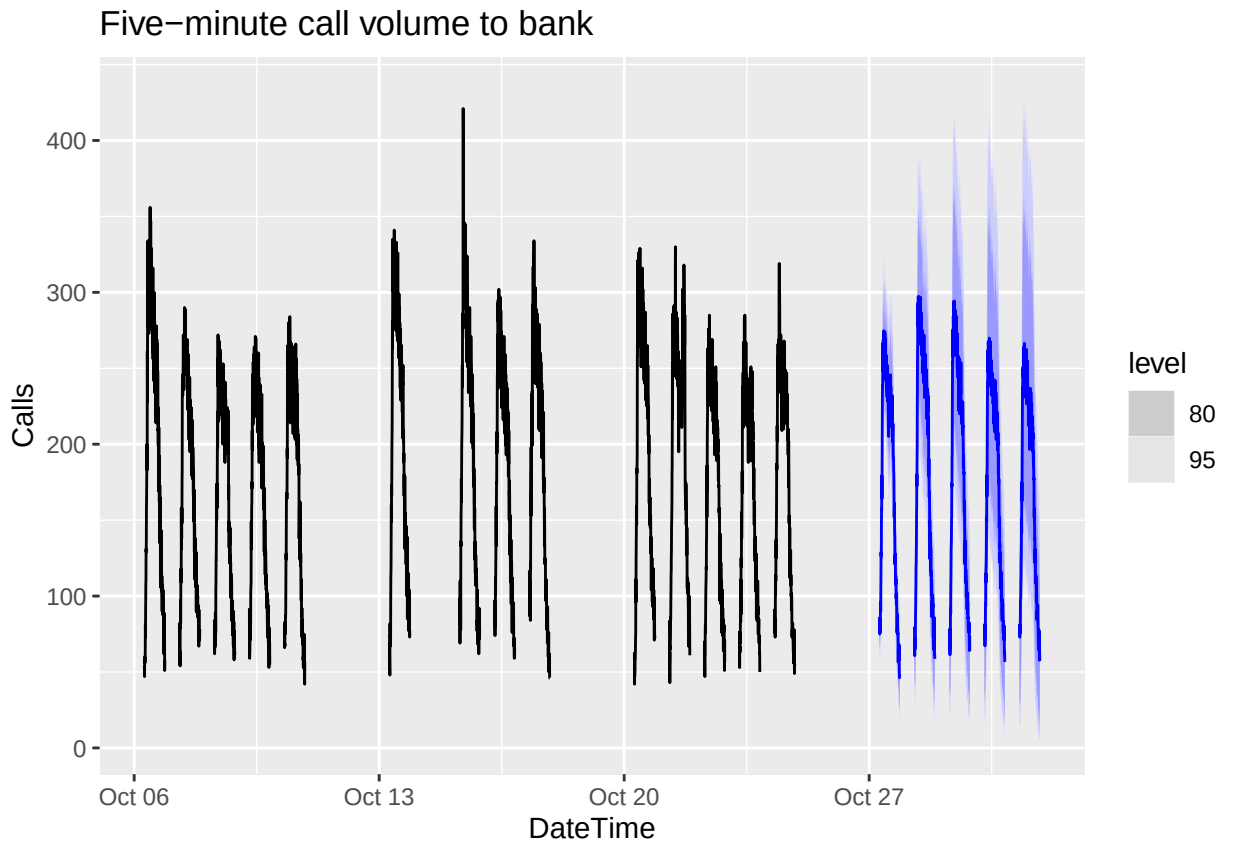
There are two seasonal patterns shown, one for the time of day (the third panel), and one for the time of week (the fourth panel). To properly interpret this graph, it is important to notice the vertical scales. In this case, the trend and the weekly seasonality have wider bars (and therefore relatively narrower ranges) compared to the other components, because 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 code is slightly more complicated than usual because we have to add back the time stamps that were lost when we re-indexed the tsibble to handle the periods of missing observations. The square root transformation used in the STL decomposition has ensured the forecasts remain positive.

```
# Forecasts from STL+ETS decomposition
my_dcmp_spec <- decomposition_model(
  STL(sqrt(Calls) ~ season(period = 169) +
                    season(period = 5*169),
      robust = TRUE),
  ETS(season_adjust ~ season("N"))
)
fc <- calls |>
  model(my_dcmp_spec) |>
  forecast(h = 5 * 169)

# Add correct time stamps to fable
fc_with_times <- bank_calls |>
  new_data(n = 7 * 24 * 60 / 5) |>
  mutate(time = format(DateTime, format = "%H:%M:%S")) |>
```

```
  filter(
    time %in% format(bank_calls$DateTime, format = "%H:%M:%S"),
    wday(DateTime, week_start = 1) <= 5
  ) |>
  mutate(t = row_number() + max(calls$t)) |>
  left_join(fc, by = "t") |>
  as_fable(response = "Calls", distribution = Calls)

# Plot results with last 3 weeks of data
fc_with_times |>
  fill_gaps() |>
  autoplot(bank_calls |> tail(14 * 169) |> fill_gaps()) +
  labs(y = "Calls",
       title = "Five-minute call volume to bank")
```




```
#> Warning: Removed 100 rows containing missing values or values outside the
#> scale range (`geom_line()`).
```

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 $\sin(2\pi kt169)$,$\cos(\frac{2\pi kt}{169})$,$\sin(\frac{2\pi kt}{845})$,and $\cos(\frac{2\pi kt}{845})$,

for k=1,2,…. 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. Again, we will use a square root transformation to ensure the forecasts and prediction intervals remain positive. 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.

```
fit <- calls |>
  model(
    dhr = ARIMA(sqrt(Calls) ~ PDQ(0, 0, 0) + pdq(d = 0) +
                  fourier(period = 169, K = 10) +
                  fourier(period = 5*169, K = 5)))
```

```
## Warning: Provided exogenous regressors are rank deficient, removing regressors:
## `fourier(period = 5 * 169, K = 5)C5_845`, `fourier(period = 5 * 169, K =
## 5)S5_845`
```

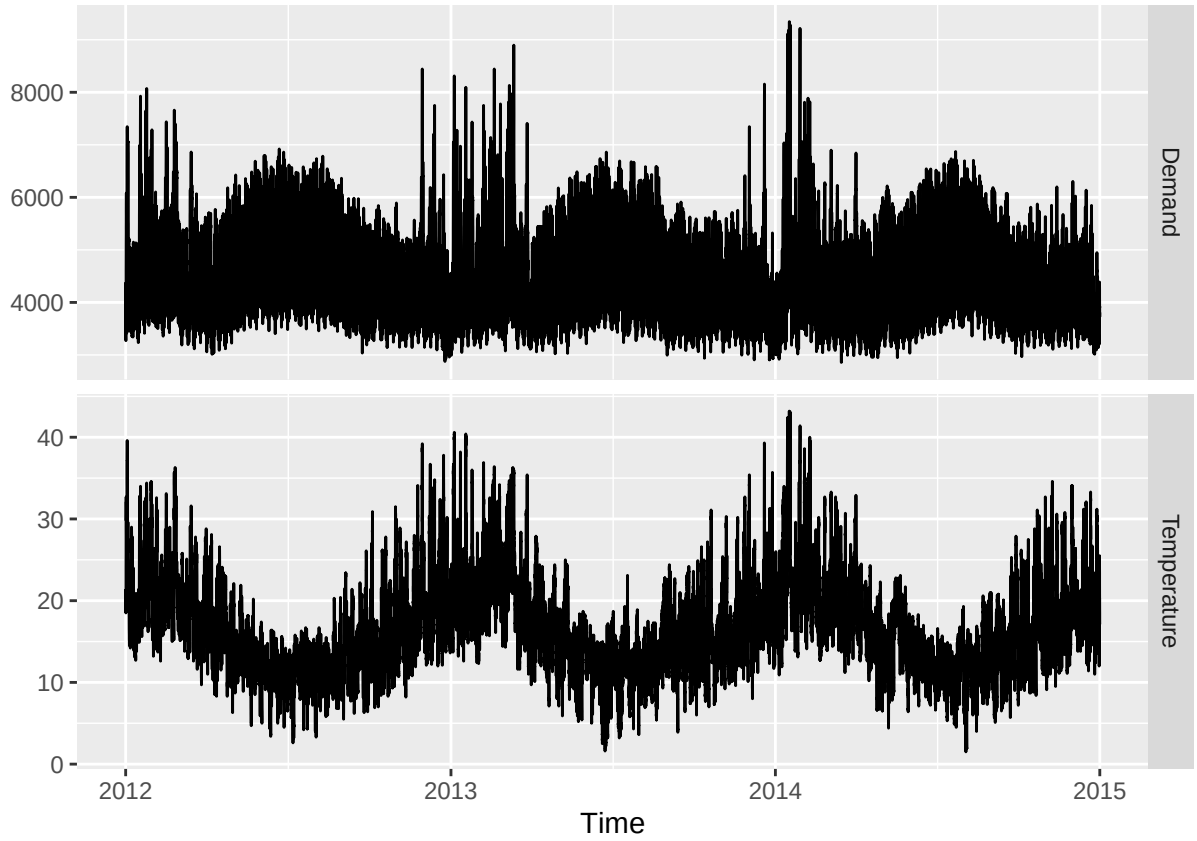
```
fc <- fit |> forecast(h = 5 * 169)

# Add correct time stamps to fable
fc_with_times <- bank_calls |>
  new_data(n = 7 * 24 * 60 / 5) |>
  mutate(time = format(DateTime, format = "%H:%M:%S")) |>
  filter(
    time %in% format(bank_calls$DateTime, format = "%H:%M:%S"),
    wday(DateTime, week_start = 1) <= 5
  ) |>
  mutate(t = row_number() + max(calls$t)) |>
  left_join(fc, by = "t") |>
  as_fable(response = "Calls", distribution = Calls)
```

This is a large model, containing 33 parameters: 4 ARMA coefficients, 20 Fourier coefficients for period 169, and 8 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 $845 = 5 \times 169$).

Example: Electricity demand

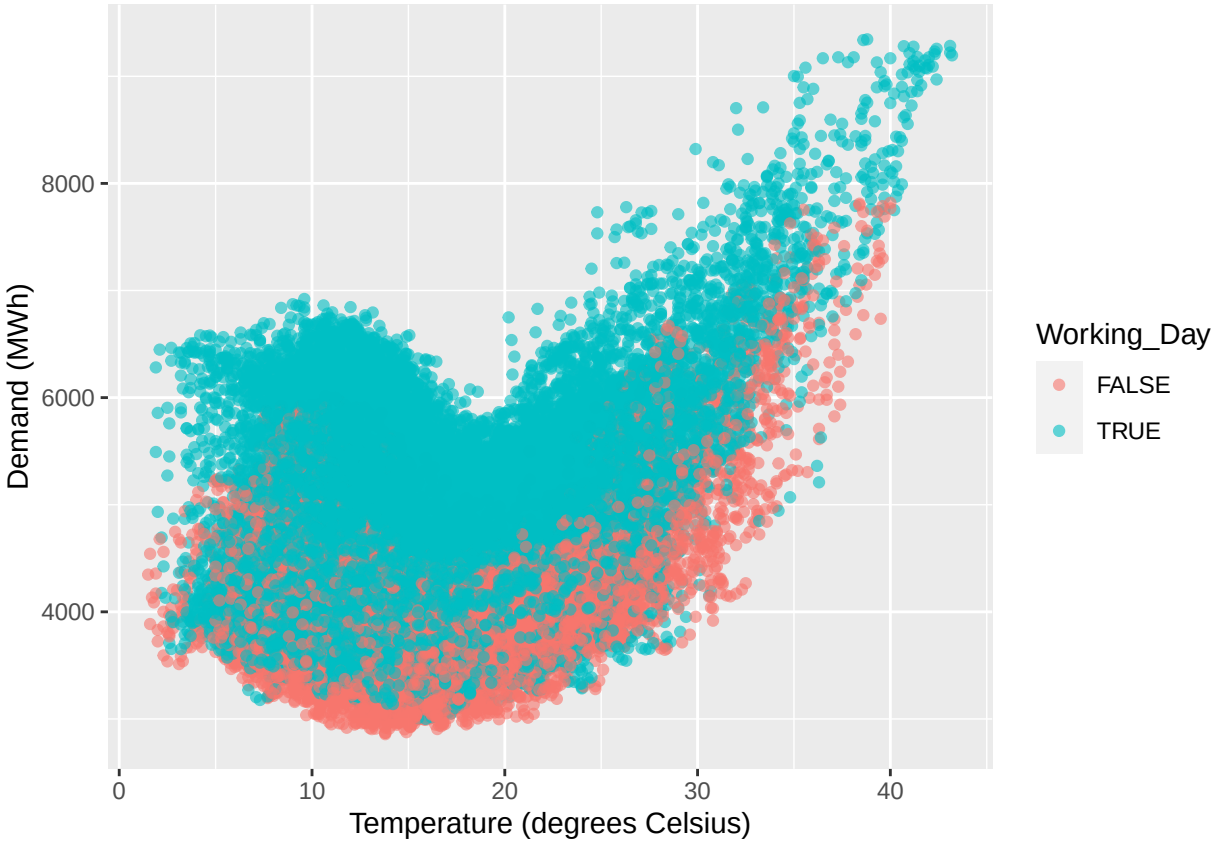
One common application of such models is electricity demand modelling. Figure 12.5 shows half-hourly electricity demand (MWh) in Victoria, Australia, during 2012–2014, along with temperatures (degrees Celsius) for the same period for Melbourne (the largest city in Victoria).

```
vic_elec |>
  pivot_longer(Demand:Temperature, names_to = "Series") |>
  ggplot(aes(x = Time, y = value)) +
  geom_line() +
  facet_grid(rows = vars(Series), scales = "free_y") +
  labs(y = "")
```



Plotting electricity demand against temperature (Figure 12.6) 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).

```
elec <- vic_elec |>
  mutate(
    DOW = wday(Date, label = TRUE),
    Working_Day = !Holiday & !(DOW %in% c("Sat", "Sun")),
    Cooling = pmax(Temperature, 18)
  )
elec |>
  ggplot(aes(x=Temperature, y=Demand, col=Working_Day)) +
  geom_point(alpha = 0.6) +
  labs(x="Temperature (degrees Celsius)", y="Demand (MWh)")
```



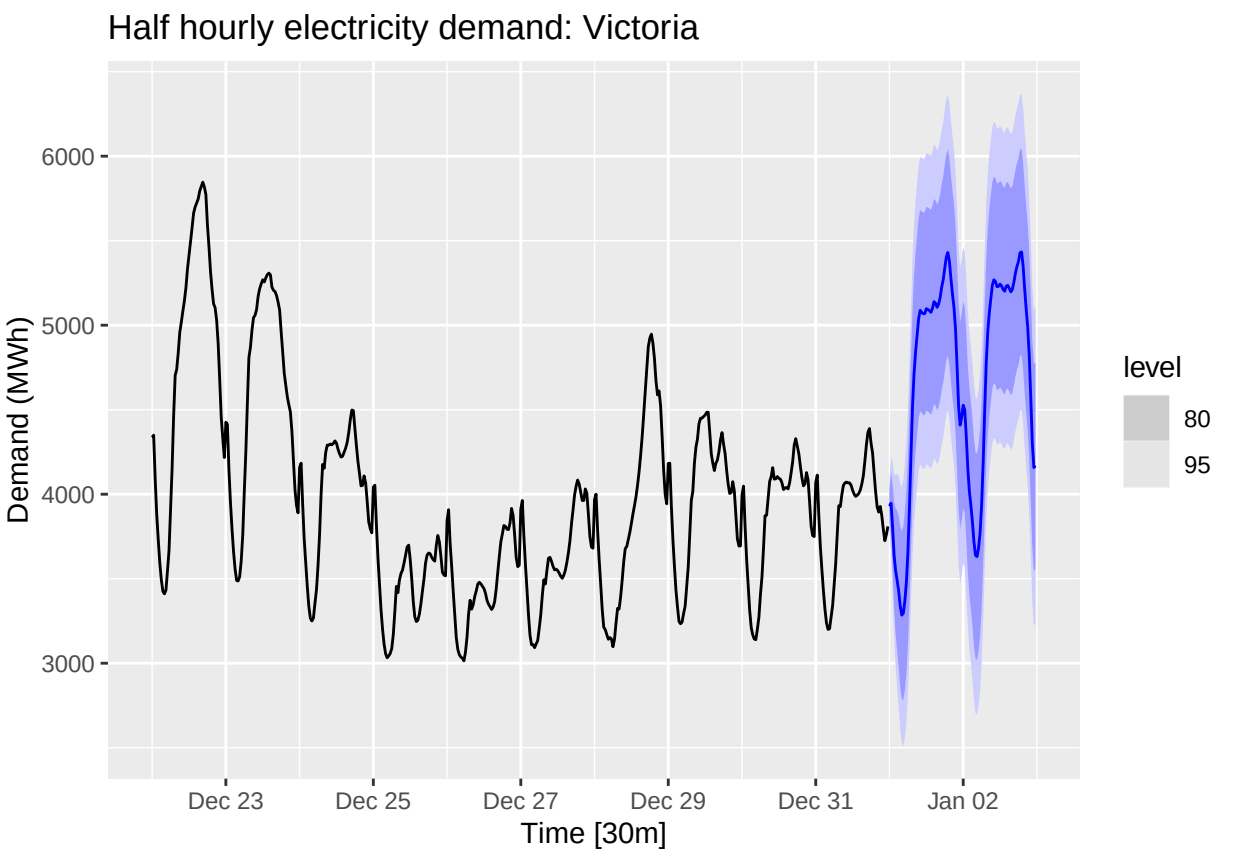
We will fit a regression model with a piecewise linear function of temperature (containing a knot at 18 degrees), and harmonic regression terms to allow for the daily seasonal pattern. Again, we set the orders of the Fourier terms subjectively, while using the AICc to select the order of the ARIMA errors.

```
fit <- elec |>
  model(
    ARIMA(Demand ~ PDQ(0, 0, 0) + pdq(d = 0) +
          Temperature + Cooling + Working_Day +
          fourier(period = "day", K = 10) +
          fourier(period = "week", K = 5) +
          fourier(period = "year", K = 3))
  )
```

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.

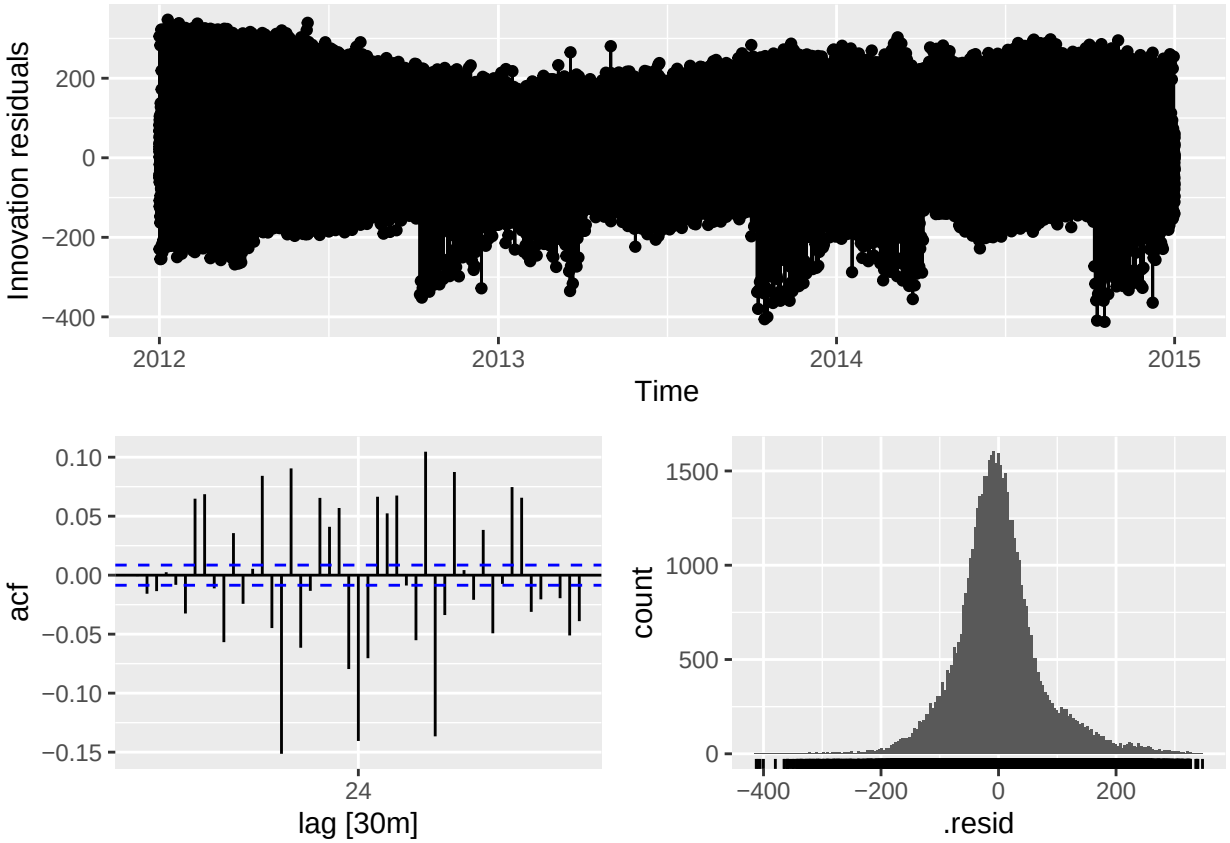
```
elec_newdata <- new_data(elec, 2*48) |>
  mutate(
    Temperature = tail(elec$Temperature, 2 * 48),
    Date = lubridate::as_date(Time),
    DOW = wday(Date, label = TRUE),
    Working_Day = (Date != "2015-01-01") &
                  !(DOW %in% c("Sat", "Sun")),
```

```
    Cooling = pmax(Temperature, 18)
  )
fc <- fit |>
  forecast(new_data = elec_newdata)

fc |>
  autoplot(elec |> tail(10 * 48)) +
  labs(title="Half hourly electricity demand: Victoria",
       y = "Demand (MWh)", x = "Time [30m]")
```



Although the short-term forecasts look reasonable, this is a crude model for a complicated process. The residuals, plotted in Figure 12.8, demonstrate that there is a lot of information that has not been captured with this model.

```
fit |> gg_tsresiduals()
```



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

12.2 Prophet model

A recent proposal is the Prophet model, available via the fable.prophet package. This model was introduced by Facebook (S. J. Taylor & 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 ε_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.

```
library(fable.prophet)
```

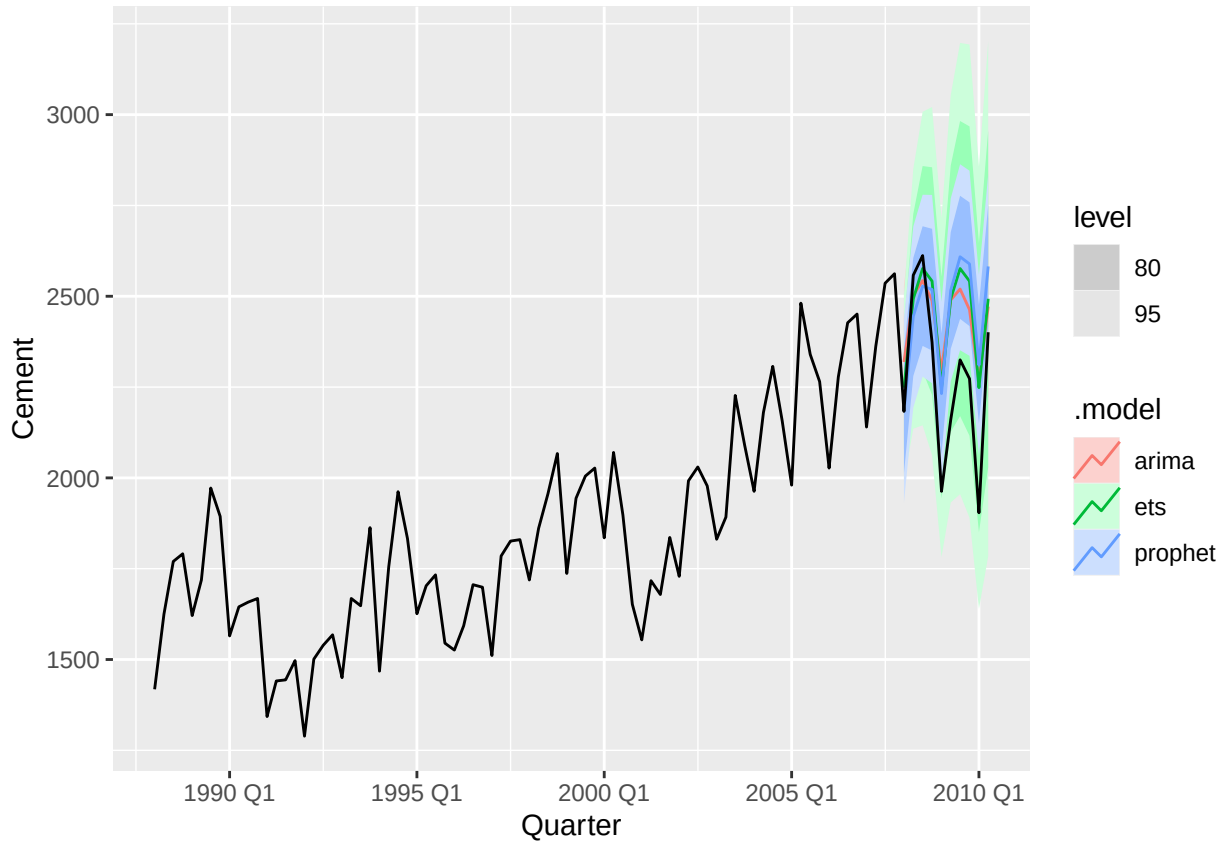
```
## Warning: package 'fable.prophet' was built under R version 4.3.3
```

```
## Loading required package: Rcpp
```

```
cement <- aus_production |>
  filter(year(Quarter) >= 1988)
train <- cement |>
  filter(year(Quarter) <= 2007)
fit <- train |>
  model(
    arima = ARIMA(Cement),
    ets = ETS(Cement),
    prophet = prophet(Cement ~ season(period = 4, order = 2,
                                      type = "multiplicative"))
  )
```

Note that the seasonal term must have the period fully specified for quarterly and monthly data, as the default values assume the data are observed at least daily.

```
fc <- fit |> forecast(h = "2 years 6 months")
fc |> autoplot(cement)
```




```
fc |> accuracy(cement)
```

```
## # A tibble: 3 x 10
##   .model  .type    ME  RMSE   MAE   MPE  MAPE  MASE RMSSE  ACF1
##   <chr>   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 arima   Test  -161.  216.  186. -7.71  8.68  1.27  1.26 0.387
## 2 ets     Test  -171.  222.  191. -8.07  8.85  1.30  1.29 0.579
## 3 prophet Test  -176.  249.  216. -8.37  9.90  1.47  1.45 0.699
```

```
#> # A tibble: 3 × 10
#>   .model  .type    ME  RMSE   MAE   MPE  MAPE  MASE RMSSE  ACF1
#>   <chr>   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 arima   Test  -161.  216.  186. -7.71  8.68  1.27  1.26 0.387
#> 2 ets     Test  -171.  222.  191. -8.07  8.85  1.30  1.29 0.579
#> 3 prophet Test  -176.  248.  216. -8.36  9.89  1.47  1.44 0.699
```

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

Example: Half-hourly electricity demand

We will fit a similar model to the dynamic harmonic regression (DHR) model from the previous section, but this time using a Prophet model. For daily and sub-daily data, the default periods are correctly specified, so that we can simply specify the period using a character string as follows.

```
fit <- elec |>
  model(
    prophet(Demand ~ Temperature + Cooling + Working_Day +
            season(period = "day", order = 10) +
```

```
            season(period = "week", order = 5) +
            season(period = "year", order = 3))
  )
fit |>
  components() |>
  autoplot()
```

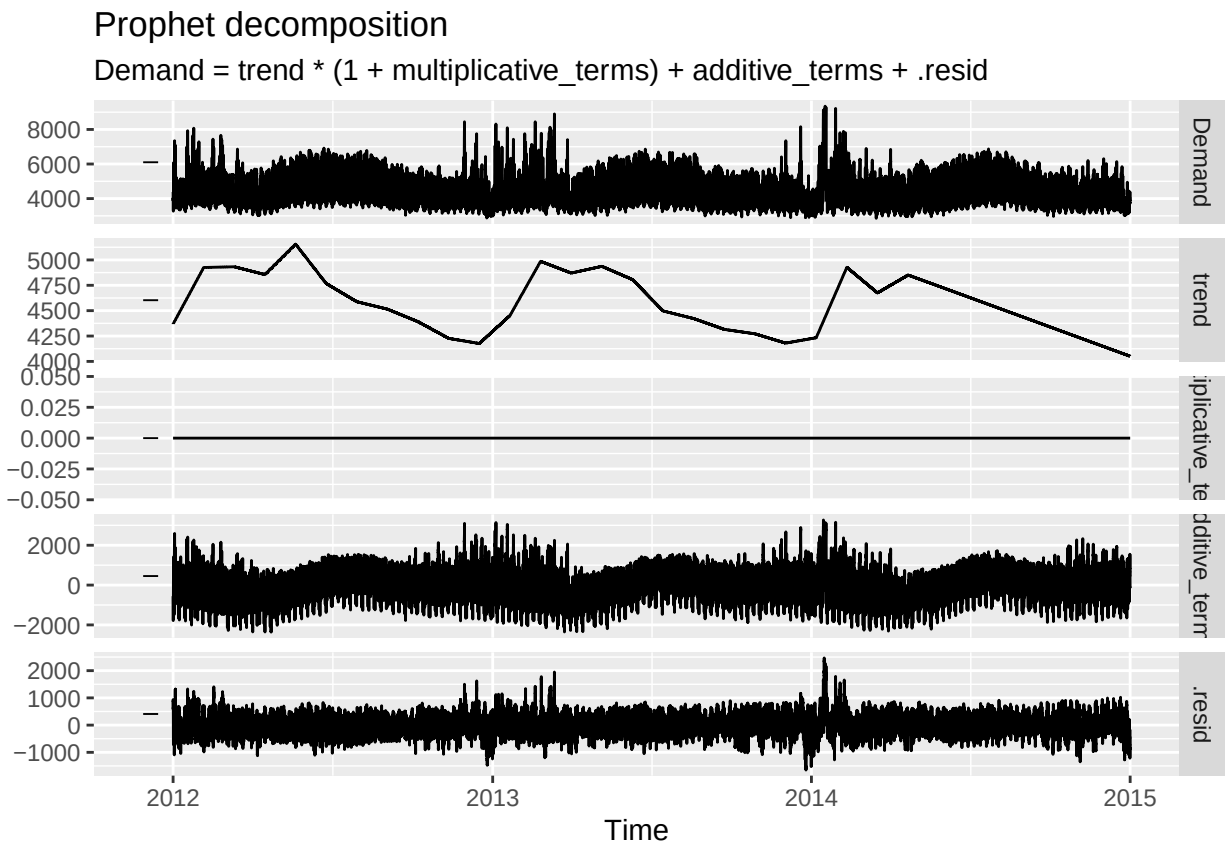
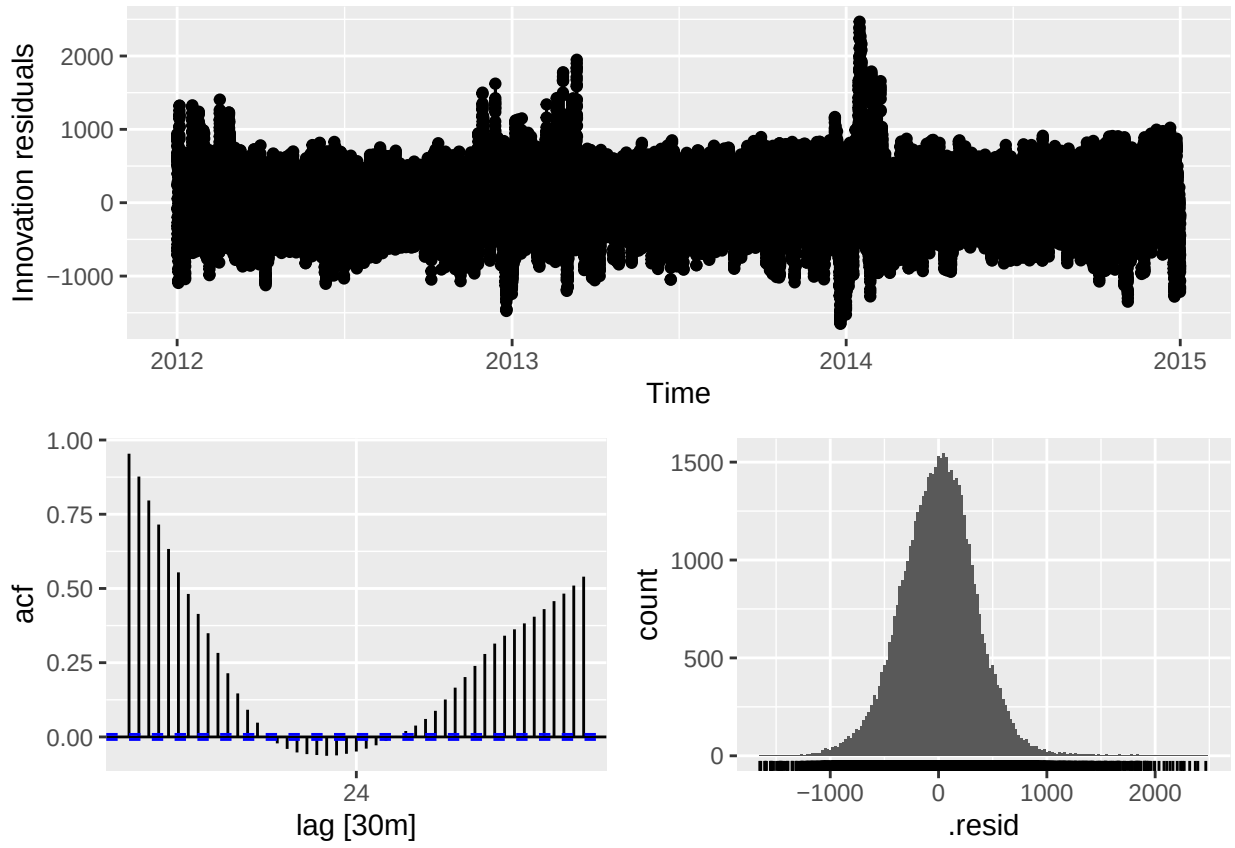


Figure 12.10 shows the trend and seasonal components of the fitted model.

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.

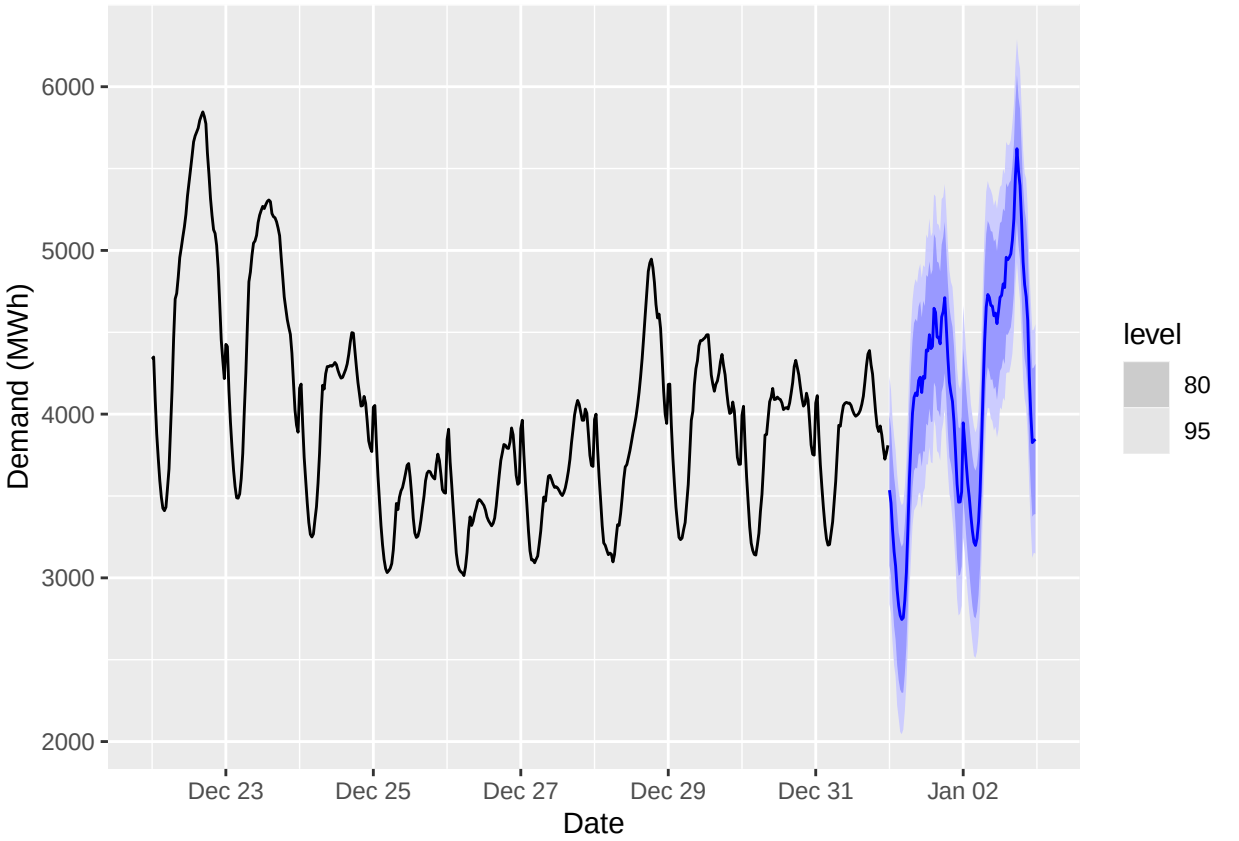
There is also substantial remaining autocorrelation in the residuals.

```
fit |> gg_tsresiduals()
```



The prediction intervals would be narrower if the autocorrelations were taken into account.

```
fc <- fit |>
  forecast(new_data = elec_newdata)
fc |>
  autoplot(elec |> tail(10 * 48)) +
  labs(x = "Date", y = "Demand (MWh)")
```



Prophet has the advantage of being much faster to estimate than the DHR models we have considered previously, and it is completely automated. However, it rarely gives better forecast accuracy than the alternative approaches, as these two examples have illustrated.

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 Section 10.2, 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 t^{th} observation of variable y_1, $y_{2,t}$ denotes the t^{th} 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.26 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

$y_{1,t} = c_1 + \phi_{11,1} y_{1,t-1} + \phi_{12,1} y_{2,t-1} + \varepsilon_{1,t}$ (12.1)

$y_{2,t} = c_2 + \phi_{21,1} y_{1,t-1} + \phi_{22,1} y_{2,t-1} + \varepsilon_{2,t}$, (12.2)

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 ℓ_{th} lag of variable y_i on itself, while the coefficient $\phi_{ij,\ell}$ captures the influence of the ℓ^{th} 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.27

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)–(12.2), for all observations up to timeT. Then the one-step-ahead forecasts are generated by

$\hat{y}_{1,T+1|T} = \hat{c_1} + \hat{\phi}_{11,1} y_{1,T} + \hat{\phi}_{12,1} y_{2,T}$

$\hat{y}_{2,T+1|T} = \hat{c_2} + \hat{\phi}_{21,1} y_{1,T} + \hat{\phi}_{22,1} y_{2,T}$.

This is the same form as (12.1)–(12.2), 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

$\hat{y}_{1,T+2|T} = \hat{c_1} + \hat{\phi}_{11,1} y_{1,T+1|T} + \hat{\phi}_{12,1} y_{2,T+1|T}$

$\hat{y}_{2,T+2|T} = \hat{c_2} + \hat{\phi}_{21,1} y_{1,T+1|T} + \hat{\phi}_{22,1} y_{2,T+1|T}$.

Again, this is the same form as (12.1)–(12.2), 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:

- forecasting a collection of related variables where no explicit interpretation is required;
- testing whether one variable is useful in forecasting another (the basis of Granger causality tests);

- impulse response analysis, where the response of one variable to a sudden but temporary change in another variable is analysed;
- 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

```
fit <- us_change |>
  model(
    aicc = VAR(vars(Consumption, Income)),
    bic = VAR(vars(Consumption, Income), ic = "bic")
  )
fit
```

```
## # A mable: 1 x 2
##               aicc              bic
##            <model>          <model>
## 1 <VAR(5) w/ mean> <VAR(1) w/ mean>
```

```
#> # A mable: 1 x 2
#>               aicc              bic
#>            <model>          <model>
#> 1 <VAR(5) w/ mean> <VAR(1) w/ mean>
```

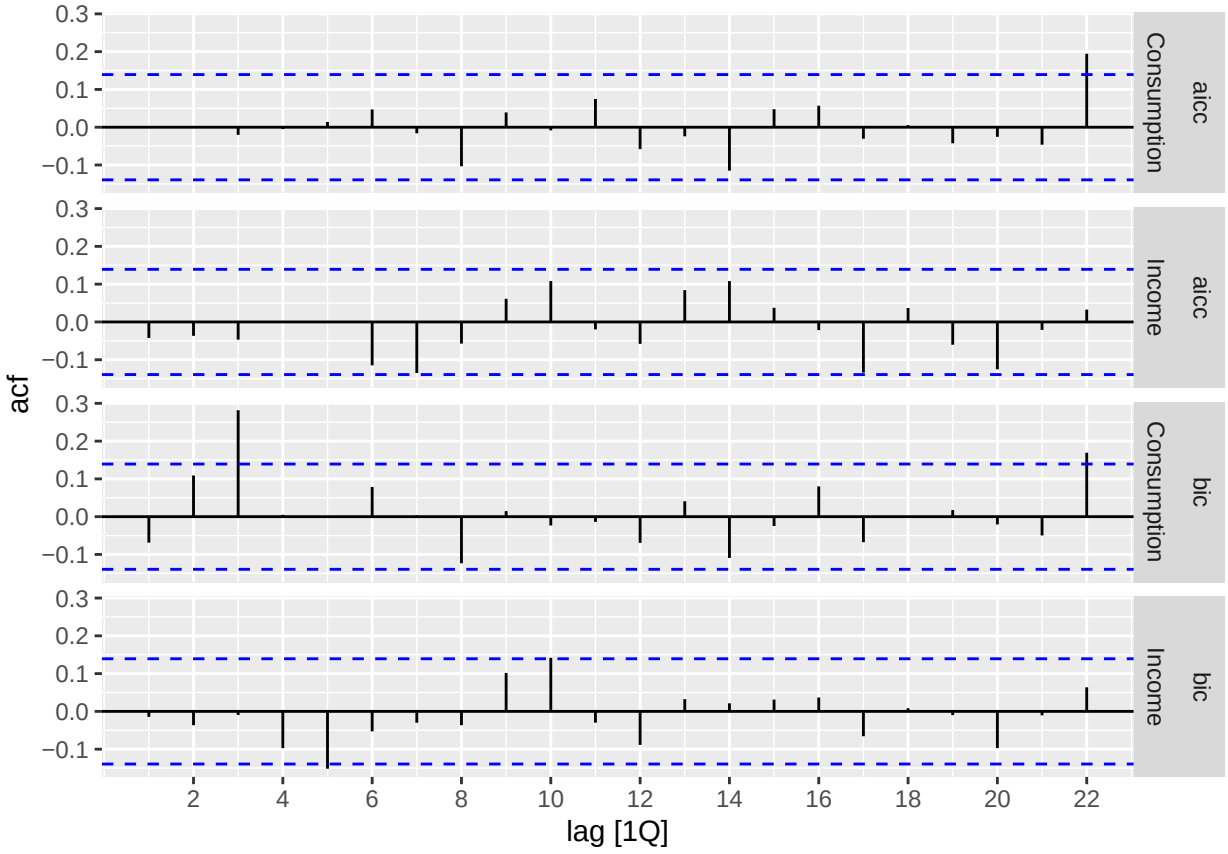
```
glance(fit)
```

```
## # A tibble: 2 x 6
##   .model sigma2        log_lik   AIC  AICc   BIC
##   <chr>  <list>          <dbl> <dbl> <dbl> <dbl>
## 1 aicc   <dbl [2 x 2]>   -373.  798.  806.  883.
## 2 bic    <dbl [2 x 2]>   -408.  836.  837.  869.
```

```
#> # A tibble: 2 × 6
#>   .model sigma2        log_lik   AIC  AICc   BIC
#>   <chr>  <list>          <dbl> <dbl> <dbl> <dbl>
#> 1 aicc   <dbl [2 × 2]>   -373.  798.  806.  883.
#> 2 bic    <dbl [2 × 2]>   -408.  836.  837.  869.
```

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.

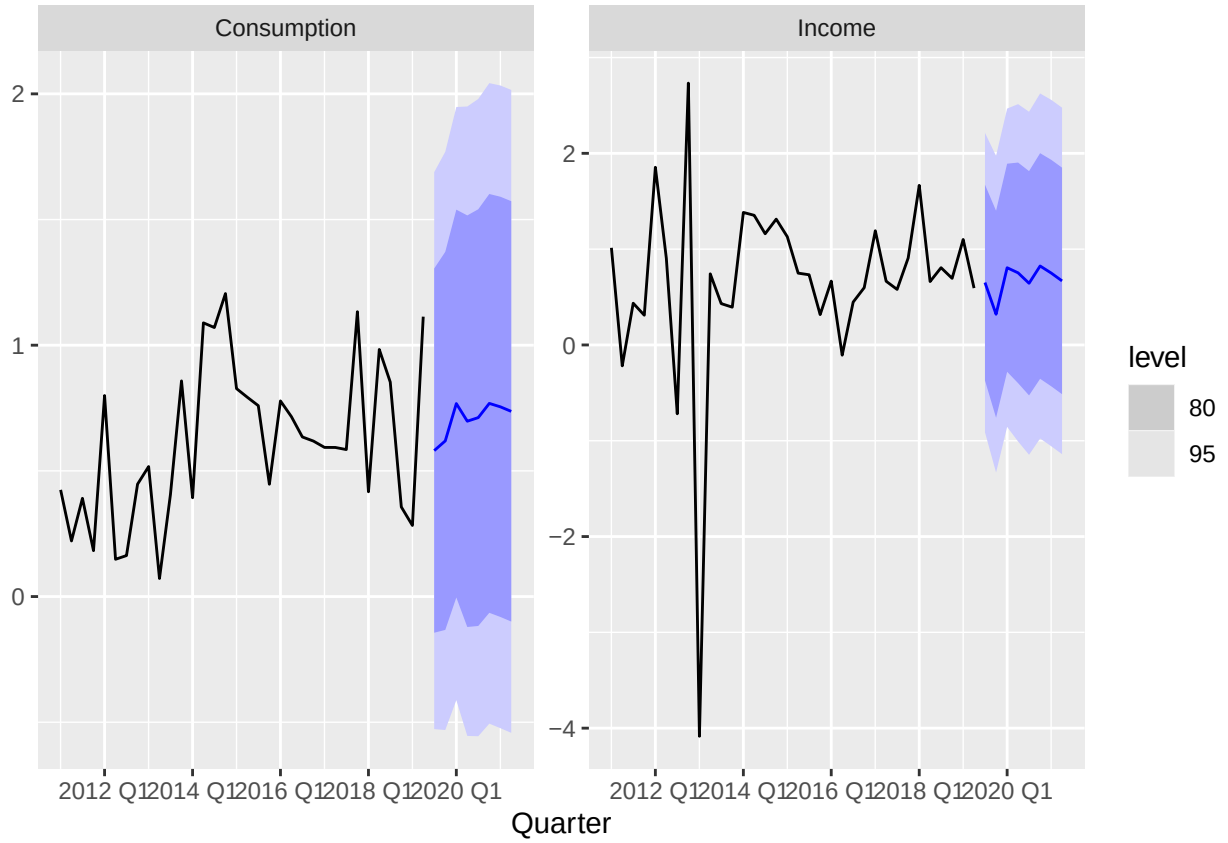
```
fit |>
  augment() |>
  ACF(.innov) |>
  autoplot()
```



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 12.14.

```
fit |>
  select(aicc) |>
  forecast() |>
  autoplot(us_change |> filter(year(Quarter) > 2010))
```



12.4 Neural network models

Artificial neural networks are forecasting methods that are based on simple mathematical models of the brain. They allow complex nonlinear relationships between the response variable and its predictors.

Neural network architecture

A neural network can be thought of as a network of "neurons" which are organised in layers. The predictors (or inputs) form the bottom layer, and the forecasts (or outputs) form the top layer. There may also be intermediate layers containing "hidden neurons".

The simplest networks contain no hidden layers and are equivalent to linear regressions. Figure 12.15 shows the neural network version of a linear regression with four predictors. The coefficients attached to these predictors are called "weights". The forecasts are obtained by a linear combination of the inputs. The weights are selected in the neural network framework using a "learning algorithm" that minimises a "cost function" such as the MSE. Of course, in this simple example, we can use linear regression which is a much more efficient method of training the model.

Once we add an intermediate layer with hidden neurons, the neural network becomes non-linear. A simple example is shown in Figure 12.16

This is known as a multilayer feed-forward network, where each layer of nodes receives inputs from the previous layers. The outputs of the nodes in one layer are inputs to the next layer. The inputs to each node are combined using a weighted linear combination. The result is then modified by a nonlinear function before being output.

The parameters b_1, b_2, b_3 and $w_{1,1}, ..., w_{4,3}$ are "learned" (or estimated) from the data. The values of the weights are often restricted to prevent them from becoming too large. The parameter that restricts the

weights is known as the "decay parameter", and is often set to be equal to 0.1.

The weights take random values to begin with, and these are then updated using the observed data. Consequently, there is an element of randomness in the predictions produced by a neural network. Therefore, the network is usually trained several times using different random starting points, and the results are averaged.

The number of hidden layers, and the number of nodes in each hidden layer, must be specified in advance. Usually, these would be selected using cross-validation.

Neural network autoregression

With time series data, lagged values of the time series can be used as inputs to a neural network, just as we used lagged values in a linear autoregression model (Chapter 9). We call this a neural network autoregression or NNAR model.

In this book, we only consider feed-forward networks with one hidden layer, and we use the notation NNAR(p,k) to indicate there are p lagged inputs and k nodes in the hidden layer. For example, a NNAR(9,5) model is a neural network with the last nine observations $(y_{t-1}, y_{t-2}, ..., y_{t-9})$ used as inputs for forecasting the output y_t, and with five neurons in the hidden layer. A NNAR(p,0) model is equivalent to an ARIMA(p,0,0) model, but without the restrictions on the parameters to ensure stationarity.

With seasonal data, it is useful to also add the last observed values from the same season as inputs. For example, an $NNAR(3,1,2)_{12}$ model has inputs y_{t-1},y_{t−2},y_{t−3}$ and y_{t-12}, and two neurons in the hidden layer. More generally, an $NNAR(p,P,k)_m$ model has inputs $(y_{t-1}, y_{t-2}, ..., y_{t-p}, y_{t-m}, yt-2m, ..., y_{t-P_m})$ and k neurons in the hidden layer. A $NNAR(p,P,0)_m$ model is equivalent to an $ARIMA(p,0,0)(P,0,0)_m$ model but without the restrictions on the parameters that ensure stationarity.

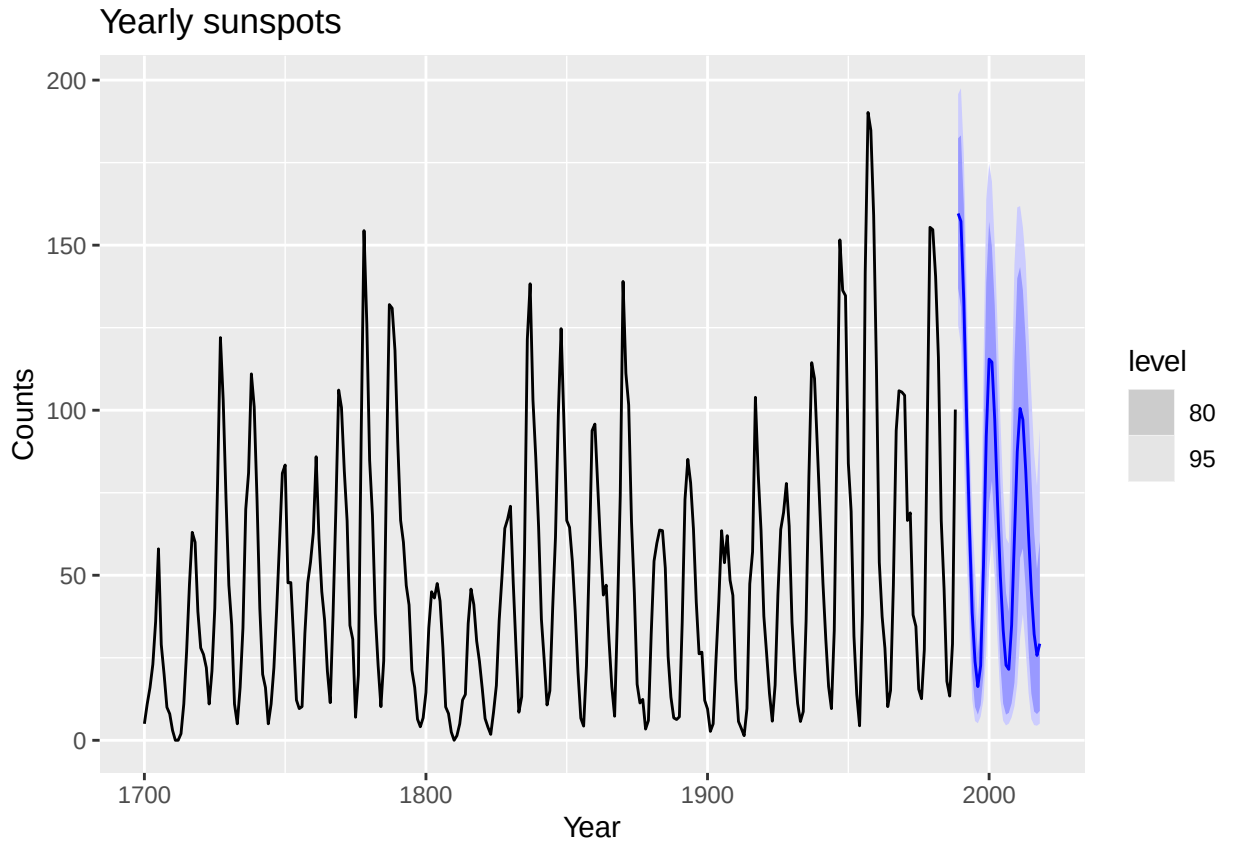
The NNETAR() function fits an $NNAR(p,P,k)_m$ model. If the values of p and P are not specified, they are selected automatically. For non-seasonal time series, the default is the optimal number of lags (according to the AIC) for a linear $AR(p)$ model. For seasonal time series, the default values are $P = 1$ and p is chosen from the optimal linear model fitted to the seasonally adjusted data. If k is not specified, it is set to $k = (p + P + 1)/2$ (rounded to the nearest integer).

When it comes to forecasting, the network is applied iteratively. For forecasting one step ahead, we simply use the available historical inputs. For forecasting two steps ahead, we use the one-step forecast as an input, along with the historical data. This process proceeds until we have computed all the required forecasts.

Example: Sunspots

The surface of the sun contains magnetic regions that appear as dark spots. These affect the propagation of radio waves, and so telecommunication companies like to predict sunspot activity in order to plan for any future difficulties. Sunspots follow a cycle of length between 9 and 14 years. In Figure 12.17, forecasts from an NNAR(9,5) are shown for the next 30 years. We have used a square root transformation to ensure the forecasts stay positive.

```
sunspots <- sunspot.year |> as_tsibble()
fit <- sunspots |>
  model(NNETAR(sqrt(value)))
fit |>
  forecast(h = 30) |>
  autoplot(sunspots) +
  labs(x = "Year", y = "Counts", title = "Yearly sunspots")
```



Here, the last 9 observations are used as predictors, and there are 5 neurons in the hidden layer. The cyclicity in the data has been modelled well. We can also see the asymmetry of the cycles has been captured by the model, where the increasing part of the cycle is steeper than the decreasing part of the cycle. This is one difference between a NNAR model and a linear AR model — while linear AR models can model cyclicity, the modelled cycles are always symmetric.

Prediction intervals

Unlike most of the methods considered in this book, neural networks are not based on a well-defined stochastic model, and so it is not straightforward to derive prediction intervals for the resultant forecasts. However, we can still compute prediction intervals using simulation where future sample paths are generated using bootstrapped residuals (as described in Section 5.5).

The neural network fitted to the sunspot data can be written as $y_t = f(y_{t-1}) + \varepsilon_t$

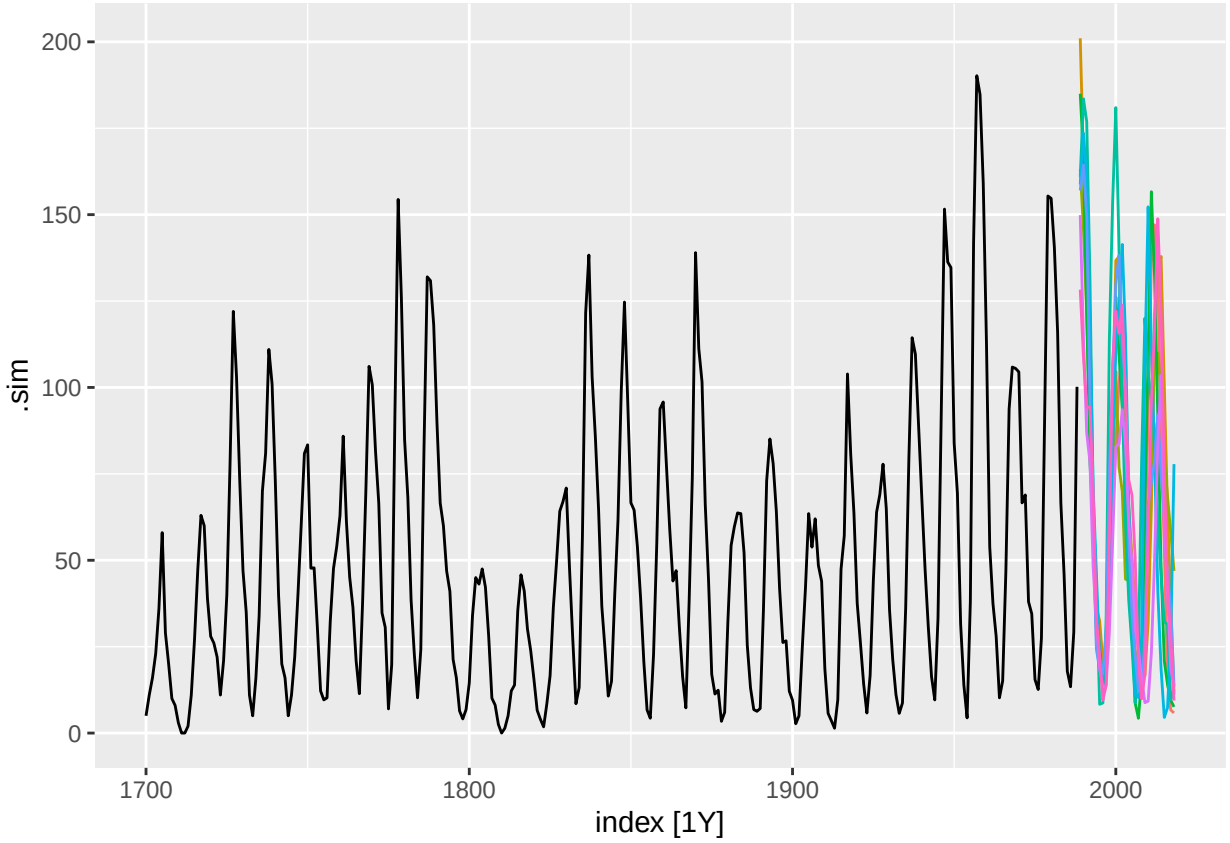
where $y_{t-1} = (y_{t-1}, y_{t-2}, ..., y_{t-9})'$ is a vector containing lagged values of the series, and f is a neural network with 5 hidden nodes in a single layer. The error series $\{\varepsilon_t\}$ is assumed to be homoscedastic (and possibly also normally distributed).

We can simulate future sample paths of this model iteratively, by randomly generating a value for ε_t, either from a normal distribution, or by resampling from the historical values. So if ε^*_{T+1} is a random draw from the distribution of errors at time $T+1$, then $y^*_{T+1} = f(y_T) + \varepsilon^*_{T+1}$ is one possible draw from the forecast distribution for y_{T+1}. Setting $y^*_{T+1} = (y^*_{T+1}, y_T, ..., y_{T-7})$, we can then repeat the process to get $y^*_{T+2} = f(y^*_{T+1}) + \varepsilon^*_{T+2}$.

In this way, we can iteratively simulate a future sample path. By repeatedly simulating sample paths, we build up knowledge of the distribution for all future values based on the fitted neural network.

Here is a simulation of 9 possible future sample paths for the sunspot data. Each sample path covers the next 30 years after the observed data.

```
fit |>
  generate(times = 9, h = 30) |>
  autoplot(.sim) +
  autolayer(sunspots, value) +
  theme(legend.position = "none")
```



If we do this many times, we can get a good picture of the forecast distributions. This is how the forecast() function produces prediction intervals for NNAR models. The times argument in forecast() controls how many simulations are done (default 1000). By default, the errors are drawn from a normal distribution. The bootstrap argument allows the errors to be "bootstrapped" (i.e., randomly drawn from the historical errors).

12.5 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.

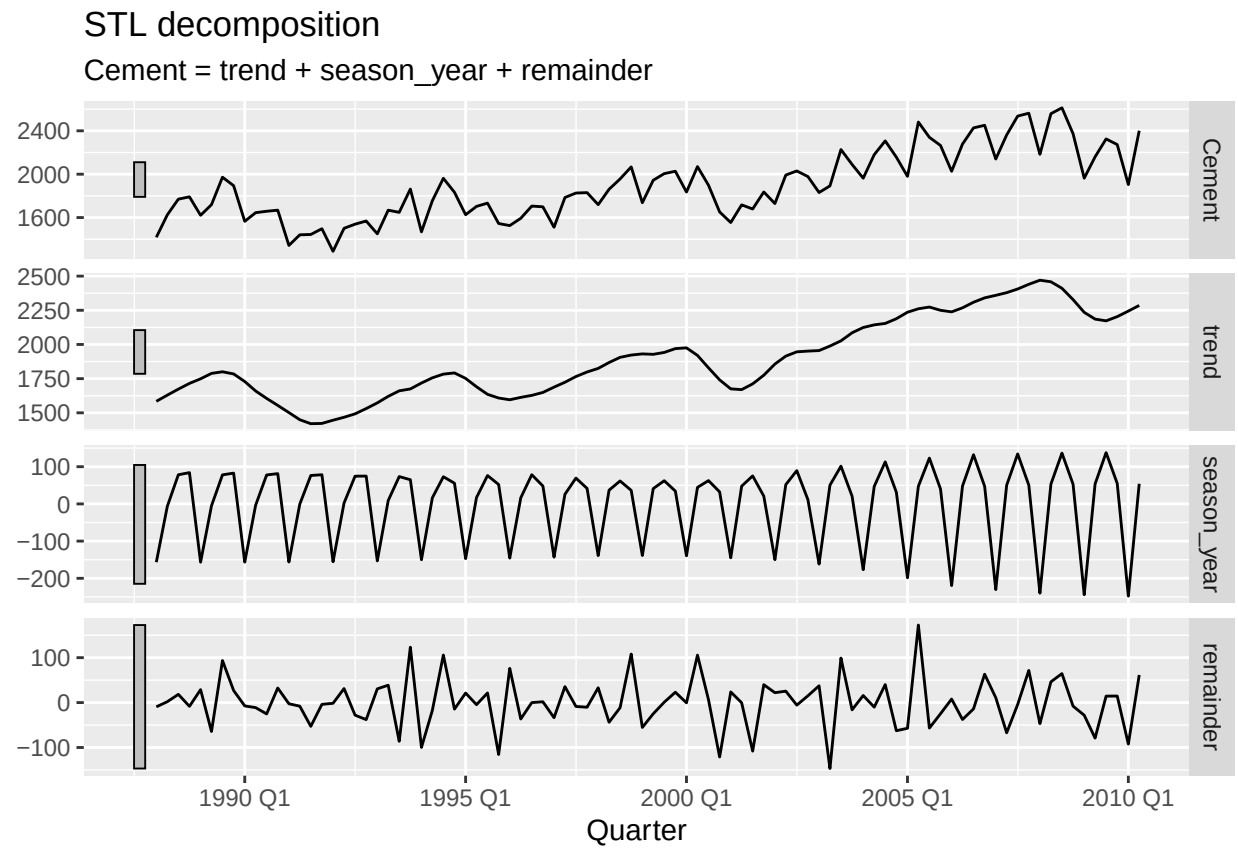
More generally, we can generate new time series that are similar to our observed series, using another type of bootstrap.

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 12.19 that the decomposition has adequately captured the trend and seasonality, and that there is no obvious remaining signal in the remainder series.

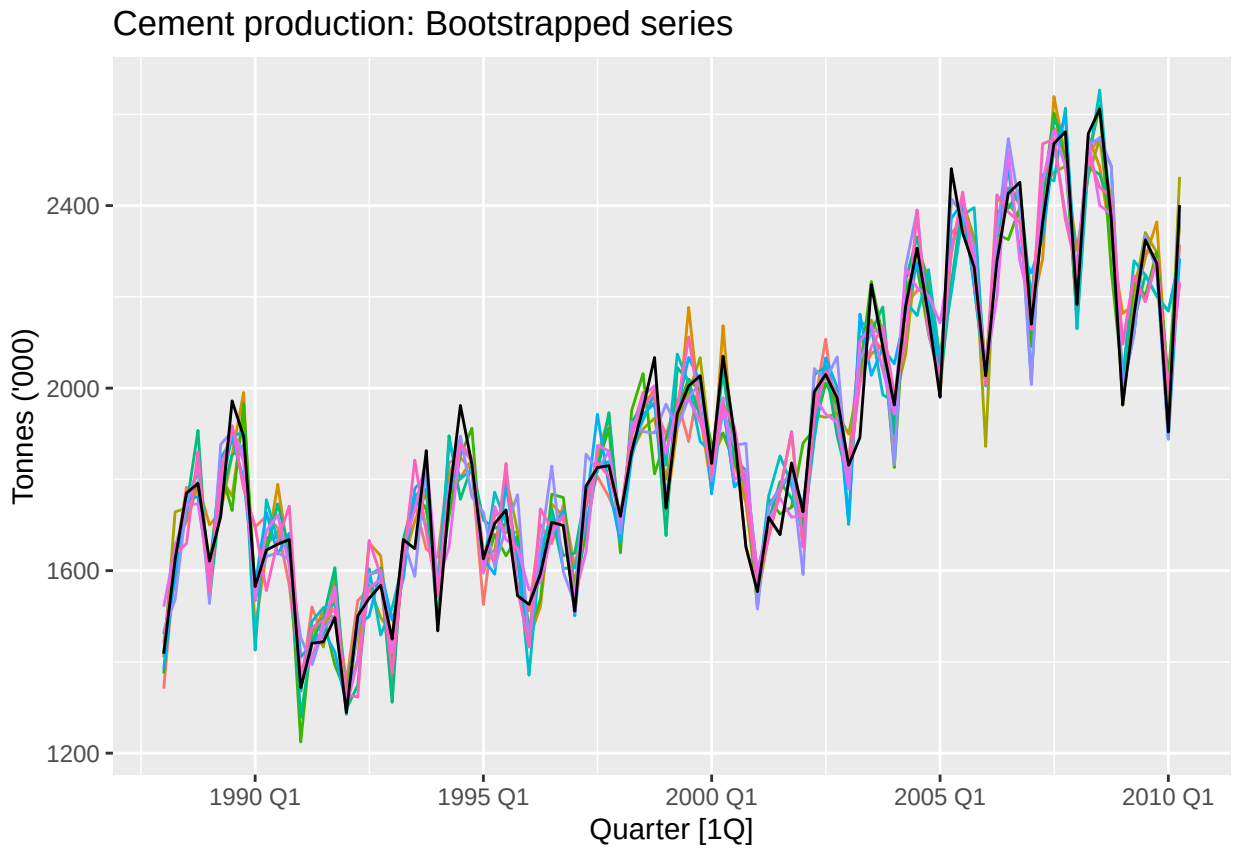
```
cement <- aus_production |>
  filter(year(Quarter) >= 1988) |>
  select(Quarter, Cement)
cement_stl <- cement |>
  model(stl = STL(Cement))
cement_stl |>
  components() |>
  autoplot()
```



Now we can generate several bootstrapped versions of the data. Usually, generate() produces simulations of the future from a model. But here we want simulations for the period of the historical data. So we use the new_data argument to pass in the original data so that the same time periods are used for the simulated data. We will use a block size of 8 to cover two years of data.

```
cement_stl |>
  generate(new_data = cement, times = 10,
           bootstrap_block_size = 8) |>
  autoplot(.sim) +
  autolayer(cement, Cement) +
  guides(colour = "none") +
  labs(title = "Cement production: Bootstrapped series",
```

```
       y="Tonnes ('000)")
```



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 "bootstrap aggregating".

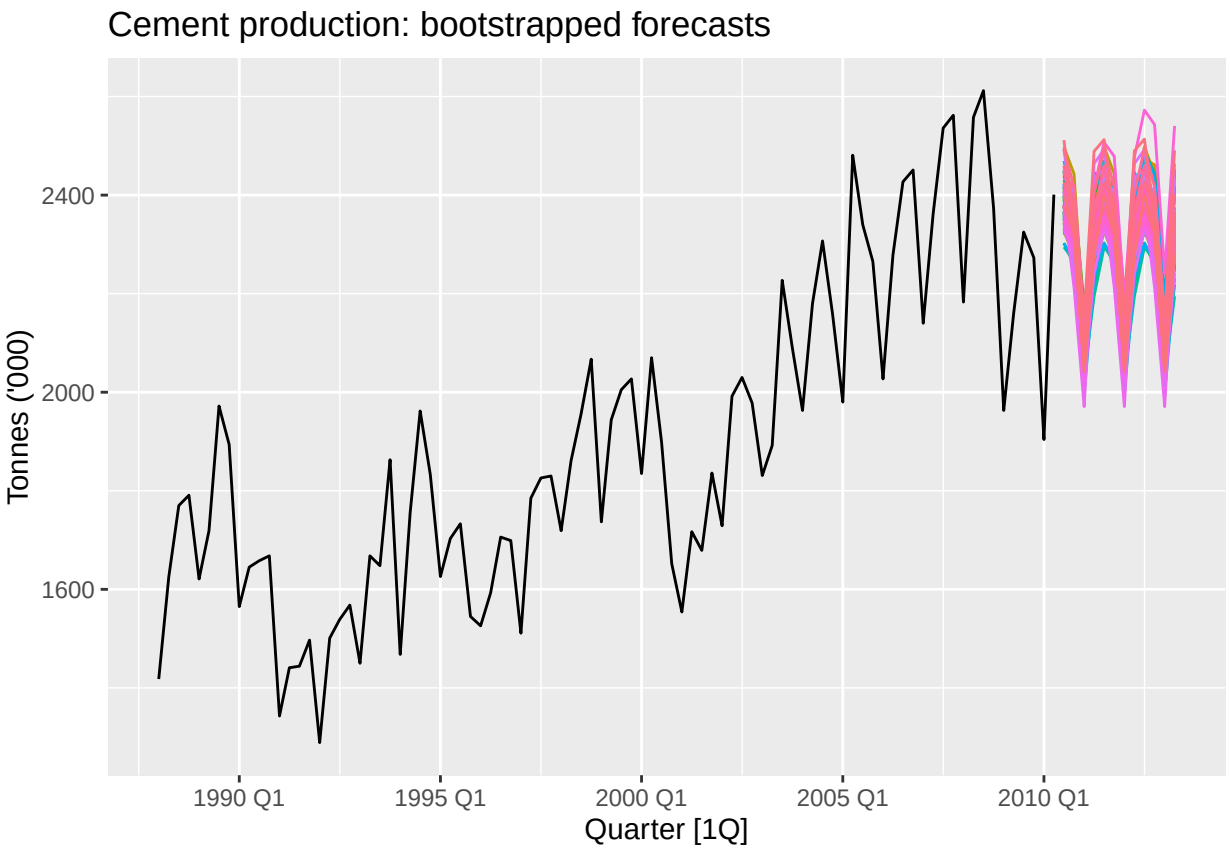
We demonstrate the idea using the cement data. First, we simulate many time series that are similar to the original data, using the block-bootstrap described above.

```
sim <- cement_stl |>
  generate(new_data = cement, times = 100,
           bootstrap_block_size = 8) |>
  select(-.model, -Cement)
```

For each of these series, we fit an ETS model. A different ETS model may be selected in each case, although it will most likely select the same model because the series are similar. However, the estimated parameters will be different, so the forecasts will be different even if the selected model is the same. This is a time-consuming process as there are a large number of series.

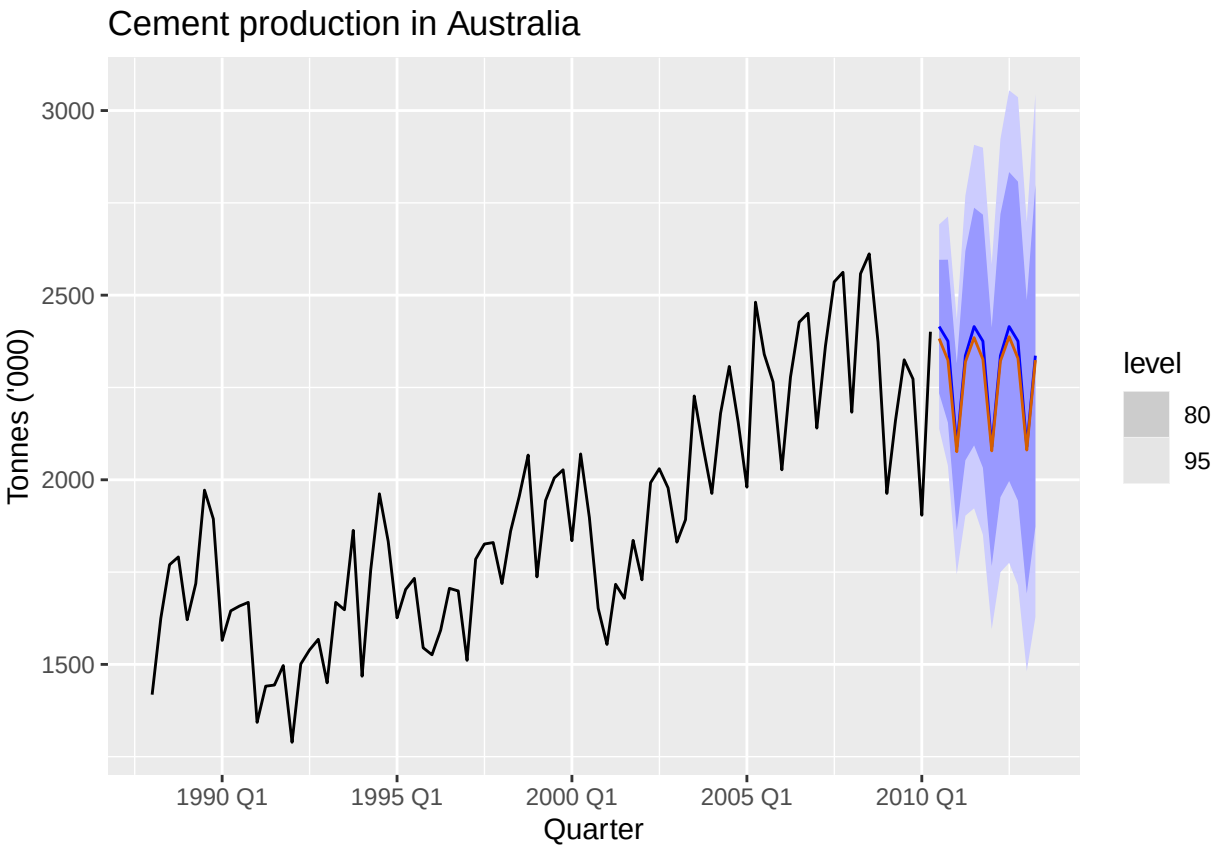
```
ets_forecasts <- sim |>
  model(ets = ETS(.sim)) |>
  forecast(h = 12)
ets_forecasts |>
  update_tsibble(key = .rep) |>
  autoplot(.mean) +
```

```
  autolayer(cement, Cement) +
  guides(colour = "none") +
  labs(title = "Cement production: bootstrapped forecasts",
       y="Tonnes ('000)")
```



Finally, we average these forecasts for each time period to obtain the "bagged forecasts" for the original data.

```
bagged <- ets_forecasts |>
  summarise(bagged_mean = mean(.mean))
cement |>
  model(ets = ETS(Cement)) |>
  forecast(h = 12) |>
  autoplot(cement) +
  autolayer(bagged, bagged_mean, col = "#D55E00") +
  labs(title = "Cement production in Australia",
       y="Tonnes ('000)")
```



Bergmeir et al. (2016) show that, on average, bagging gives better forecasts than just applying ETS() directly. Of course, it is slower because a lot more computation is required.