Forecasting with ARIMA models

Seun Odeyemi

2/23/2018

```
# runif(1, 0, 10^8)
set.seed(77159275) #for reproducibility of results
```

```
rm(list =ls())
```

```
devtools::session_info()
```

```
##  setting  value
##  version  R version 3.4.3 (2017-11-30)
##  system   x86_64, linux-gnu
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  tz       Zulu
##  date     2018-03-05
##
##  package   * version date       source
##  backports   1.1.2   2017-12-13 CRAN (R 3.4.3)
##  base      * 3.4.3   2017-12-01 local
##  compiler    3.4.3   2017-12-01 local
##  datasets  * 3.4.3   2017-12-01 local
##  devtools    1.13.5  2018-02-18 CRAN (R 3.4.3)
##  digest      0.6.15  2018-01-28 CRAN (R 3.4.3)
##  evaluate    0.10.1  2017-06-24 CRAN (R 3.4.3)
##  graphics  * 3.4.3   2017-12-01 local
##  grDevices * 3.4.3   2017-12-01 local
##  htmltools   0.3.6   2017-04-28 CRAN (R 3.4.3)
##  knitr       1.20    2018-02-20 CRAN (R 3.4.3)
##  magrittr    1.5     2014-11-22 CRAN (R 3.4.3)
##  memoise     1.1.0   2017-04-21 CRAN (R 3.4.3)
##  methods   * 3.4.3   2017-12-01 local
##  Rcpp        0.12.15 2018-01-20 CRAN (R 3.4.3)
##  rmarkdown   1.9     2018-03-01 CRAN (R 3.4.3)
##  rprojroot   1.3-2   2018-01-03 CRAN (R 3.4.3)
##  stats     * 3.4.3   2017-12-01 local
##  stringi     1.1.6   2017-11-17 CRAN (R 3.4.3)
##  stringr     1.3.0   2018-02-19 CRAN (R 3.4.3)
##  tools       3.4.3   2017-12-01 local
##  utils     * 3.4.3   2017-12-01 local
##  withr       2.1.1   2017-12-19 CRAN (R 3.4.3)
##  yaml        2.1.17  2018-02-27 CRAN (R 3.4.3)
```

Loading some useful libraries

```
#library(XLConnect)
library(dplyr)
library(ggplot2)
```

```
#library(forecast)
library(fpp2)
library(readxl)
library(data.table)
```

Set Working Directory

```
setwd("/home/sdotserver1/projects/")
```

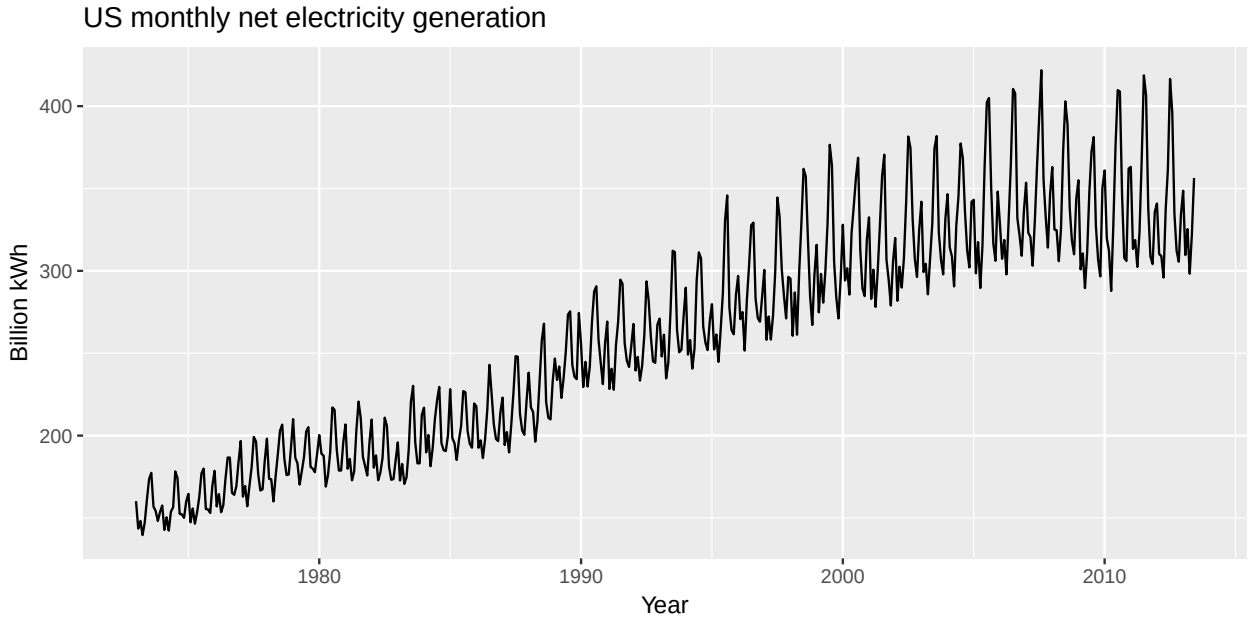
Transformations for variance stabilization

With ETS models, you used multiplicative errors and multiplicative seasonality to handle time series that have variance which increases with the level of the series. An alternative approach is to transform the time series. If the data show increasing variation as the level of the series increases, then a **transformation** can be useful.

*$y_1, ..., y_n$: original observations, $w_1, ..., w_n$: transformed observations.

*Mathematical transformations for stabilizing variation include square root ($w_t = \sqrt{y_t}$), cube root ($w_t = \sqrt[3]{y_t}$), logarithm ($w_t = \log(y_t)$), and inverse ($w_t = -1/y_t$). You can think of these on a scale of transformations with increasing strength the further along the scale you go, the greater the effect of the transformation. Let's illustrate this with some data.

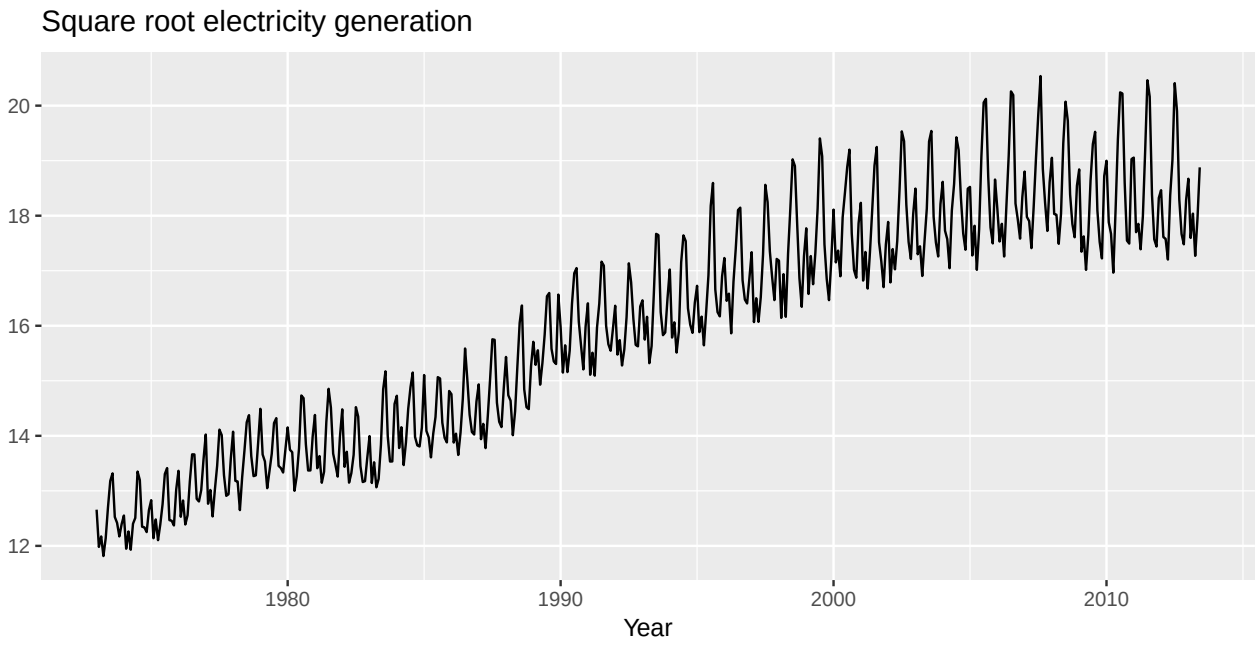
```
# usmelec contains monthly total net electricity generation in the US from
# January 1973 - June 2013
autoplot(usmelec) +
  xlab("Year") +
  ylab("Billion kWh") +
  ggtitle("US monthly net electricity generation")
```



There is an upward trend and strong seasonality driven by the use of heating and air conditioning. The size of the seasonal fluctuation is greater when the level of the series is higher. When you use transformations, **you**

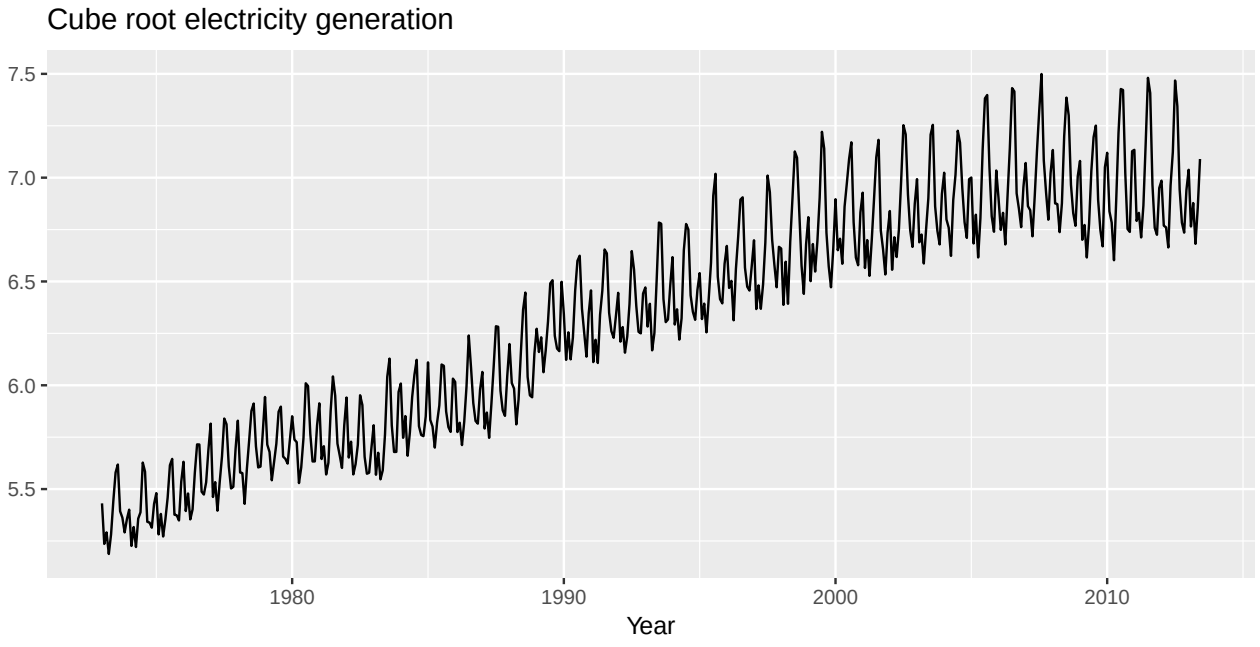
are attempting to make those fluctuations *approximately even* throughout the series. A square root has a relatively small effect in this case.

```
autoplot(usmelec ^ 0.5) +
  xlab("Year") + ylab("") +
  ggtitle("Square root electricity generation")
```



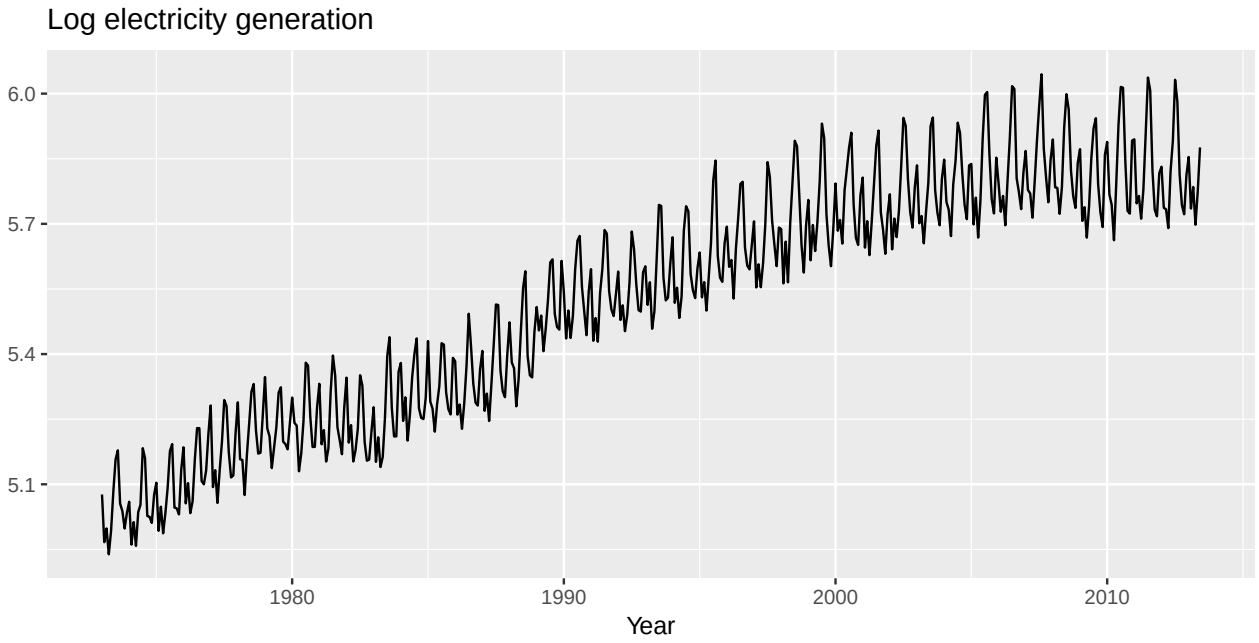
Cube root is stronger.

```
autoplot(usmelec ^ 0.33333) +
  xlab("Year") + ylab("") +
  ggtitle("Cube root electricity generation")
```



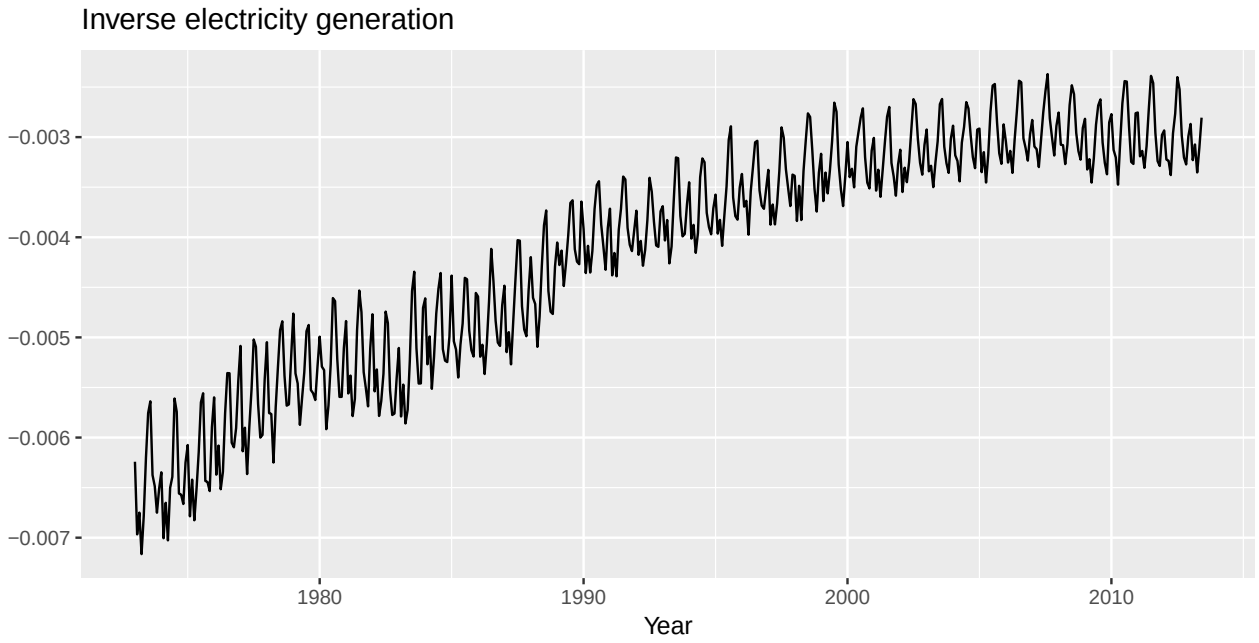
The logarithm transformation is stronger still.

```
autoplot(log(usmelec)) +
  xlab("Year") + ylab("") +
  ggtitle("Log electricity generation")
```



But the fluctuations at the top end of the series are still larger than the fluctuations at the bottom end. So, let's try an inverse transformation.

```
autoplot(-1/usmelec) +
  xlab("Year") + ylab("") +
  ggtitle("Inverse electricity generation")
```



Now, we seem to have gone too far. The seasonal fluctuations at the top are now smaller than the seasonal fluctuations at the bottom end. So, we need something in-between the log transformation and the inverse transformation if you want to stabilize the variance of this series. These four transformations are closely

related to the family of the **Box-Cox transformations**.

Box-Cox Transformations for Time Series

Box-Cox can be expressed as

$$w_t = \{log(y_t)\ \lambda = 0\}$$

$$OR$$

$$w_t = \{(y_t^{\lambda} - 1)\ \lambda \neq 0\}$$

$\lambda = 0\ \lambda \neq 0$

It contains a single parameter λ that contains how strong the transformation is:

$\lambda = 1$ *: No substantial transformation. We simply substract* ***1*** *from all transformations.* $\lambda = \frac{1}{2}$: Similar to the square root plus linear transformation $\lambda = \frac{1}{3}$ *: Similar to the cube root plus linear transformation* $\lambda = 0$: Similar to the natural logarithm transformation $^*\lambda = -1$: Similar to the inverse tranformation

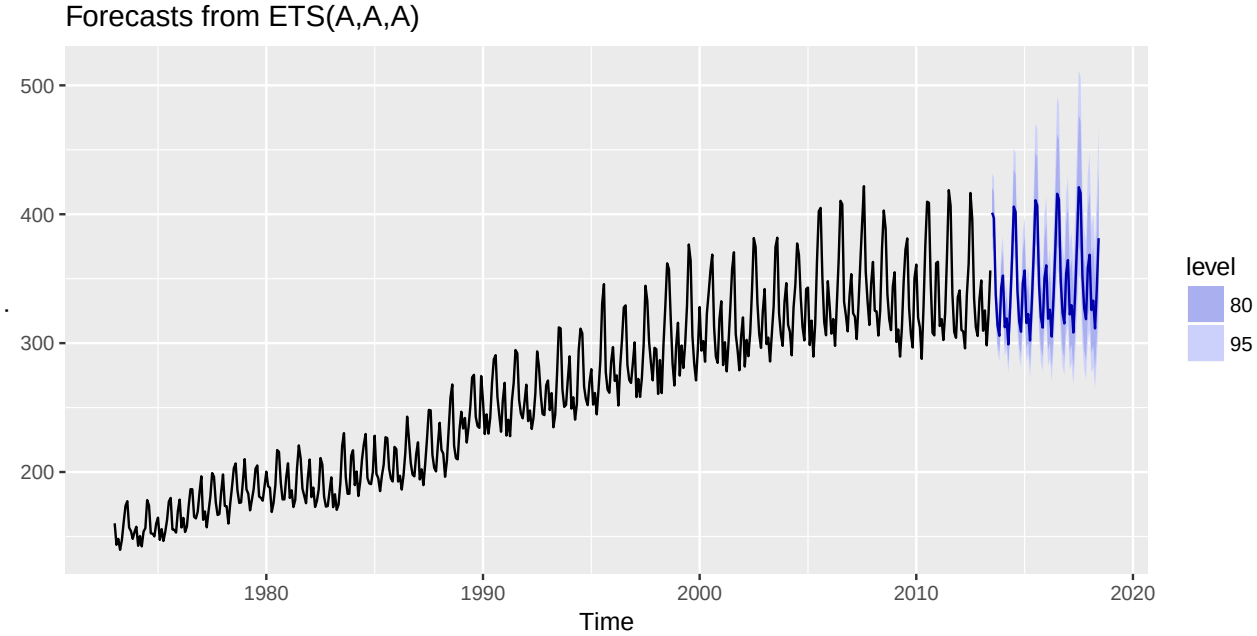
You can use the lambda values in-between these values to get other transformations. You might try a few values yourself until you find something that looks about right. Or you can use the `BoxCox.lambda()` function which returns an estimate of lambda that should roughly balance the variation across the series.

```
BoxCox.lambda(usmelec)
```

```
## [1] -0.5738331
```

Here it is has given a value between -1 and 0 as you will expect given the graphs shown above. Once we have chosen a lambda, we simply need to add it to the modeling function you are using and R will take care of the rest.

```
usmelec %>%
  ets(lambda = -0.57) %>%
  forecast(h = 60) %>%
  autoplot()
```



Here we have $\lambda = -0.57$ in the `ets()` function. R will transform the time series using the chosen `BoxCox` transformation and fit an `ets` model. When you pass the resulting model to the `forecast()` function, it

passes information about the transformation as well so the `forecast()` function will produce forecasts from the ets model, and then back-transforms them by undoing the `BoxCox`transformation to give a forecast on the original scale.

Notice how the seasonal fluctuations in the forecasts are much the same size as those in the end of the data. It is not very common to use a `BoxCox` transformation with a `ets` model like this as `ets` models are capable of handling the increasing variance in the series directly by using multiplicative error and seasonal components in the model. But soon you will be using `arima` models, and you will need transformations that handle time series with increasing variance.

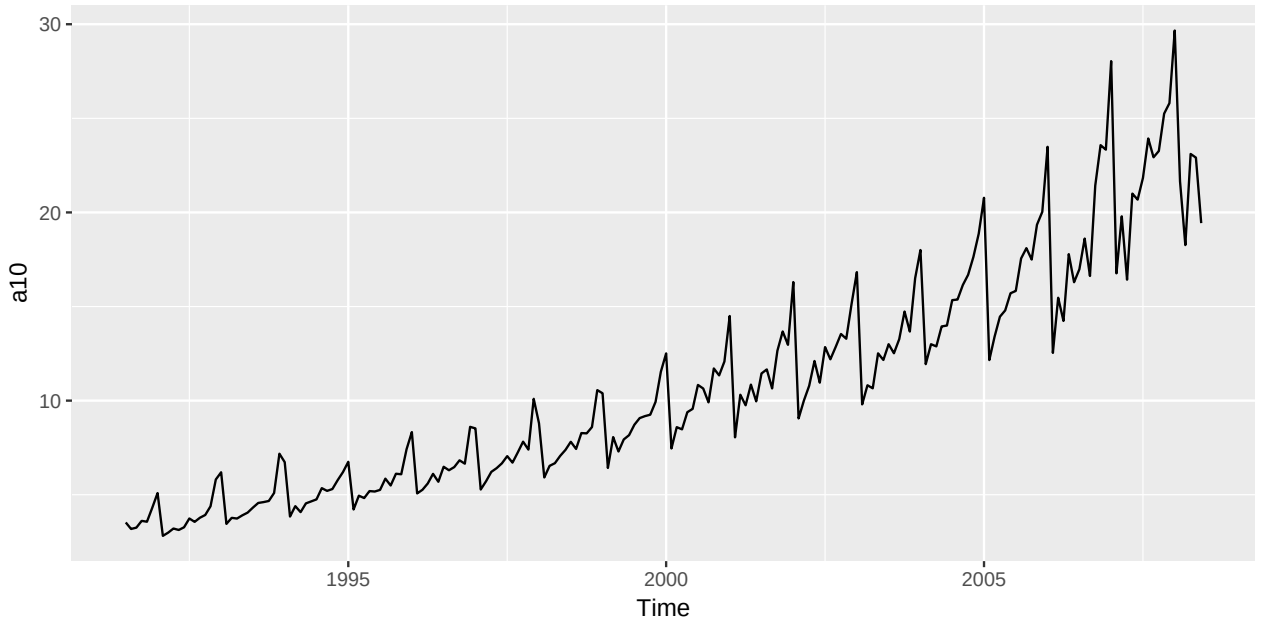
Practice Box-Cox Transformations

Here, you will use a Box-Cox transformation to stabilize the variance of the pre-loaded a10 series, which contains monthly anti-diabetic drug sales in Australia from 1991-2008.

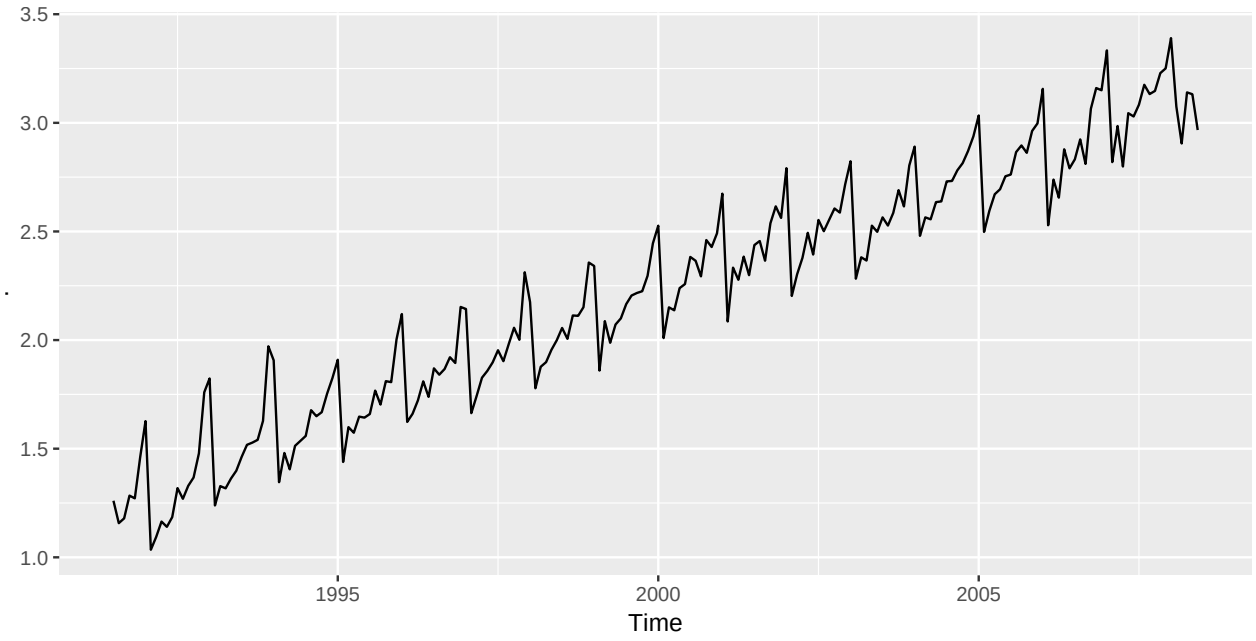
In this exercise, you will need to experiment to see the effect of the λ argument on the transformation. Notice that small changes in λ make little difference to the resulting series. You want to find a value of λ that makes the seasonal fluctuations of roughly the same size across the series.

Recall from the video that the recommended range for lambda values is $-1 \leq \lambda \leq 1$.

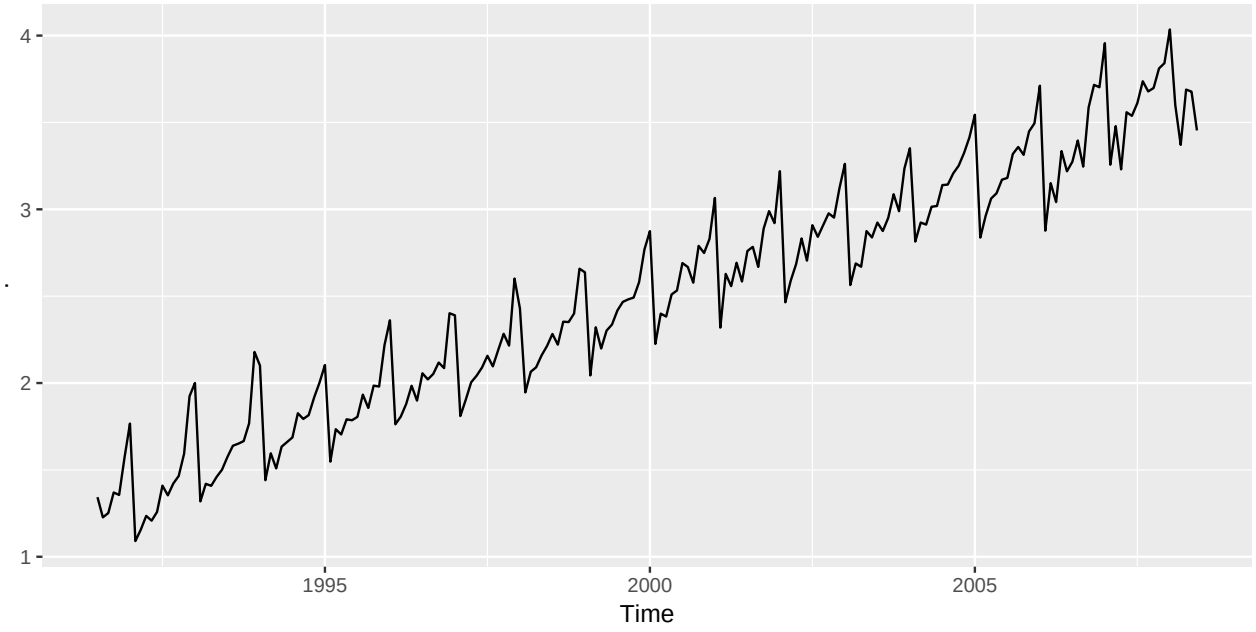
```
par(mfrow=c(1,2), las=1)
# Plot the series
autoplot(a10)
```



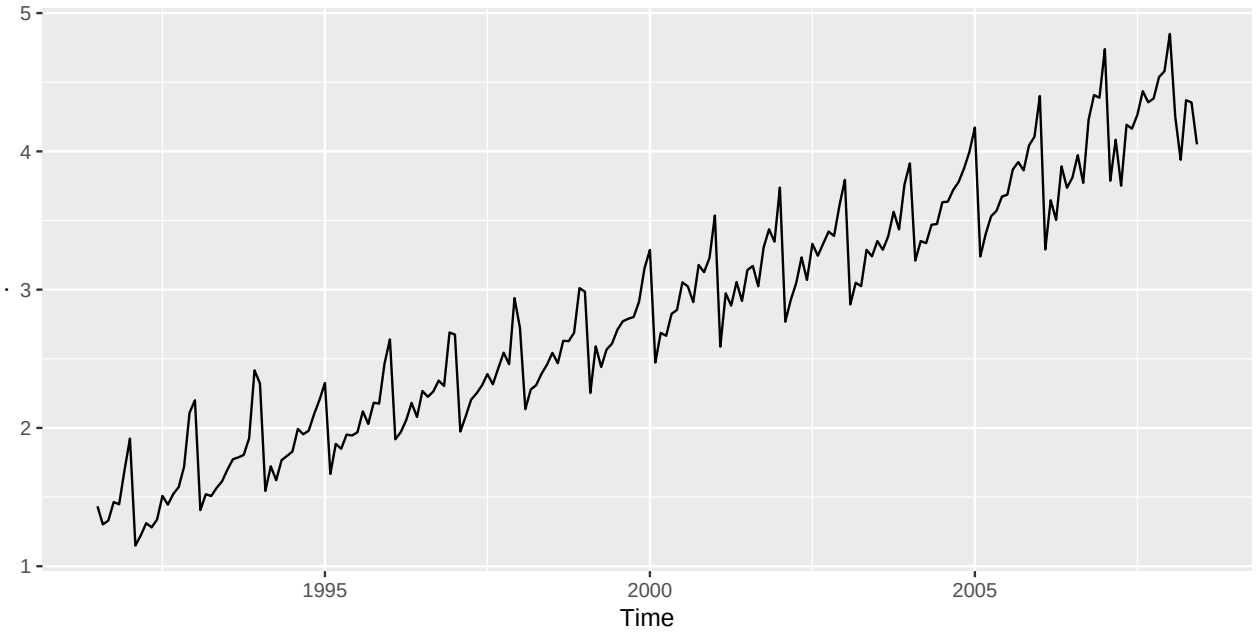

```
# Try four values of lambda in Box-Cox transformations
a10 %>% BoxCox(lambda = 0.0) %>% autoplot()
```



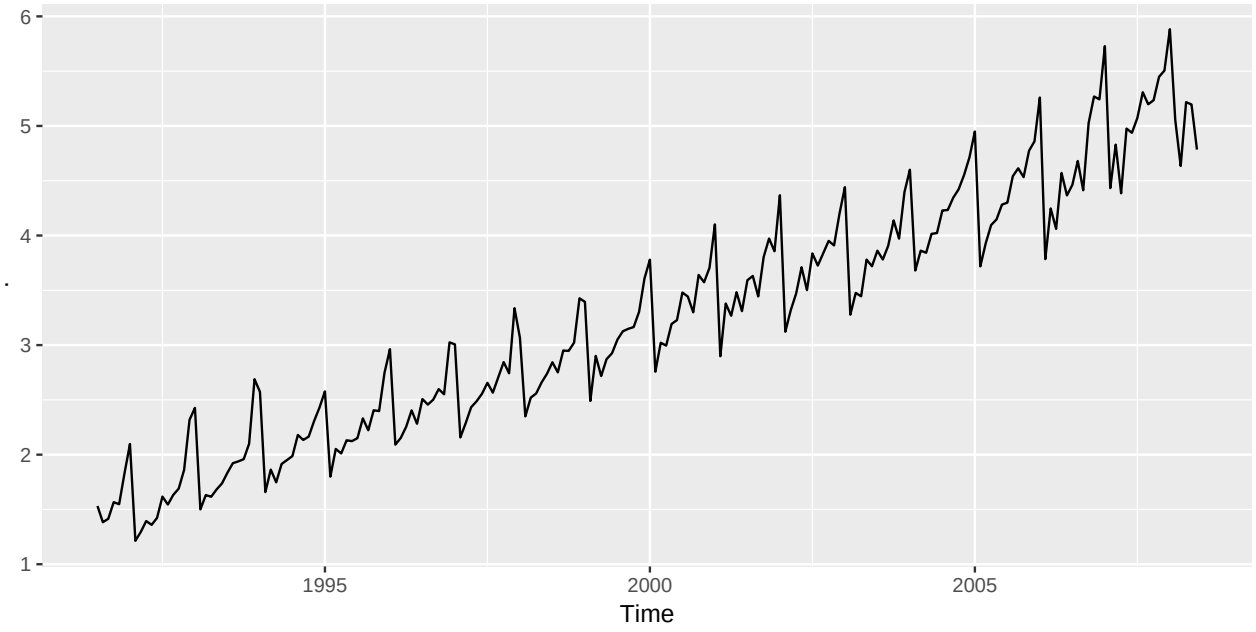

```
a10 %>% BoxCox(lambda = 0.1) %>% autoplot()
```




```
a10 %>% BoxCox(lambda = 0.2) %>% autoplot()
```




```
a10 %>% BoxCox(lambda = 0.3) %>% autoplot()
```




```
# Compare with BoxCox.lambda()
BoxCox.lambda(a10)
```

```
## [1] 0.1313326
```

Good job! It seems like a lambda of .13 would work well.

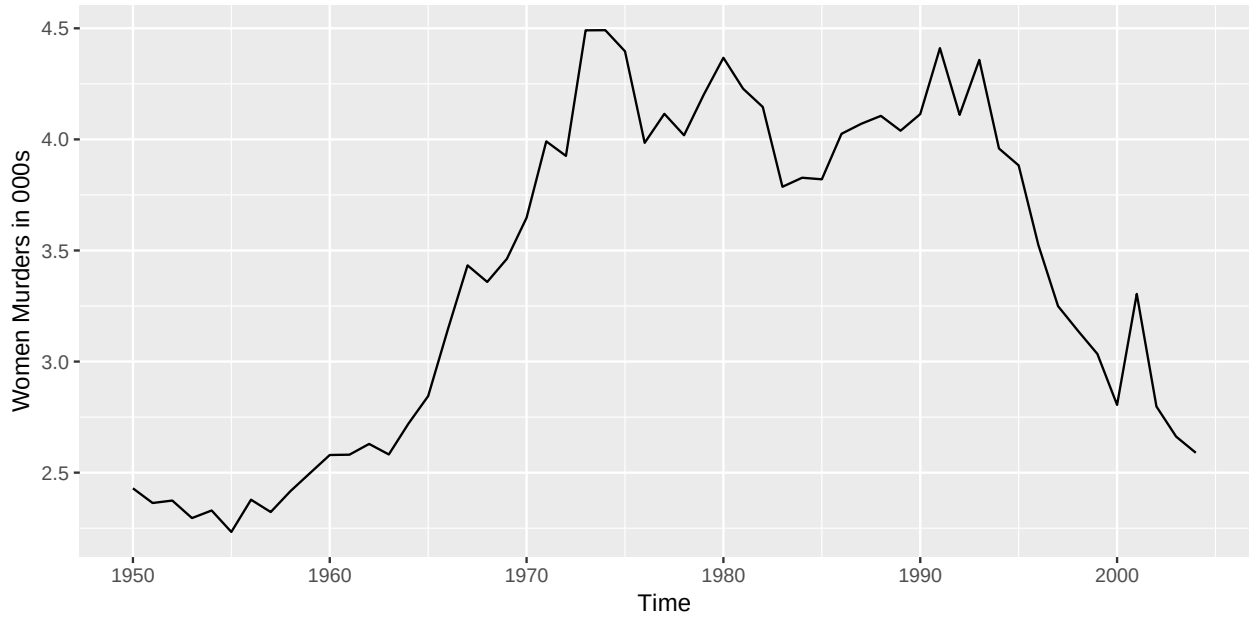
Non-seasonal differencing for stationarity

Differencing is a way of making a time series *stationary*; this means that you remove any systematic patterns such as trend and seasonality from the data. A **white noise** series is considered a special case of a stationary time series.

With non-seasonal data, you use **lag-1** differences to model changes between observations rather than the observations directly. You have done this before by using the diff() function.

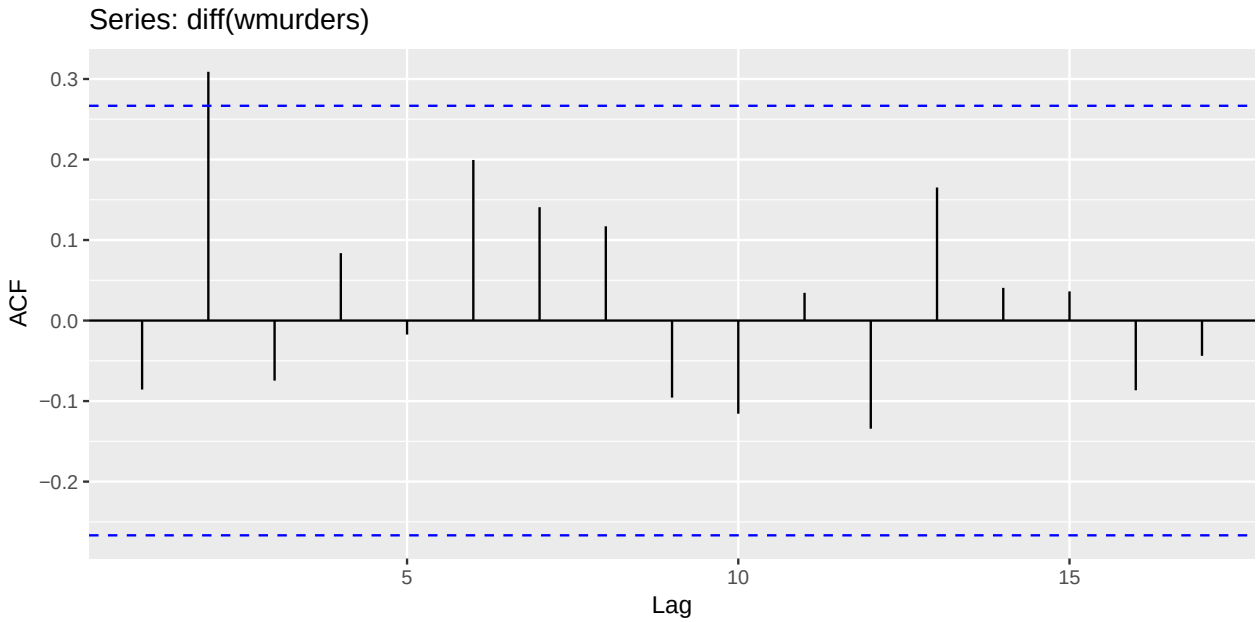
In this exercise, you will use the pre-loaded `wmurders` data, which contains the annual female murder rate in the US from 1950-2004.

```
# Plot the US female murder rate
autoplot(wmurders) +
  ylab("Women Murders in 000s")
```




```
# Plot the differenced murder rate
autoplot(diff(wmurders))
```


```
# Plot the ACF of the differenced murder rate
ggAcf(diff(wmurders))
```



Great! It seems like the data look like white noise after differencing.

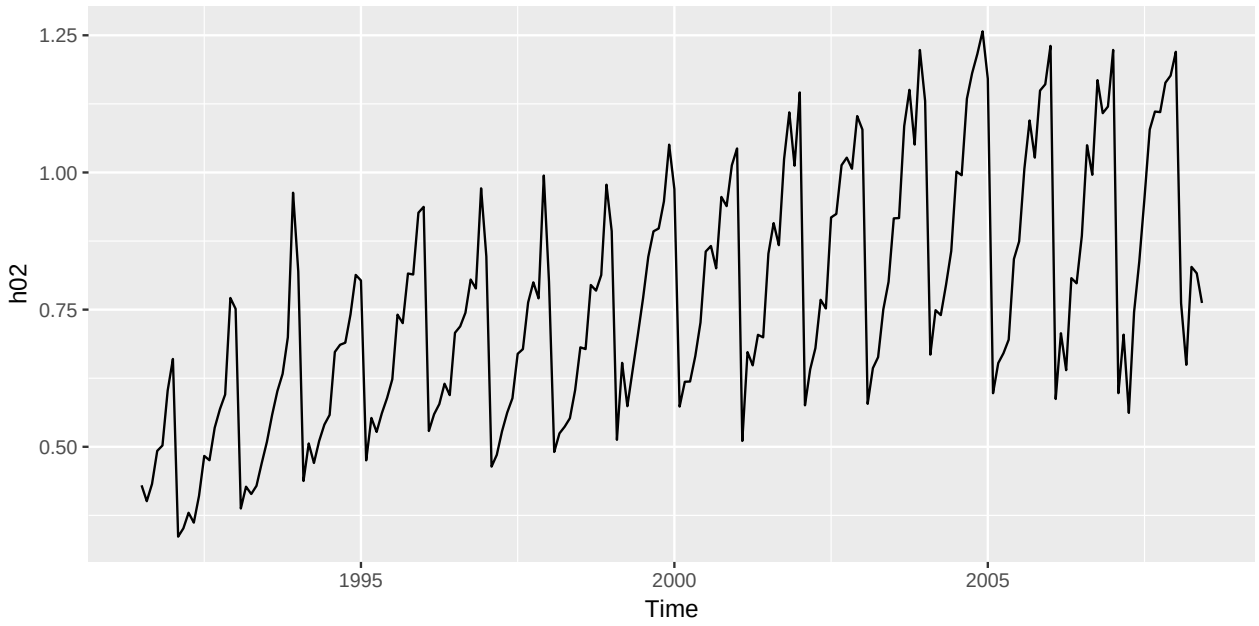
Seasonal differencing for stationarity

With seasonal data, differences are often taken between observations in the same season of consecutive years, rather than in consecutive periods. For example, with quarterly data, one would take the difference between Q1 in one year and Q1 in the previous year. This is called **seasonal differencing**.

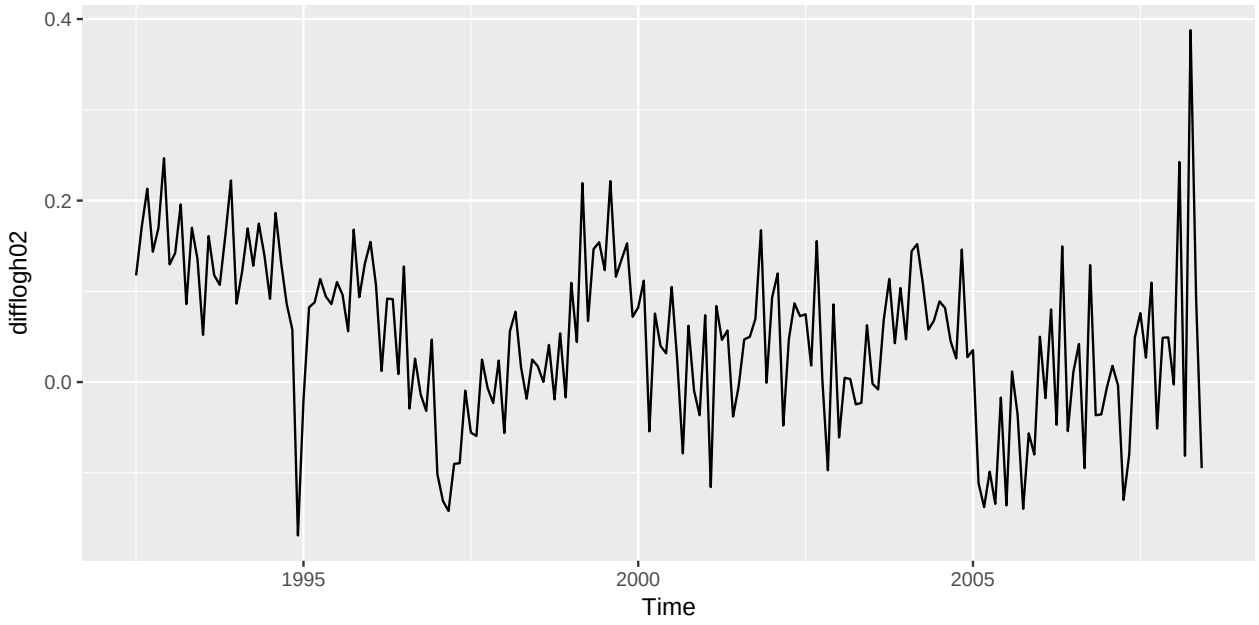
Sometimes you need to apply both seasonal differences and lag-1 differences to the same series, thus, calculating the differences in the differences. Lag is a period of time between one event or phenomenon and another.

In this exercise, you will use differencing and transformations simultaneously to make a time series look stationary. The data set here is h02, which contains 17 years of monthly corticosteroid drug sales in Australia. It has been loaded into your workspace.

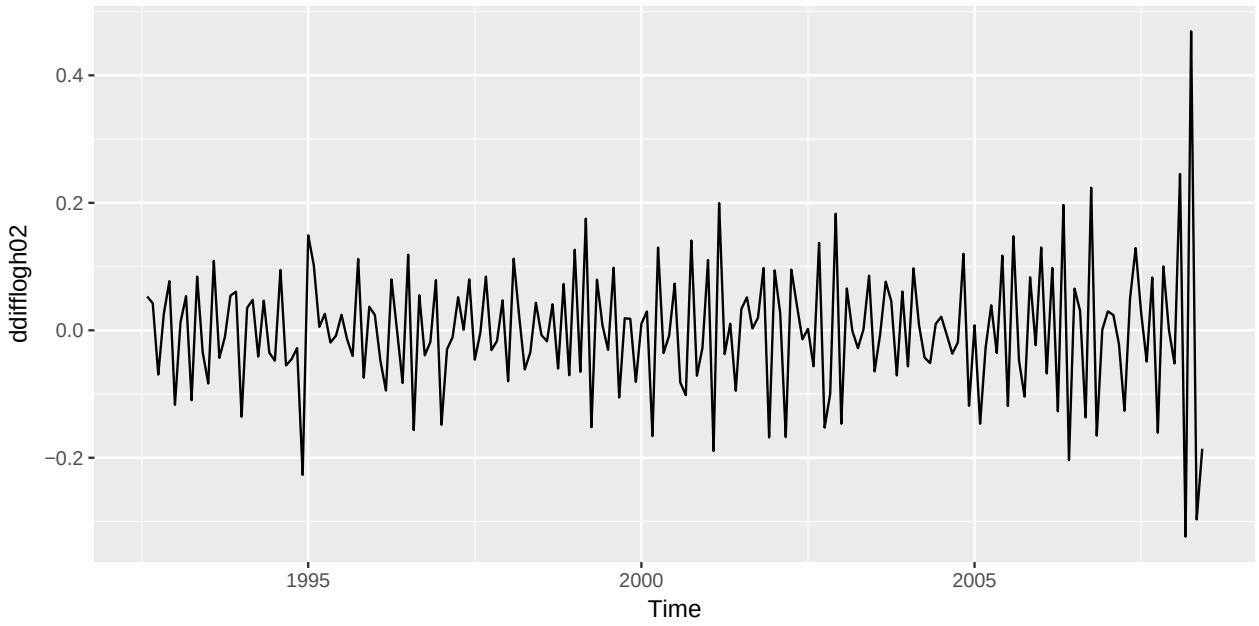
```
# Plot the data
autoplot(h02)
```



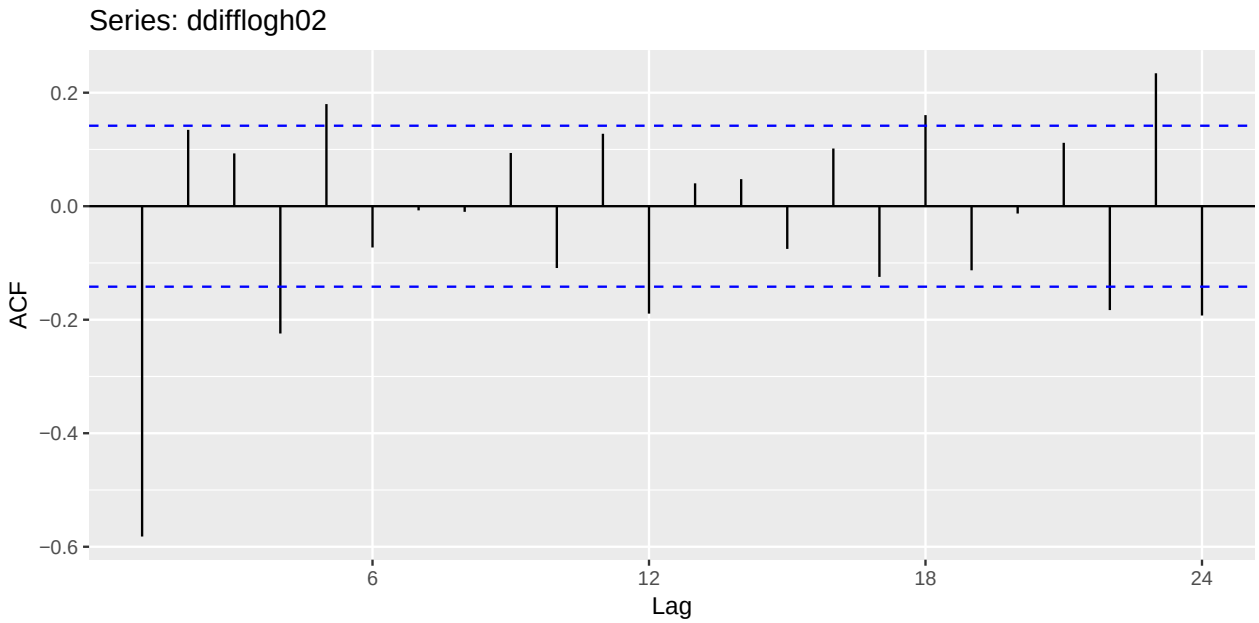

```
# Take logs and seasonal differences of h02
difflogh02 <- diff(log(h02), lag = 12)

# Plot difflogh02
autoplot(difflogh02)
```




```
# Take another difference and plot
ddifflogh02 <- diff(difflogh02)
autoplot(ddifflogh02)
```




```
# Plot ACF of ddifflogh02
ggAcf(ddifflogh02)
```



Great! The data doesn't look like white noise after the transformation, but you could develop an ARIMA model for it.

ARIMA Models

ARIMA stands for **Autoregressive Integrated Moving Average** models. Let's break that down to its parts:

*Autoregressive (AR) models: $y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + e_t$

*where $e_t \sim$ white noise

An autoregressive model is simply is a multiple regression of a time series against the lagged values of that series. The ***lagged observations*** (last p observations) are used as predictors in the regression equation.

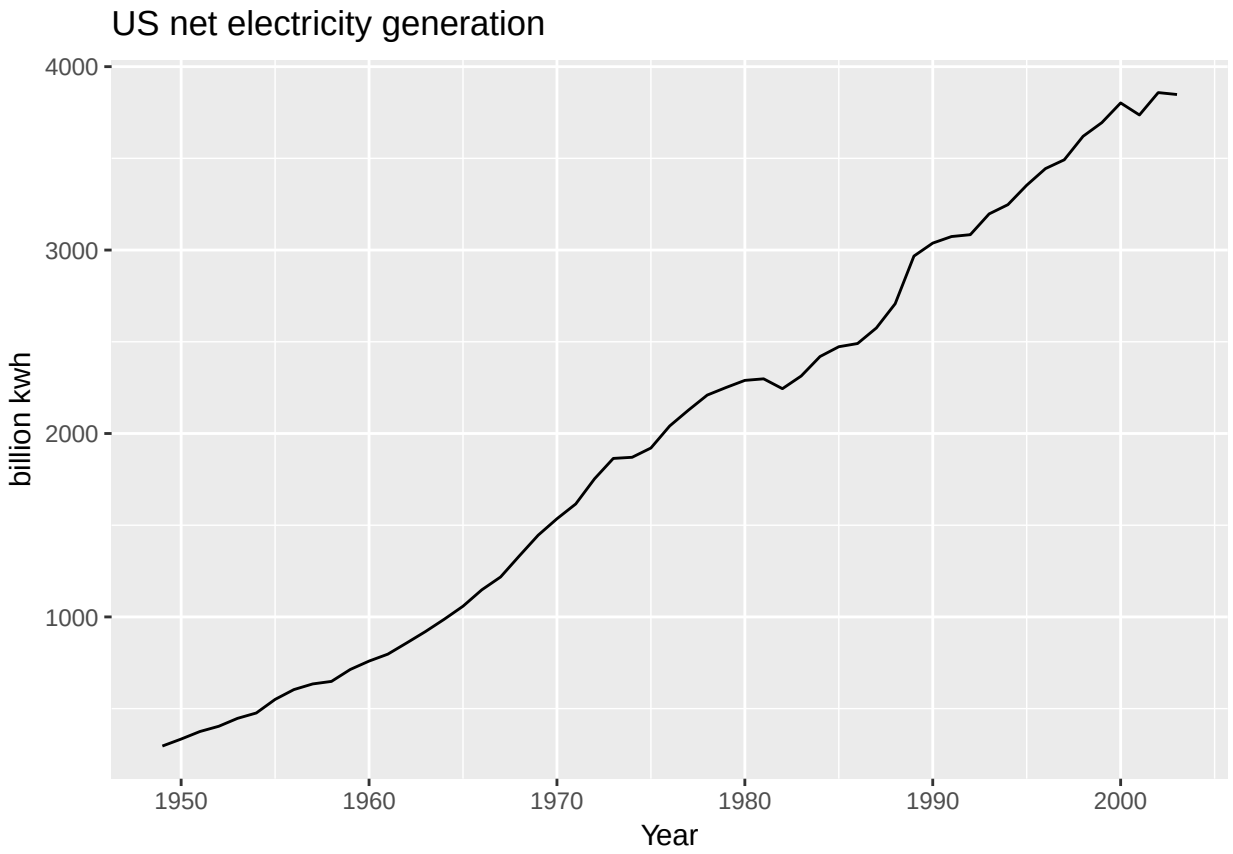
Moving Average (MA) models: $y_t = c + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} +$ where $e_t \sim$ white noise

A moving average model can also be thought of as a regression, but instead of regressing against lagged observations we regress against ***lagged errors***. The last q errors are used as predictors in the equation. When you put this together you get an **ARMA** model with the last p observations and last q errors used as predictors in the equation.

$$y_t = c + \phi_1 y_{t-1} + ... + \phi_p y_{t-p} + \theta_1 e_{t-1} + ... + \theta_q e_{t-q} + e_t$$

ARMA models can only work with stationary data so you need to difference the data first. That brings us to the **I** in ARIMA, which stands for **Integrated** – i.e. the opposite of differencing. If our time series needs to be differenced d times to make it stationary, then the resulting model is called an **ARIMA (p,d,q) model**. To apply an ARIMA model on a data, you need to decide on the value of p, d, and q, and whether or not to include the constant, c – the intercept in these equations.

```
autoplot(usnetelec) +
  xlab("Year") +
  ylab("billion kwh") +
  ggtitle("US net electricity generation")
```



The `auto.arima()` function chooses the ARIMA model given the time series.

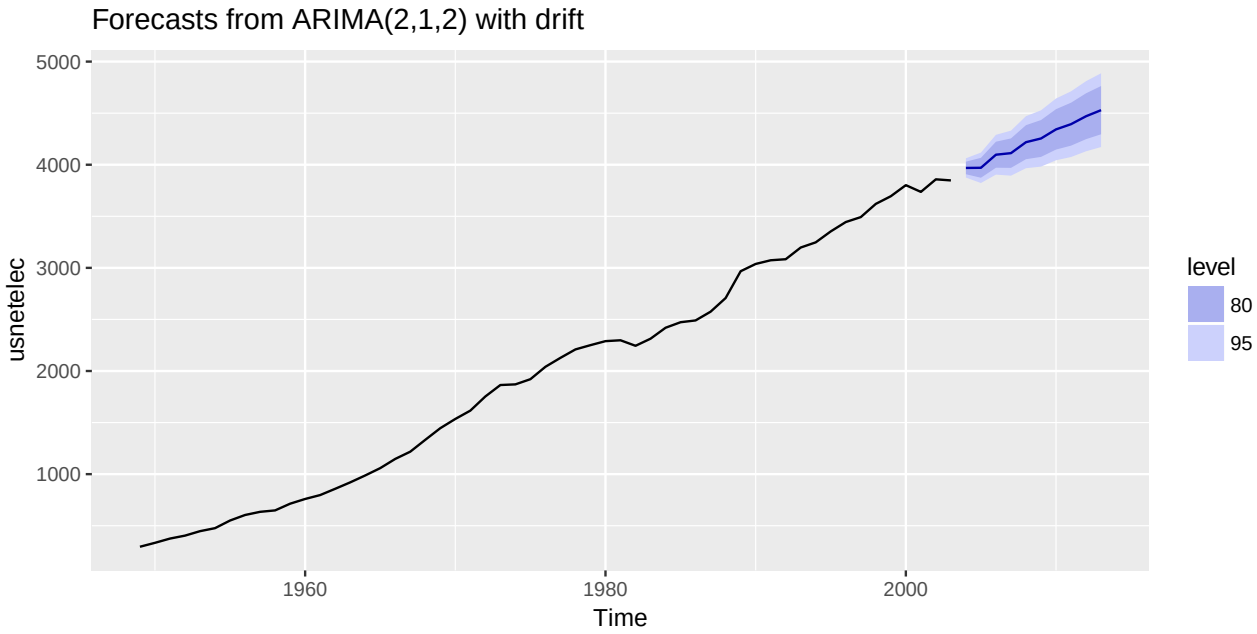
```
fit <- auto.arima(usnetelec)
summary(fit)
```

```
## Series: usnetelec
```

```
## ARIMA(2,1,2) with drift
##
## Coefficients:
##           ar1      ar2     ma1     ma2    drift
##       -1.3032  -0.4332  1.5284  0.8340  66.1585
## s.e.   0.2122   0.2084  0.1417  0.1185   7.5595
##
## sigma^2 estimated as 2262:  log likelihood=-283.34
## AIC=578.67   AICc=580.46   BIC=590.61
##
## Training set error measures:
##                      ME     RMSE     MAE        MPE     MAPE      MASE
## Training set 0.04640184 44.89414 32.3328 -0.6177064 2.101204 0.4581279
##                     ACF1
## Training set 0.02249247
```

In this case it has selected an ARIMA (2,1,2) model with drift. So the data has been differenced once, two past observations, and two past errors have been used in the equation. The *drift* here refers to the coefficient, c. It is called a drift coefficient when there is differencing. The rest of the output tell you about the values of the parameters and other model information. Notice that the AIC_c value is given, as it was for ETS models. `auto.arima()` is selecting the value of p and q by minimizing the AIC_c value just like the `ets()` function did. **However, you cannot compare an ARIMA AIC_c value with an `ets()` AIC_c value, you can only compare AIC_c values with models of the same class. You also cannot compare the AIC_c values between models with different amounts of differencing**.

```
fit %>% forecast() %>% autoplot()
```



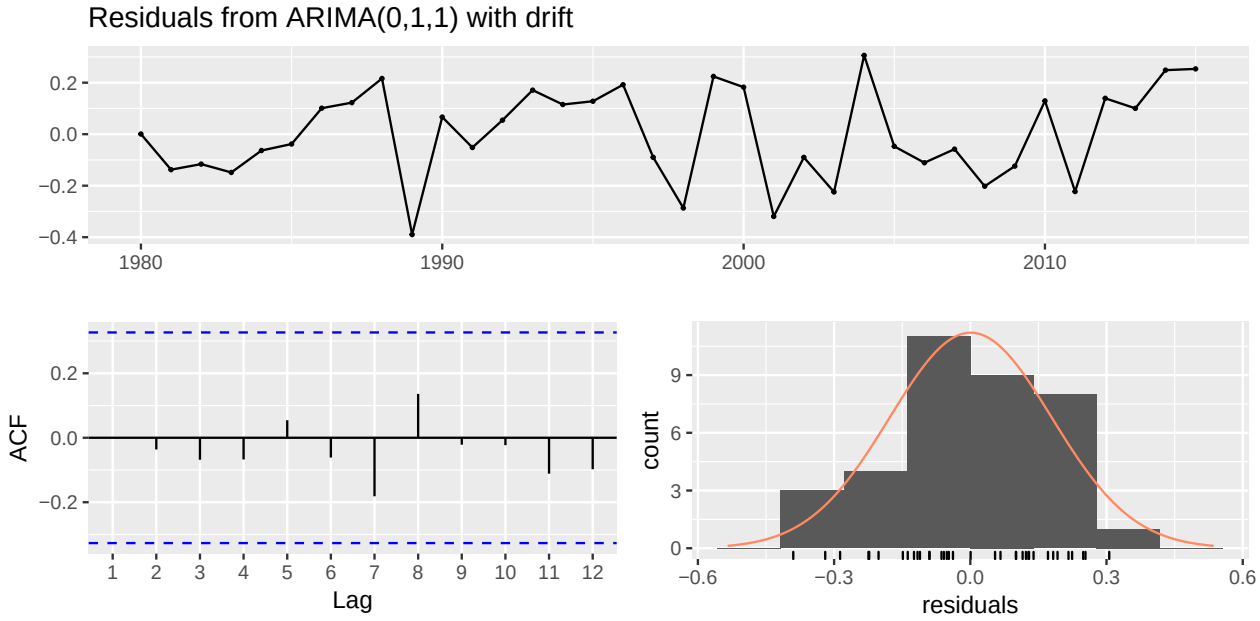
The forecast looks pretty good. The upward trend has been captured nicely.

`auto.arima()` is using an algorithm Dr. Hyndman developed with Khandakar – the Hyndman-Khandakar algorithm – which chooses the number of differences d using a tool called the *unit root tests* and it selects the value of p and q by minimizing the AIC_c. The parameters are estimated using Maximum Likelihood Estimation (MLE). One issue here is that the model space is very large as p and q can take on any non-negative values. So to save time, we only try some of the possible models. That means it is possible that `auto.arima()` returns a model that is not actually the one with the minimum AIC_c value. There is a whole datacamp class on ARIMA models if you want to delve deep into these kinds of time series models.

Automatic ARIMA models for non-seasonal time series

In this exercise, you will automatically choose an ARIMA model for the pre-loaded austa series, which contains the annual number of international visitors to Australia from 1980-2015. You will then check the residuals **(recall that a p-value greater than 0.05 indicates that the data resembles white noise)** and produce some forecasts. Other than the modelling function, this is identicial to what you did with ETS forecasting.

```
# Fit an automatic ARIMA model to the austa series
fit <- auto.arima(austa)

# Check that the residuals look like white noise
checkresiduals(fit)
```



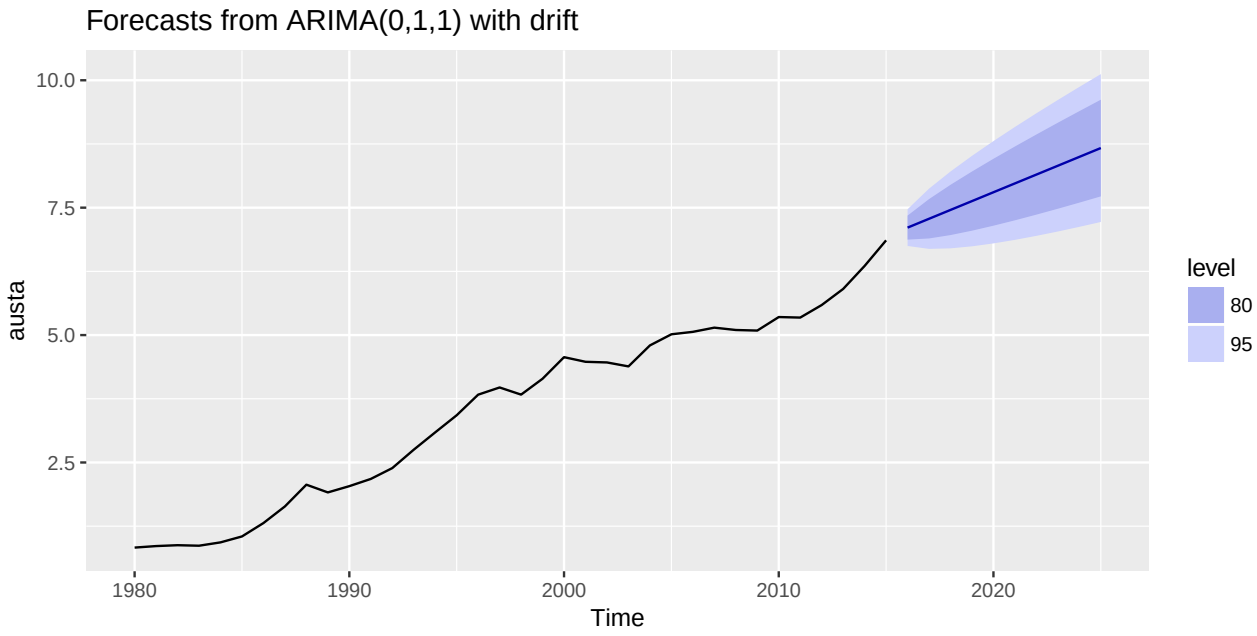

```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(0,1,1) with drift
## Q* = 3.2552, df = 8, p-value = 0.9173
##
## Model df: 2.   Total lags used: 10
```

```
residualsok <- TRUE

# Summarize the model
summary(fit)
```

```
## Series: austa
## ARIMA(0,1,1) with drift
##
## Coefficients:
##          ma1   drift
##       0.3006  0.1735
## s.e.  0.1647  0.0390
##
## sigma^2 estimated as 0.03376:  log likelihood=10.62
```

```
## AIC=-15.24   AICc=-14.46   BIC=-10.57
##
## Training set error measures:
##                       ME      RMSE       MAE       MPE     MAPE      MASE
## Training set 0.0008313383 0.1759116 0.1520309 -1.069983 5.513269 0.7461559
##                      ACF1
## Training set -0.000571993
```

```
# Find the AICc value and the number of differences used
AICc <- -14.46
d <- 1

# Plot forecasts of fit
fit %>% forecast(h = 10) %>% autoplot()
```



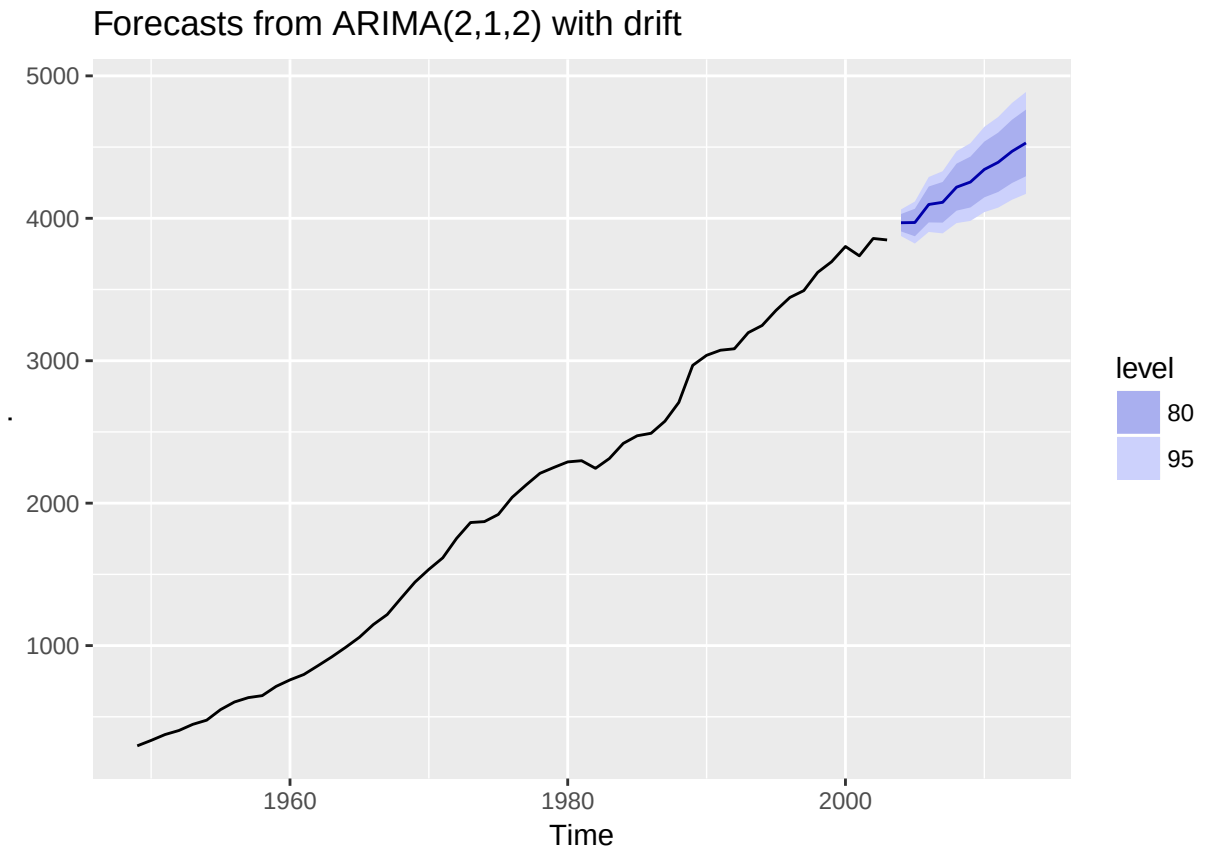
Good job. It looks like the ARIMA model created a pretty good forecast for you.

Forecasting with ARIMA Models

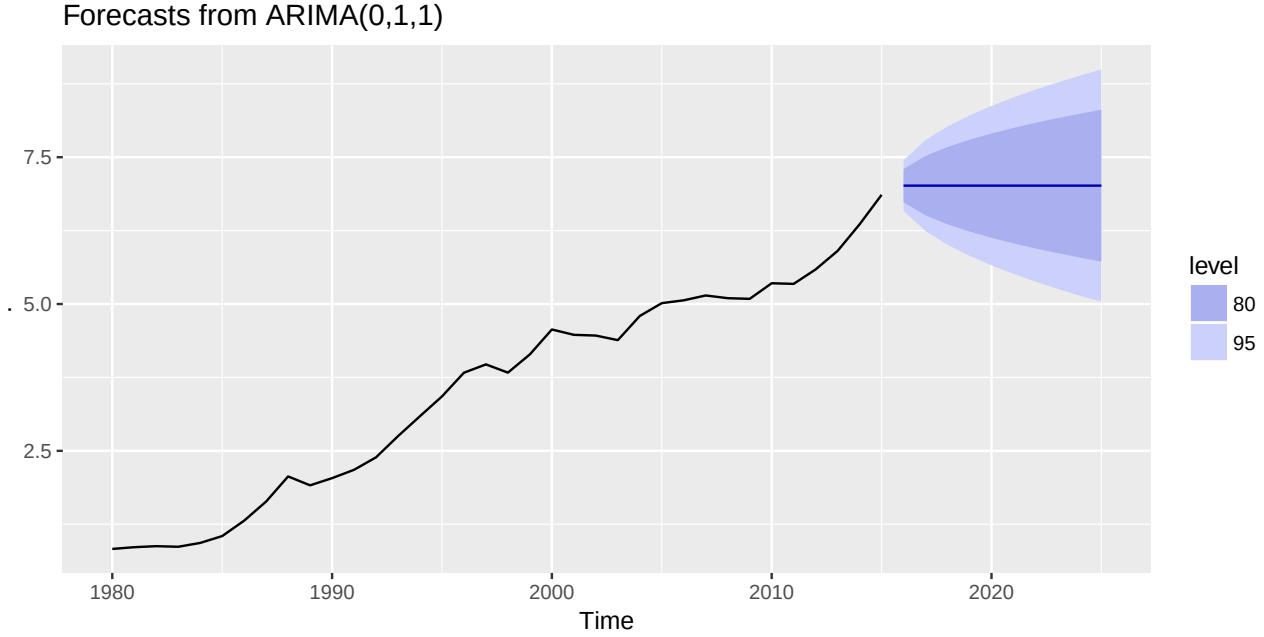
The automatic method in the previous exercise chose an ARIMA(0,1,1) with drift model for the austa data, that is, $y_t = c + y_t - 1 + \theta e_t - 1 + e_t$. You will now experiment with various other ARIMA models for the data to see what difference it makes to the forecasts.

The `Arima()` function can be used to select a specific ARIMA model. Its first argument, order, is set to a vector that specifies the values of p, d and q. The second argument, include.constant, is a booolean that determines if the constant c, or drift, should be included. Below is an example of a pipe function that would plot forecasts of usnetelec from an ARIMA(2,1,2) model with drift:

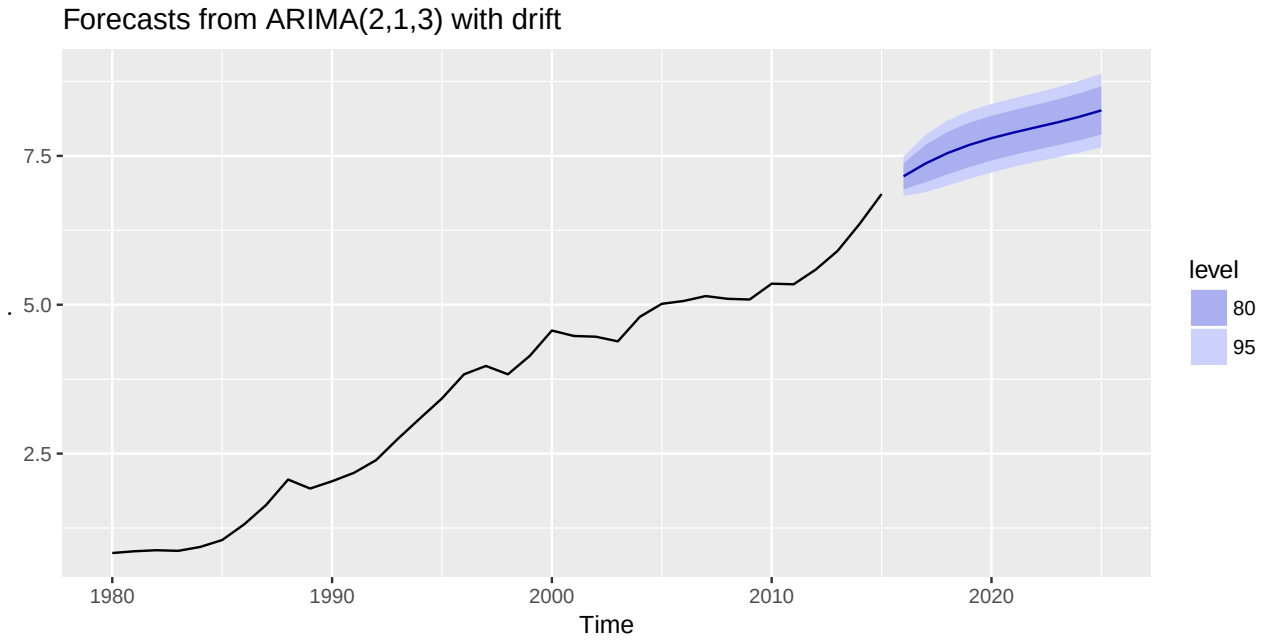
```
usnetelec %>%
    Arima(order = c(2,1,2), include.constant = TRUE) %>%
    forecast() %>%
    autoplot()
```



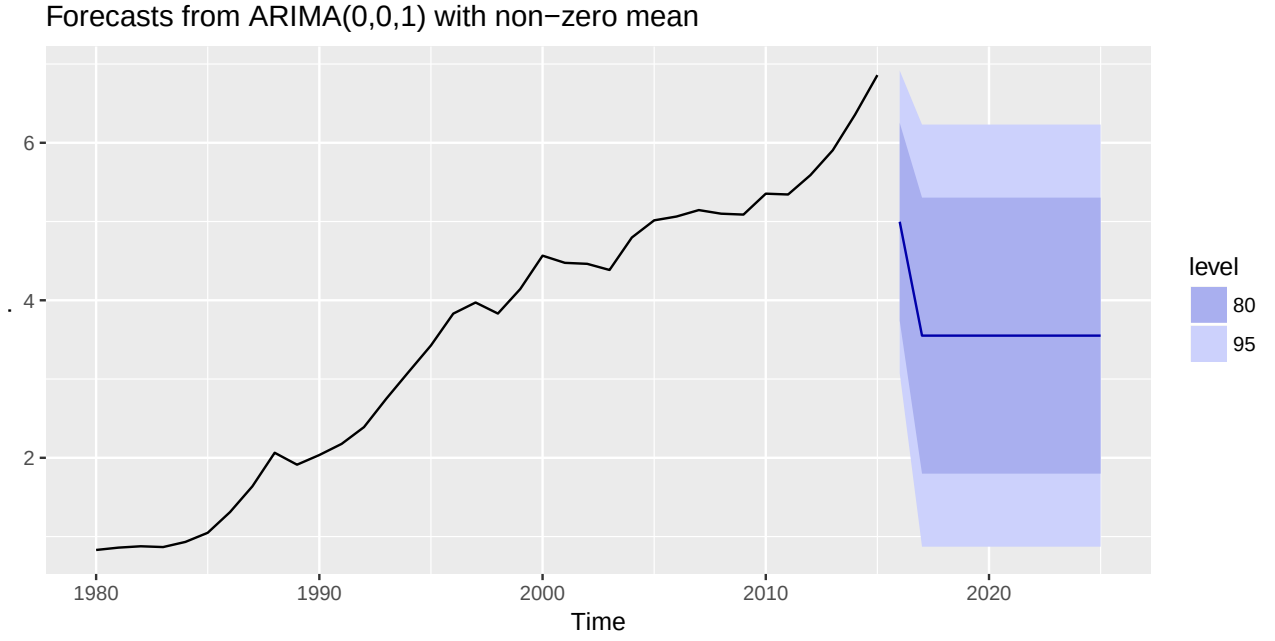

```
# Plot forecasts from an ARIMA(0,1,1) model with no drift
austa %>% Arima(order = c(0, 1, 1), include.constant =   FALSE) %>% forecast() %>% autoplot()
```



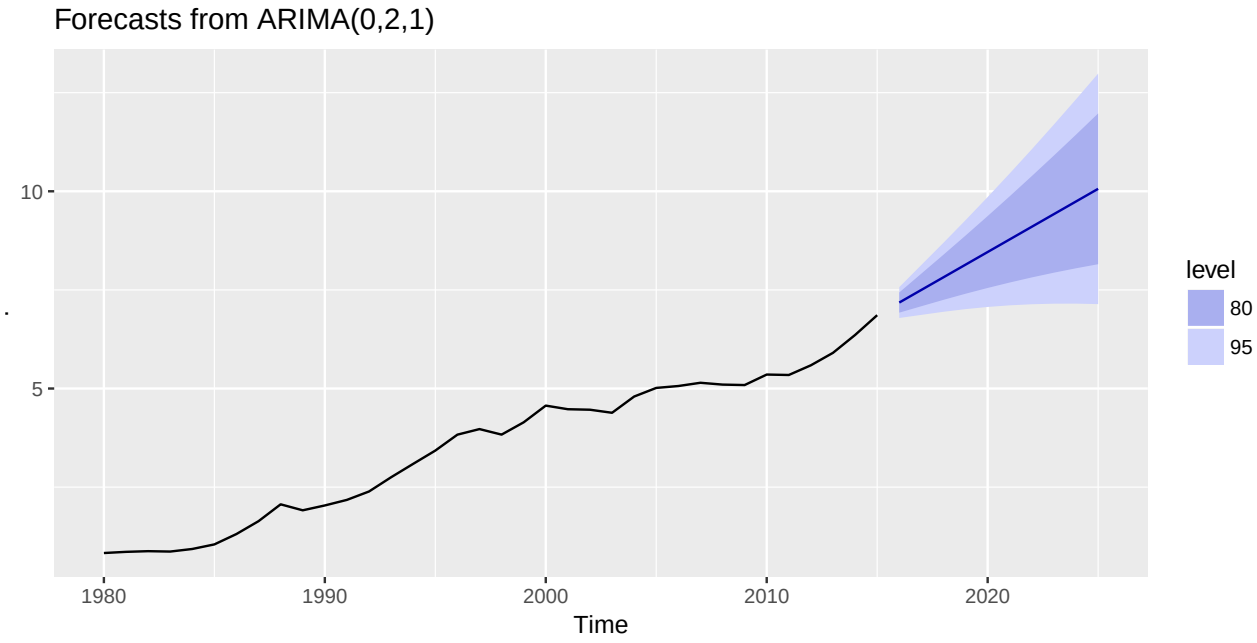

```
# Plot forecasts from an ARIMA(2,1,3) model with drift
austa %>% Arima(order = c(2, 1, 3), include.constant = TRUE) %>% forecast() %>% autoplot()
```




```
# Plot forecasts from an ARIMA(0,0,1) model with a constant
austa %>% Arima(order = c(0, 0, 1),  include.constant = TRUE) %>% forecast() %>% autoplot()
```




```
# Plot forecasts from an ARIMA(0,2,1) model with no constant
austa %>% Arima(order = c(0, 2, 1), include.constant = FALSE) %>% forecast() %>% autoplot()
```



Good job. The model specification makes a big impact on the forecast!

Comparing auto.arima() and ets() on non-seasonal data

The AICc statistic is useful for selecting between models in the same class. For example, you can use it to select an ETS model or to select an ARIMA model. However, you cannot use it to compare ETS and ARIMA models because they are in different model classes.

Instead, you can use time series cross-validation to compare an ARIMA model and an ETS model on the austa data. Because `tsCV()` requires functions that return forecast objects, you will set up some simple functions that fit the models and return the forecasts. The arguments of `tsCV()` are a time series, forecast function, and forecast horizon h. Examine this code snippet from the second chapter:

```
# e <- matrix(NA_real_, nrow = 1000, ncol = 8)
# for (h in 1:8)
#   e[, h] <- tsCV(goog, naive, h = h)
```

Furthermore, recall that pipe operators in R take the value of whatever is on the left and pass it as an argument to whatever is on the right, step by step, from left to right. Here's an example based on code you saw in an earlier chapter:

Plot 20-year forecasts of the lynx series modeled by ets()

```
# lynx %>% ets() %>% forecast(h = 20) %>% autoplot()
```

In this exercise, you will compare the MSE of two forecast functions applied to austa, and plot forecasts of the function that computes the best forecasts. Once again, austa has been loaded into your workspace.

```
# Set up forecast functions for ETS and ARIMA models
fets <- function(x, h) {
  forecast(ets(x), h = h)
}
farima <- function(x, h) {
  forecast(auto.arima(x), h = h)
}
```

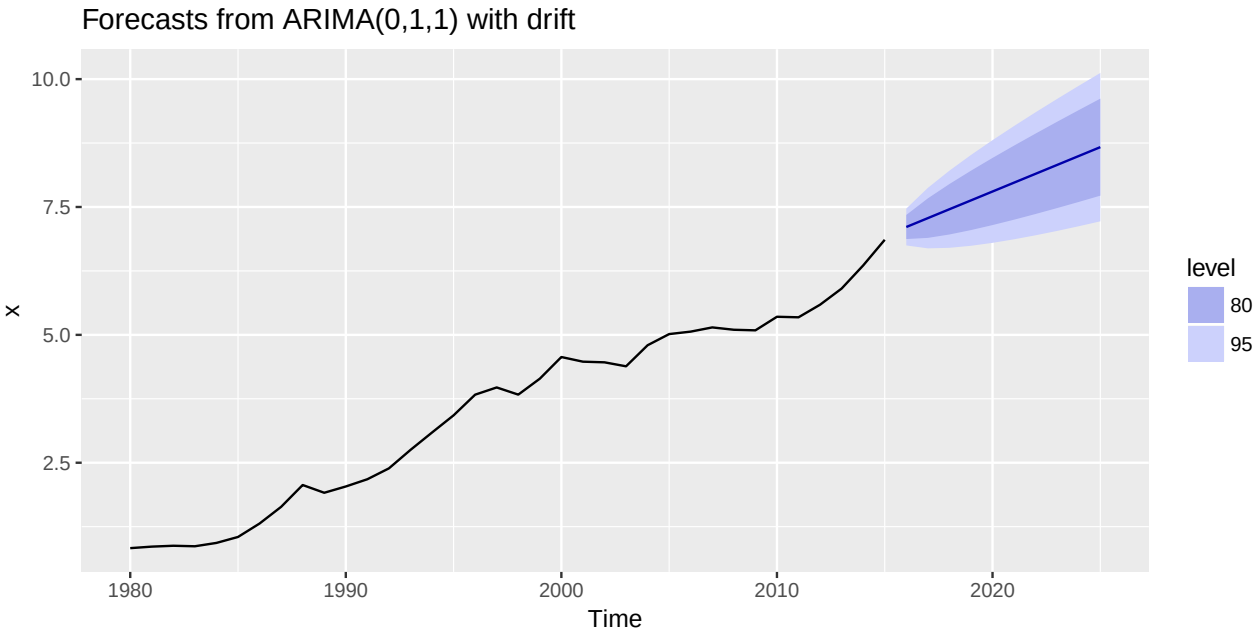
```
# Compute CV errors for ETS as e1
e1 <- tsCV(austa, fets, h = 1)

# Compute CV errors for ARIMA as e2
e2 <- tsCV(austa, farima, h = 1)

# Find MSE of each model class
mean(e1^2, na.rm = TRUE)
```

```
## [1] 0.05623684
```

```
mean(e2^2, na.rm = TRUE)
```

```
## [1] 0.04336277
```

```
# Plot 10-year forecasts using the best model class
austa %>% farima(h = 10) %>% autoplot()
```



Great! Now you know how to compare across model classes.

Seasonal ARIMA Models

ARIMA models can also handle seasonal time series. You just need to add seasonal differencing and a whole lot more lag times into the model. An ARIMA (p, d, q) model involves:

d = Number of lag-1 differences p = Number of ordinary AR lags: $y_{t-1}, y_{t-2}, ..., y_{t-p}$ *q = Number of ordinary MA lags:* $\epsilon_{t-1}, \epsilon_{t-2}, ..., \epsilon_{t-q}$ p lagged observations and q lagged errors

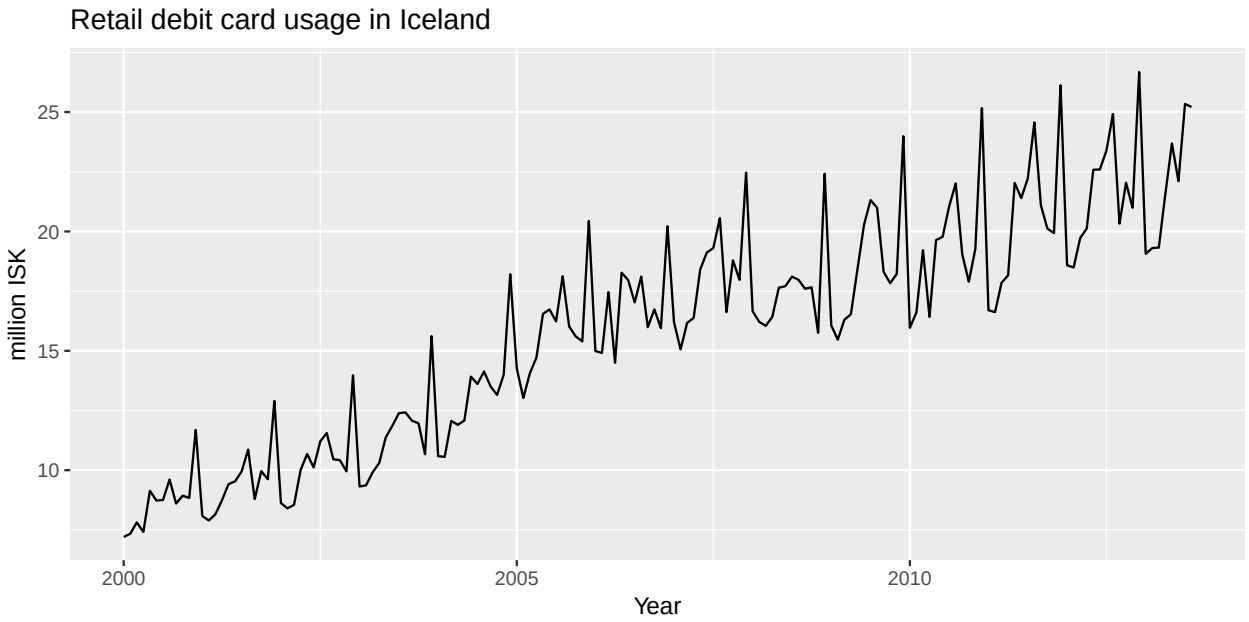
For seasonal ARIMA models, we have another P, D, and Q.

D = Number of seasonal differences P = Number of seasonal AR lags: $y_{t-m}, y_{t-2m}, ..., y_{t-Pm}$ *Q = Number of seasonal MA lags:* $\epsilon_{t-m}, \epsilon_{t-2m}, ..., \epsilon_{t-Qm}$ P is number of seasonally lagged observation and Q is the number of seasonally lagged errors. *m = Number of observations in each year of data.

As you can imagine, these models can be very complicated to write down. They are also no longer linear as the seasonal part get multiplied with the non-seasonal part of the model. Let's look at an example of

monthly data.

```
autoplot(debitcards) +
  xlab("Year") + ylab("million ISK") +
  ggtitle("Retail debit card usage in Iceland")
```



This is the total usage of debitcards in Iceland from January 2000 to August 2013. There is increasing variation so you need to use a `BoxCox` transformation. To keep it simple, let's use a log transformation and set $\lambda = 0$. We apply `auto.arima()` to the data setting $\lambda = 0$

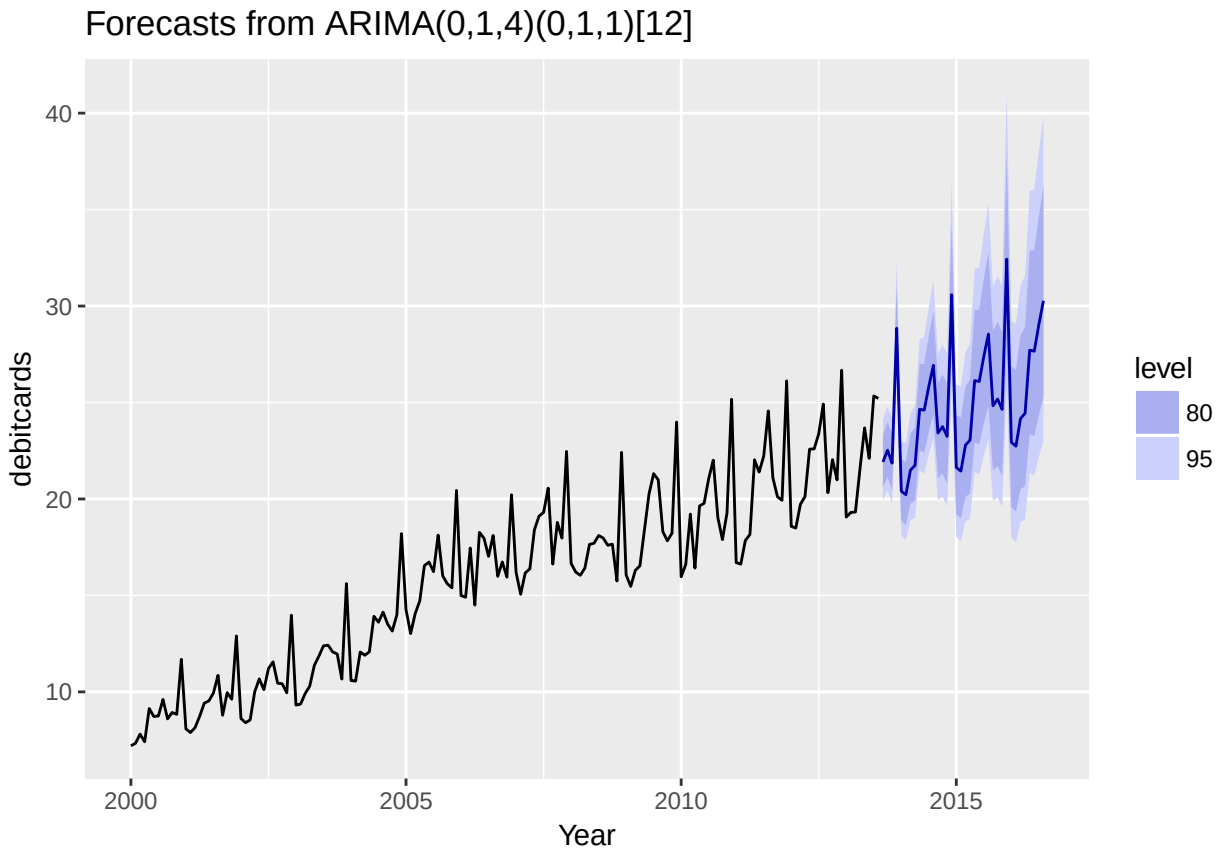
```
fit <- auto.arima(debitcards, lambda = 0)
fit
```

```
## Series: debitcards
## ARIMA(0,1,4)(0,1,1)[12]
## Box Cox transformation: lambda= 0
##
## Coefficients:
##           ma1     ma2     ma3      ma4     sma1
##       -0.7959  0.0856  0.2628  -0.1751  -0.8144
## s.e.   0.0825  0.0988  0.0999   0.0798   0.1118
##
## sigma^2 estimated as 0.002321:  log likelihood=239.33
## AIC=-466.67   AICc=-466.08   BIC=-448.56
```

The resulting model is an ARIMA $(0, 1, 4)(0, 1, 1)$ i.e. both seasonal and lag-1 differences have been used indicated by the 1 in the middle part of the model. It also tells us that four lagged errors and one seasonally lagged errors have been selected. The 0s indicates that no AR terms have been used. The rest of the output tells us about the model coefficients and other model information.

ARIMA models can be hard to interpret and those coefficients don't have a neat explanation in terms of the original data. But they are very powerful models which can handle a very wide range types of time series. You can pass the model object to the `forecast()` function to get forecasts

```
fit %>%
  forecast(h = 36) %>%
  autoplot() + xlab("Year")
```



We are forecasting three years ahead so we can clearly see increasing trend and increasing variation due to the `BoxCox` transformation. Notice, how the seasonal part of the model have captured the seasonal part quite well. A nice feature of seasonal ARIMA models is that they allow the seasonality to change over time. The forecasted seasonal pattern will be most affected by the shape of the seasonality near the end of the series and not so much the seasonal pattern at the start of the series.

You might wonder where the trend comes from in this model as there is no drift term here. **It turns out that whenever you do two lots of differencing of any kind, the forecast will have a trend without needing to include a constant**. Here, you have done both ordinary and seasonal differencing so there is a trend in the forecast.

Automatic ARIMA models for Seasonal Time Series

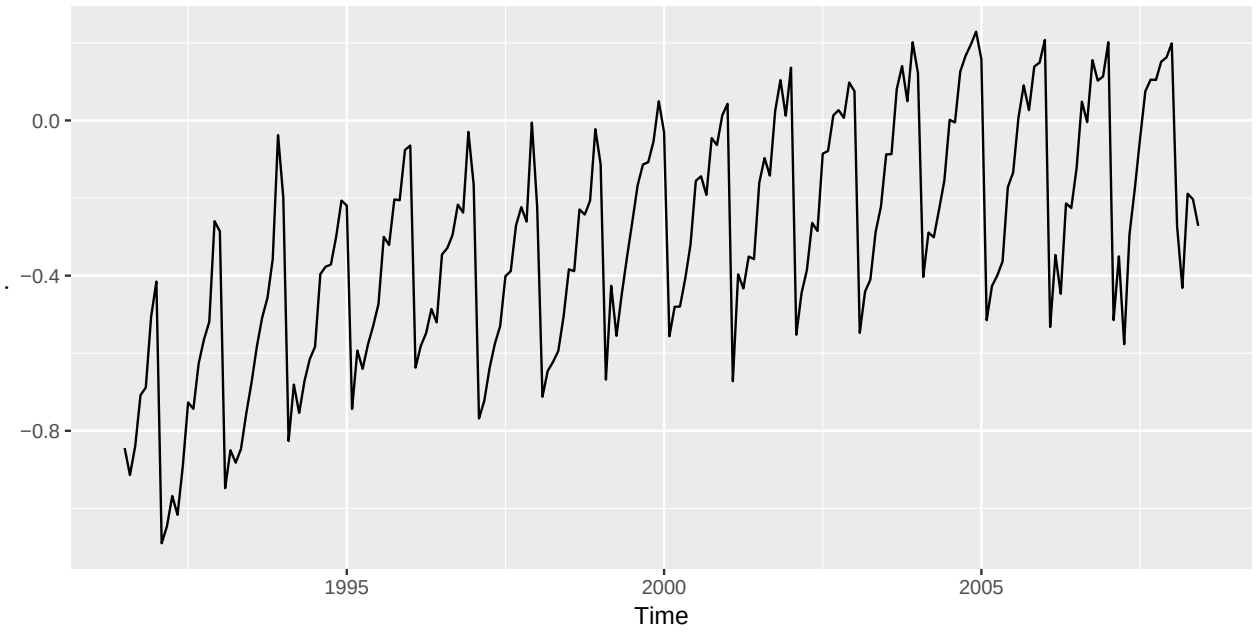
As you learned in the video, the auto.arima() function also works with seasonal data. Note that setting lambda = 0 in the auto.arima() function – applying a log transformation - means that the model will be fitted to the transformed data, and that the forecasts will be back-transformed onto the original scale.

After applying summary() to this kind of fitted model, you may see something like the output below which corresponds with (p,d,q)(P,D,Q)m(P,D,Q)[m]:

*ARIMA(0,1,4)(0,1,1)[12]

In this exercise, you will use these functions to model and forecast the pre-loaded h02 data, which contains monthly sales of cortecosteroid drugs in Australia.

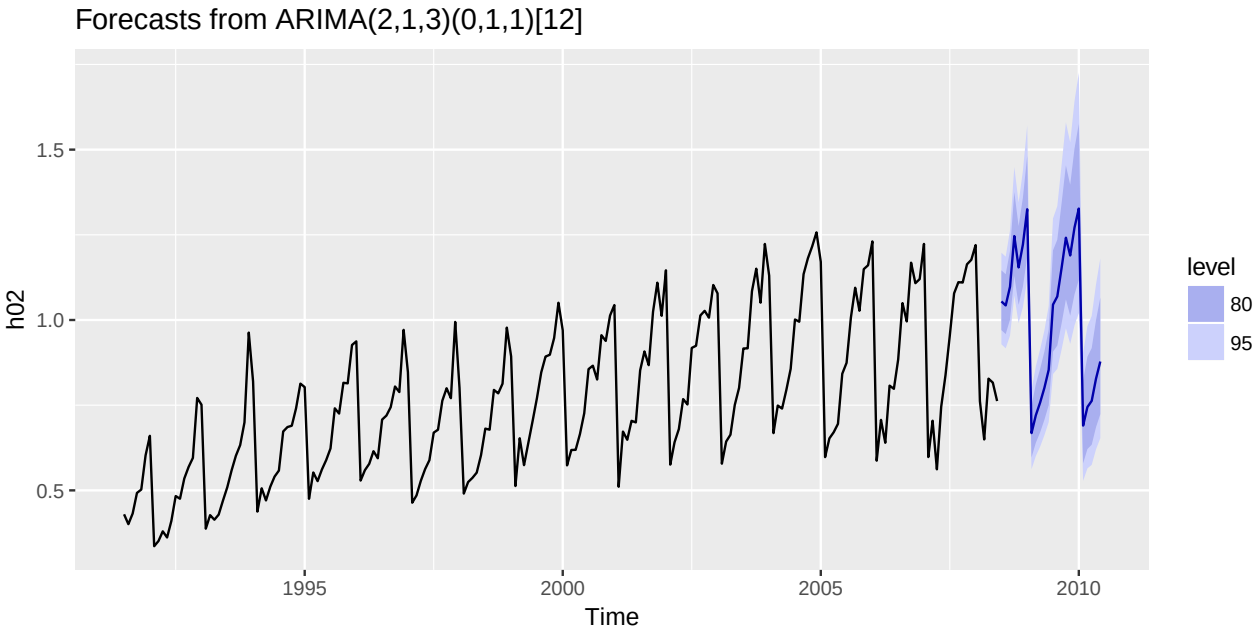
```
# Check that the logged h02 data have stable variance
# h02 %>% BoxCox(lambda = 0.0) %>% autoplot()
h02 %>% log() %>% autoplot()
```




```
# Fit a seasonal ARIMA model to h02 with lambda = 0
fit <- auto.arima(h02, lambda = 0)

# Summarize the fitted model
summary(fit)
```

```
## Series: h02
## ARIMA(2,1,3)(0,1,1)[12]
## Box Cox transformation: lambda= 0
##
## Coefficients:
##           ar1      ar2     ma1     ma2      ma3     sma1
##       -1.0194  -0.8351  0.1717  0.2578  -0.4206  -0.6528
## s.e.   0.1648   0.1203  0.2079  0.1177   0.1060   0.0657
##
## sigma^2 estimated as 0.004203:  log likelihood=250.8
## AIC=-487.6   AICc=-486.99   BIC=-464.83
##
## Training set error measures:
##                       ME       RMSE        MAE       MPE    MAPE
## Training set -0.003823286 0.05006017 0.03588577 -0.643286 4.52991
##                    MASE         ACF1
## Training set 0.5919957 -0.007519926
```

```
# Record the amount of lag-1 differencing and seasonal differencing used
d <- 1
D <- 1

# Plot 2-year forecasts
fit %>% forecast(h = 24) %>% autoplot()
```



Exploring auto.arima() options

The `auto.arima()` function needs to estimate a lot of different models, and various short-cuts are used to try to make the function as fast as possible. This can cause a model to be returned which does not actually have the smallest AICc value. To make `auto.arima()` work harder to find a good model, add the optional argument `stepwise = FALSE` to look at a much larger collection of models.

Here, you will try finding an ARIMA model for the pre-loaded `euretail` data, which contains quarterly retail trade in the Euro area from 1996-2011. Inspect it in the console before beginning this exercise.

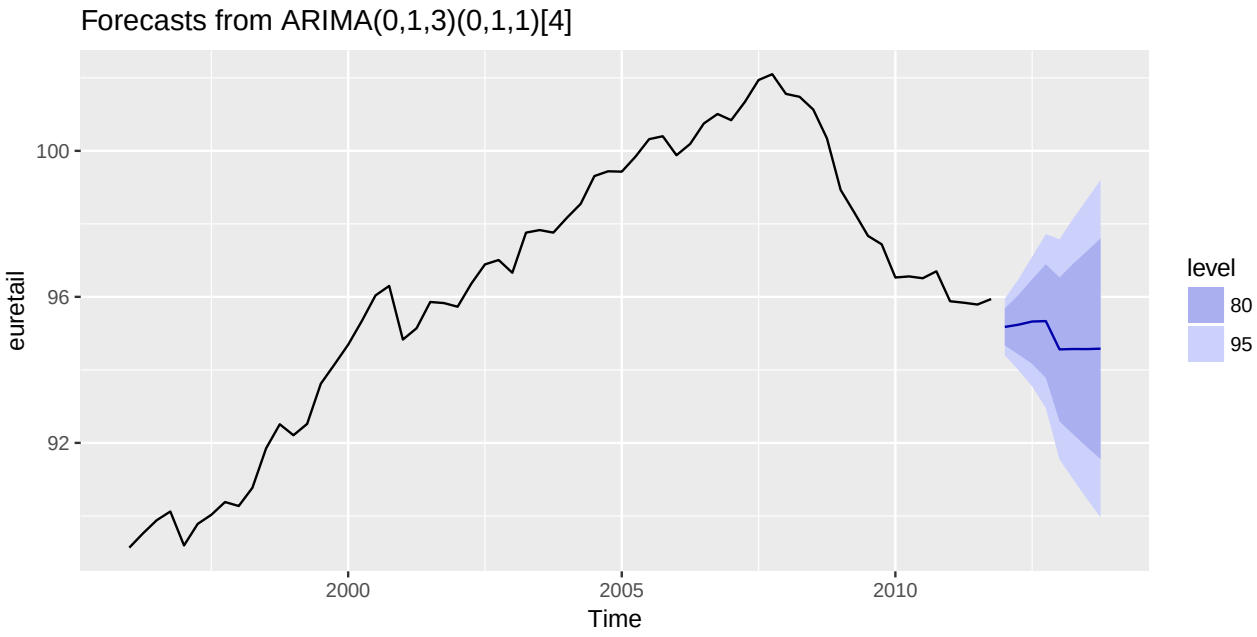
```
# Find an ARIMA model for euretail
fit1 <- auto.arima(euretail)

# Don't use a stepwise search
fit2 <- auto.arima(euretail, stepwise = FALSE)
summary(fit1)
```

```
## Series: euretail
## ARIMA(1,1,2)(0,1,1)[4]
##
## Coefficients:
##          ar1      ma1     ma2     sma1
##       0.7362  -0.4663  0.2163  -0.8433
## s.e.  0.2243   0.1990  0.2101   0.1876
##
## sigma^2 estimated as 0.1587:  log likelihood=-29.62
## AIC=69.24   AICc=70.38   BIC=79.63
##
## Training set error measures:
##                       ME      RMSE       MAE         MPE      MAPE
## Training set -0.03183461 0.3693126 0.2788867 -0.03031231 0.2889988
##                    MASE       ACF1
## Training set 0.2268601 0.01482898
```

```
summary(fit2)
```

```
## Series: euretail
## ARIMA(0,1,3)(0,1,1)[4]
##
## Coefficients:
##          ma1     ma2     ma3     sma1
##       0.2630  0.3694  0.4200  -0.6636
## s.e.  0.1237  0.1255  0.1294   0.1545
##
## sigma^2 estimated as 0.156:  log likelihood=-28.63
## AIC=67.26   AICc=68.39   BIC=77.65
##
## Training set error measures:
##                       ME      RMSE       MAE         MPE      MAPE
## Training set -0.02965298 0.3661147 0.2787802 -0.02795377 0.2885545
##                    MASE        ACF1
## Training set 0.2267735 0.006455781
```

```
# AICc of better model
AICc <- 68.39

# Compute 2-year forecasts from better model
fit2 %>% forecast(h = 8) %>% autoplot()
```



Comparing auto.arima() and ets() on seasonal data

What happens when you want to create training and test sets for data that is more frequent than yearly? If needed, you can use a vector in form `c(year, period)` for the `start` and/or `end` keywords in the `window()` function. You must also ensure that you're using the appropriate values of `h` in forecasting functions. Recall that h should be equal to the length of the data that makes up your test set.

For example, if your data spans 15 years, your training set consists of the first 10 years, and you intend to forecast the last 5 years of data, you would use `h = 12 * 5` not `h = 5` because your test set would include

60 monthly observations. If instead your training set consists of the first 9.5 years and you want forecast the last 5.5 years, you would use `h = 66` to account for the extra 6 months.

In the final exercise for this chapter, you will compare seasonal ARIMA and ETS models applied to the quarterly cement production data `qcement`. Because the series is very long, you can afford to use a training and test set rather than time series cross-validation. This is much faster.

```
# Use 20 years of the qcement data beginning in 1988
train <- window(qcement, start = c(1988, 1), end = c(2007, 4))

# Fit an ARIMA and an ETS model to the training data
fit1 <- auto.arima(train, stepwise = FALSE)
fit2 <- ets(train)
summary(fit1)
```

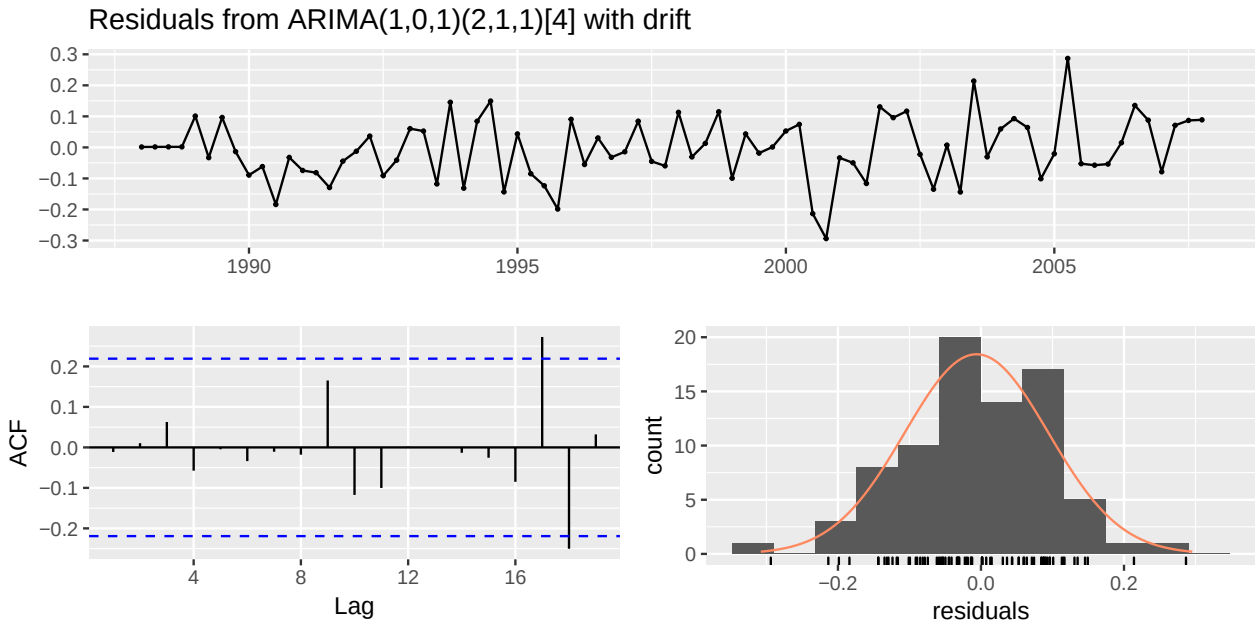
```
## Series: train
## ARIMA(1,0,1)(2,1,1)[4] with drift
##
## Coefficients:
##          ar1      ma1   sar1     sar2     sma1   drift
##       0.8886  -0.2366  0.081  -0.2345  -0.8979  0.0105
## s.e.  0.0842   0.1334  0.157   0.1392   0.1780  0.0029
##
## sigma^2 estimated as 0.01146:  log likelihood=61.47
## AIC=-108.95   AICc=-107.3   BIC=-92.63
##
## Training set error measures:
##                        ME      RMSE        MAE        MPE     MAPE
## Training set -0.006205705 0.1001195 0.07988903 -0.6704455 4.372443
##                    MASE        ACF1
## Training set 0.5458078 -0.01133907
```

```
summary(fit2)
```

```
## ETS(M,N,M)
##
## Call:
##  ets(y = train)
##
##   Smoothing parameters:
##     alpha = 0.7341
##     gamma = 1e-04
##
##   Initial states:
##     l = 1.6439
##     s=1.031 1.0439 1.0103 0.9148
##
##   sigma:  0.0585
##
##        AIC       AICc        BIC
## -2.1967020 -0.6411464 14.4774845
##
## Training set error measures:
##                       ME      RMSE        MAE       MPE     MAPE      MASE
## Training set 0.01406512 0.1022079 0.07958478 0.4938163 4.371823 0.5437292
##                      ACF1
```

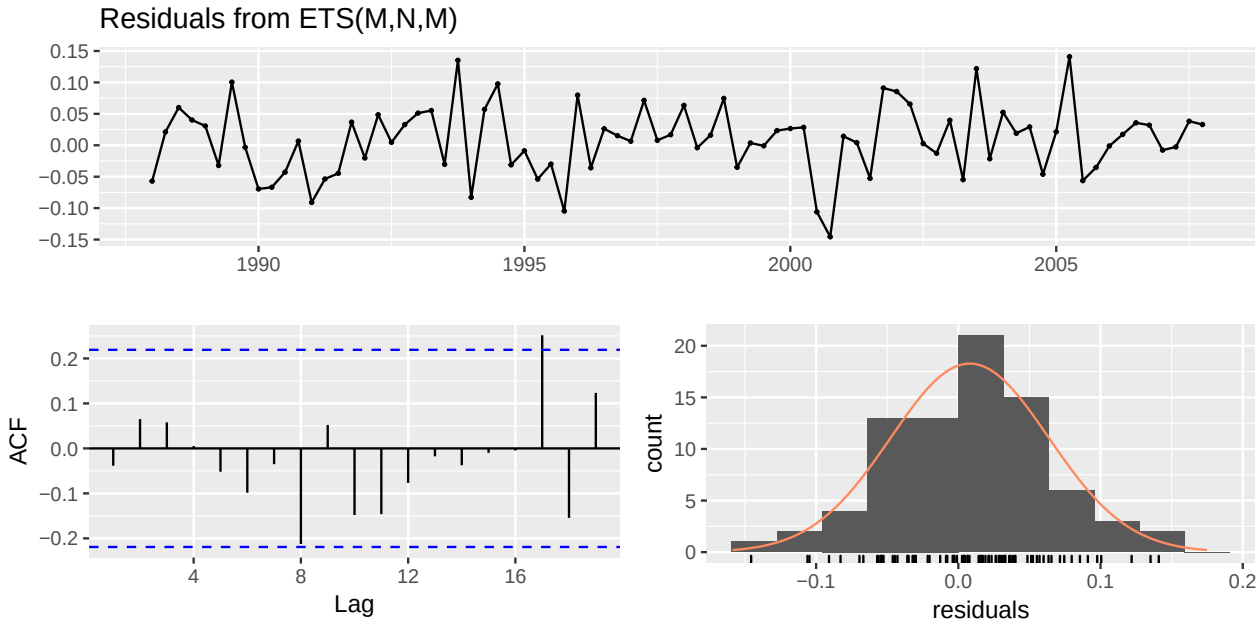
```
## Training set -0.03346295
```

```
# Check that both models have white noise residuals
# ggAcf(fit1$residuals)
# ggAcf(fit2$residuals)
checkresiduals(fit1)
```




```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(1,0,1)(2,1,1)[4] with drift
## Q* = 3.3058, df = 3, p-value = 0.3468
##
## Model df: 6.   Total lags used: 9
```

```
checkresiduals(fit2)
```



```
##
##  Ljung-Box test
##
## data:  Residuals from ETS(M,N,M)
## Q* = 6.3457, df = 3, p-value = 0.09595
##
## Model df: 6.   Total lags used: 9
```

```
# Produce forecasts for each model
fc1 <- forecast(fit1, h = 25)
fc2 <- forecast(fit2, h = 25)

# Use accuracy() to find better model based on RMSE
accuracy(fc1, qcement)
```

```
##                       ME      RMSE        MAE        MPE     MAPE
## Training set -0.006205705 0.1001195 0.07988903 -0.6704455 4.372443
## Test set     -0.158835254 0.1996098 0.16882205 -7.3332836 7.719241
##                   MASE        ACF1 Theil's U
## Training set 0.5458078 -0.01133907        NA
## Test set     1.1534049  0.29170452 0.7282225
```

```
accuracy(fc2, qcement)
```

```
##                       ME      RMSE        MAE        MPE     MAPE
## Training set  0.01406512 0.1022079 0.07958478  0.4938163 4.371823
## Test set     -0.13495518 0.1838791 0.15395142 -6.2508985 6.986077
##                   MASE        ACF1 Theil's U
## Training set 0.5437292 -0.03346295        NA
## Test set     1.0518076  0.53438368  0.680556
```

```
bettermodel <- fit2
```

HINT

*In window(), start is equal to the starting year of the qcement data, 1988, and end is in the format c(YYYY, Q). Use head() and tail() on qcement if you're not sure what these dates are.

*The last data point in qcement is in the first quarter of 2014, and the last data point in the training set is in the fourth quarter of 2007. Therefore, h in forecast() is equal to $1 + (4*(2013 - 2007))$.

*You do not need to set up a test set. Just pass the whole of qcement as the test set to accuracy() and it will find the relevant part to use in comparing with the forecasts.