FORECASTING AND PREDICTIVE MODELING

LECTURE 7

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 6

- Time series decomposition
- Time series features

Today

- Time series features
- The forecasting workflow

Time series features

- Remember, a feature is a numerical value which describes some property of a time series.
- We have talked about simple features (mean, maximum, etc.) and autocorrelation - related features.

Features based on the STL decomposition I

- Remember, the STL decomposition means to split the time series into three components: trend, seasonal and remainder. STL assumes that there is an additive decomposition:

$$y_t = T_t + S_t + R_t$$

Features based on the STL decomposition II

- We can measure how strong the trend in the data is, using the following formula:

$$F_T = max(0, 1 - \frac{Var(R_t)}{Var(T_t + R_t)})$$

- where *Var* represents the variance of the data. $T_t + R_t$ represents the seasonally adjusted data (where the seasonal component is removed). If the data is strongly trended, the variation of the seasonally adjusted data should be a lot larger than the variation in the remainder (and F_T is close to 1), while in case of data with little or no trend the two variations are similar.

Features based on the STL decomposition III

- We can define the strength of the seasonality in the same manner, but this time using the detrended data (the data from which the trend component is removed)

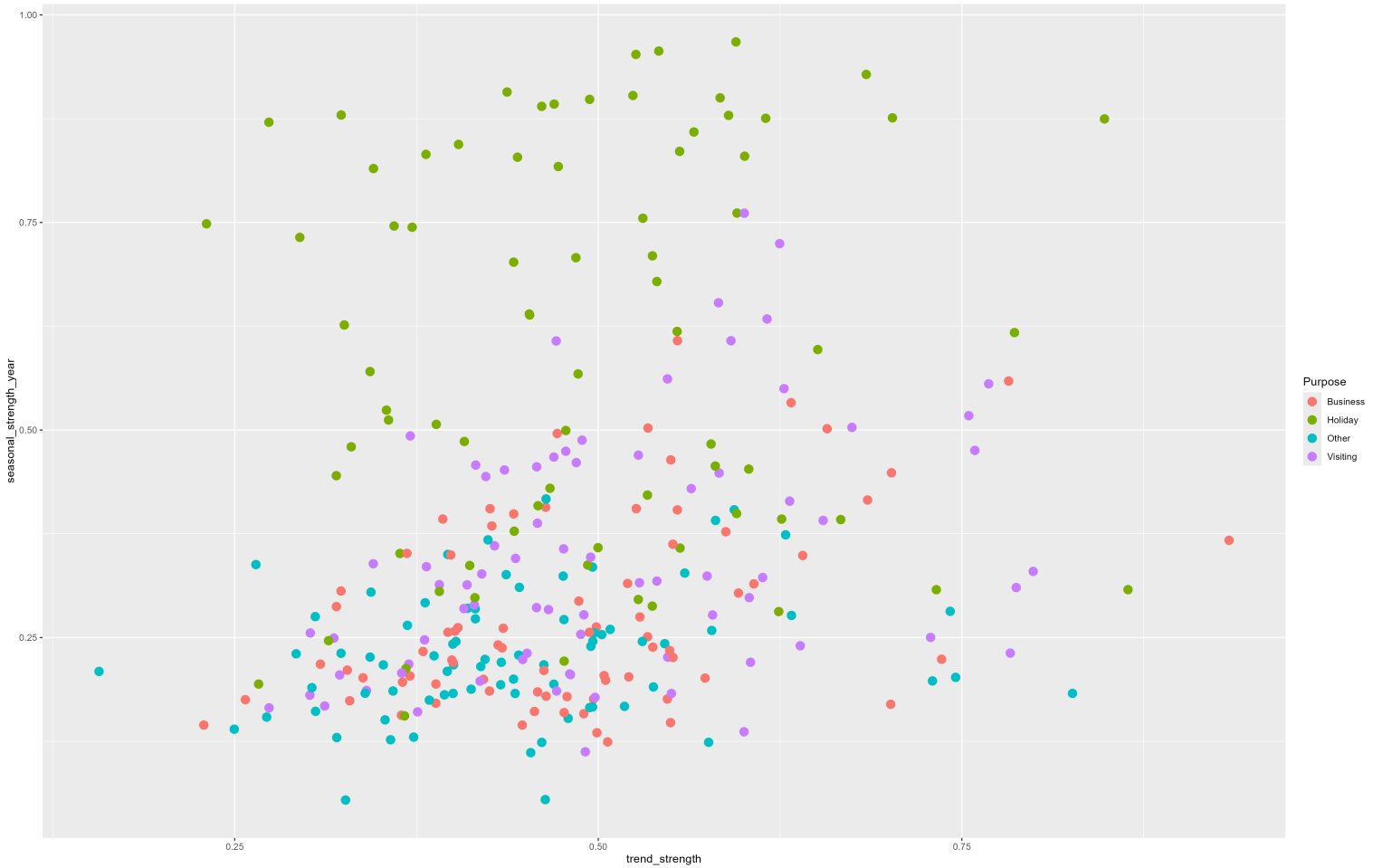
$$F_S = max(0, 1 - \frac{Var(R_t)}{Var(S_t + R_t)})$$

- A value of F_S close to 0 means no seasonality.
- F_T and F_S can be useful, when we have a lot of time series and we need to find the ones with the most trend or the most seasonality. The function *feat_stl* will compute these two features, together with some other features.

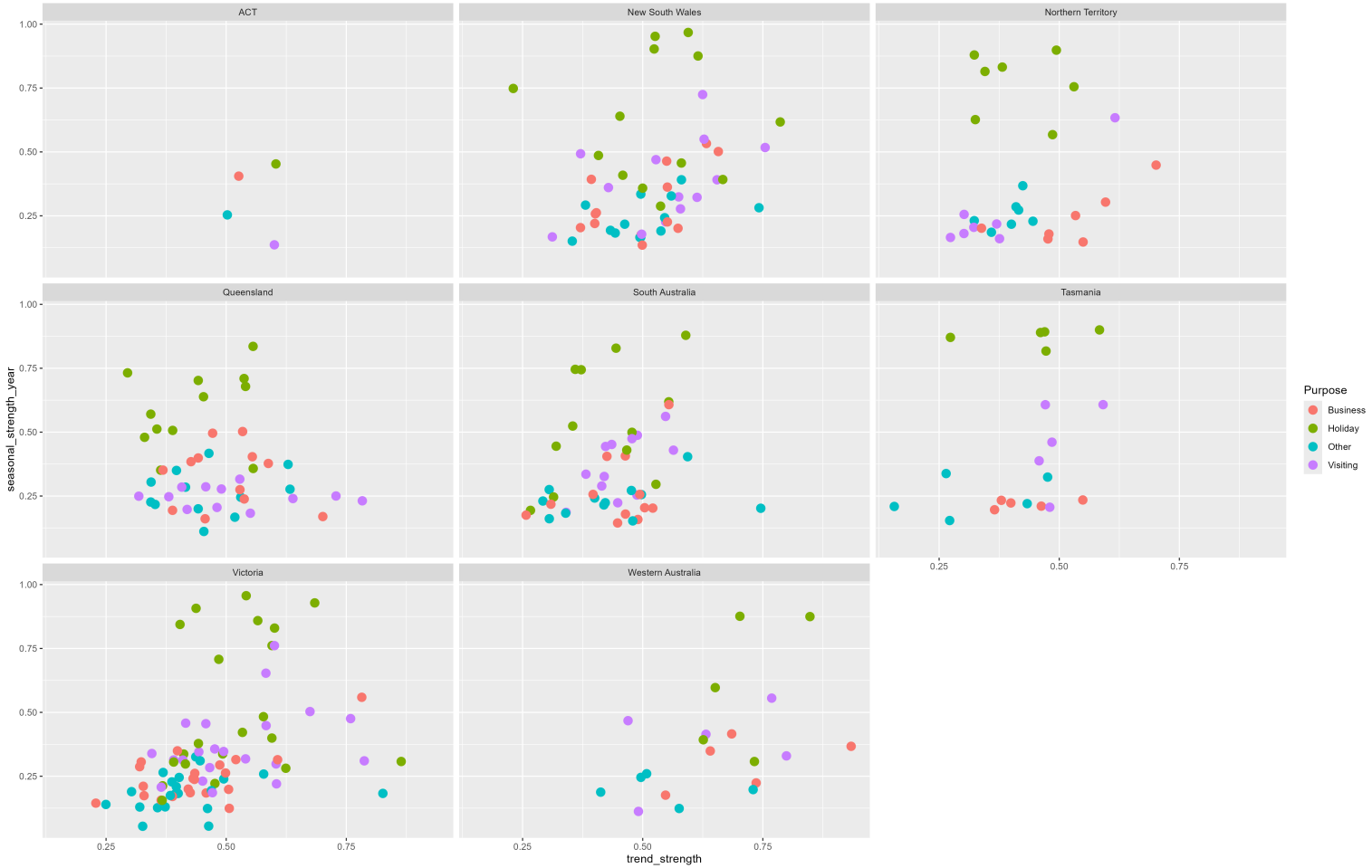
```
tourism |>
  features(Trips, feat_stl)

tourism |>
  features(Trips, feat_stl) |> arrange(desc(trend_strength))
```

- We can check how trended and seasonal are these time series by plotting trend strength against seasonal strength: on the entire time series or split by states

```
tourism |>
 features(Trips, feat_stl) |>
 ggplot(aes(x = trend_strength, y = seasonal_strength_year, col = Purpose)) +
 geom_point()

tourism |>
 features(Trips, feat_stl) |>
 ggplot(aes(x = trend_strength, y = seasonal_strength_year, col = Purpose)) +
 geom_point() +
 facet_wrap(vars(State))
```

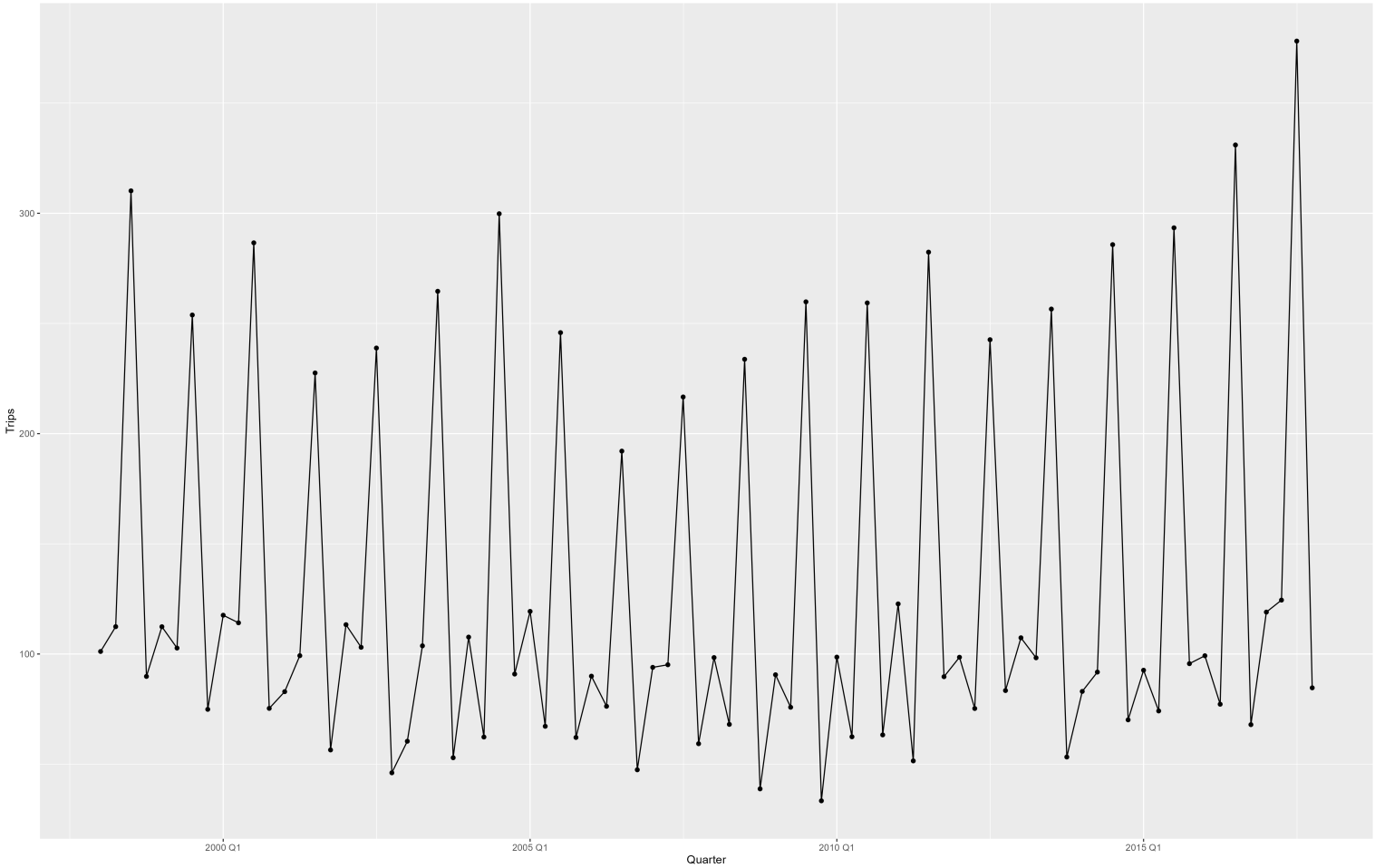


- What do you observe on the plot?


ACT
New South Wales
Northern Territory
Queensland
South Australia
Tasmania
Victoria
Western Australia
seasonal_strength_year
trend_strength
1.00
0.75
0.50
0.25
Purpose
Business
Holiday
Other
Visiting

- Find the series with the strongest seasonal component

```
max_strength <- tourism |>
  features(Trips, feat_stl) |>
  filter(seasonal_strength_year == max(seasonal_strength_year))
max_strength

max_strength |> left_join(tourism, by = c("State", "Region", "Purpose"),
      multiple = "all") |>
  ggplot(mapping = aes(x = Quarter, y = Trips)) +
  geom_line() +
  geom_point()

max_strength |> left_join(tourism, by = c("State", "Region", "Purpose"),
      multiple = "all") |>
  ggplot(mapping = aes(x = Quarter, y = Trips)) +
  geom_line() +
  geom_point()  +
  facet_grid(vars(State, Region, Purpose))
```


Trips
300
200
100
2000 Q1
2005 Q1
2010 Q1
2015 Q1
Quarter

Other STL features

- *seasonal_peak_year* - which month or quarter contains the largest values (the peak of the season). For example, in tourism data, if the peak is Q1, people are travelling more in winter (except if we are talking about Australia, because Q1 is summer there).
- *seasons_through_year* - which month or quarter contains the lowest values
- *spikiness* - measures the prevalence of spikes in the remainder
- *linearity* - how linear the trend component is
- *curvature* - measures the curvature of the trend component
- *stl_e_acf1* - the first autocorrelation value of the remainder
- *stl_e_acf10* - the sum of the squares of the first 10 autocorrelation features

- All features can be computed using the *feature_set* function, with the *pkgs* parameter set to "feasts". This will give us a total of 48 new features for each time series.

```
tourism_features <- tourism |>
  features(Trips, feature_set(pkgs = "feasts"))
view(tourism_features)
```

A few other features I

- A few other features which might be important:
  - partial autocorrelation related features (autocorrelation at lag k after removing the effect of the previous observations)
  - feat_spectral - Shannon spectral entropy which shows how easy is to forecast a time series. If it has a lot of trend and season, it is easier to forecast (entropy close to 0).
  - box_pierce and lung_box statistics for checking if a time series is white noise.
  - unitroot_kpss
  - unitroot_ndiffs and unit_nsdiffs
  - var_tiled_mean - the variance in the tiled means (means of non-overlapping sequences). The length of the tile is either 10 or the seasonal length.

A few other features II

- shift_level_max - find where there is the maximum shift between two consecutive sliding windows. Useful to find sudden junmps in data.
- n_crossing_points - computes the number of times the series crosses the median.
- guerrero

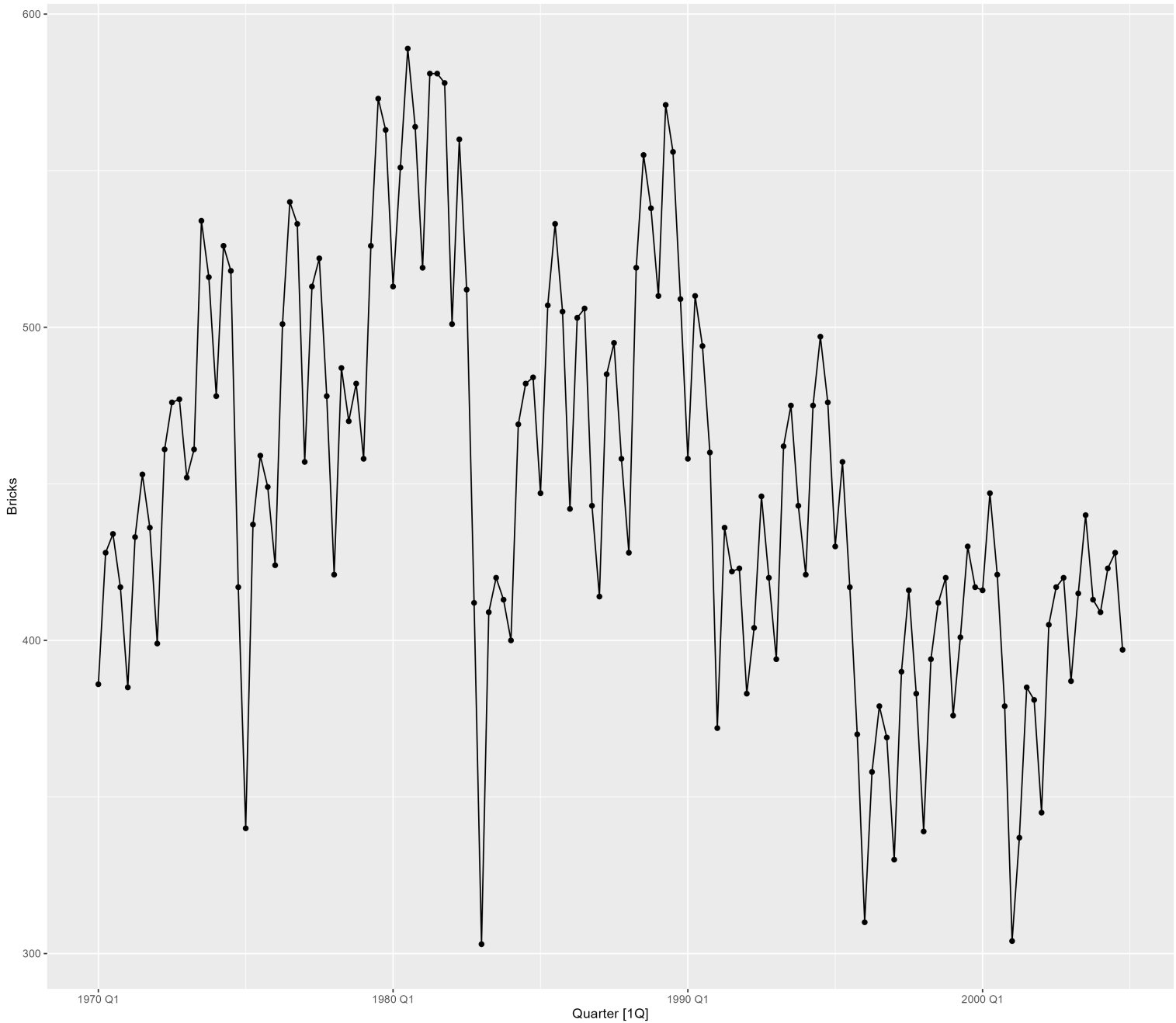
The forecasting workflow

- In general a forecasting workflow is organized in the following way:
  1. Data preparation (data cleaning/tidying)
     - Load, filter, transform, identify missing values, outliers, etc.
  2. Data visualization
  3. Define/specify a model
  4. Train/estimate the model $\Rightarrow$ mable (model table)
  5. Check performance
  6. Forecast $\Rightarrow$ fable (forecast table)
- Steps 2 - 5 can be repeated several times.

Simple forecasting methods

- They are simple methods, not really used for actual forecasting, rather as baseline to evaluate other methods.
- We will talk about 4 simple forecasting methods:
  - mean
  - naive method
  - seasonal naive method
  - drift method
- We will look at them using as example the Australian clay brick production between 1970 and 2004.

```
bricksData <- aus_production |>
  filter(year(Quarter) >= 1970 & year(Quarter) <= 2004) |>
  select(Bricks)
autoplot(bricksData) + geom_point()
```


600
500
400
300
Bricks
1970 Q1
1980 Q1
1990 Q1
2000 Q1
Quarter [1Q]

The mean method I

- In the *mean* method, the forecast for all future values is simply the mean of all historical observations.

$$\hat{y}_{T+h|T} = \frac{y_1 + y_2 + \cdots + y_T}{T}$$

- **Obs:**
  - $\hat{y}$ denotes the estimated (forecast) values of y
  - $\hat{y}_{T+h|T}$ means the forecast/estimated value of y for the time stamp $T + h$, given all the observations $y_1, ..., y_T$.

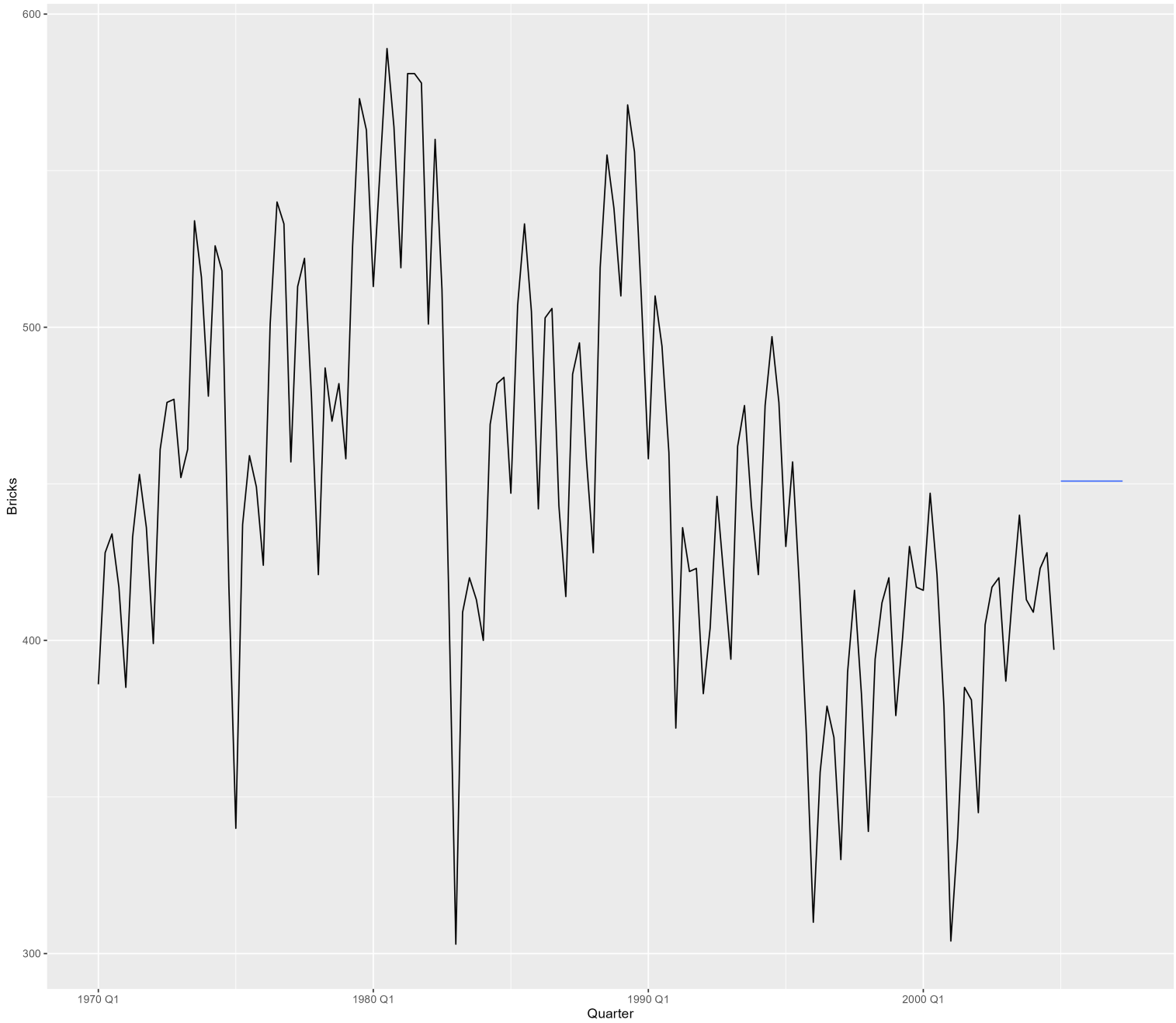
- In R, it can be computed with the *MEAN* function, passed as argument to the *model* function. Calling the *tidy* function for the model, will give us details about the model parameters.

The mean method II

```
bricksData |>
    model(MEAN(Bricks)) -> brickModel
tidy(brickModel)
```

```
  .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
```

```
brickModel |>
    forecast(h = 10) -> brickForecast
brickForecast
brickForecast |>
    autoplot(level = NULL) +
    autolayer(bricksData, Bricks)
```


600
500
400
300
Bricks
1970 Q1
1980 Q1
1990 Q1
2000 Q1
Quarter

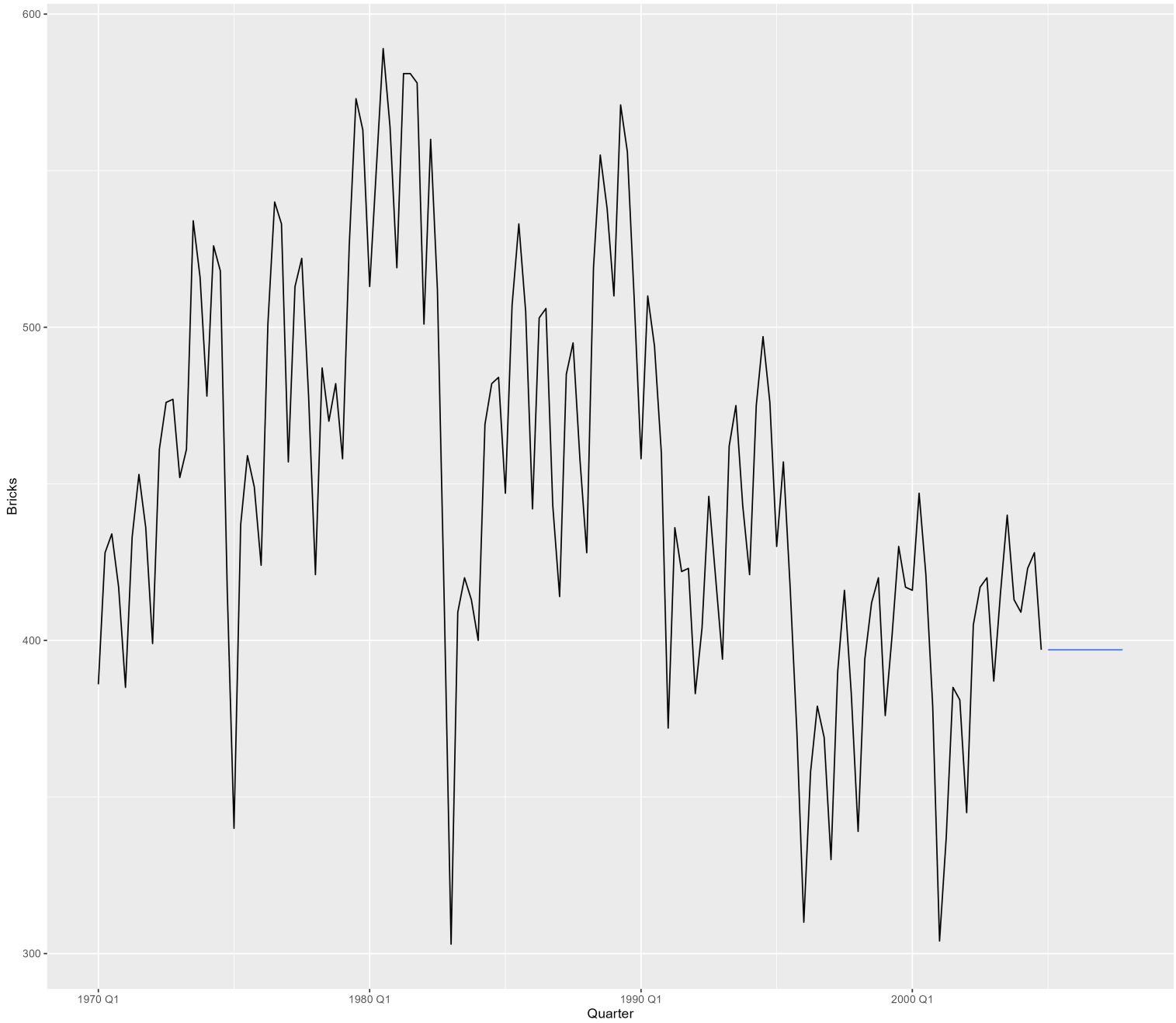
The naive method I

- In the *naive* method, the forecast of all future values is simply the value of the last observation.

$$\hat{y}_{T+h|T} = y_T$$

- In R, it can be computed with the *NAIVE* function, passed as argument to the *model* function.

```
bricksData |>
    model(NAIVE(Bricks)) -> brickNaiveModel

brickNaiveModel |>
    forecast(h = 10) -> brickNaiveForecast
brickNaiveForecast
brickNaiveForecast |>
    autoplot(level = NULL) +
    autolayer(bricksData, Bricks)
```


600
500
400
300
Bricks
1970 Q1
1980 Q1
1990 Q1
2000 Q1
Quarter

- Remember, we have talked about the random walk process (stationary time series where all the autocorrelations are white noise).
- For such time series the naive model is actually the optional forcasting method.
- This is why, in R, we can use the **RW** function instead of **NAIVE**.

The seasonal naive method I

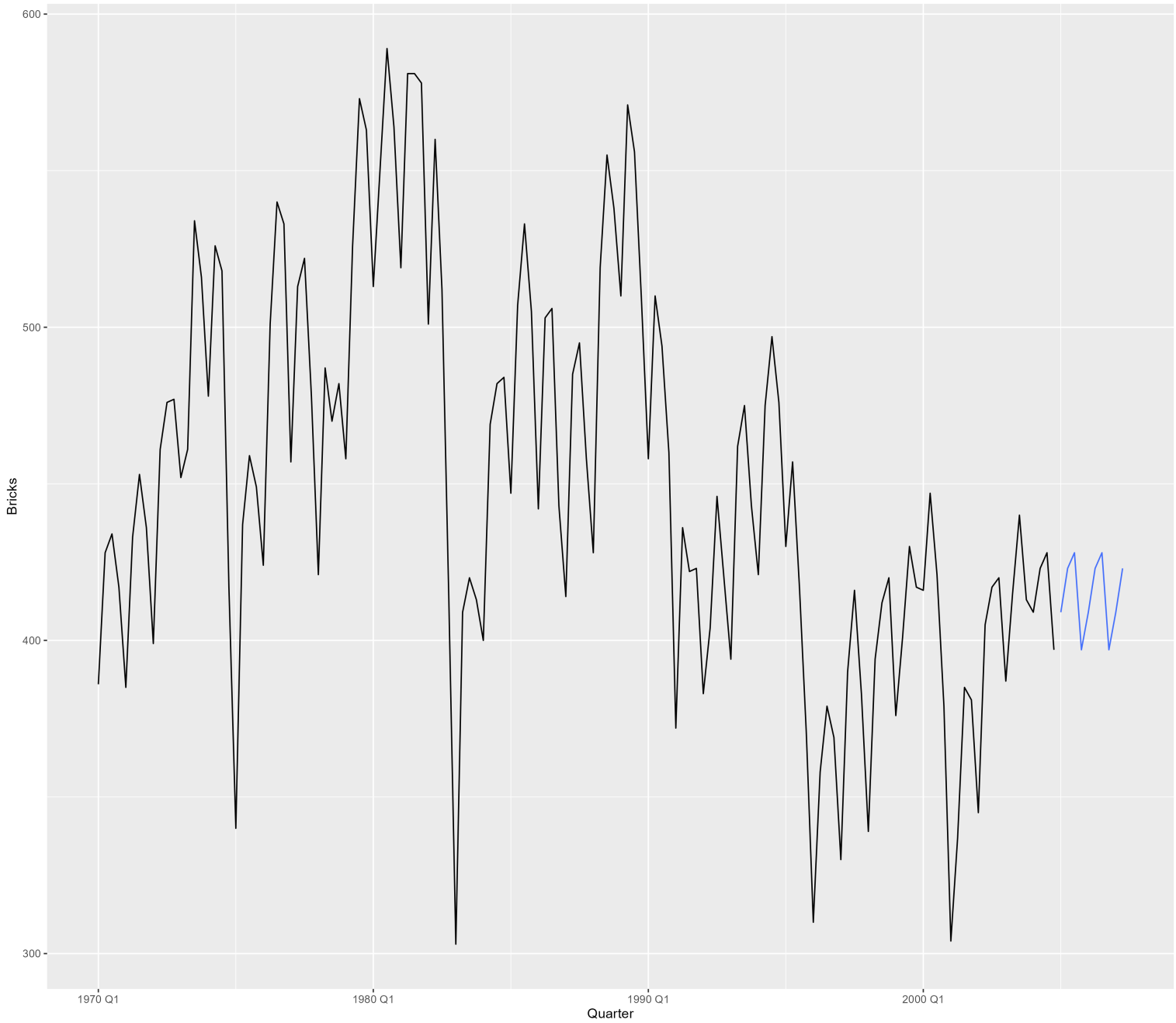
- This method can be used in case of seasonal data. Like in case of the naive method, we use as forecast the last observation, but from the same season (so we are repeating the last season over and over again).

$$\hat{y}_{T+h|T} = y_{T+h-m*(k+1)}$$

- where m is the seasonal period (ex. 4 in case of quarterly data) and k is the integer part of $\frac{h-1}{m}$ (the number of complete seasons in the forecast period, before $T + h$)
- In R it can be computed with the *SNAIVE* function, which can take as parameter the seasonal lag.

The seasonal naive method II

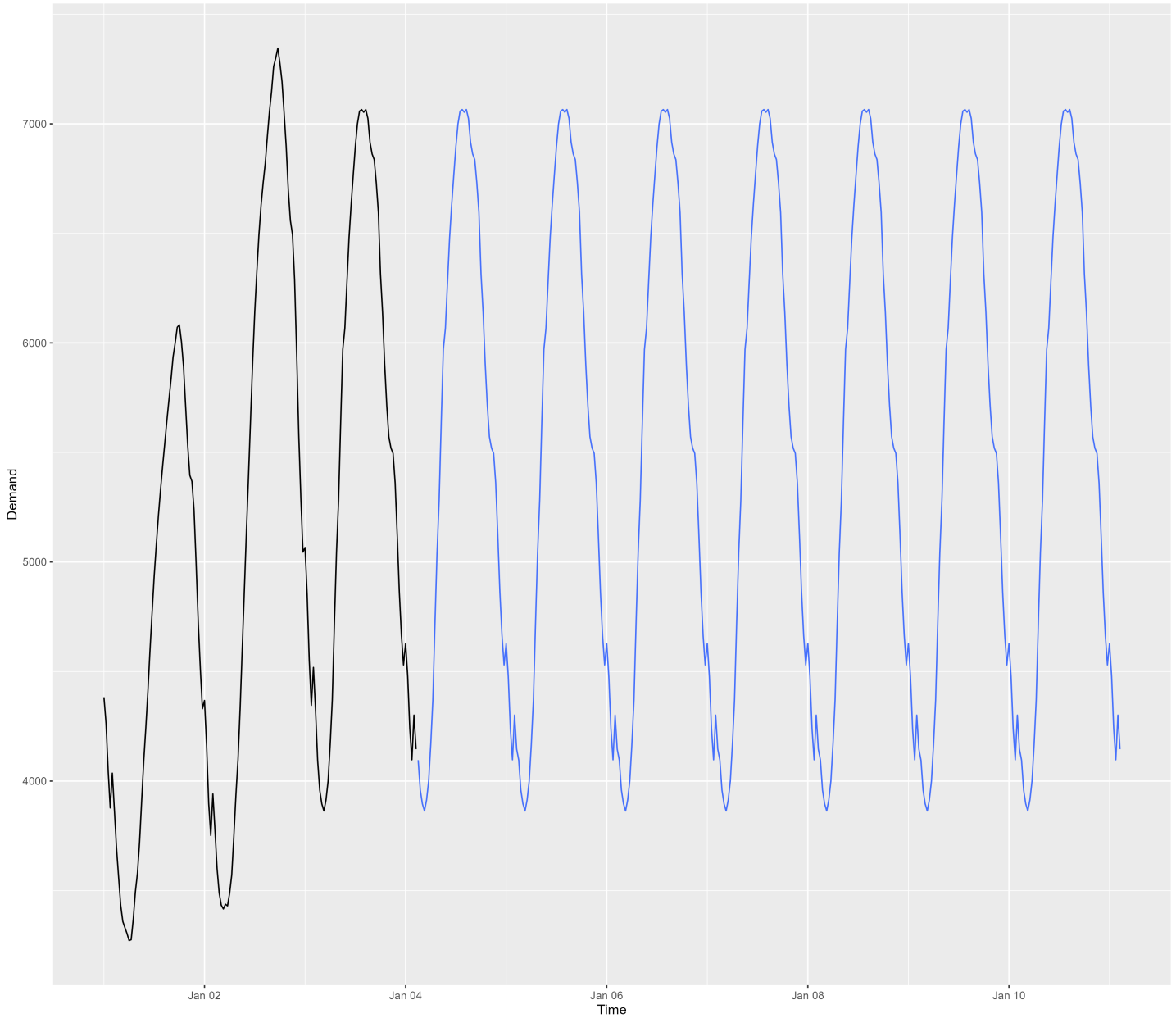
```
bricksData |>
    model(SNAIVE(Bricks ~ lag("year"))) -> brickSNaiveModel

brickSNaiveModel |>
    forecast(h = 10) -> brickSNaiveForecast

brickSNaiveForecast
brickSNaiveForecast |>
    autoplot(level = NULL) +
    autolayer(bricksData, Bricks)
```


600
500
400
300
Bricks
1970 Q1
1980 Q1
1990 Q1
2000 Q1
Quarter

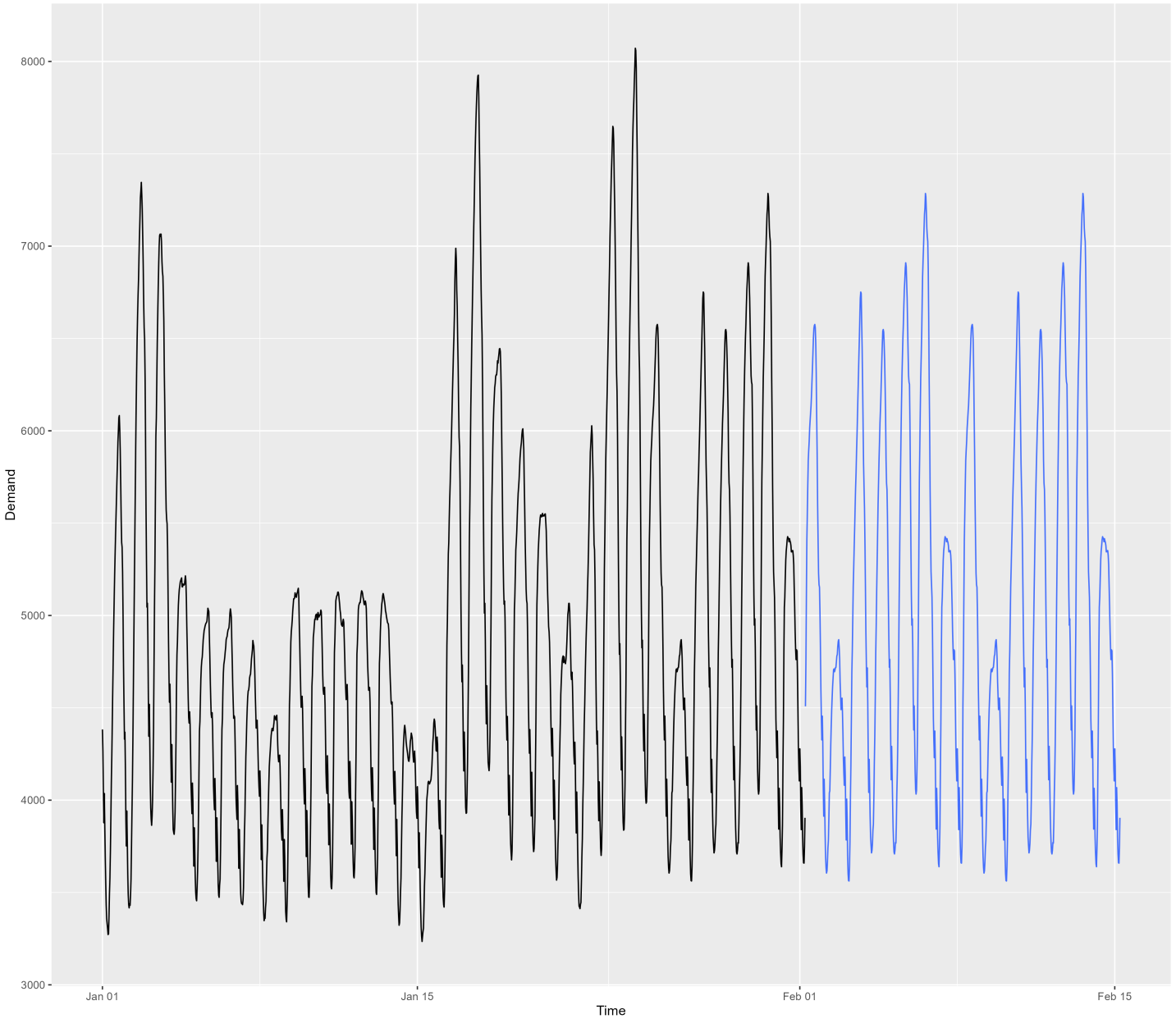
- When the data has multiple possible seasonal patterns, you can specify for what seasonal length you want the forecast to be.

```
vic_elec |> head (n = 150) -> vic_elec_short
vic_elec_short |> autoplot()
vic_elec_short |> model(SNAIVE(Demand ~ lag("day"))) -> vic_elec_SNaiveDayModel
vic_elec_SNaiveDayModel |> forecast(h = "7 days") -> vic_elec_SNaiveDayForecast
vic_elec_SNaiveDayForecast |>
  autoplot(level= NULL) +
  autolayer(vic_elec_short, Demand)

vic_elec |> head (n = 1500) -> vic_elec_short
vic_elec_short |> autoplot()
vic_elec_short |> model(SNAIVE(Demand ~ lag("week"))) -> vic_elec_SNaiveDayModel
vic_elec_SNaiveDayModel |> forecast(h = "2 weeks") -> vic_elec_SNaiveDayForecast
vic_elec_SNaiveDayForecast |>
  autoplot(level= NULL) +
  autolayer(vic_elec_short, Demand)


vic_elec |> head (n = 1500) -> vic_elec_short
vic_elec_short |> autoplot()
vic_elec_short |> model(SNAIVE(Demand ~ lag("month"))) -> vic_elec_
    SNaiveDayModel
vic_elec_SNaiveDayModel |> forecast(h = "1 month") -> vic_elec_SNaiveDayForecast
vic_elec_SNaiveDayForecast |>
  autoplot(level= NULL) +
  autolayer(vic_elec_short, Demand)
```

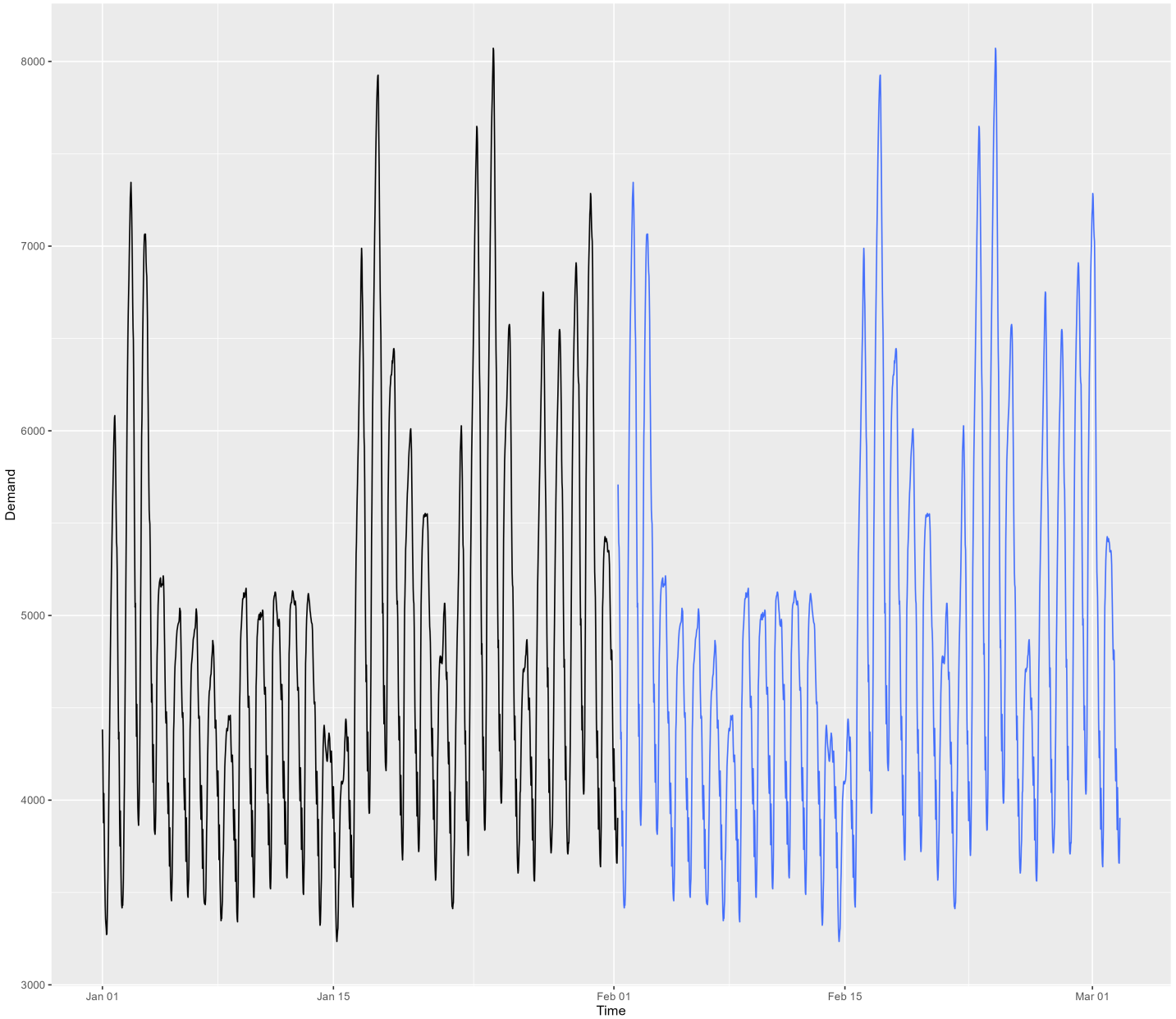
- Daily seasonality, forecast for 7 days.



- Weekly seasonality, forecast for 2 weeks



- Monthly seasonality, forecast for 1 month



The drift method I

- Suitable for data which has a trend, in case of the drift model, we allow the predictions to increase / decrease. The amount of change (drift) is the average change seen in the historical data. It is like drawing a line connecting the first and last historical observation and extrapolating from it for forecasting.

$$\hat{y}_{T+h|T} = y_T + h * (\frac{y_T - y_1}{T - 1})$$

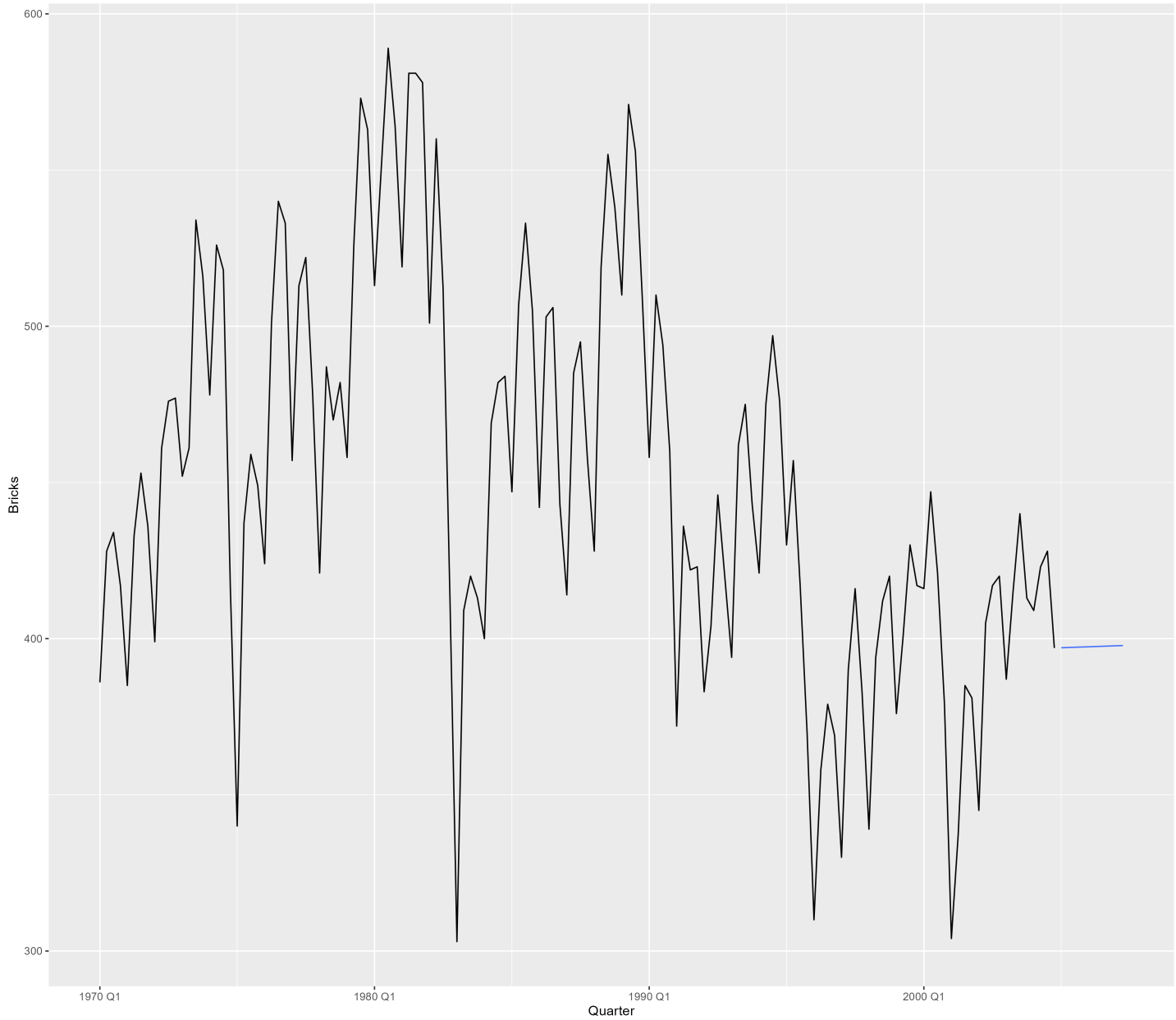
- In R it can be computed with the *RW* (random walk) function to which we pass as parameter the *drift* function.

```
bricksData |>
    model(RW(Bricks ~ drift())) -> bricksDriftModel
tidy(bricksDriftModel)
```

The drift method II

```
  .model               term  estimate std.error statistic p.value
  <chr>                <chr>    <dbl>     <dbl>     <dbl>   <dbl>
1 RW(Bricks ~ drift()) b       0.0791      3.77    0.0210   0.983
```

```
bricksDriftModel |>
    forecast(h = 10) -> bricksDriftForecast

bricksDriftForecast
bricksDriftForecast |>
    autoplot() +
    autolayer(bricksData, Bricks)
```

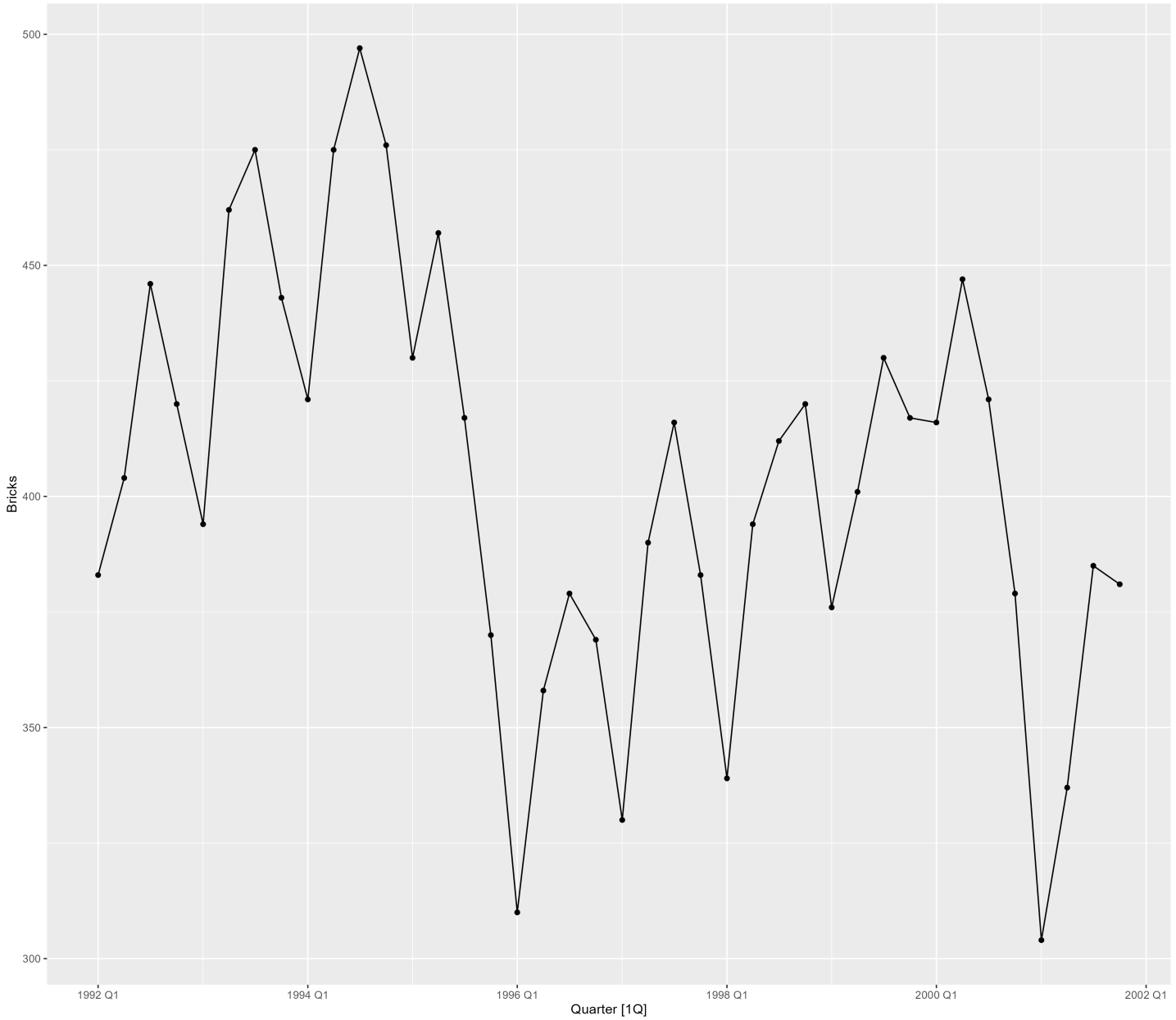

600
500
400
300
Bricks
1970 Q1
1980 Q1
1990 Q1
2000 Q1
Quarter

Example 1

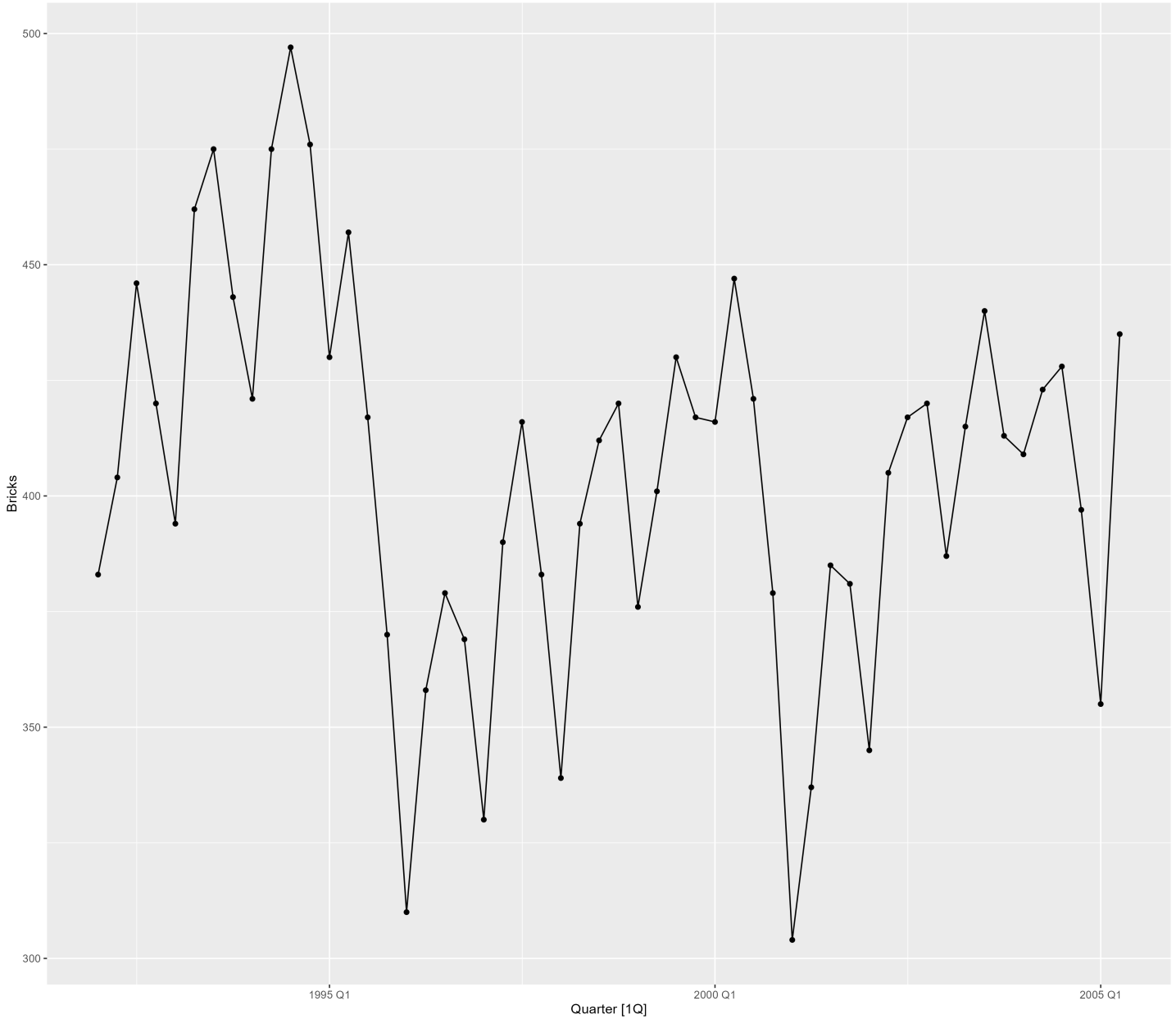
- Let's see all four naive models on the same plot for brick production
- Also, let us leave some data at the end of the time series, as *testing data* to see how well the simple models' forecast matches the reality.

```
bricksTrainData <- aus_production |>
  filter(year(Quarter) >= 1992 & year(Quarter) < 2002) |>
  select(Bricks)

bricksAllData = aus_production |>
  filter(year(Quarter) >= 1992 & !is.na(Bricks)) |>
  select(Bricks)

autoplot(bricksTrainData) +
  geom_point()

autoplot(bricksAllData) +
  geom_point()

bricksTrainData |>
  model (mean = MEAN(Bricks),
         naive = NAIVE(Bricks),
         snaive = SNAIVE(Bricks ~ lag("year")),
         drift = RW(Bricks ~ drift())) -> brickModels

brickModels |>
    forecast(h = 14) -> brickForecast
view(brickForecast)

brickForecast |>
  autoplot(bricksTrainData, level = NULL) +
  autolayer (bricksAllData, colour = "black") +
  guides(colour = guide_legend(title = "Forecasts"))
```

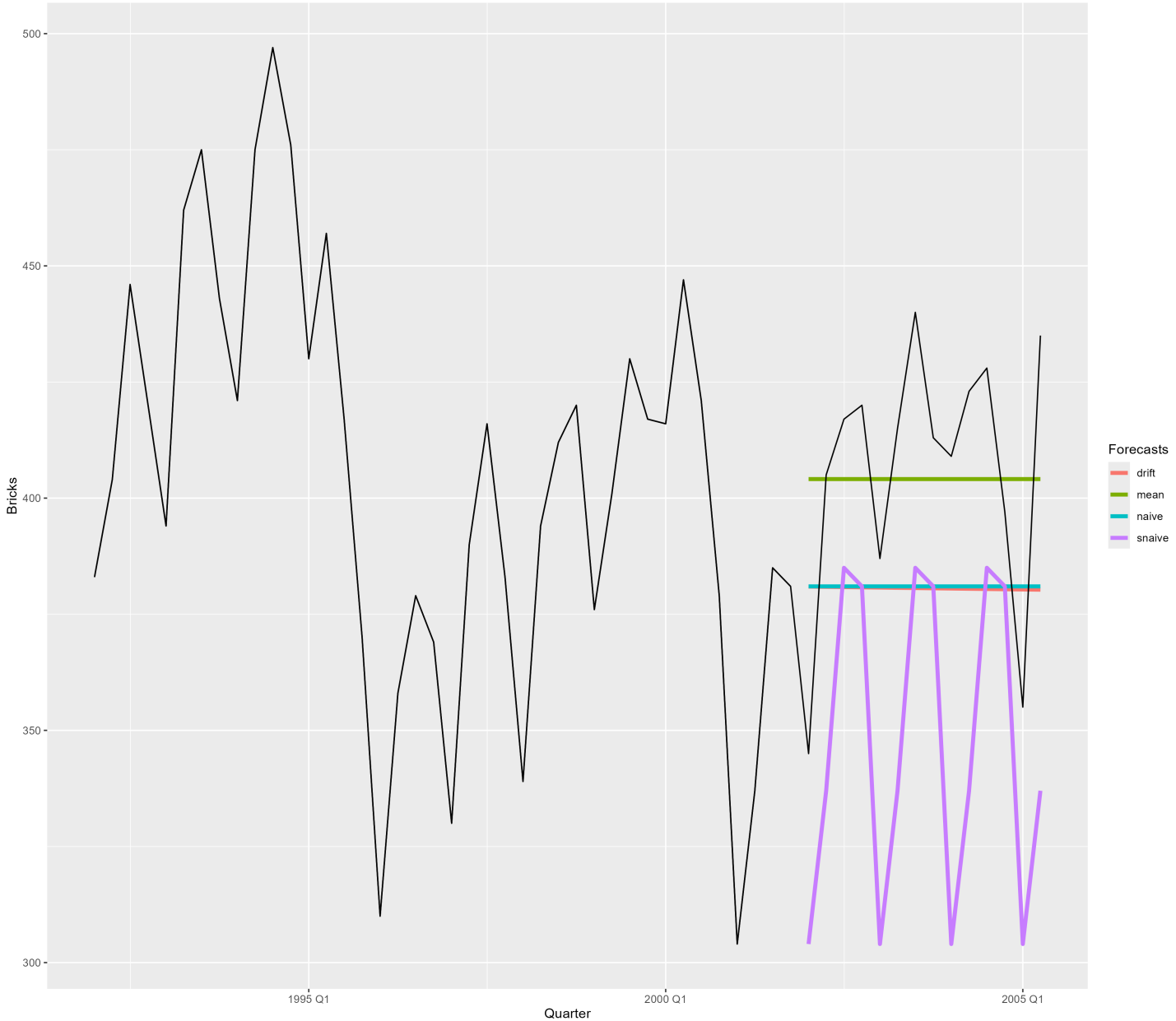
- *Training data*



- All data



- Actual data and forecasts of the four baseline methods

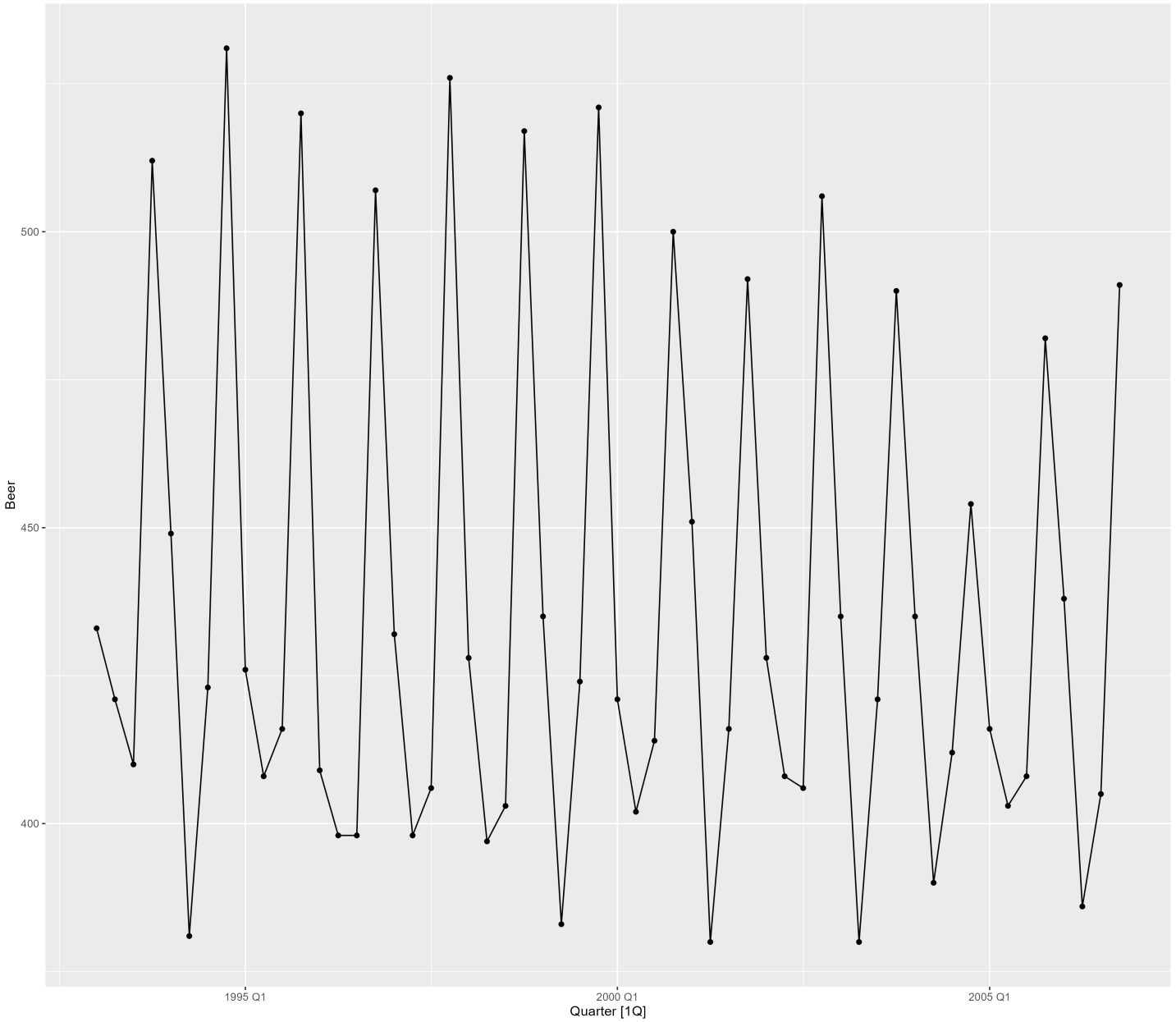


Example 2

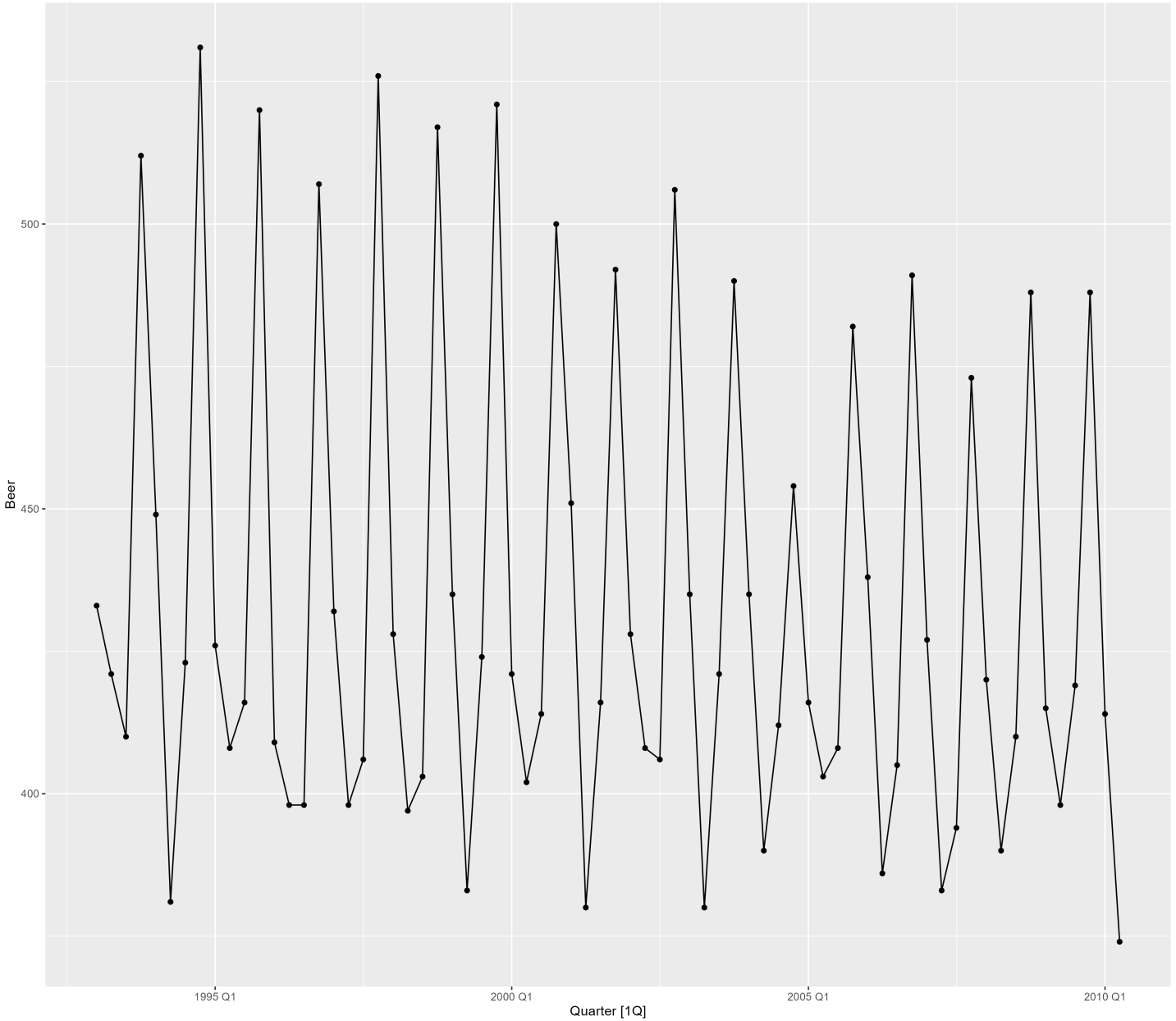
- Let's try the same example on the beer production time series from the *aus_production* data set.

```
beerTrainData <- aus_production |>
  filter(year(Quarter) > 1992 & year(Quarter) < 2007) |>
  select(Beer)

autoplot(beerTrainData) +
  geom_point()

beerAllData <- aus_production |>
  filter(year(Quarter) > 1992) |>
  select(Beer)

autoplot(beerAllData) +
  geom_point()

beerTrainData |>
  model (mean = MEAN(Beer),
         naive = NAIVE(Beer),
         snaive = SNAIVE(Beer ~ lag("year")),
         drift = RW(Beer ~ drift()))  -> beerModels

beerModels |>
    forecast(h = 14) -> beerForecast
view(beerForecast)

beerForecast |>
  autoplot(beerTrainData, level = NULL) +
  autolayer (beerAllData, colour = "black") +
  guides(colour = guide_legend(title = "Forecasts"))
```

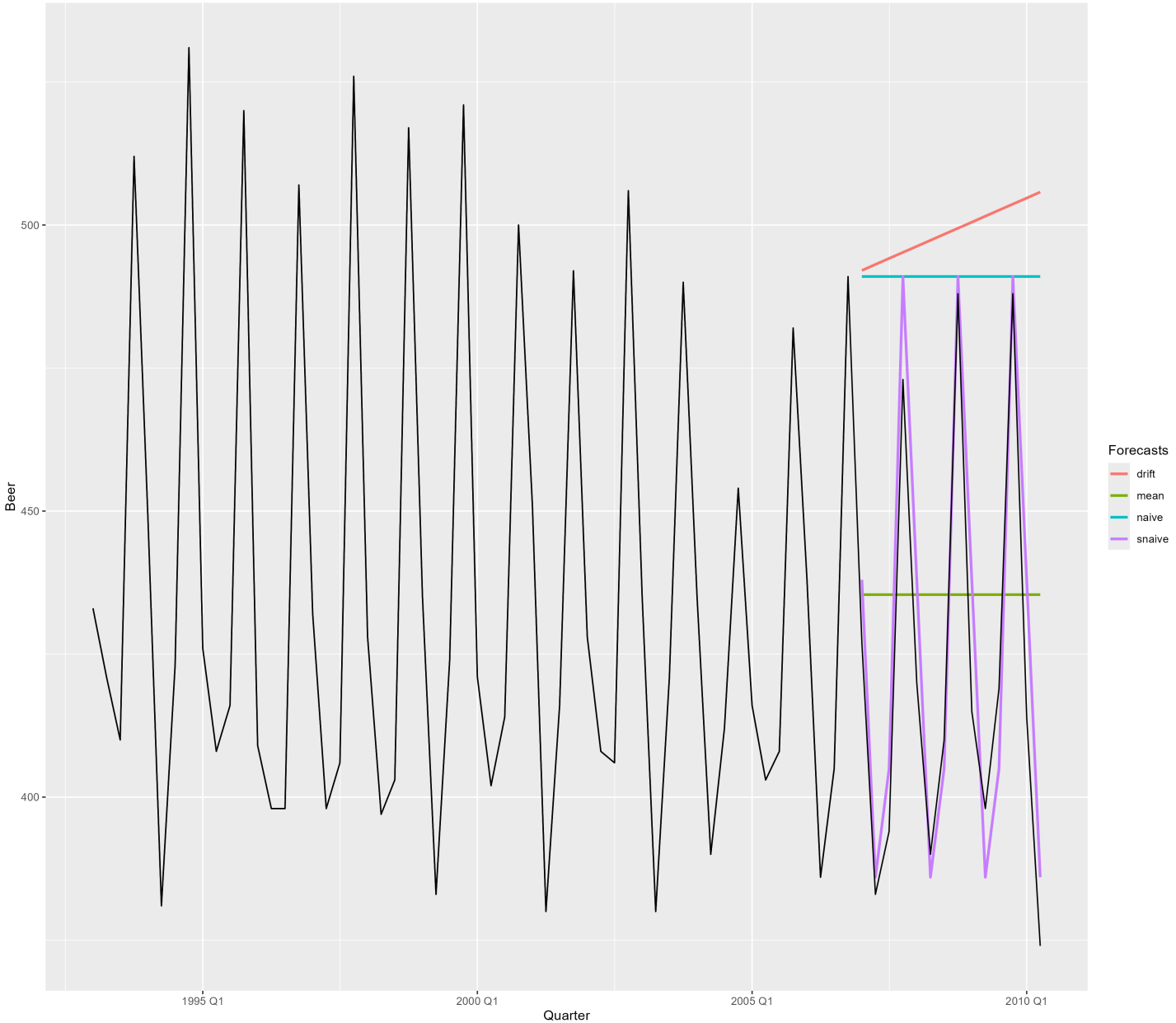
- *Training data*



- All data



- Actual data and forecasts of the four baseline methods



Example 3

- Let's see a third example using Google stock prices.
- The *gafa_stock* data set contains information about Google, Amazon, Facebook and Apple stock prices between 2014 - 2018.
- We will only use Google data, for *training* the model we will use data from 2015 and we will forecast the first month of 2016.
- Since it is stock prices, the time interval is trading days, which are not consecutive days and are not accepted by some functions. So we will first create a new index for our data (just consecutive numbers), then we will repeat the analysis done in the previous 2 examples.

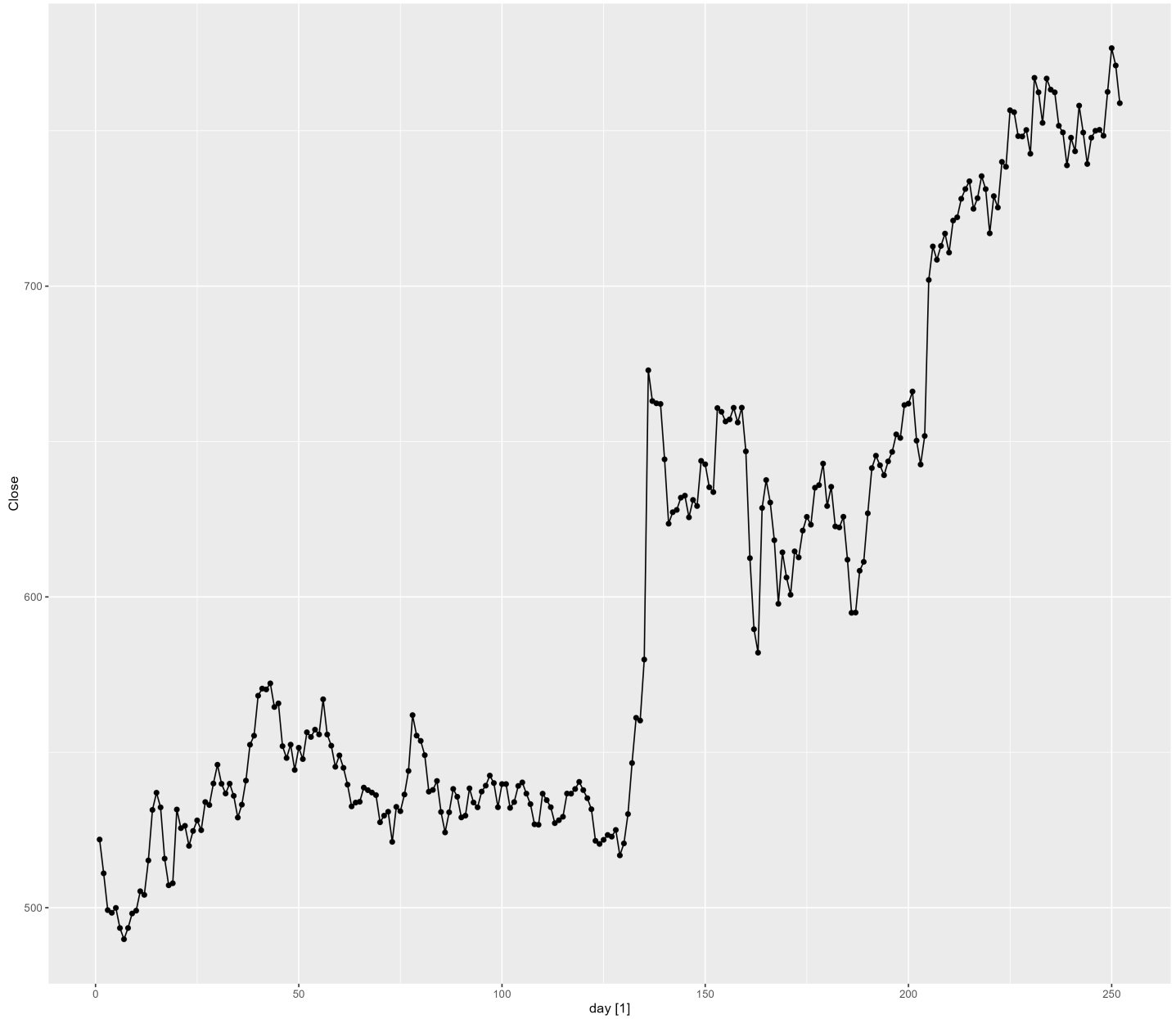
Example 3 I

```
gafa_stock
gafa_stock |>
  filter(Symbol == "GOOG"& year(Date) > 2014 & yearmonth(Date) <= yearmonth("
     2016 Jan")) -> googleData
view(googleData)

#we need to know how many observations we have for train and test.
#Since data is for trading days, this is not obvious
google2016DataCount <- googleData |>
  filter(year(Date) == 2016) |> count()
google2016DataCount
google2015DataCount <- googleData |>
  filter(year(Date) == 2015) |> count()
google2015DataCount


googleData <- googleData |>
  mutate(day = row_number()) |> #new attribute with consecutive numbers
  update_tsibble(index = day, regular = TRUE) |> #make the new attribute the
     index
  select(Close)

googleData
googleTrain <- googleData |> head(n = google2015DataCount$n[1])
googleTest <- googleData |> tail(n = google2016DataCount$n[1])
```

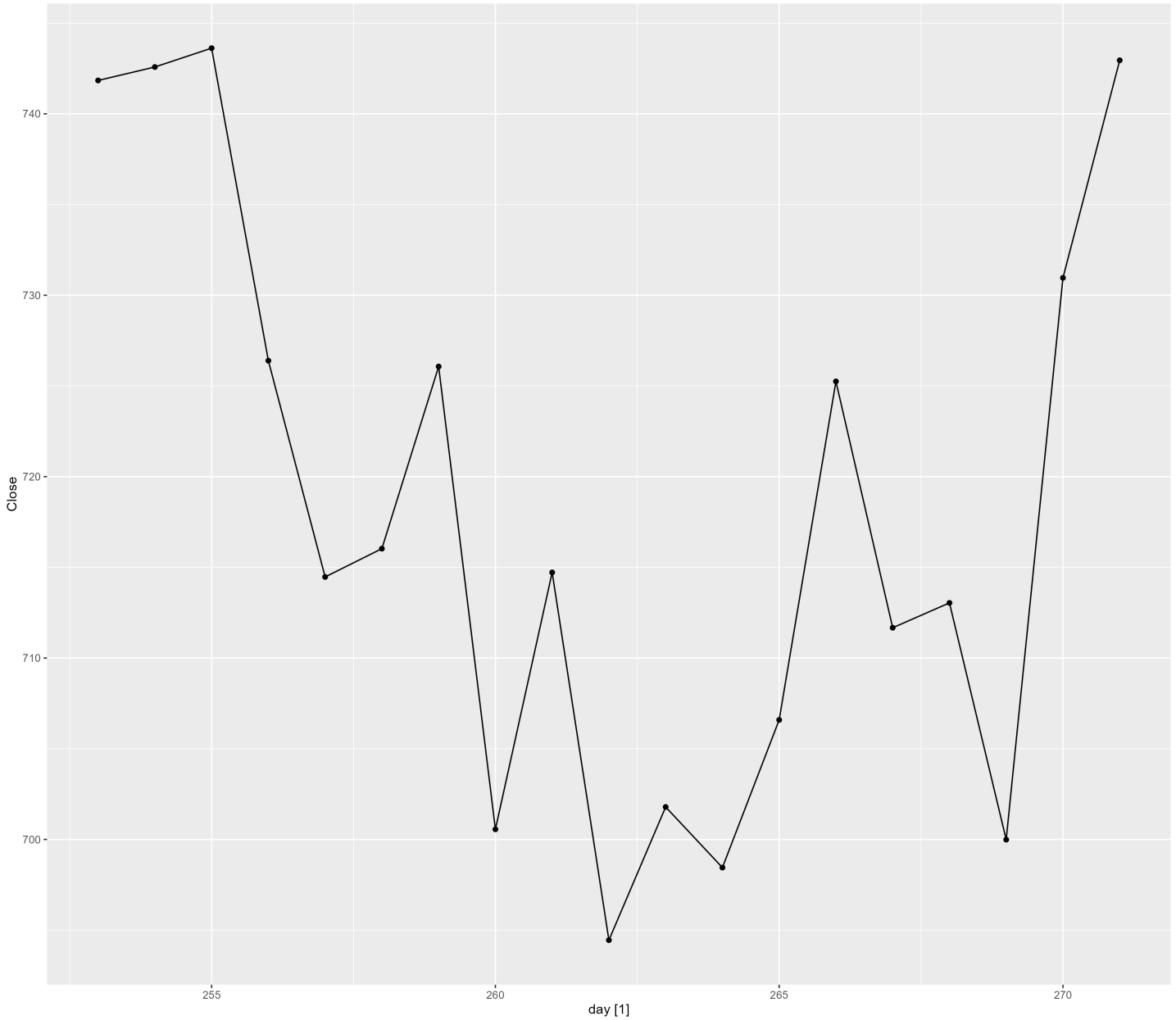
Example 3 II

```
autoplot(googleTrain) +
  geom_point()

autoplot(googleTest) +
  geom_point()

googleTrain |>
  model (mean = MEAN(Close),
         naive = NAIVE(Close),
         drift = RW(Close ~ drift()),
        snaive = SNAIVE(Close ~ lag(5))) -> googleModels

googleModels |>
  forecast(h = google2016DataCount$n[1]) -> googForecast
view(googForecast)

googForecast |>
  autoplot(googleTrain, level = NULL, size = 1.5) +
  autolayer (googleData, colour = "black") +
  guides(colour = guide_legend(title = "Forecasts"))
```

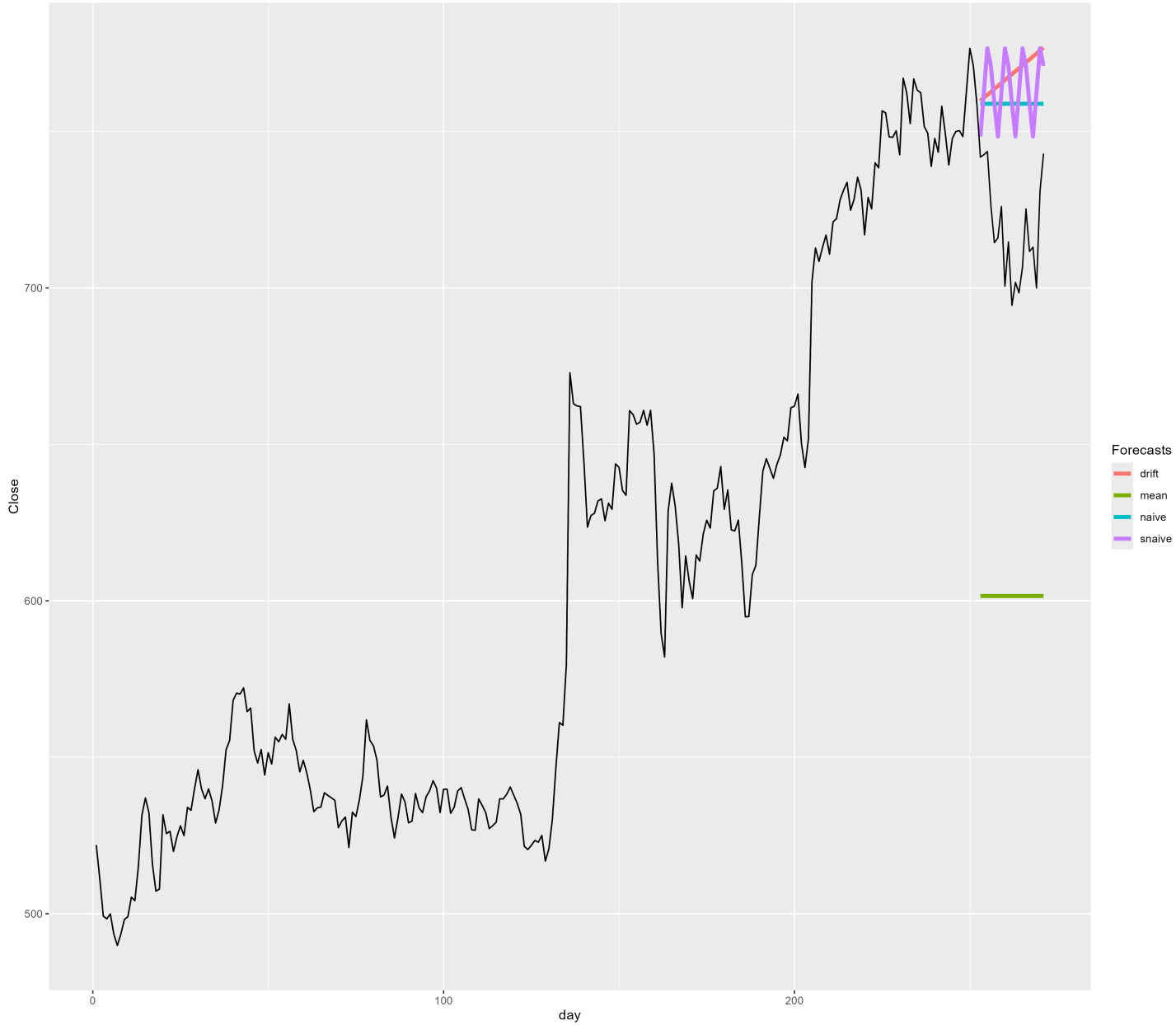
- Google training data



- Google testing data



- Actual data and forecasts of the four baseline methods



Observations

- The seasonal naive baseline might not make sense in case of the Google data, since originally the data is for irregular intervals (trading days). The above example considers a seasonality of 5 days, but due to national holidays, this is probably not correct (i.e., not every week is made of 5 trading days).

Fitted values

- Once you start building forecasting models (like the previous baseline), there are two important questions to consider:
  - How well that model fits the historical data?
  - How well that model can predict new data?
- Having an already trained model, for each observation from a time series we can do the following: use the model to forecast the value of this observation, using all the previous values. These are called **fitted values** and are denoted by $\hat{y}_{t|t-1}$ or more simply $\hat{y}_t$.
- Are fitted values true forecasts?

- It depends on what kind of model produced them. If the model has some parameters, then it is very likely that the parameter of the model was estimated using all available observations, including y_t and even observations for time periods after t. In this case we do not have real forecasts.

- On the other hand, if the model has no parameters, these are real forecasts.

- Which of the discussed baseline models have parameters to be estimated from data?

- The *naive* and *seasonal naive* baseline models have no parameters (they just repeat the last observation). When we use them, fitted values are real forecasts.
- In case of the *mean* model, we compute the mean of all values, so in this case we do not have real forecasts.
- Similarly, in case of the *drift* model we use all data to estimate the slope of the line between the first and last observation, so we do not have real forecasts.

- **Obs.** This is not the same as time series cross-validation, because for fitted values we build one single model, while in case of time series cross validation we have several. For example, for the mean model:
  - When talking about *fitted values* we compute the mean of all T observations and then this mean is used to compute the fitted values (actually, this mean is the fitted value).
  - When talking about *time series cross-validation* we estimate the mean of the first t observations and use it to forecast observation $t+1$. Then we compute the mean of the first $t+1$ observations and use it to forecast observation $t+2$, etc.

Residuals

- The *residual* of a time series model is the difference between the observation and the fitted value:

$$e_t = y_t - \hat{y}_t$$

- The fitted values and the residuals can be obtained using the *augment* function, passing to it as parameter the fitted models. The function which adds three new columns to the data:
  - *.fitted*: the fitted values
  - *.resid*: the residuals
  - *.innov*: the *innovation residuals*
- *Innovation residuals* are important when the perform transformations (for example: $w_t = log(y_t)$) on the data before forecasting. In this case we have the regular residuals $(y_t - \hat{y}_t)$ and the innovation residuals which are computed for the transformed values: $w_t - \hat{w}_t$.

```
augment(beerModels)
```

Residual diagnostics

- You can use residuals to check if the model has captured all the information in the data. A good model will give innovation residuals with the following property:
  - The innovation residuals are uncorrelated.
  - The innovation residuals have zero mean, otherwise they are biased. (Otherwise, if residuals have mean m, simply add this value to all forecasts).
- Any model not respecting these two conditions can be improved. Models respecting these conditions *might* be improved. You should not use the residuals to select a good forecasting method from several methods.

- Optionally, it would be good to have the following two properties as well (useful for computing the prediction interval):
  - The innovation residuals have constant variance (called *homoscedasticity*)
  - The innovation residuals are normally distributed
- Models not respecting these might still be good, but the prediction interval needs to be obtained with different methods. Sometimes using a Box-Cox transformation might help with these properties, but otherwise there is not much you can do to achieve them.

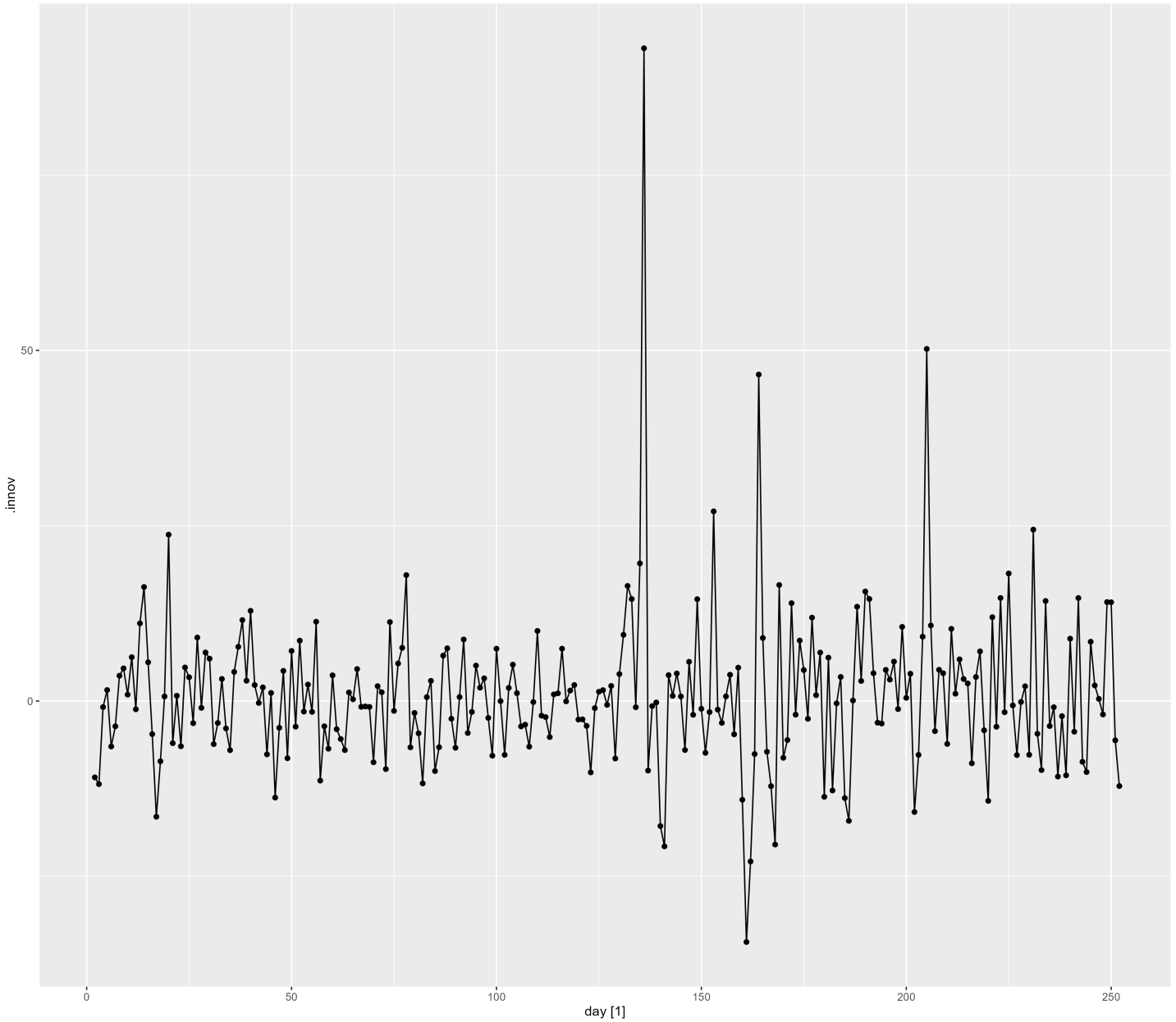
Example

- Let's look at the fitted values and residuals for the 2015 Google stock price data set (used previously). For stock market prices, in many cases the naive model is the best (the one where each forecast value is equal to the previous one). Consequently, the residuals are simply the differences between consecutive observations.

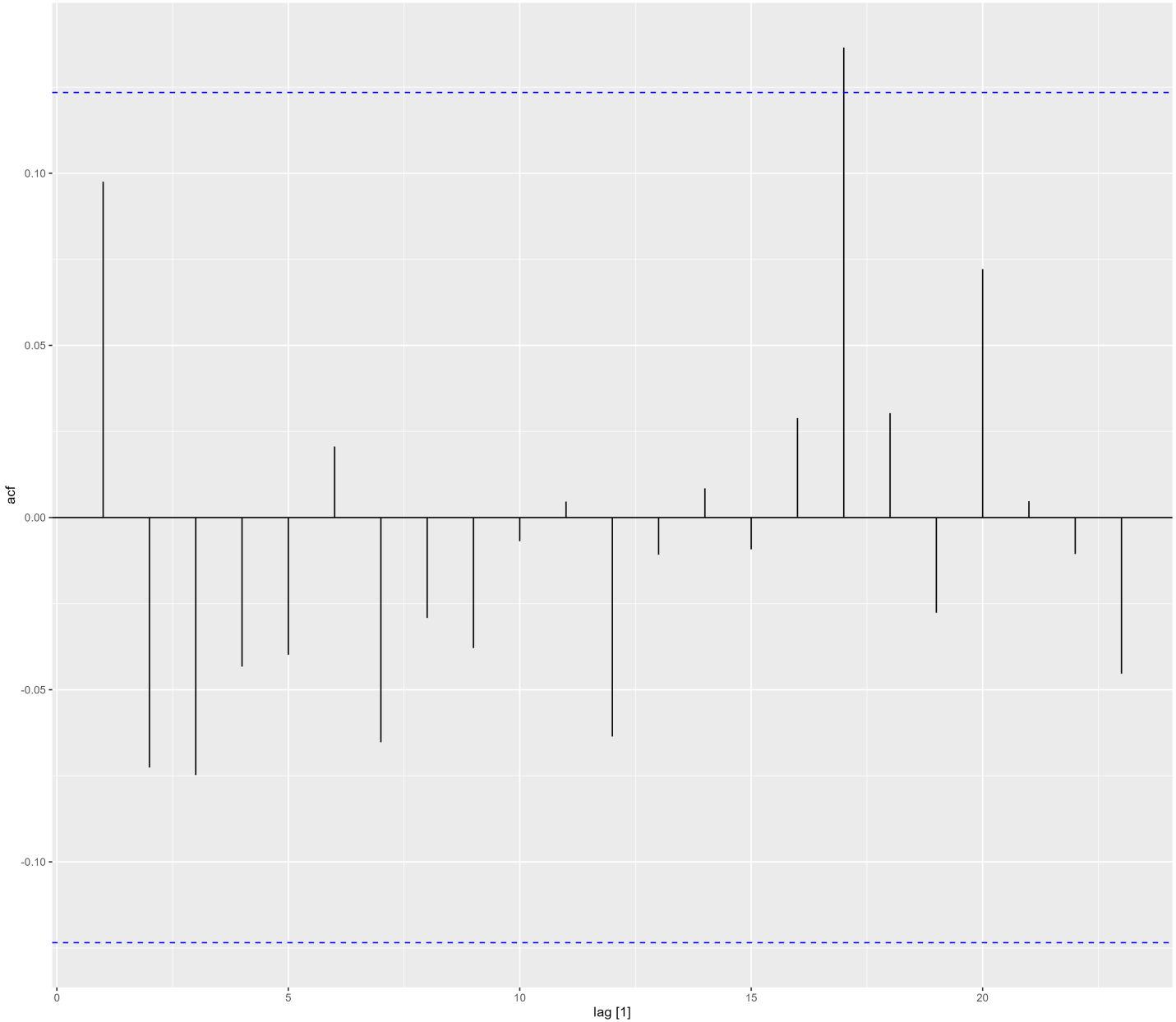
```r
google2015Data <- googleTrain #just rename it for clarity
google2015Data |>
  model(naive = NAIVE(Close)) |>
  augment() -> google2015NaiveModel

google2015NaiveModel

google2015NaiveModel |>
  ggplot(mapping = aes(x = day)) +
  geom_line(mapping = aes(y = Close, color = "Data"), size = 1) +
  geom_line(mapping = aes(y = .fitted, color = "Fitted"), size = 1)

autoplot(google2015NaiveModel, .innov) +
  geom_point()

#check for a mean of 0 (it is 0.94 when the innovation residuals are between -34
      and 93. It is OK)
google2015NaiveModel$.innov |> tail(251) |> mean()
google2015NaiveModel$.innov |> tail(251) |> min()
google2015NaiveModel$.innov |> tail(251) |> max()

google2015NaiveModel |>
  ACF(.innov) |>
  autoplot()
```

- Actual values vs. fitted values for the naive model on Google data

- Innovation residuals for the naive model on Google stock price



- ACF of innovation residuals



Example

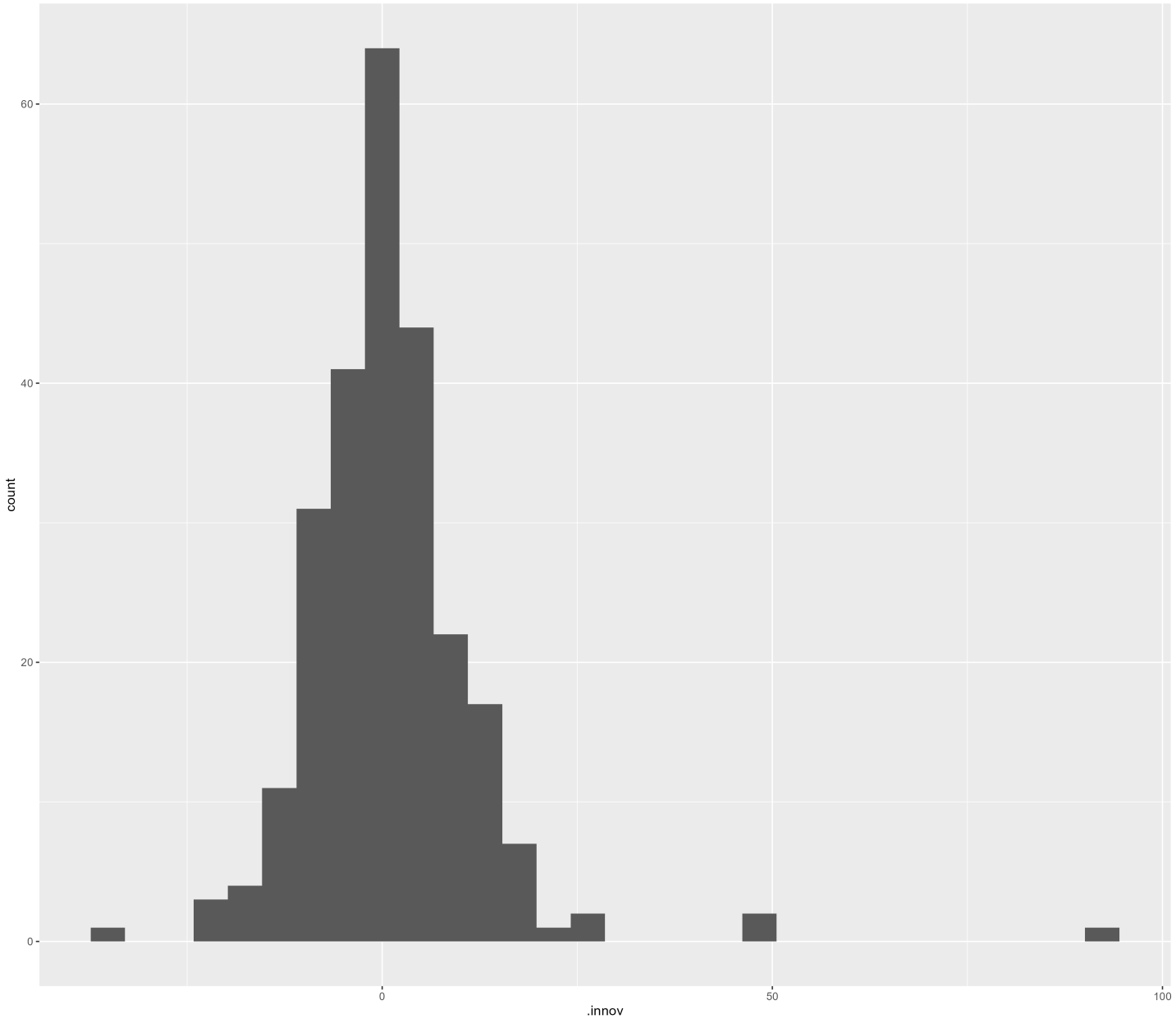
- Since the mean is around 1 (when the values range from -35 to 93) we can say that it is almost zero.
- The ACF plot shows that there is no autocorrelation in data.
- We can conclude that the naive model captures all information from data.

- Let's see the other two properties: normality and constant variance.
- Constance variance can be checked on the autoplot of the innovation residuals (Figure 14). For normality we need a histogram.

```
google2015NaiveModel |>
  ggplot(mapping = aes(x = .innov)) +
  geom_histogram()
```

- We can see that the residuals seem to have a constant variance (with some minor exceptions), but they are not really normally distributed (the right side contains too many values).

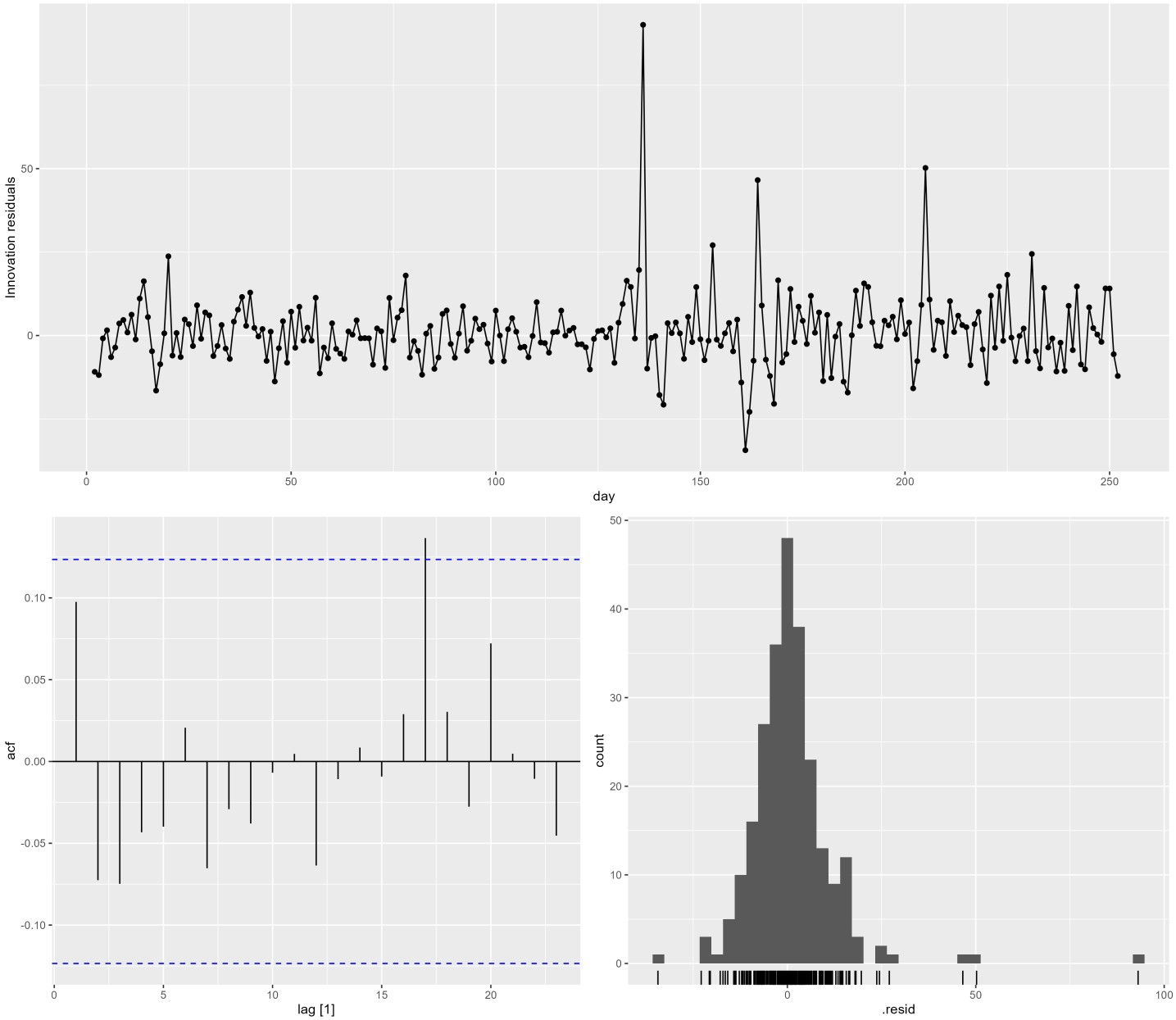
- Histogram of innovation residuals



- In this example we have used three different plots: a time series plot, an autocorrelation plot and a histogram to check the innovation residuals. We can create all three plots at once with the *gg_tsresiduals* function, which takes as parameter a model (not the augmented one, just the regular model).

```
google2015Data |>
 model(naive = NAIVE(Close)) |>
 gg_tsresiduals()
```

- Residual diagnostics for the Naive model on Google



Autocorrelation tests

- Besides looking at the blue line and determining whether the data is autocorrelated, we can have a more formal test for the autocorrelation of the residuals, which treats all residuals r_k as a group and tests whether the first l autocorrelations are significantly different from what would be expected from white noise.
- There are two related tests: *Box-Pierce* and *Ljung-Box*.

Box-Pierce test

- The *Box-Pierce test* computes the following value:

$$Q = T \sum_{k=1}^{l} r_k^2$$

- where l is the maximum lag considered and T is the number of observations. If Q is small, that means that we have no autocorrelation in the data.
- For non-seasonal data the suggested value for l is 10, while for seasonal data it is suggested to use $2 * m$, where m is the period of the season.
- However, l should not exceed the value of $\frac{T}{5}$, so if the previous values are larger than this limit, $\frac{T}{5}$ should be used.

Ljung-Box text

- Another test to see if there is autocorrelation in the data, which is more accurate than the *Box-Pierce* test is the *Ljung-Box* test, which computes the following value:

$$Q^* = T * (T+2) \sum_{k=1}^{l} (T-k)^{-1} * r_k^2$$

- where the same notations are used as for the Box-Pierce test.
- Large values of Q* suggest that the autocorrelation does not come from white noise.

- What does *large* Q and Q* value mean? You need to look at the p-value. The *null hypothesis* of these tests is that the data is significantly different from the white noise. If the *p-value* returned by the test is high (ex. greater than 0.05) we can *reject* the null hypothesis and conclude that the data is not significantly different from white noise.

```
google2015NaiveModel |>
    features(.innov, box_pierce, lag = 10)
```

```
.model      bp_stat bp_pvalue
  <chr>       <dbl>     <dbl>
1 naive        7.74     0.654
```

- *bp_stat* is the Q value and *bp_pvalue* is the probability of seeing such a Q value for white noise residuals.

```
google2015NaiveModel |>
    features(.innov, ljung_box, lag = 10)
```

```
  .model lb_stat  lb_pvalue
  <chr>    <dbl>      <dbl>
1 naive     7.91      0.637
```

- Both tests returned high p-values, meaning that it is very probably to have such a Q (and Q* for the Ljung-Box test) for white noise data.