Chapter 5 Review

Anthony Tricarico

Table of contents

1 Intro

This is an explanation of the code used for chapter 5. This chapter of the textbook is focused on producing our first forecasts after fitting various models to our time series data. The models in this chapter are meant to be simple so that we can then use them as benchmark methods for more advanced models that will be developed in the next chapters.

Also, this review will assume that you already know and have acquired basic familiarity with the functions explained up until the third chapter. Note that we will not cover all the mathematical aspects included in the book to keep these explanations simple enough, but of course if you feel like it you can just read through the chapter to understand what is really going on behind the scenes. Finally, if you did not cover basic topics in probability theory and hypothesis testing yet, this is the right moment to do so because we will use these concepts frequently in the following sections of this review and those concepts will be useful in the future (e.g., for your quantitative methods class).

2 Basic functions for Modeling and Forecasting

This section is an overview of the most important functions that we will use to model our data and produce the first forecasts. First let's import the library that contains the functions we will use.

```r
library(fpp3)
```

```
Registered S3 method overwritten by 'tsibble':
  method               from 
  as_tibble.grouped_df dplyr

-- Attaching packages -------------------------------------------- fpp3 1.0.1 --

v tibble      3.2.1     v tsibble     1.1.5
v dplyr       1.1.4     v tsibbledata 0.4.1
v tidyr       1.3.1     v feasts      0.4.1
v lubridate   1.9.3     v fable       0.4.1
v ggplot2     3.5.1     

-- 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()
```

2.1 TSLM()

This function allows you to fit a linear model using the components of a time series (i.e., trend or seasonality).

```r
TSLM(GDP_per_capita ~ trend()) ①
```

① This is how you specify a formula, on the left of the ~ there is the dependent variable and on its right stands the independent variable (i.e., trend)

```
<TSLM model definition>
```

In order to use this formula to fit a linear model you need to use the `model()` function.

2.2 `model()`

This is how you use the model function to fit a model to your data:

```
gdppc <- mutate(global_economy, "GDP_per_capita" = GDP / Population)     ①

(fit <- model(gdppc, trend_model = TSLM(GDP_per_capita ~ trend())))      ②
```

① create the `gdpcc` table by adding to the `global_economy` dataset a new column specifying the GDP per capita.
② we fit the model using the `model()` function and assign its result to the `fit` variable. The result is shown above.

```
# A mable: 263 x 2
# Key:     Country [263]
   Country             trend_model
   <fct>                   <model>
 1 Afghanistan              <TSLM>
 2 Albania                  <TSLM>
 3 Algeria                  <TSLM>
 4 American Samoa           <TSLM>
 5 Andorra                  <TSLM>
 6 Angola                   <TSLM>
 7 Antigua and Barbuda      <TSLM>
 8 Arab World               <TSLM>
 9 Argentina                <TSLM>
10 Armenia                  <TSLM>
# i 253 more rows
```

with `model()` as with many other functions you used so far, you just need to specify where the data that are used in the model are contained (`gdppc` in our case) and the name of the column where you want to store the models produced by the formula we described in Section 2.1 . Notice that the name of the column or the formula you want to use for modeling will change based on the model that you want to specify.

2.2.1 `report()`

This is used to get the results of the model you previously fit to the data.

```
report(filter(fit,Country == 'Sweden')) # see output and evaluate
```

```
Series: GDP_per_capita
Model: TSLM

Residuals:
     Min       1Q   Median       3Q      Max
-11170.4  -2193.1   -505.7   3524.9  10850.2

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6394.78    1324.72  -4.827 1.11e-05 ***
trend()      1060.34      39.06  27.150  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4979 on 56 degrees of freedom
Multiple R-squared: 0.9294, Adjusted R-squared: 0.9281
F-statistic: 737.1 on 1 and 56 DF, p-value: < 2.22e-16
```

The code above reports the result only for Sweden since we filtered for its model among the many contained in `fit` as you can see from the previous output. Usually `report()` will only work for single models and if we try using the following we get a warning which prompts us to use `filter()` or `select()` to get the output we are looking for:

```r
report(fit)
```

```
Warning in report.mdl_df(fit): Model reporting is only supported for individual
models, so a glance will be shown. To see the report for a specific model, use
`select()` and `filter()` to identify a single model.
```

```
# A tibble: 256 x 16
   Country        .model r_squared adj_r_squared sigma2 statistic  p_value    df
   <fct>          <chr>      <dbl>         <dbl>  <dbl>     <dbl>    <dbl> <int>
 1 Afghanistan    trend~     0.756         0.749 8.86e3     111.  1.43e-12     2
 2 Albania        trend~     0.846         0.841 4.24e5     176.  1.55e-14     2
 3 Algeria        trend~     0.752         0.748 6.01e5     170.  1.30e-18     2
 4 American Samoa trend~     0.759         0.742 5.12e5      44.1 1.11e- 5     2
 5 Andorra        trend~     0.860         0.857 2.85e7     284.  2.68e-21     2
 6 Angola         trend~     0.664         0.655 9.71e5      71.2 4.71e-10     2
 7 Antigua and B~ trend~     0.950         0.948 9.43e5     736.  6.31e-27     2
 8 Arab World     trend~     0.811         0.807 8.92e5     207.  5.18e-19     2
 9 Argentina      trend~     0.784         0.780 3.38e6     196.  1.29e-19     2
10 Armenia        trend~     0.846         0.840 3.45e5     143.  4.47e-12     2
# i 246 more rows
# i 8 more variables: log_lik <dbl>, AIC <dbl>, AICc <dbl>, BIC <dbl>,
#   CV <dbl>, deviance <dbl>, df.residual <int>, rank <int>
```

2.3 `forecast()`

This is the function that we use to actually make forecasts after fitting a model to our data.

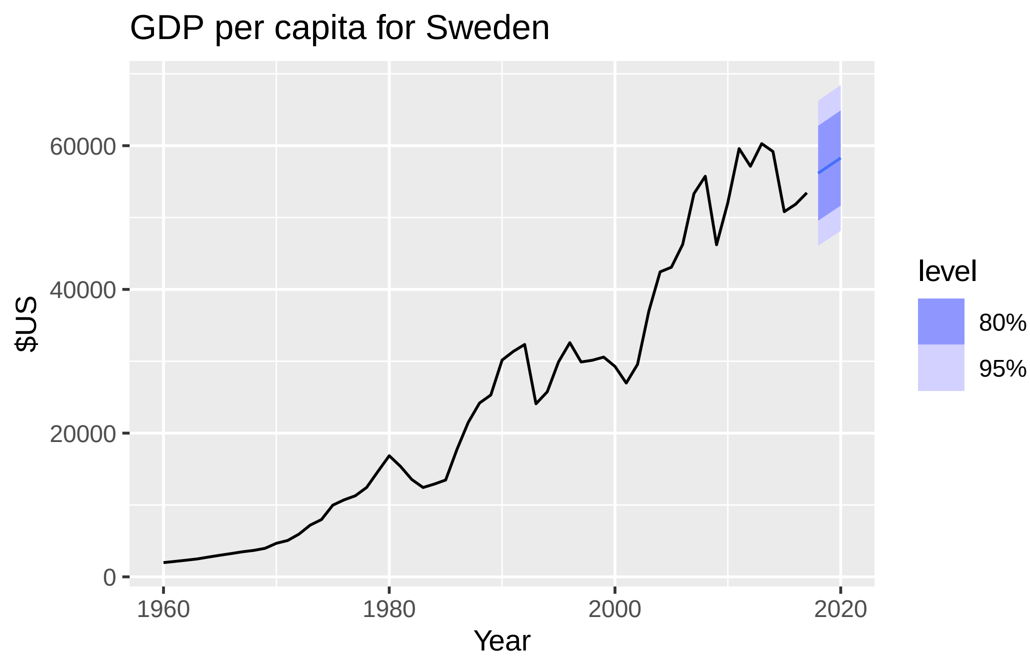
```
forecast(fit, h = "3 years")
```

```
# A fable: 789 x 5 [1Y]
# Key:     Country, .model [263]
   Country        .model       Year
   <fct>          <chr>       <dbl>
 1 Afghanistan    trend_model  2018
 2 Afghanistan    trend_model  2019
 3 Afghanistan    trend_model  2020
 4 Albania        trend_model  2018
 5 Albania        trend_model  2019
 6 Albania        trend_model  2020
 7 Algeria        trend_model  2018
 8 Algeria        trend_model  2019
 9 Algeria        trend_model  2020
10 American Samoa trend_model  2018
# i 779 more rows
# i 2 more variables: GDP_per_capita <dist>, .mean <dbl>
```

The arguments in the function is just the name of the model you previously declared and the number of periods you want to use in your forecast. In this case, we are saying that we want forecasts to be produced for three years ahead of the last one. Since, the `global_economy` dataset contains data up until 2017, the forecast will be for the three years after (2018, 2019, and 2020).

We can now filter our forecasts to only get the forecast value for Sweden and plot our forecasts with the following code:

```
fore <- filter(forecast(fit, h = "3 years"), Country == "Sweden")

autoplot(fore, gdppc) +
  labs(y = "$US",
       title = "GDP per capita for Sweden") # color='black'
```



Notice that the plot also contains 80% and 95% confidence levels (or intervals) for your forecasts, as represented by the dark blue and light blue ranges, respectively.

3 Mean method

The mean method of forecasting simply produces forecasts for future periods that are equal to the mean of the time series considered.

```
recent_prod <- filter_index(aus_production, "1970 Q1" ~ "2004 Q4")
bricks <- dplyr::select(recent_prod, Bricks)
mean_fit <- model(bricks, MEAN(Bricks))                              (1)
```

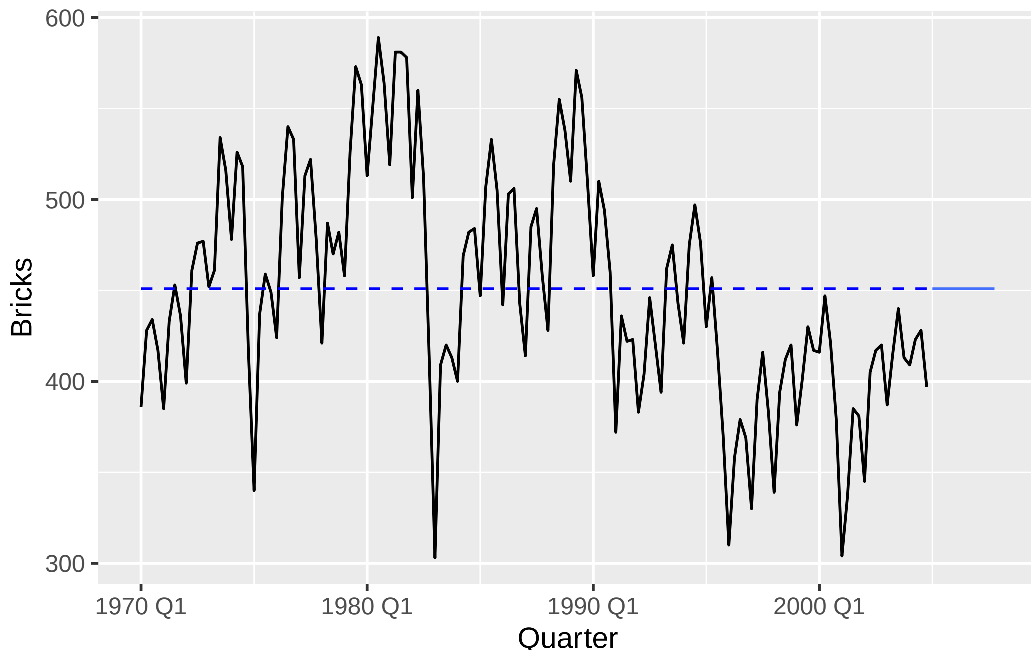
(1) This is how you specify that you want to use the `MEAN()` method for forecasting. As its argument you pass in the name of the column for which you want to get the forecasts (`Bricks` in this case)

```
tidy(mean_fit)  # extract output (1)
```

```
# A tibble: 1 x 6
  .model       term  estimate std.error statistic   p.value
  <chr>        <chr>    <dbl>     <dbl>     <dbl>     <dbl>
1 MEAN(Bricks) mean      451.      5.34      84.4 2.58e-121
```

Using the `tidy()` function you can have a look at a brief report of the mean_fit model. You can see specifically that the estimate is equal to the mean of the time series considered. Also, you get the p-value which tells you about the significance of the model. Since the p-value is very close to 0, this model is statistically significant and we can use it to produce forecasts.

```r
results_list <- mean_fit$'MEAN(Bricks)'[[1]] # extract output (2)
mean_results <- results_list$fit

mean_fc <- forecast(mean_fit, h = 12)
bricks_mean = mutate(bricks,hline=mean_fc$.mean[1]) # add a dashed line
autoplot(mean_fc, bricks, level = NULL) +
  autolayer(bricks_mean,hline,linetype='dashed',color='blue')
```



The lines of code above produce the plot that shows that the forecasts are indeed equal to the mean of the time series.

4 Naive method

This method produces forecasts that are just equal to the last observed value in the time series.

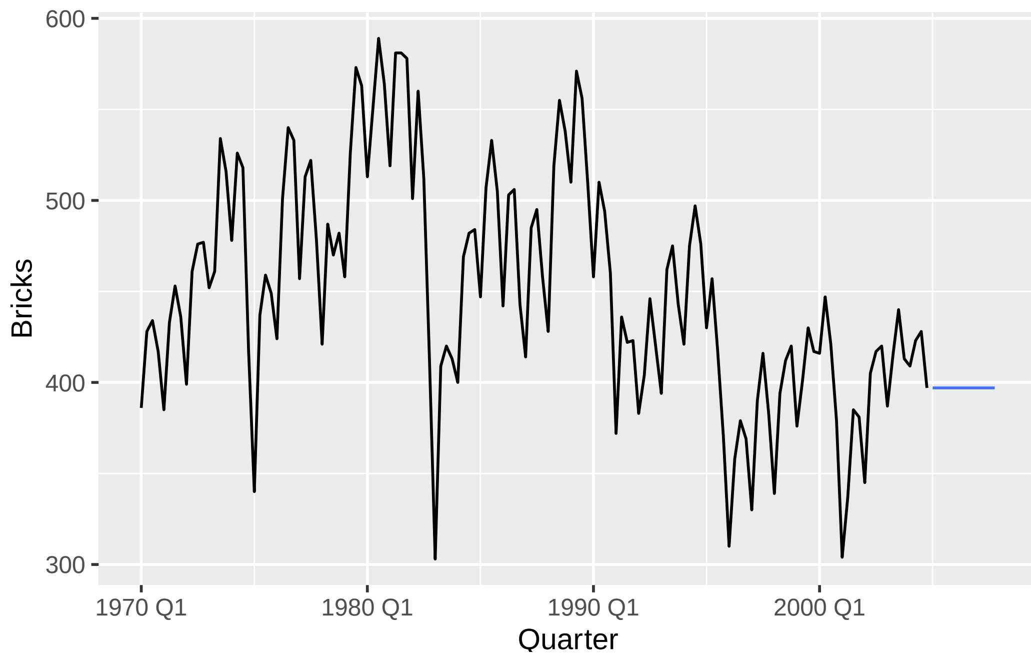
```r
naive_fit <- model(bricks,NAIVE(Bricks))    # (1)
```

(1) Specify the `NAIVE()` model and fit it to the data

```r
naive_fc <- forecast(naive_fit, h = 12)    # (1)
```

(1) Produce forecasts using the model specified

```r
autoplot(naive_fc, bricks, level = NULL)
```



The plot above just shows how the forecasts produced with this method are equal to the last value observed in the series.

5 Seasonal Naive

This method is similar to the Naive method but produces forecasts in the future that are equal to the last seasonal trend observed in the data.

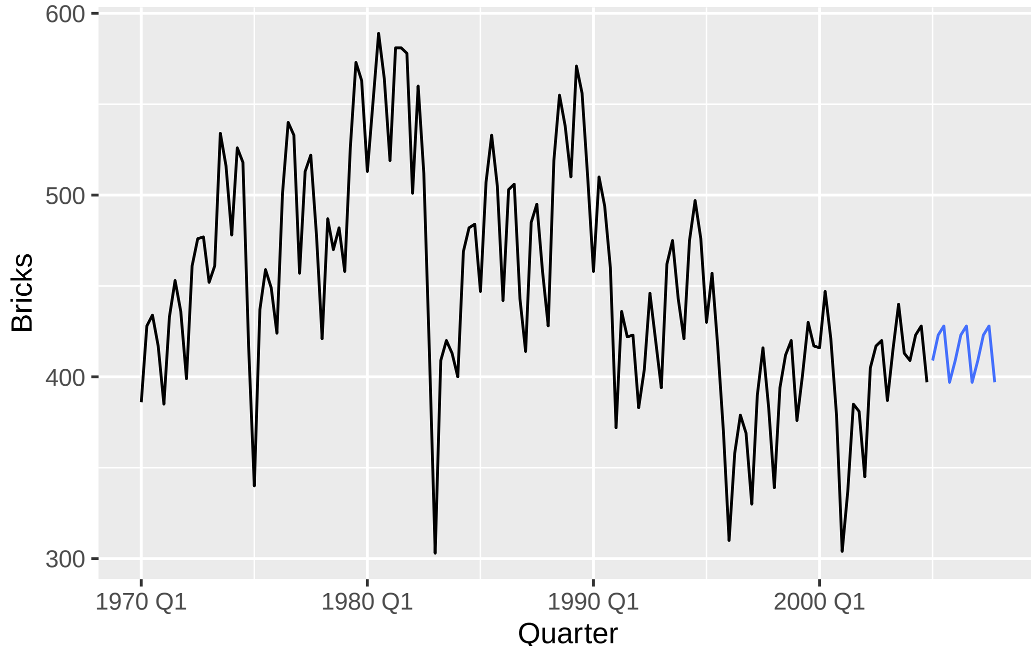
```
snaive_fit <- model(bricks,SNAIVE(Bricks ~ lag("year")))    ①
```

① specify `SNAIVE()` model indicating that the seasonal trend is observed at a yearly interval

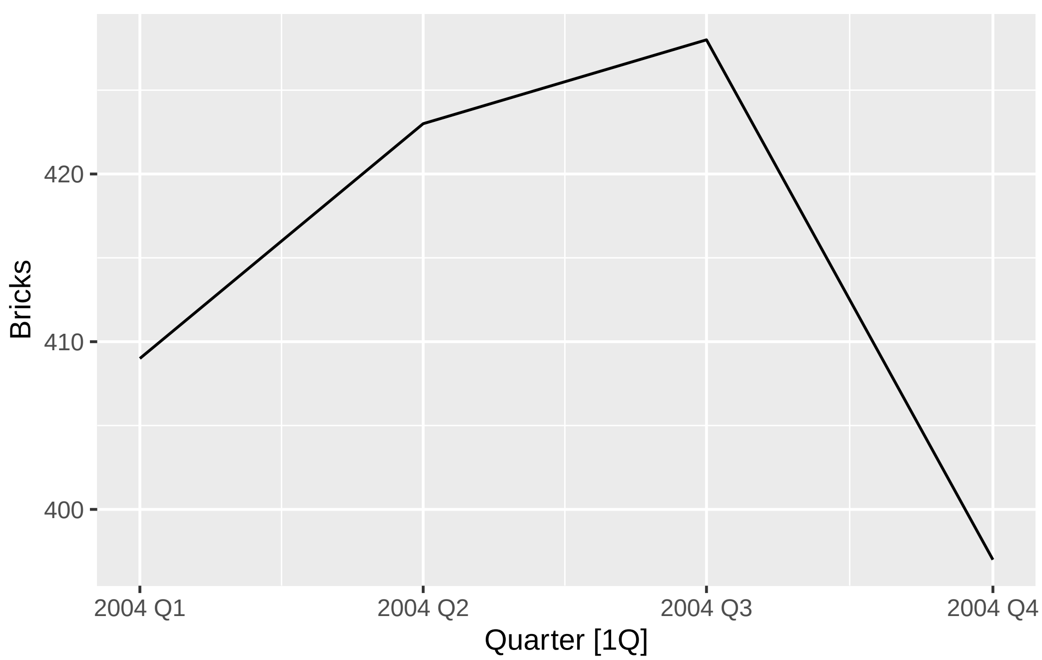
```
snaive_fc <- forecast(snaive_fit, h = 12)
```

```
autoplot(snaive_fc, bricks, level = NULL)    ①
```

① Here the `level` argument is set to `NULL` so that the plot does not contain forecast confidence intervals




```
bricks %>%
  filter_index("Q1 2004"~"Q4 2004") %>%
  autoplot(.vars = Bricks)
```

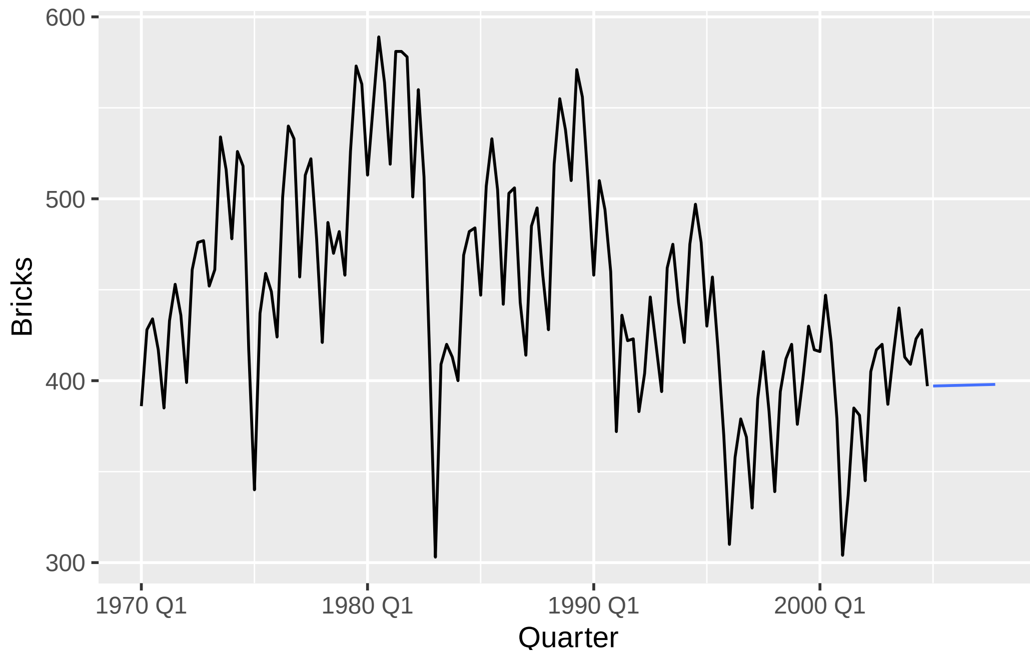


Notice that the yearly trend looks something like the plot above, which is exactly the shape that the future forecasts follow.

6 Drift method

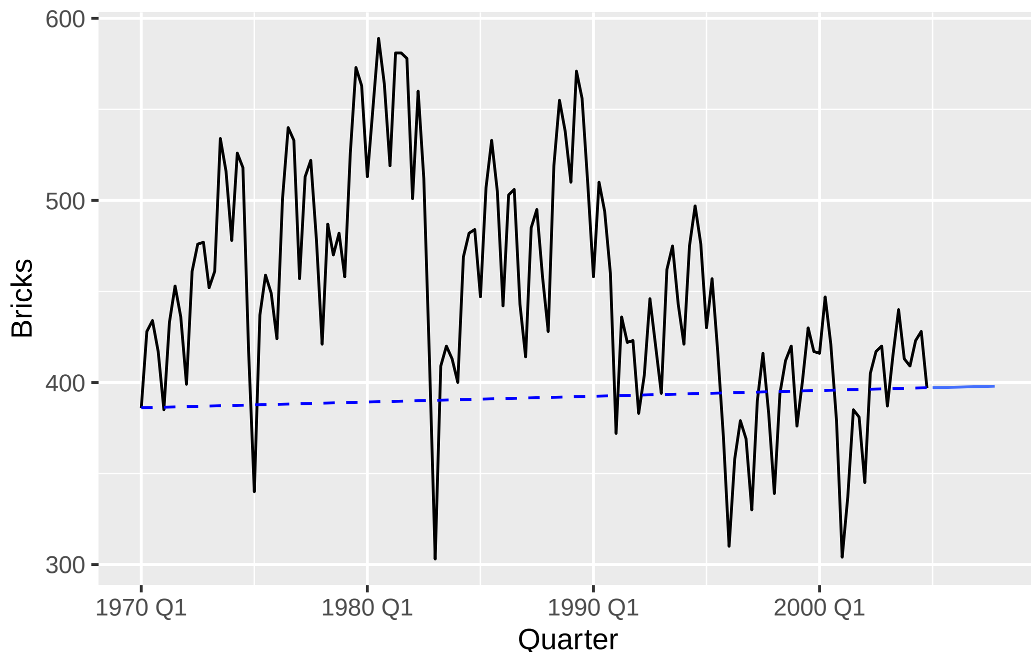
This method interpolates between the first and last observation and the line obtained is then "stretched" into future periods to produce forecasts.

```
drift_fit <- model(bricks, RW(Bricks ~ drift()))
drift_fc <- forecast(drift_fit, h = 12)
autoplot(drift_fc, bricks, level = NULL)
```



To have a basic idea of how this works, we can convince ourselves that the line used in the forecast is actually the line going from the first to the last observation by looking at this plot.

```
T <- length(bricks$Bricks) #getting length of Bricks column
b <- (bricks$Bricks[T] - bricks$Bricks[1])/(T - 1) #equation of a line: slope (row140-row1)/(140-
a <- bricks$Bricks[1]
y <- a + b * seq(1,T,by=1)

DashDR <- tibble(y,Date=bricks$Quarter)
DashDRts <- as_tsibble(DashDR,index=Date)

autoplot(drift_fc, bricks, level = NULL)+
  autolayer(DashDRts,y,color='blue',linetype='dashed')
```



Notice that T, b, a are used to compute the interpolated line and that this follows from the equation of a line of the form $y = mx + q$ where m is the slope parameter (b in the code) and q is the intercept (a in the code, which is just the first observation in the time series).

7 Train / Test Split

To test how well our model performs we need to test in on data on which it was not trained. This is often done to prevent the problem of *overfitting* which refers to the fact that when a parameters of a model are estimated those perform well on the data that the model has already seen but performs poorly on new data (which is actually what should not happen). To mitigate this problem we divide our dataset into a *train* and a *test* portion.

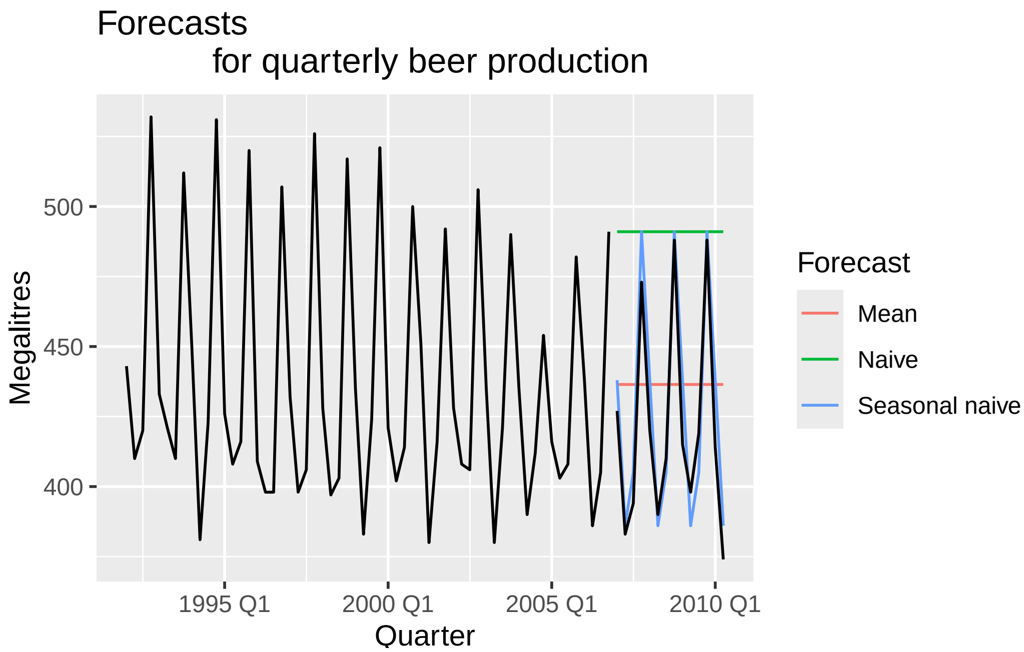
```
train <- filter_index(aus_production, "1992 Q1" ~ "2006 Q4")   ①
```

① Our train dataset is made only of observations from 1992 to 2006.

```
beer_fit <- model(train, Mean = MEAN(Beer), Naive = NAIVE(Beer),
'Seasonal naive' = SNAIVE(Beer))                                ①
beer_fc <- forecast(beer_fit, h = 14)                           ②
```

① fit three different models using the MEAN, NAIVE, and SNAIVE methods
② produce forecasts based on these three models

```
autoplot(beer_fc, train, level = NULL) +
  autolayer(filter_index(aus_production, "2007 Q1" ~ .), .vars = Beer,
  colour = "black") + labs(y = "Megalitres", title = "Forecasts
            for quarterly beer production") +
  guides(colour = guide_legend(title = "Forecast"))
```



Now we can plot how well the three different models perform and we can see that the Seasonal Naive produces more accurate forecasts as it is closer to the original series (black line). Notice, that by the way they were constructed, the models did not see all the data in the series but they were trained only on data up to 2006. Nonetheless, the Seasonal Naive performs pretty well when it tries to make forecasts on data it did not see. In a later section we will see how we can quantify how well a model performs.

7.1 More Examples

```
recent_GOOG <- filter(gafa_stock, Symbol == "GOOG",
                      year(Date) >= 2015)

goog <- mutate(recent_GOOG, day = row_number())                                    (1)

google_stock <- update_tsibble(goog, index = day, regular = TRUE)

google_2015 <- filter(google_stock, year(Date) == 2015) # Filter the year of interest

google_fit <- model(google_2015, # Fit the models
Mean = MEAN(Close), Naive = NAIVE(Close),
Drift = NAIVE(Close ~ drift()))
```

(1) the `row_number()` function adds a sequence of numbers representing the row number of each observation (basically a sequence, starts from 1 and ends with the last observation)

By now you should be able to see what's going on in these lines of code. We are filtering, mutating and updating the original tsibble to take as index the day column we just created before. The last three lines is where we fit models to data.

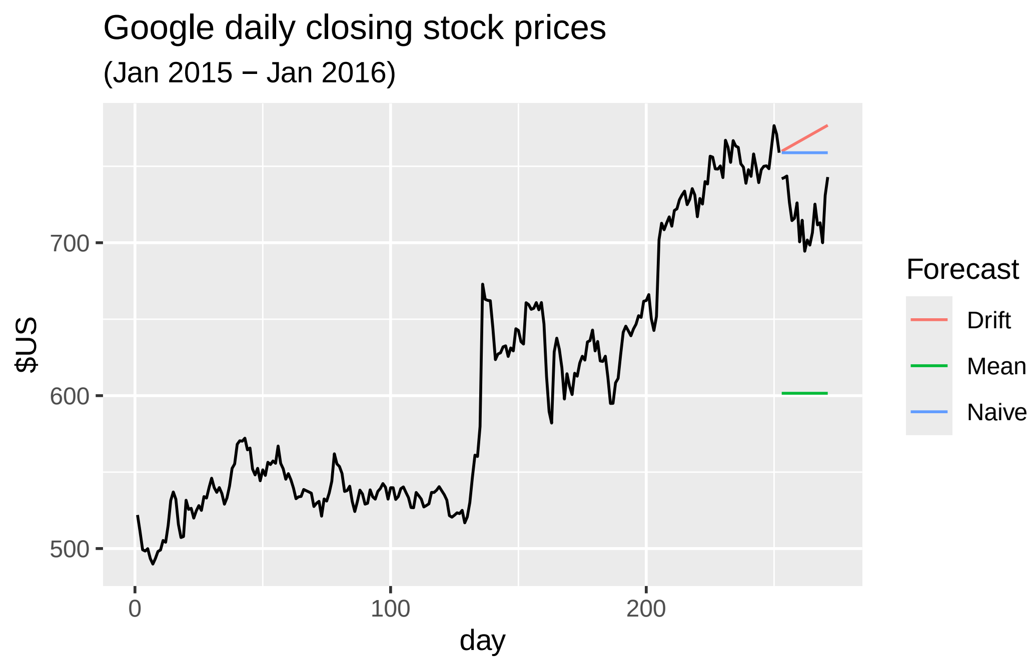
```r
google_jan_2016 <- filter(google_stock,
        yearmonth(Date) == yearmonth("2016 Jan"))   # (1)

(google_fc <- forecast(google_fit, new_data = google_jan_2016))   # (2)
```

(1) Produce forecasts for the trading days in January 2016
(2) the `new_data` argument is used to specify for which datapoints the forecasts should be produced. In this example, we will be producing forecasts only for the trading days in Jan 2016

```
# A fable: 57 x 11 [1]
# Key:     Symbol, .model [3]
   Symbol .model   day
   <chr>  <chr>  <int>
 1 GOOG   Mean     253
 2 GOOG   Mean     254
 3 GOOG   Mean     255
 4 GOOG   Mean     256
 5 GOOG   Mean     257
 6 GOOG   Mean     258
 7 GOOG   Mean     259
 8 GOOG   Mean     260
 9 GOOG   Mean     261
10 GOOG   Mean     262
# i 47 more rows
# i 8 more variables: Close <dist>, .mean <dbl>, Date <date>, Open <dbl>,
#   High <dbl>, Low <dbl>, Adj_Close <dbl>, Volume <dbl>
```

```r
autoplot(google_fc, google_2015, level = NULL) +
  autolayer(google_jan_2016, Close, colour = "black") +   # (1)
  labs(y = "$US", title = "Google daily closing stock prices",
       subtitle = "(Jan 2015 - Jan 2016)") +
  guides(colour = guide_legend(title = "Forecast"))
```

(1) Remember that `autolayer()` just adds another layer to the plot produced with `autoplot()` instead of starting a new plot from a white, empty canvas.



The plot just shows what each model predicts for the Jan 2016 period.

8 Residuals

Residuals are the errors that our model makes when being tested. Basically, they represent the difference between the true observed value and what the model predicted that value to be. We now start with an example to understand how residuals fit in our discussion.

```
beer_fit1 <- model(train, SNAIVE(Beer))
(mean_fitted <- augment(beer_fit1))                                        (1)
```

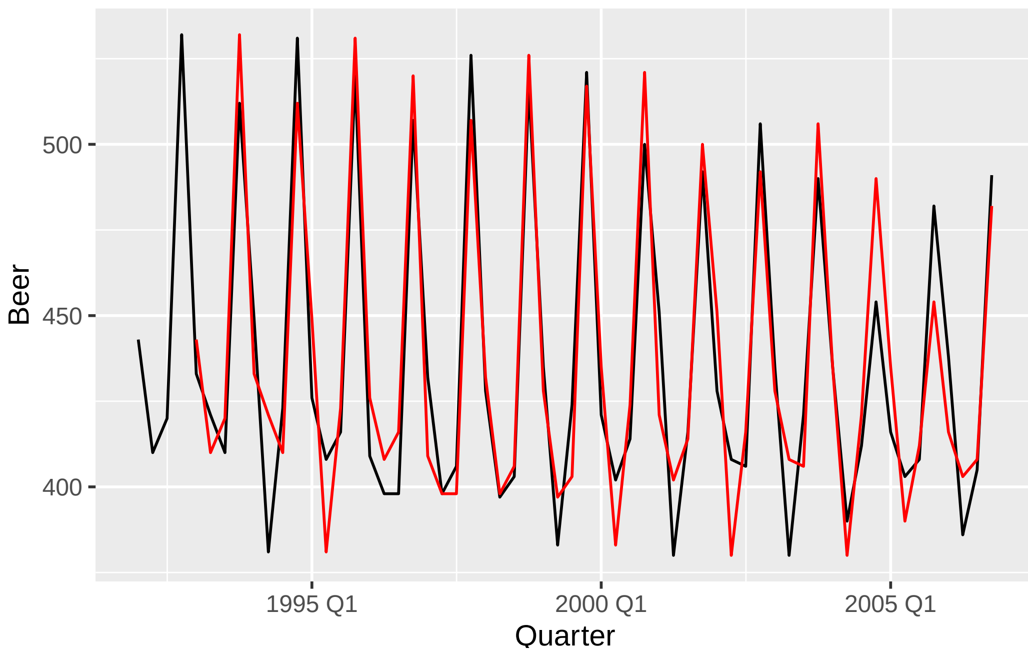
(1) fitted values for a single method, along with residuals and innovation residuals

```
# A tsibble: 60 x 6 [1Q]
# Key:       .model [1]
   .model       Quarter  Beer .fitted .resid .innov
   <chr>          <qtr> <dbl>   <dbl>  <dbl>  <dbl>
 1 SNAIVE(Beer) 1992 Q1   443      NA     NA     NA
 2 SNAIVE(Beer) 1992 Q2   410      NA     NA     NA
 3 SNAIVE(Beer) 1992 Q3   420      NA     NA     NA
 4 SNAIVE(Beer) 1992 Q4   532      NA     NA     NA
 5 SNAIVE(Beer) 1993 Q1   433     443    -10    -10
 6 SNAIVE(Beer) 1993 Q2   421     410     11     11
 7 SNAIVE(Beer) 1993 Q3   410     420    -10    -10
 8 SNAIVE(Beer) 1993 Q4   512     532    -20    -20
 9 SNAIVE(Beer) 1994 Q1   449     433     16     16
10 SNAIVE(Beer) 1994 Q2   381     421    -40    -40
# i 50 more rows
```

We saw how by using `augment()` on a model we previously fit to our data we can get some more information about how it performs. For instance, we can see that the .resid and .innovation column contain the residuals and the innovation residuals for each estimate provided by the model. The difference between residuals and innovation residuals is just that innovation residuals are more interpretable when dealing with data that has gone through transformations. However, since that was not the case with our example, we see that the residuals match perfectly with the innovation ones.

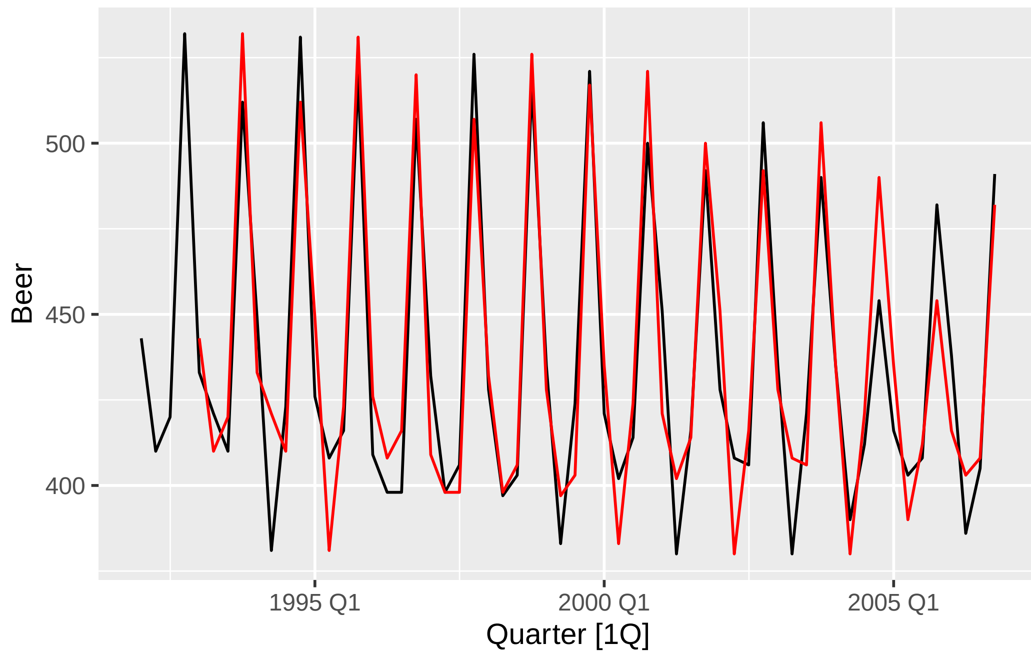
```
ggplot(mean_fitted, aes(x = Quarter)) +
  geom_line(aes(y = Beer),color='black') +
  geom_line(aes(y = .fitted),color='red')
```

```
Warning: Removed 4 rows containing missing values or values outside the scale range
(`geom_line()`).
```



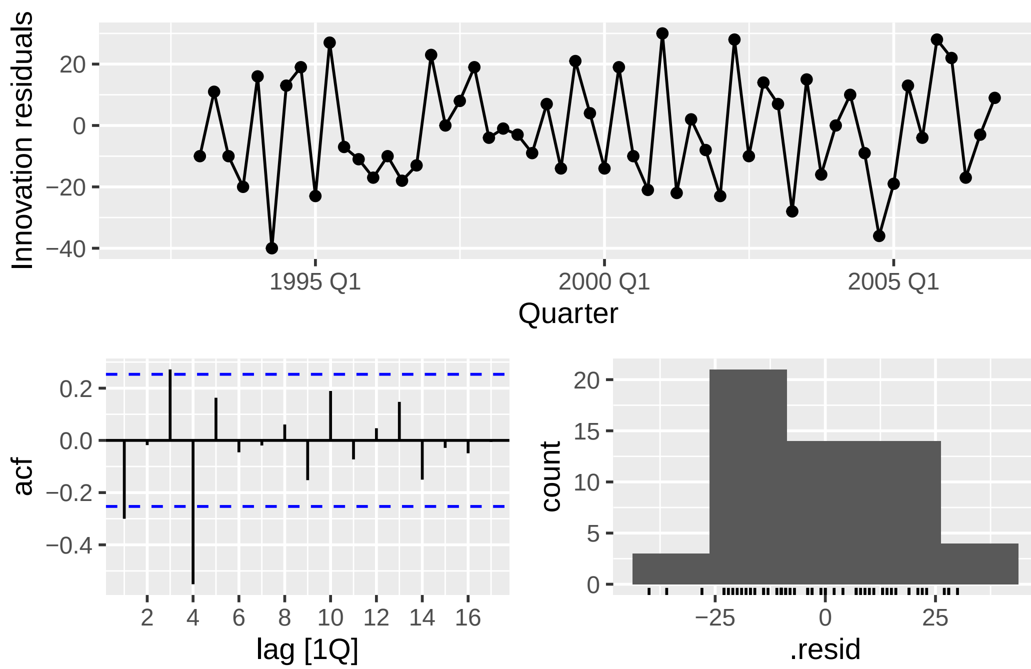

```
autoplot(mean_fitted,.vars = Beer) + # alternative command
  autolayer(mean_fitted,.fitted,color='red')
```

```
Warning: Removed 4 rows containing missing values or values outside the scale range
(`geom_line()`).
```



In the plot we see that the predictions produced are very close to the actual series, meaning that the model performs well on the data.

```r
gg_tsresiduals(beer_fit1)
```



The `gg_tsresiduals()` is a very useful function that allows you to check different assumptions that are imposed on residuals through plots. The usual assumption that are imposed on residuals is that they exhibit:

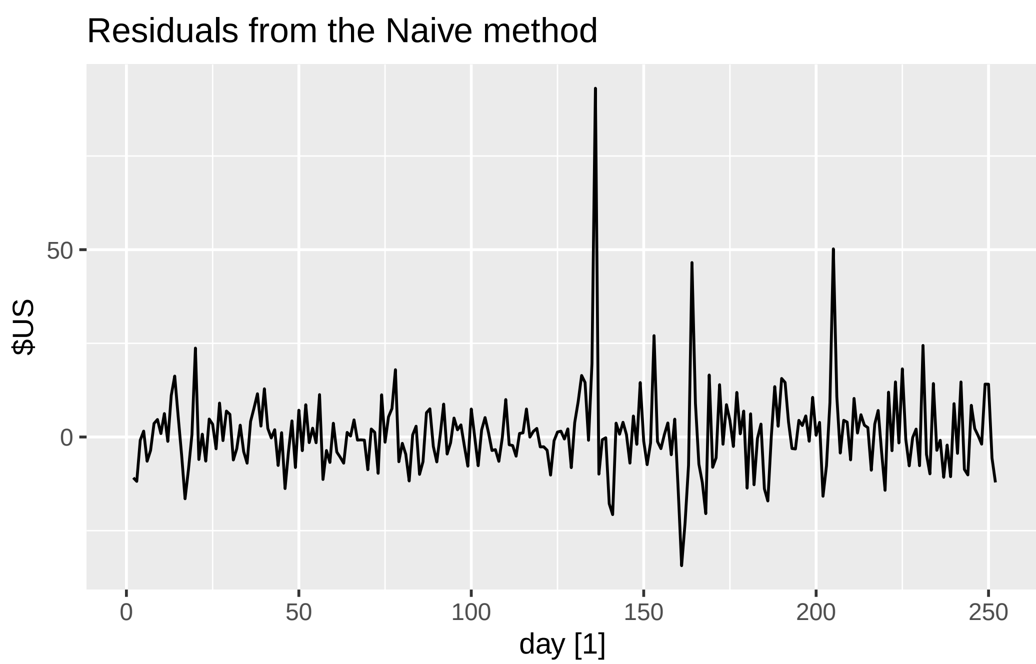
1. homoskedasticity (or exhibit no heteroskedasticity), meaning that there should be no visible pattern in how residuals are distributed over time. This is visible from the plot at the top.

2. no significant autocorrelation, this follows from the first assumption as autocorrelation would imply a pattern in the time series that should be exploited but that our model does not capture. You can check this assumption from the ACF plot on the bottom left.
3. residuals are normally distributed, you can check this from the histogram at the bottom right.

8.1 Assumptions on Residuals (continued)

We continue the discussion here more in-depth.

```r
aug <- augment(model(google_2015, NAIVE(Close)))
autoplot(aug, .innov) +
  labs(y = "$US", title = "Residuals from the Naive method")
```



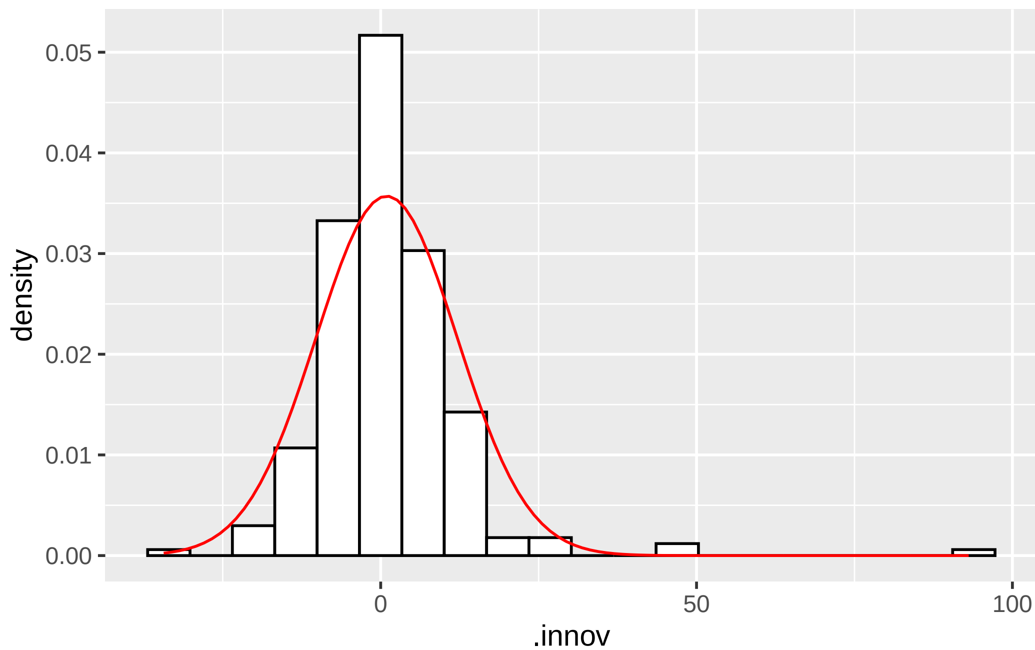
In the plot is a graphical representation of the residuals throughout the entire model we estimated using the NAIVE method.

We can also check that the distribution of the residuals closely mirrors a normal distribution with the following lines of code

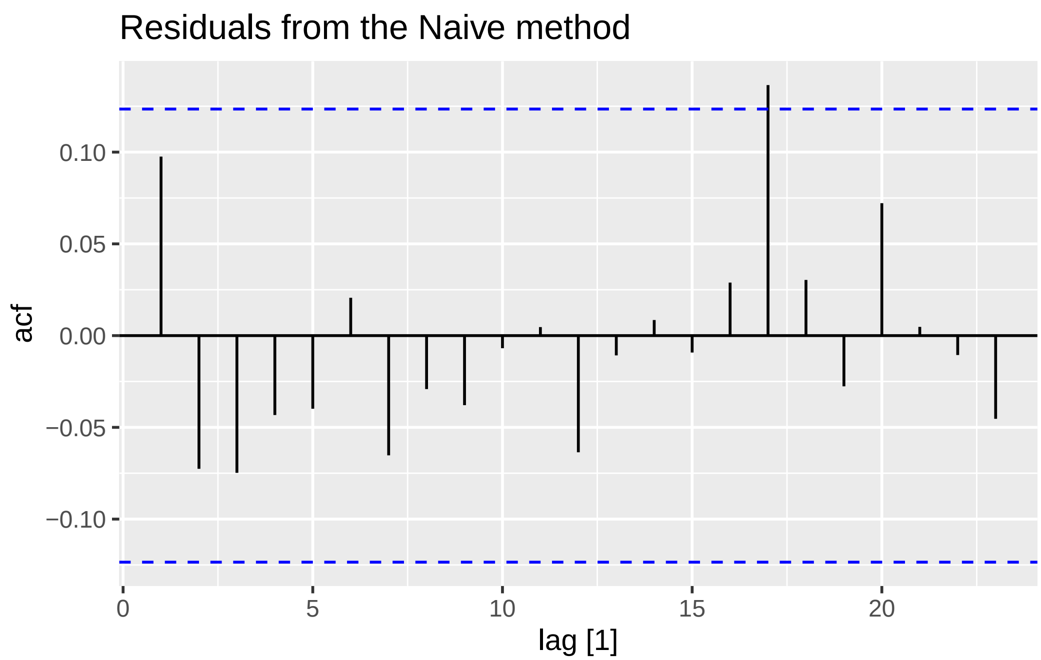
```r
p0 <- ggplot(aug, aes(x = .innov)) +
  geom_histogram(aes(y=after_stat(density)), bins = 20,                    ①
  color="black", fill="white")

p0 + stat_function(fun = dnorm, colour = "red",                            ②
  args = list(mean = mean(aug$.innov,na.rm=TRUE),
  sd = sd(aug$.innov,na.rm=TRUE)))
```

① The `after_stat()` function used as a y `aesthetic` inside the `geom_histogram()` layer of the ggplot, is used to specify that we want the `density` to be plotted on the y axis as opposed to the default behavior of the function which is to plot the `count`.

② The `stat_function()` is used to add a layer which plots a probability distribution specified by the argument `fun`. In this case, we are plotting the density of the normal distribution (`dnorm()`) in red. Since it is another layer, it is superimposed to the histogram. Since each normal distribution is uniquely identified by a `mean` and a `standard deviation`, we need to pass these values inside the `dnorm()` function using the `args` argument as a `list`.




```
autoplot(ACF(aug, .innov)) +
  labs(title = "Residuals from the Naive method")
```



By computing the ACF we can also plot it and check for the presence of significant autocorrelation of residuals at different time lags.

8.2 Portmanteau Tests

These kinds of test are used when we want to assess the level of autocorrelation in a time series across a number k of lags, where $k \in \mathbb{N}$ (i.e., notation for saying that k is a natural number). There are two main tests that your book discusses.

8.2.1 Box-Pierce Test

```
features(aug, .innov, box_pierce, lag = 10)
```

```
# A tibble: 1 x 4
  Symbol .model       bp_stat bp_pvalue
  <chr>  <chr>          <dbl>     <dbl>
1 GOOG   NAIVE(Close)    7.74     0.654
```

With this we compute the Box-Pierce statistic on the data and obtain its related p-value. Generally a p-value greater than 0.1 will be a good way of assessing that the residuals are randomly distributed and that there is no significant pattern arising from their actual distribution.

8.2.2 Ljung-Box Test

```r
features(aug, .innov, ljung_box, lag = 10)
```

```
# A tibble: 1 x 4
  Symbol .model       lb_stat lb_pvalue
  <chr>  <chr>          <dbl>     <dbl>
1 GOOG   NAIVE(Close)    7.91     0.637
```

Another approach is using the Ljung-Box test. It has been shown that the two tests yield comparable results, however, this test has been shown to be more precise overall so it is recommended that you use this one from now on. Remember that to use the `features()` function you need to specify the augmented model fit (aug in the example), select the .innov column containing the innovation residuals, and then specify that you want to compute the ljung_box statistic with a lag of 10 (due to the data being non-seasonal in this case).

> **! Important**
>
> The book suggests using ℓ=10 (i.e., `lag = 10`) for non-seasonal data and ℓ=2m for seasonal data, where m is the period of seasonality. However, the test is not good when ℓ is large, so if these values are larger than T/5, then use ℓ=T/5. T in this case refers to the total number of observations that you are using to carry out the test (i.e., number of rows in your dataset).

8.2.2.1 More examples

We can now use this method to check if the residuals produced by other models present patterns or are autocorrelated.

```r
fit <- model(google_2015, RW(Close ~ drift()))
tidy(fit)                                                    (1)
```

(1) remember that the estimate obtained using the drift method is $\frac{(y_T - y_1)}{(T-1)}$ = (759-522)/251 = 0.9439931

```
# A tibble: 1 x 7
  Symbol .model              term  estimate std.error statistic p.value
  <chr>  <chr>               <chr>    <dbl>     <dbl>     <dbl>   <dbl>
1 GOOG   RW(Close ~ drift()) b        0.944     0.705      1.34   0.182
```

```r
features(augment(fit), .innov, ljung_box, lag=10)
```

```
# A tibble: 1 x 4
  Symbol .model              lb_stat lb_pvalue
  <chr>  <chr>                 <dbl>     <dbl>
1 GOOG   RW(Close ~ drift())    7.91     0.637
```

then we compute the Ljung Box statistic and see that the p-value is > .1 and therefore we can assume that residuals are randomly distributed (no pattern)

```r
hilo(forecast(model(google_2015, NAIVE(Close)), h = 10))
```

```
# A tsibble: 10 x 7 [1]
# Key:       Symbol, .model [1]
   Symbol .model         day
   <chr>  <chr>        <dbl>
 1 GOOG   NAIVE(Close)   253
 2 GOOG   NAIVE(Close)   254
 3 GOOG   NAIVE(Close)   255
 4 GOOG   NAIVE(Close)   256
 5 GOOG   NAIVE(Close)   257
 6 GOOG   NAIVE(Close)   258
 7 GOOG   NAIVE(Close)   259
 8 GOOG   NAIVE(Close)   260
 9 GOOG   NAIVE(Close)   261
10 GOOG   NAIVE(Close)   262
# i 4 more variables: Close <dist>, .mean <dbl>, `80%` <hilo>, `95%` <hilo>
```

Finally we use the `hilo()` function to extract confidence (prediction) intervals for each prediction obtained from a NAIVE model.

8.3 Bootstrapped residuals

We can use our model to generate more observations. These observations would be generated based on what has been observed in previous periods, but they would still vary. In this case, we are building a simulation to test our model on multiple different future possibilities. The method that is used to obtain new random observations starting from a sample is called **bootstrap**.

To generate these bootstrapped residuals we can use the `generate()` function in R. Alternatively, we could also just set the `bootstrap` argument in the `forecast()` to `TRUE` and specify how many `times` we want to run the bootstrap simulation.

```r
(fore <- forecast(model(google_2015, NAIVE(Close)), h = 10,
        bootstrap = TRUE, times = 1000))                    ①
```

① This specifies that the bootstrapping process will perform 1000 simulations. Each simulation generates a possible future path for the time series. These simulations are then aggregated to produce the final forecast distribution.

```
# A fable: 10 x 5 [1]
# Key:     Symbol, .model [1]
   Symbol .model         day        Close .mean
   <chr>  <chr>        <dbl>       <dist> <dbl>
 1 GOOG   NAIVE(Close)   253 sample[1000]  759.
 2 GOOG   NAIVE(Close)   254 sample[1000]  759.
 3 GOOG   NAIVE(Close)   255 sample[1000]  759.
```

```
 4 GOOG   NAIVE(Close)   256 sample[1000]  759.
 5 GOOG   NAIVE(Close)   257 sample[1000]  759.
 6 GOOG   NAIVE(Close)   258 sample[1000]  760.
 7 GOOG   NAIVE(Close)   259 sample[1000]  759.
 8 GOOG   NAIVE(Close)   260 sample[1000]  760.
 9 GOOG   NAIVE(Close)   261 sample[1000]  760.
10 GOOG   NAIVE(Close)   262 sample[1000]  760.
```

Finally we take a look at the final forecast distribution along with the prediction intervals

```
hilo(fore)
```

```
# A tsibble: 10 x 7 [1]
# Key:       Symbol, .model [1]
   Symbol .model         day        Close .mean                    `80%`
   <chr>  <chr>        <dbl>       <dist> <dbl>                   <hilo>
 1 GOOG   NAIVE(Close)   253 sample[1000]  759. [747.8160, 771.8960]80
 2 GOOG   NAIVE(Close)   254 sample[1000]  759. [742.1871, 774.8240]80
 3 GOOG   NAIVE(Close)   255 sample[1000]  759. [738.6650, 779.4271]80
 4 GOOG   NAIVE(Close)   256 sample[1000]  759. [735.5861, 781.8397]80
 5 GOOG   NAIVE(Close)   257 sample[1000]  759. [731.5328, 787.0748]80
 6 GOOG   NAIVE(Close)   258 sample[1000]  760. [730.2786, 791.5545]80
 7 GOOG   NAIVE(Close)   259 sample[1000]  759. [727.1139, 795.2611]80
 8 GOOG   NAIVE(Close)   260 sample[1000]  760. [724.4122, 797.9961]80
 9 GOOG   NAIVE(Close)   261 sample[1000]  760. [721.9938, 801.4880]80
10 GOOG   NAIVE(Close)   262 sample[1000]  760. [719.6038, 804.1328]80
# i 1 more variable: `95%` <hilo>
```

9 Final lines of code

9.1 More examples on model fitting and diagnosis of residuals

```
us_retail_employment <- filter(us_employment, year(Month) >= 1990,
                        Title == "Retail Trade")                            ①

US_model_0 <- model(us_retail_employment,
              STL(Employed ~ trend(window = 7), robust = TRUE))             ②
```

① filter the dataset
② specify the model formula using an STL-based decomposition using only the trend component

```
US_model_1 <- select(components(US_model_0), -.model)                       ①

(US_fore <- forecast(model(US_model_1, NAIVE(season_adjust))))              ②
```

① get rid of the `.model` column
② produce forecasts using the NAIVE method on the seasonally adjusted series

```
# A fable: 24 x 5 [1M]
# Key:     Series_ID, .model [1]
   Series_ID     .model                   Month
   <chr>         <chr>                    <mth>
 1 CEU4200000001 NAIVE(season_adjust) 2019 Oct
 2 CEU4200000001 NAIVE(season_adjust) 2019 Nov
 3 CEU4200000001 NAIVE(season_adjust) 2019 Dec
 4 CEU4200000001 NAIVE(season_adjust) 2020 Jan
 5 CEU4200000001 NAIVE(season_adjust) 2020 Feb
 6 CEU4200000001 NAIVE(season_adjust) 2020 Mar
 7 CEU4200000001 NAIVE(season_adjust) 2020 Apr
 8 CEU4200000001 NAIVE(season_adjust) 2020 May
 9 CEU4200000001 NAIVE(season_adjust) 2020 Jun
10 CEU4200000001 NAIVE(season_adjust) 2020 Jul
# i 14 more rows
# i 2 more variables: season_adjust <dist>, .mean <dbl>
```

Finally, we plot the results from our model which is based on seasonally adjusted data.

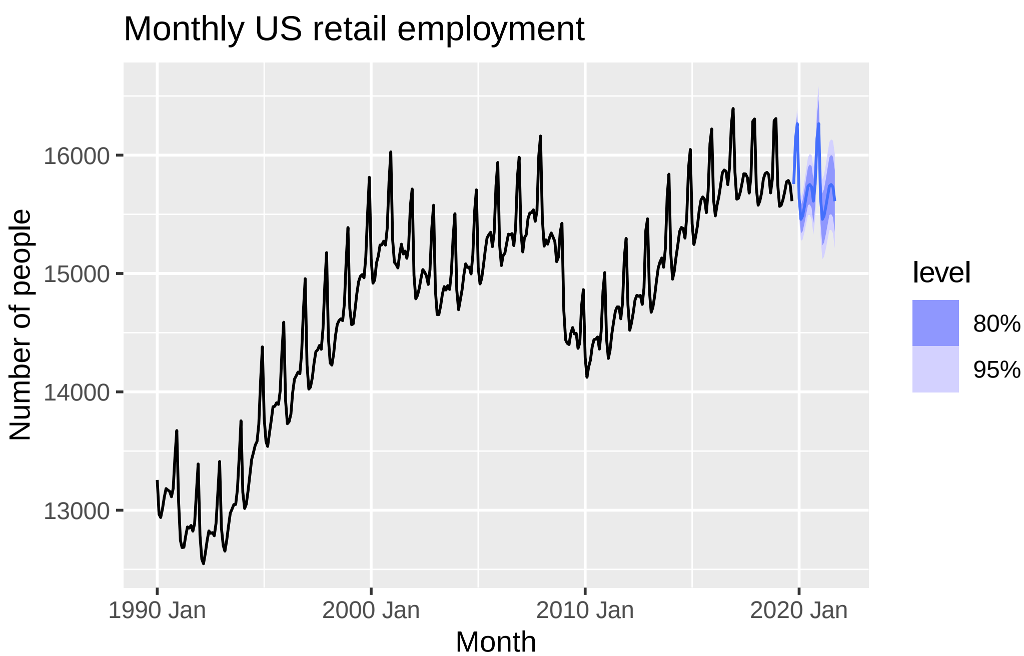
```r
autoplot(US_fore, US_model_1) +
  labs(y = "Number of people", title = "US retail employment")
```

Another similar example follows below.

```r
fit_dcmp <- model(us_retail_employment,
    stlf = decomposition_model(STL(Employed ~ trend(window = 7),
          robust = TRUE), NAIVE(season_adjust)))
```

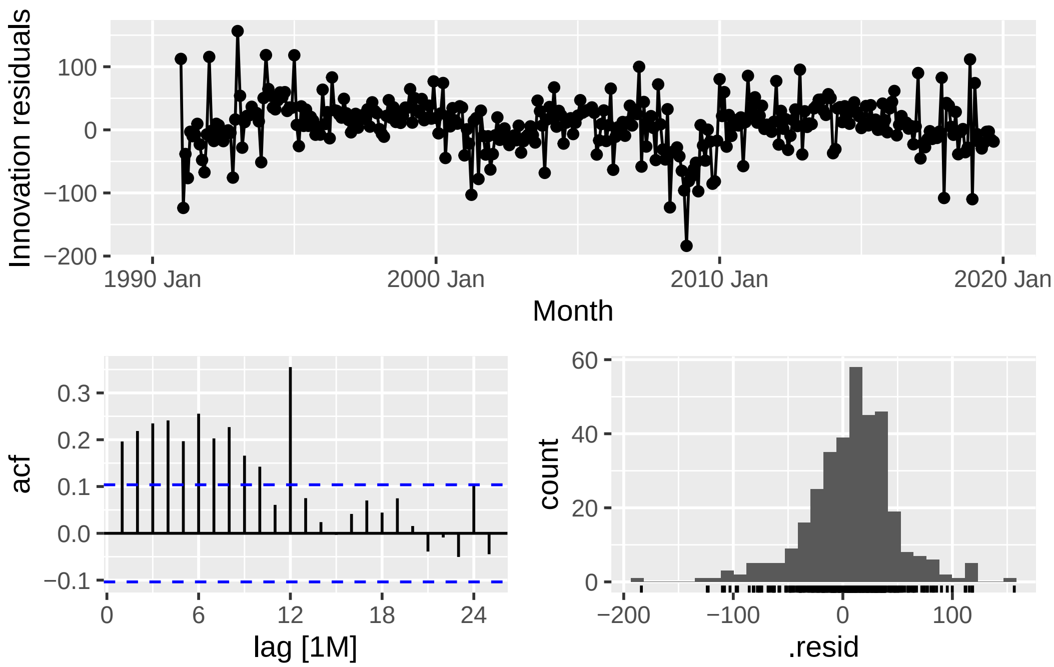
We start from the decomposing the model using an STL decomposition as before and then we fit a NAIVE model to the seasonally adjusted time series. We then plot the results.

```r
autoplot(forecast(fit_dcmp), us_retail_employment) +
  labs(y = "Number of people", title = "Monthly US retail employment")
```



We can also plot the residuals of the model by using:

```r
gg_tsresiduals(fit_dcmp)
```



From here we notice some patterns in how those are distributed and the autocorrelation at different time lags. We now want to compute the mean of the innovation residuals.

```
features(augment(fit_dcmp), .innov, list(avg = ~ mean(.,na.rm=TRUE)))
```

```
# A tibble: 1 x 3
  Series_ID     .model   avg
  <chr>         <chr>  <dbl>
1 CEU4200000001 stlf    7.84
```

A simpler way to get the mean of the residuals **when you only estimate one model** is shown below.

```
mean(augment(fit_dcmp)$.innov, na.rm=TRUE)
```

```
[1] 7.836419
```

9.2 Subsetting

Subsetting refers to extracting only some rows in a dataset. Subsetting is usually a synonym of filtering the data and this is also shown in code. Here is an example:

```
filter(aus_production, year(Quarter) >= 1995)   ①
```

① only get the rows where the year of the observation is greater than or equal to 1995

```
# A tsibble: 62 x 7 [1Q]
   Quarter  Beer Tobacco Bricks Cement Electricity   Gas
     <qtr> <dbl>   <dbl>  <dbl>  <dbl>       <dbl> <dbl>
 1 1995 Q1   426    4714    430   1626       41768   131
 2 1995 Q2   408    3939    457   1703       43686   167
 3 1995 Q3   416    6137    417   1733       46022   181
 4 1995 Q4   520    4739    370   1545       42800   145
 5 1996 Q1   409    4275    310   1526       43661   133
 6 1996 Q2   398    5239    358   1593       44707   162
 7 1996 Q3   398    6293    379   1706       46326   184
 8 1996 Q4   507    5575    369   1699       43346   146
 9 1997 Q1   432    4802    330   1511       43938   135
10 1997 Q2   398    5523    390   1785       45828   171
# i 52 more rows
```

In case you do not have a specific filtering condition to use, you can use slice to get a subset of your data.

```
slice(aus_production, n()-19:0)                                    (1)
```

(1) last 20 observations obtained using the `n()` function which returns the total number of rows (n) in the aus_production dataset. Then the slice only selects the rows from n-19 to the end.

```
# A tsibble: 20 x 7 [1Q]
   Quarter  Beer Tobacco Bricks Cement Electricity   Gas
     <qtr> <dbl>   <dbl>  <dbl>  <dbl>       <dbl> <dbl>
 1 2005 Q3   408      NA     NA   2340       56043   221
 2 2005 Q4   482      NA     NA   2265       54992   180
 3 2006 Q1   438      NA     NA   2027       57112   171
 4 2006 Q2   386      NA     NA   2278       57157   224
 5 2006 Q3   405      NA     NA   2427       58400   233
 6 2006 Q4   491      NA     NA   2451       56249   192
 7 2007 Q1   427      NA     NA   2140       56244   187
 8 2007 Q2   383      NA     NA   2362       55036   234
 9 2007 Q3   394      NA     NA   2536       59806   245
10 2007 Q4   473      NA     NA   2562       56411   205
11 2008 Q1   420      NA     NA   2183       59118   194
12 2008 Q2   390      NA     NA   2558       56660   229
13 2008 Q3   410      NA     NA   2612       64067   249
14 2008 Q4   488      NA     NA   2373       59045   203
15 2009 Q1   415      NA     NA   1963       58368   196
16 2009 Q2   398      NA     NA   2160       57471   238
17 2009 Q3   419      NA     NA   2325       58394   252
18 2009 Q4   488      NA     NA   2273       57336   210
19 2010 Q1   414      NA     NA   1904       58309   205
20 2010 Q2   374      NA     NA   2401       58041   236
```

```r
slice(group_by(aus_retail, State, Industry), 1:12) # working with groups
```

```
# A tsibble: 1,824 x 5 [1M]
# Key:       State, Industry [152]
# Groups:    State, Industry [152]
   State                        Industry           `Series ID`    Month Turnover
   <chr>                        <chr>              <chr>          <mth>    <dbl>
 1 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Apr      4.4
 2 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 May      3.4
 3 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Jun      3.6
 4 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Jul      4
 5 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Aug      3.6
 6 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Sep      4.2
 7 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Oct      4.8
 8 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Nov      5.4
 9 Australian Capital Territory Cafes, restaurant~ A3349849A   1982 Dec      6.9
10 Australian Capital Territory Cafes, restaurant~ A3349849A   1983 Jan      3.8
# i 1,814 more rows
```

9.3 Forecast errors

```r
recent_production <- filter(aus_production, year(Quarter) >= 1992)   # (1)

beer_train <- filter(recent_production, year(Quarter) <= 2007)   # (2)
```

(1) get rows where year is after 1992 (included)
(2) train / test split

```r
beer_fit <- model(beer_train, Mean = MEAN(Beer),
    Naive = NAIVE(Beer),
    'Seasonal naive' = SNAIVE(Beer),
    Drift = RW(Beer ~ drift()))
```

Fit three models to our training dataset. Finally, produce forecasts using our model.

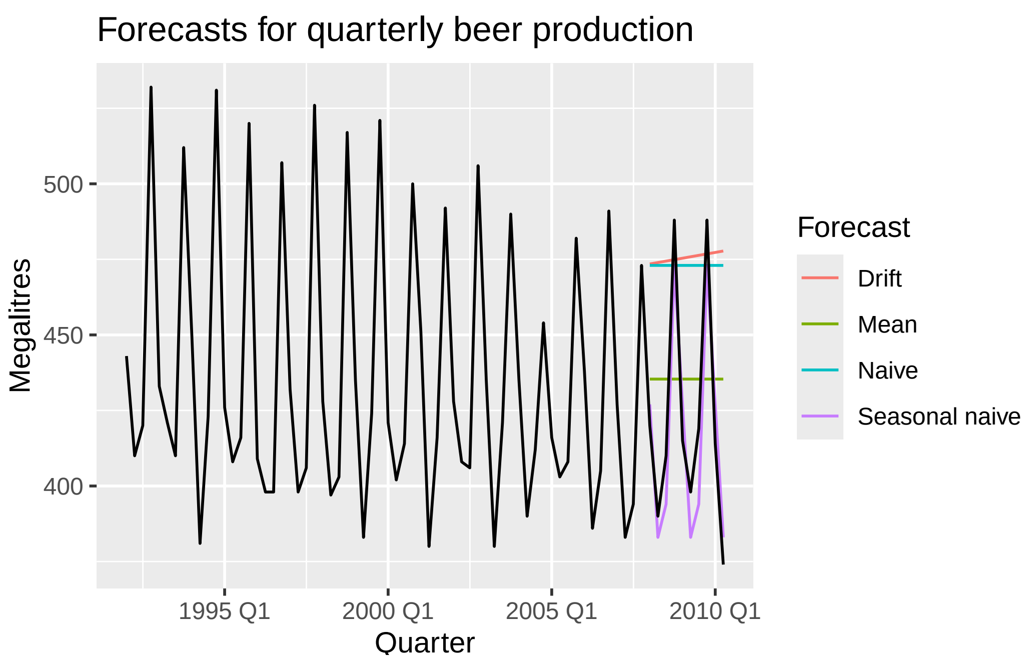
```r
(beer_fc <- forecast(beer_fit, h = 10))
```

```
# A fable: 40 x 4 [1Q]
# Key:     .model [4]
   .model Quarter
   <chr>    <qtr>
 1 Mean   2008 Q1
 2 Mean   2008 Q2
 3 Mean   2008 Q3
```

```
 4 Mean   2008 Q4
 5 Mean   2009 Q1
 6 Mean   2009 Q2
 7 Mean   2009 Q3
 8 Mean   2009 Q4
 9 Mean   2010 Q1
10 Mean   2010 Q2
# i 30 more rows
# i 2 more variables: Beer <dist>, .mean <dbl>
```

And plot the forecasts without plotting the prediction intervals (i.e., `level = NULL`).

```
autoplot(beer_fc, recent_production, level = NULL) +
  labs(y = "Megalitres", title = "Forecasts for quarterly beer production") +
  guides(colour = guide_legend(title = "Forecast"))
```



Finally, we evaluate the performance of our model by producing an accuracy table.

```
accTable <- accuracy(beer_fc, recent_production)
select(accTable,.model,RMSE,MAE,MAPE)
```

```
# A tibble: 4 x 4
  .model          RMSE   MAE  MAPE
  <chr>          <dbl> <dbl> <dbl>
1 Drift           64.9  58.9 14.6
2 Mean            38.4  34.8  8.28
3 Naive           62.7  57.4 14.2
4 Seasonal naive  14.3  13.4  3.17
```

This allows us to get the main accuracy measures for the different models fit to the data. From the table we see that the Seasonal naive model scores the lowest on the error metrics, and therefore is the best model among those proposed.