Chapter 5: The forecaster's toolbox

Ankit Gupta

27/12/2023

In this chapter, we discuss some general tools that are useful for many different forecasting situations. We will describe some benchmark forecasting methods, procedures for checking whether a forecasting method has adequately utilised the available information, techniques for computing prediction intervals, and methods for evaluating forecast accuracy.

Each of the tools discussed in this chapter will be used repeatedly in subsequent chapters as we develop and explore a range of forecasting methods.

5.1 A tidy forecasting workflow

The process of producing forecasts can be split up into a few fundamental steps:

(1) Preparing data

(2) Data Visualisation

(3) Specifying a model

(4) Model estimation

(5) Accuracy and performance evaluation

(6) Producing forecasts

The process of producing forecasts for time series data can be broken down into a few steps.

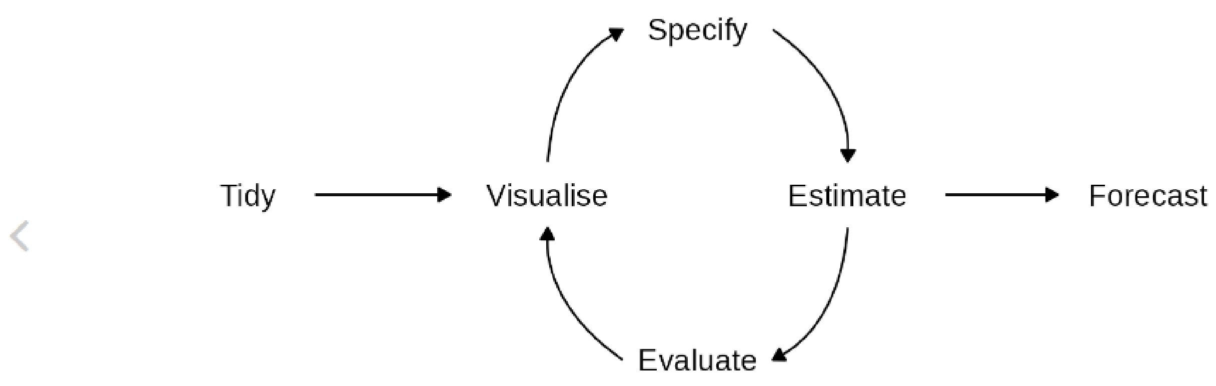


Figure 1: Process Workflow

Example

```
library(fpp3)
```

```
## -- Attaching packages -------------------------------------------- fpp3 0.5 --
## v tibble      3.2.1     v tsibble     1.1.3
## v dplyr       1.1.3     v tsibbledata 0.4.1
## v tidyr       1.3.0     v feasts      0.3.1
## v lubridate   1.9.3     v fable       0.3.3
## v ggplot2     3.4.4     v fabletools  0.3.4
```

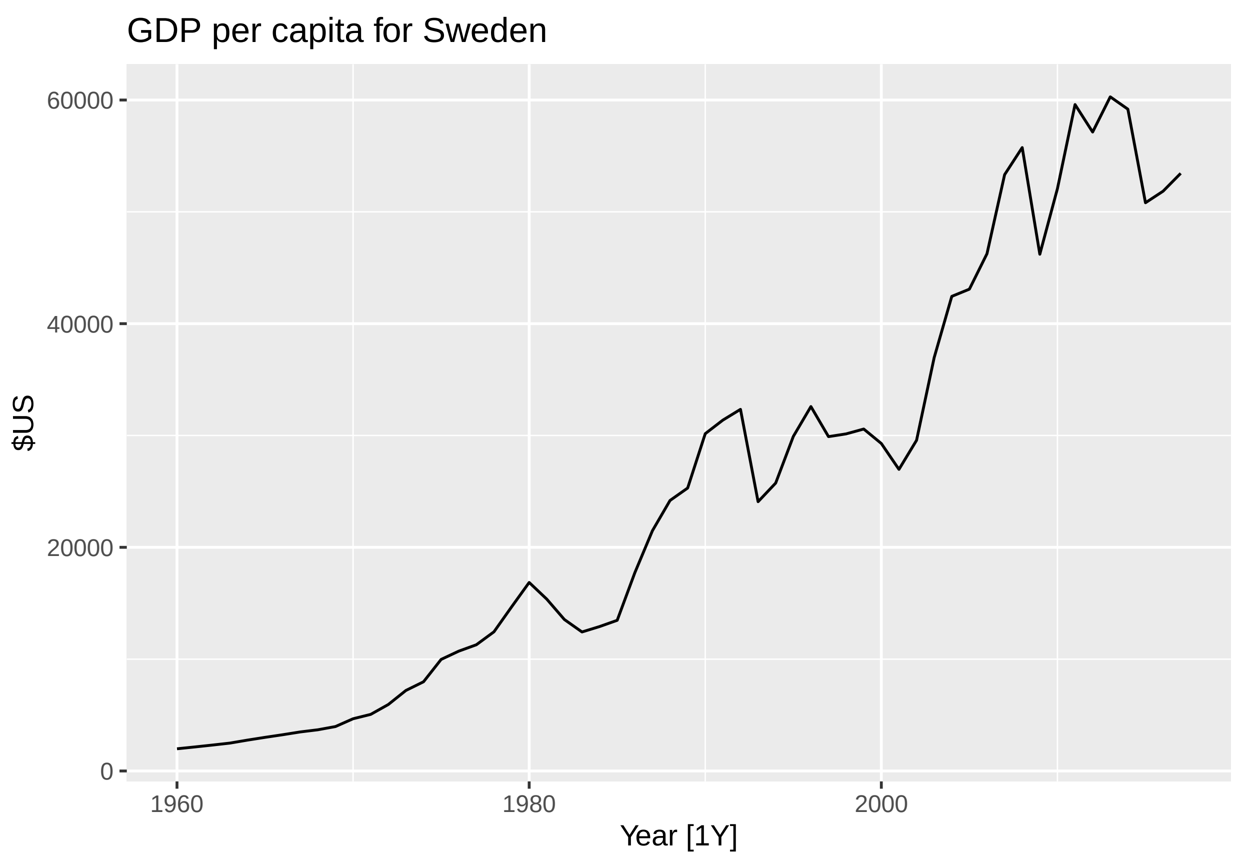
```
## -- Conflicts ------------------------------------------------- fpp3_conflicts --
## x lubridate::date()    masks base::date()
## x dplyr::filter()      masks stats::filter()
## x tsibble::intersect() masks base::intersect()
## x tsibble::interval()  masks lubridate::interval()
## x dplyr::lag()         masks stats::lag()
## x tsibble::setdiff()   masks base::setdiff()
## x tsibble::union()     masks base::union()
```

```
gdppc <- global_economy |>
  mutate(GDP_per_capita=GDP/Population) |>
  select(Year, Country, GDP, Population, GDP_per_capita)
gdppc
```

```
## # A tsibble: 15,150 x 5 [1Y]
## # Key:       Country [263]
##     Year Country             GDP Population GDP_per_capita
##    <dbl> <fct>             <dbl>      <dbl>          <dbl>
##  1  1960 Afghanistan  537777811.    8996351           59.8
##  2  1961 Afghanistan  548888896.    9166764           59.9
##  3  1962 Afghanistan  546666678.    9345868           58.5
##  4  1963 Afghanistan  751111191.    9533954           78.8
##  5  1964 Afghanistan  800000044.    9731361           82.2
##  6  1965 Afghanistan 1006666638.    9938414          101.
##  7  1966 Afghanistan 1399999967.   10152331          138.
##  8  1967 Afghanistan 1673333418.   10372630          161.
##  9  1968 Afghanistan 1373333367.   10604346          130.
## 10  1969 Afghanistan 1408888922.   10854428          130.
## # i 15,140 more rows
```

Data Visualisation

```
gdppc |>
  filter(Country=="Sweden") |>
  autoplot(GDP_per_capita)+
  labs(title = "GDP per capita for Sweden", y= "$US")
```



It is having an upward trend. Since it is yearly data, so it does not have the seasonal component.

Model estimation

Here, we use a linear model using the time trend. I am not saying that this is the best model for time series. It is just for demo.

```
fit <- gdppc |>
  model(trend_model=TSLM(GDP_per_capita ~ trend()))
```

```
## Warning: 7 errors (1 unique) encountered for trend_model
## [7] 0 (non-NA) cases
```

```
fit
```

```
## # 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
```

Here, w edenote the model notationally as:

$$y_t = \beta_0 + \beta_1 t + \epsilon_t$$

where $\epsilon_t \ N(0, \sigma^2)$

Here, TSML is a linear model for time series. Here y variable is GPP_per_capita and x variable(explanatory variable) is time variable "t" which is showing by trend(). trend() (a "special" function specifying a linear trend when it is used within TSLM().

A "mable" as shown in the above table, is a model table where each cell corresponds to a fitted model.

Evaluation

We are skipping it for now

Producing forecasts

```
fit |> forecast(h="3 years")
```

```
## # A fable: 789 x 5 [1Y]
## # Key:     Country, .model [263]
##    Country        .model       Year   GDP_per_capita  .mean
##    <fct>          <chr>       <dbl>           <dist>  <dbl>
##  1 Afghanistan    trend_model  2018     N(526, 9653)   526.
##  2 Afghanistan    trend_model  2019     N(534, 9689)   534.
##  3 Afghanistan    trend_model  2020     N(542, 9727)   542.
##  4 Albania        trend_model  2018  N(4716, 476419)  4716.
##  5 Albania        trend_model  2019  N(4867, 481086)  4867.
##  6 Albania        trend_model  2020  N(5018, 486012)  5018.
##  7 Algeria        trend_model  2018  N(4410, 643094)  4410.
##  8 Algeria        trend_model  2019  N(4489, 645311)  4489.
##  9 Algeria        trend_model  2020  N(4568, 647602)  4568.
## 10 American Samoa trend_model  2018 N(12491, 652926) 12491.
## # i 779 more rows
```

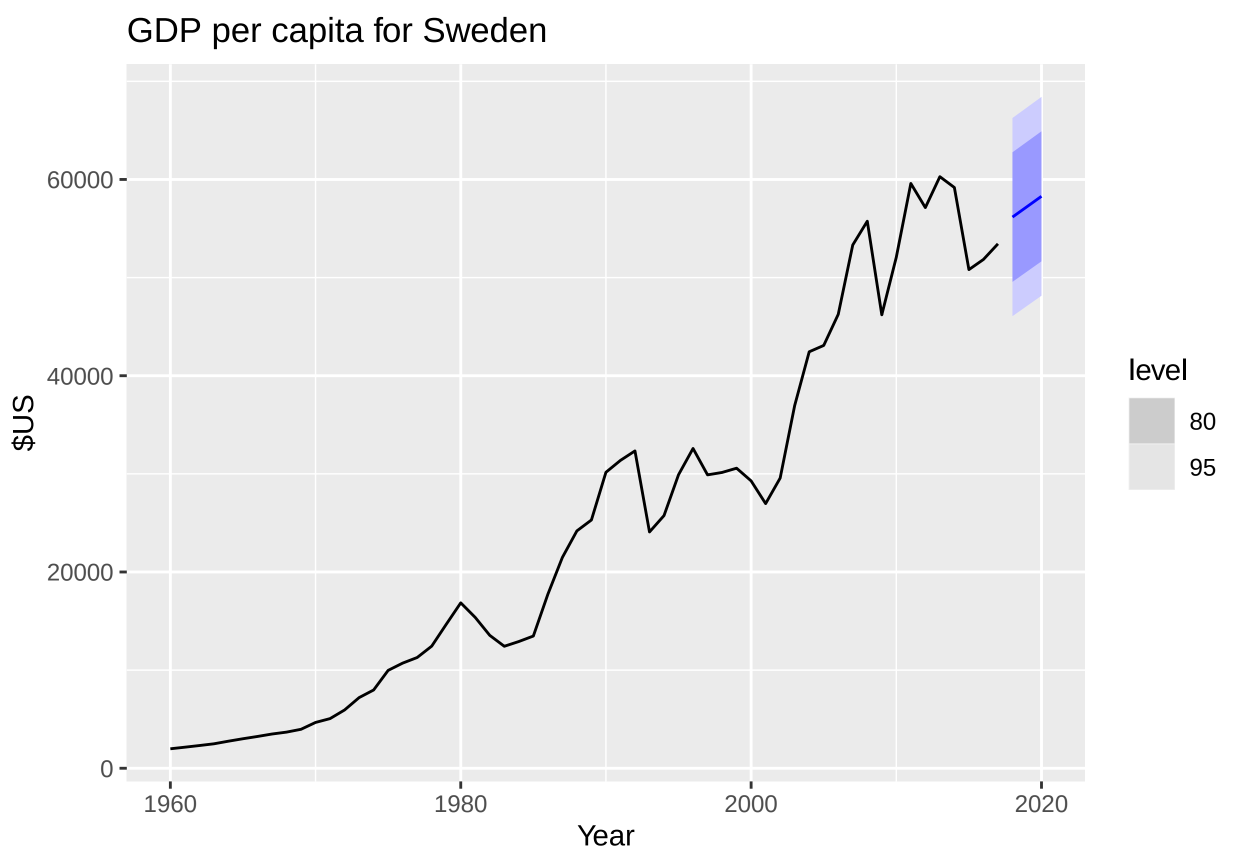
A "fable" is a forecast table with point forecasts and distributions. It is similar to a forecast. It tells something about future that is not true but it is informative about the future and very similar to what forecast is.

h=3 years means it is forecasting for next 3 years.

Our forecast is a whole distribution and a point forecast is something which we select from a distribution. N(.,.) represents the normal distribution. The GDP_per_capita column contains the forecast distribution, while the .mean column contains the point forecast. The point forecast is the mean (or average) of the forecast distribution.

Visualizing Forecasts

```
fit |>
  forecast(h= "3 years") |>
  filter(Country == "Sweden") |>
  autoplot(gdppc)+labs(title = "GDP per capita for Sweden", y="$US")
```



Summary

Data preparation (tidy)

The first step in forecasting is to prepare data in the correct format. This process may involve loading in data, identifying missing values, filtering the time series, and other pre-processing tasks. The functionality provided by tsibble and other packages in the tidyverse substantially simplifies this step.

Many models have different data requirements; some require the series to be in time order, others require no missing values. Checking your data is an essential step to understanding its features and should always be done before models are estimated.

We will model GDP per capita over time; so first, we must compute the relevant variable.

Plot the data (visualise)

As we have seen in Chapter 2, visualisation is an essential step in understanding the data. Looking at your data allows you to identify common patterns, and subsequently specify an appropriate model.

Define a model (specify)

There are many different time series models that can be used for forecasting, and much of this book is dedicated to describing various models. Specifying an appropriate model for the data is essential for producing appropriate forecasts.

Models in fable are specified using model functions, which each use a formula (y ~ x) interface. The response variable(s) are specified on the left of the formula, and the structure of the model is written on the right.

Train the model (estimate)

Once an appropriate model is specified, we next train the model on some data. One or more model specifications can be estimated using the model() function.

Check model performance (evaluate)

Once a model has been fitted, it is important to check how well it has performed on the data. There are several diagnostic tools available to check model behaviour, and also accuracy measures that allow one model to be compared against another. Sections 5.8 and 5.9 go into further details.

Produce forecasts (forecast)

With an appropriate model specified, estimated and checked, it is time to produce the forecasts using forecast(). The easiest way to use this function is by specifying the number of future observations to forecast. For example, forecasts for the next 10 observations can be generated using h = 10. We can also use natural language; e.g., h = "2 years" can be used to predict two years into the future.

In other situations, it may be more convenient to provide a dataset of future time periods to forecast. This is commonly required when your model uses additional information from the data, such as exogenous regressors. Additional data required by the model can be included in the dataset of observations to forecast.

5.2 Some simple forecasting methods

All these methods are benchmark methods.

(1) *Mean(y): Average method:*

- Forecast of all future values is equal to mean of historical data $y_1, y_2, ..., y_T$ So, it is s simple method. We take the simple average of all historical data and it becomes the point forecast for all the future values.
- Forecasts: $\hat{y}_{T+h|T} = \overline{y} = \frac{(y_1+y_2+...y_T)}{T}$
- It is not a great forecast but it is a benchmark for which we compare the other methods.

(2) *Naive(y): Naive method:*

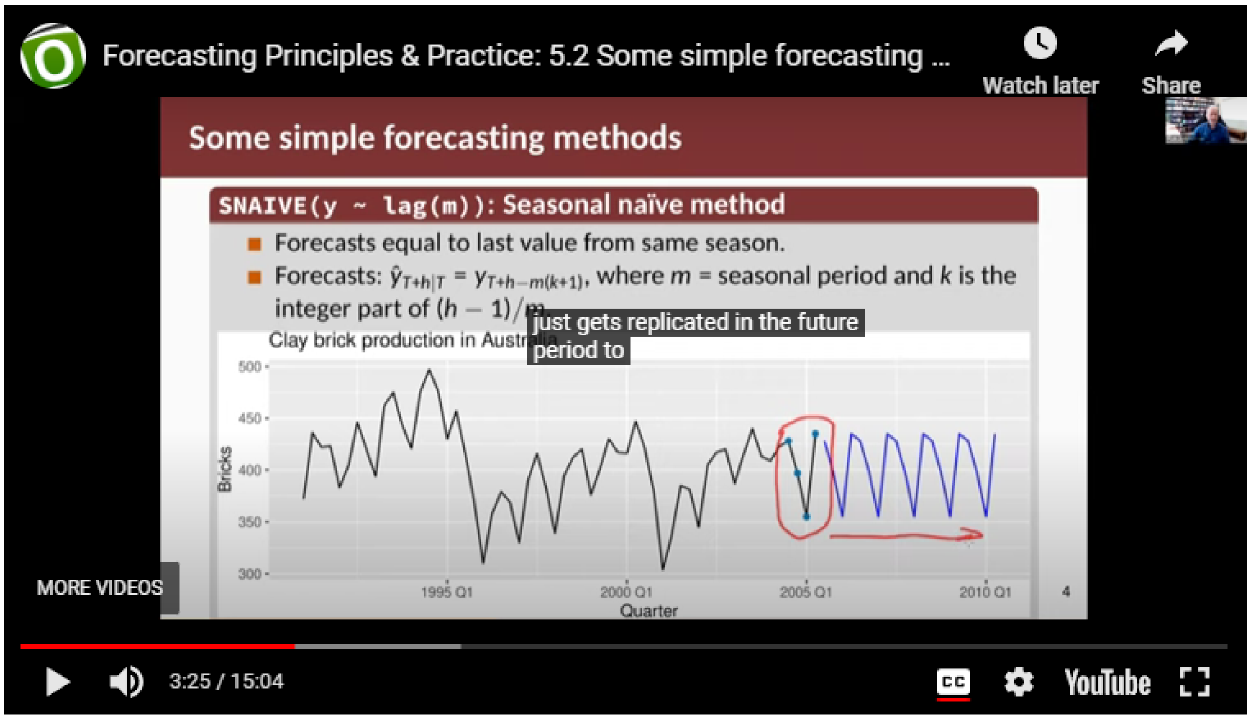
- Forecasts equal to last observed value.
- Forecasts: $\hat{y}_{T+h|T} = y_T$
- Consequence of efficient market hypothesis. It means if we have a financial data and we have a efficient market then the efficient market hypothesis says that this is the best forecast that is the stock price or exchange rate.

(3) *SNaive(y~lag(m)): Seasonal Naive method:*

- Forecasts equal to the last value from the same season. It means if we have a quarterly data and we are trying to predict quarter 1 then we take the last observed values from quarter 1 and similarly if we have to predict quarter 2 then we have to take the last observed value from quarter 2. It means last full year data is replicated into the future.
- Forecasts: $\hat{y}_{T+h|T} = y_{T+h-m(k+1)}$ where $m = \textit{seasonal period}$ and k is the integer part of $\frac{h-1}{m}$.

where m=the seasonal period, and k is the integer part of (h−1)/m (i.e., the number of complete years in the forecast period prior to time T+h).

For example, with monthly data, the forecast for all future February values is equal to the last observed February value. With quarterly data, the forecast of all future Q2 values is equal to the last observed Q2 value (where Q2 means the second quarter). Similar rules apply for other months and quarters, and for other seasonal periods.


Forecasting Principles & Practice: 5.2 Some simple forecasting ...
Watch later
Share
MORE VIDEOS
3:25 / 15:04
YouTube

Some simple forecasting methods

SNAIVE(y ~ lag(m)): Seasonal naïve method

- Forecasts equal to last value from same season.
- Forecasts: $\hat{y}_{T+h|T} = y_{T+h-m(k+1)}$, where m = seasonal period and k is the integer part of $(h-1)/m$.

just gets replicated in the future period to

Figure 2: Last year data is replicated in future to give future point estimate

(4) *RW(y~drift()): Drift method:*

- forecasts equal to the last value plus average change.
- Forecasts:

$$\hat{y}_{T+h|T} = y_T + \frac{h}{T-1}\Sigma_{t=2}^{T}(y_t - y_{t-1})$$

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

* It is equivalent to extrapolating a line drawn between first and last observations.

- Here the summation is from t=2 to t=T so total period is T-2+1=T-1 and so we divide it by T-1 to get the average and "h" is number of steps ahead (forecast period) and so we multiplied it by "h".

[Extrapolation line which connect first and last observation is extended fot future forecast] (img2.png)

Now, we see how to implement these methods in R.

Model fitting

```
brick_fit <- aus_production |>
  filter(!is.na(Bricks)) |>
  model(
    Seasonal_naive = SNAIVE(Bricks),
    Naive=NAIVE(Bricks),
    Drift=RW(Bricks~drift()),
    Mean=MEAN(Bricks)
  )
brick_fit
```

```
## # A mable: 1 x 4
##   Seasonal_naive   Naive         Drift    Mean
##          <model> <model>       <model> <model>
## 1       <SNAIVE> <NAIVE> <RW w/ drift>  <MEAN>
```

Here, drift model uses the RW() function which stands for the random walk. In the Naive model, we can also replace NAIVE() with RW(), we get the same result.

Producing forecasts

```
brick_fc <- brick_fit |>
  forecast(h="5 years")
```

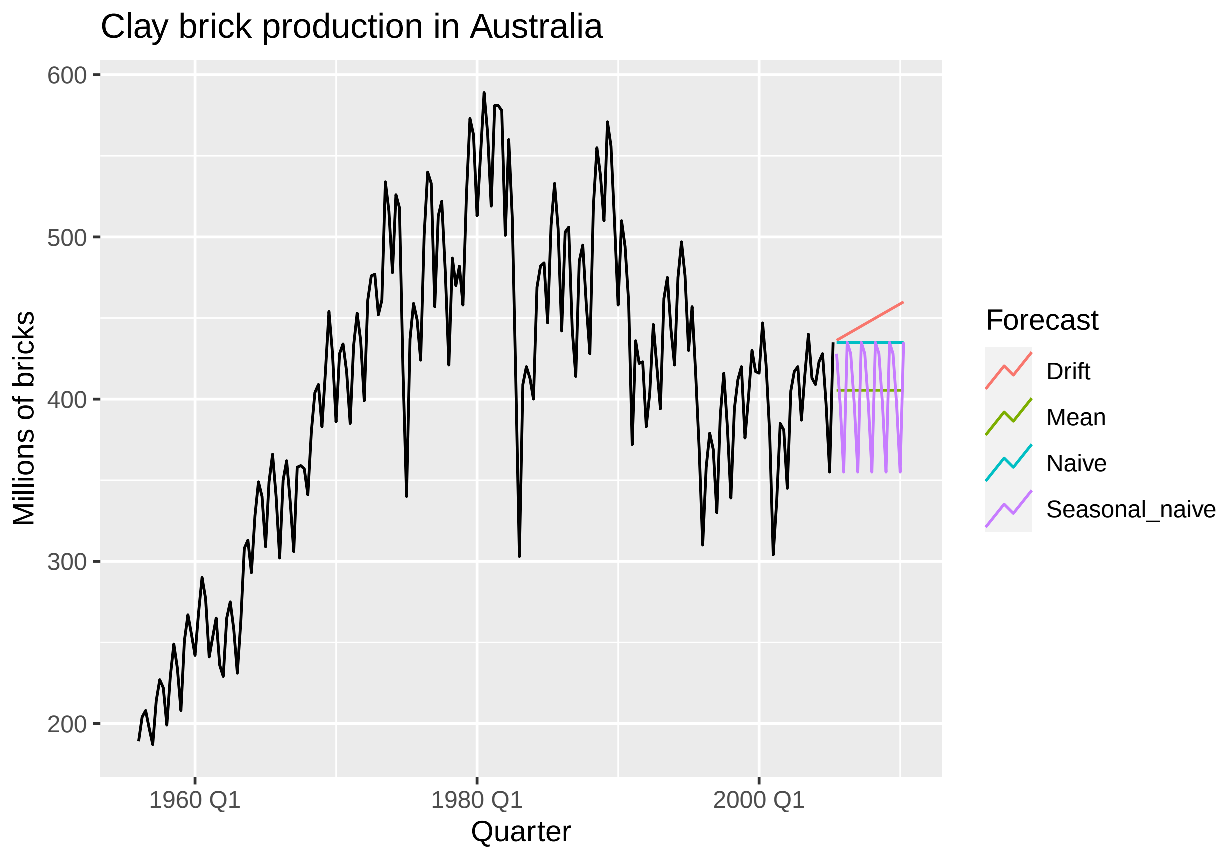
```
brick_fc
```

```
## # A fable: 80 x 4 [1Q]
## # Key:     .model [4]
##    .model         Quarter       Bricks .mean
##    <chr>            <qtr>       <dist> <dbl>
##  1 Seasonal_naive 2005 Q3 N(428, 2336)   428
##  2 Seasonal_naive 2005 Q4 N(397, 2336)   397
##  3 Seasonal_naive 2006 Q1 N(355, 2336)   355
##  4 Seasonal_naive 2006 Q2 N(435, 2336)   435
##  5 Seasonal_naive 2006 Q3 N(428, 4672)   428
##  6 Seasonal_naive 2006 Q4 N(397, 4672)   397
##  7 Seasonal_naive 2007 Q1 N(355, 4672)   355
##  8 Seasonal_naive 2007 Q2 N(435, 4672)   435
##  9 Seasonal_naive 2007 Q3 N(428, 7008)   428
## 10 Seasonal_naive 2007 Q4 N(397, 7008)   397
## # i 70 more rows
```

Visualizing Forecasts

```
brick_fc |>
  autoplot(aus_production, level = NULL)+
  labs(title = "Clay brick production in Australia",
       y="Millions of bricks")+
  guides(colour=guide_legend(title="Forecast"))
```

```
## Warning: Removed 20 rows containing missing values (`geom_line()`).
```



Here, by specifying "level=NULL", we mean don't plot the prediction intervals. By default it plots 80%,95% prediction intervals. Another argument is historical data i.e. aus_production here to plot the historical data as well.

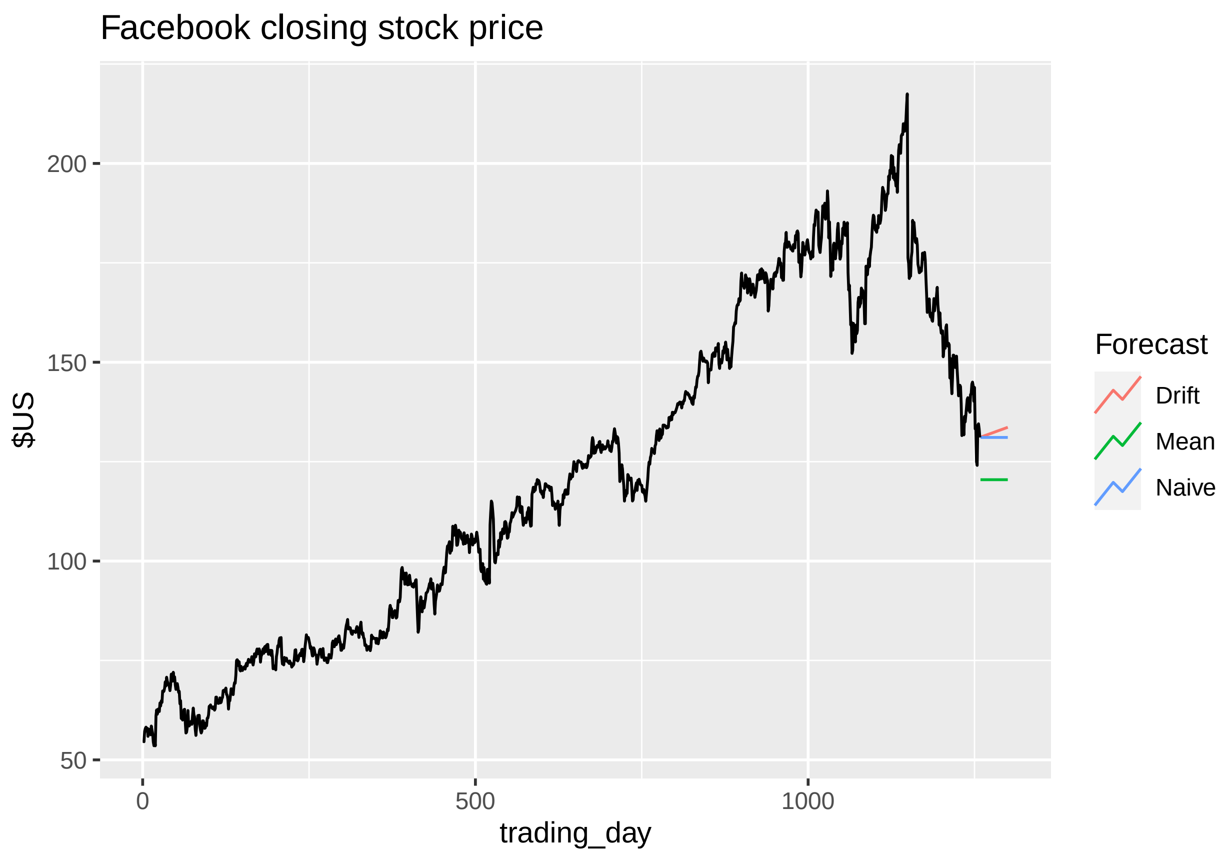
Here, in this case the plausible forecast is probably the seasonal naive forecast out all 4 benchmarking methods.

We are interested in more sophisticated forecast which is better than all 4 benchmarking forecasts.

Facebook closing stock price

```
# Extract training data
fb_stock <- gafa_stock |>
  filter(Symbol=="FB") |>
  mutate(trading_day=row_number()) |>
  update_tsibble(index = trading_day,regular = TRUE)

# Specify, estimate and forecast
fb_stock |>
  model(
  Mean=MEAN(Close),
  Naive=NAIVE(Close),
  Drift=RW(Close ~ drift())
) |>
  forecast(h=42) |>
  autoplot(fb_stock,level=NULL)+
  labs(title = "Facebook closing stock price",y="$US")+
  guides(colour=guide_legend(title="Forecast"))
```



Here trading day represents when we got trading in the data. For example, if on monday, trading does not happened then it will not be showing. We have not fitted the seasonal naive because seasonality is not presented in the data. h=42 says that it forecasts for next 42 trading days ahead.

Here, MEAN() is not a very sensible model for stock prices because they wander around a lot. The NAIVE() and DRIFT() methods are not too bad here becasue they start from very recent observation.

—————————— Summary ——————————

Some forecasting methods are extremely simple and surprisingly effective. We will use four simple forecasting methods as benchmarks throughout this book.

To illustrate them, we will use quarterly Australian clay brick production between 1970 and 2004

```
bricks <- aus_production |>
  filter_index("1970 Q1" ~ "2004 Q4") |>
  select(Bricks)
bricks
```

```
## # A tsibble: 140 x 2 [1Q]
##    Bricks Quarter
##     <dbl>   <qtr>
##  1    386 1970 Q1
##  2    428 1970 Q2
##  3    434 1970 Q3
##  4    417 1970 Q4
##  5    385 1971 Q1
##  6    433 1971 Q2
##  7    453 1971 Q3
##  8    436 1971 Q4
##  9    399 1972 Q1
## 10    461 1972 Q2
## # i 130 more rows
```

The filter_index() function is a convenient shorthand for extracting a section of a time series.

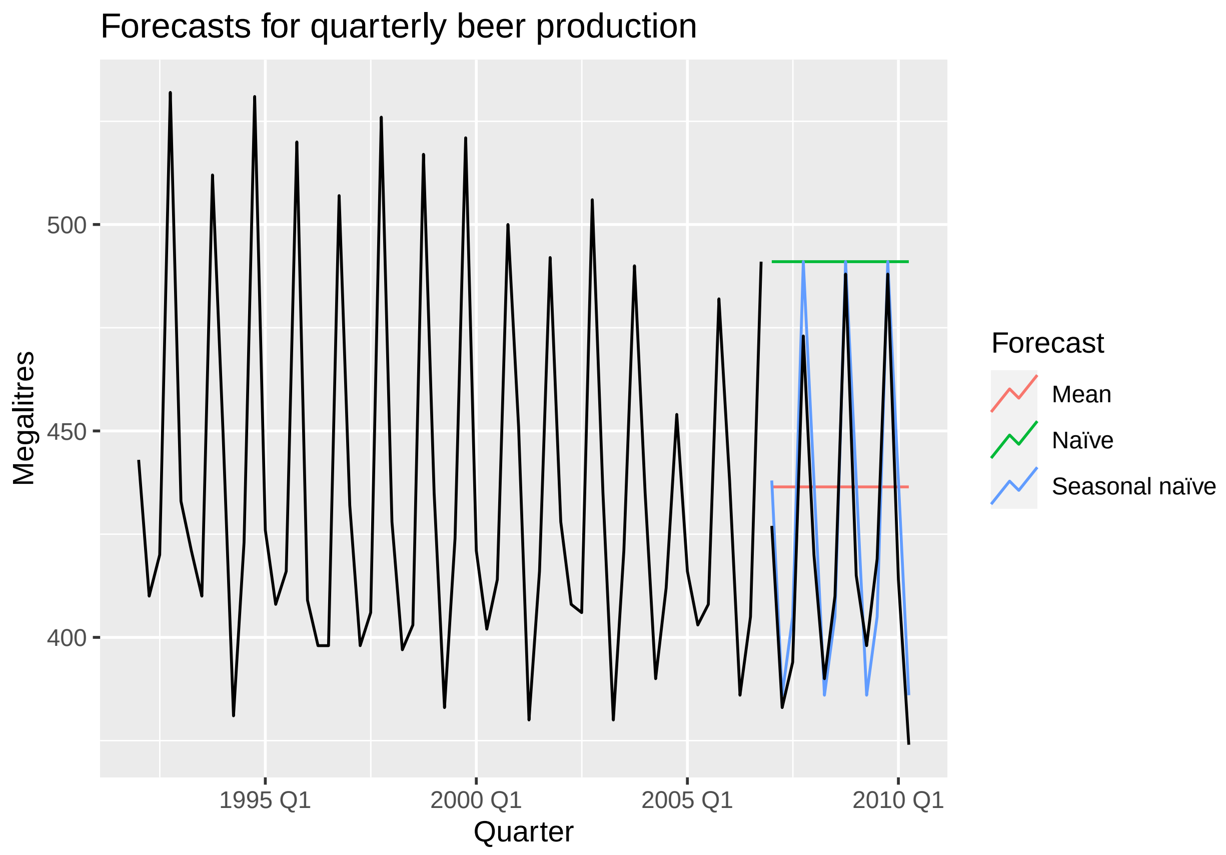
a naïve forecast is optimal when data follow a random walk (see Section 9.1), these are also called random walk forecasts and the RW() function can be used instead of NAIVE.

Example: Australian quarterly beer production

Figure 5.7 shows the first three methods applied to Australian quarterly beer production from 1992 to 2006, with the forecasts compared against actual values in the next 3.5 years.

```r
# Set training data from 1992 to 2006
train <- aus_production |>
  filter_index("1992 Q1" ~ "2006 Q4")
# Fit the models
beer_fit <- train |>
  model(
    Mean = MEAN(Beer),
    `Naïve` = NAIVE(Beer),
    `Seasonal naïve` = SNAIVE(Beer)
  )
# Generate forecasts for 14 quarters
beer_fc <- beer_fit |> forecast(h = 14)
# Plot forecasts against actual values
beer_fc |>
  autoplot(train, level = NULL) +
  autolayer(
    filter_index(aus_production, "2007 Q1" ~ .),
    colour = "black"
  ) +
  labs(
    y = "Megalitres",
    title = "Forecasts for quarterly beer production"
  ) +
  guides(colour = guide_legend(title = "Forecast"))
```

```
## Plot variable not specified, automatically selected `.vars = Beer`
```



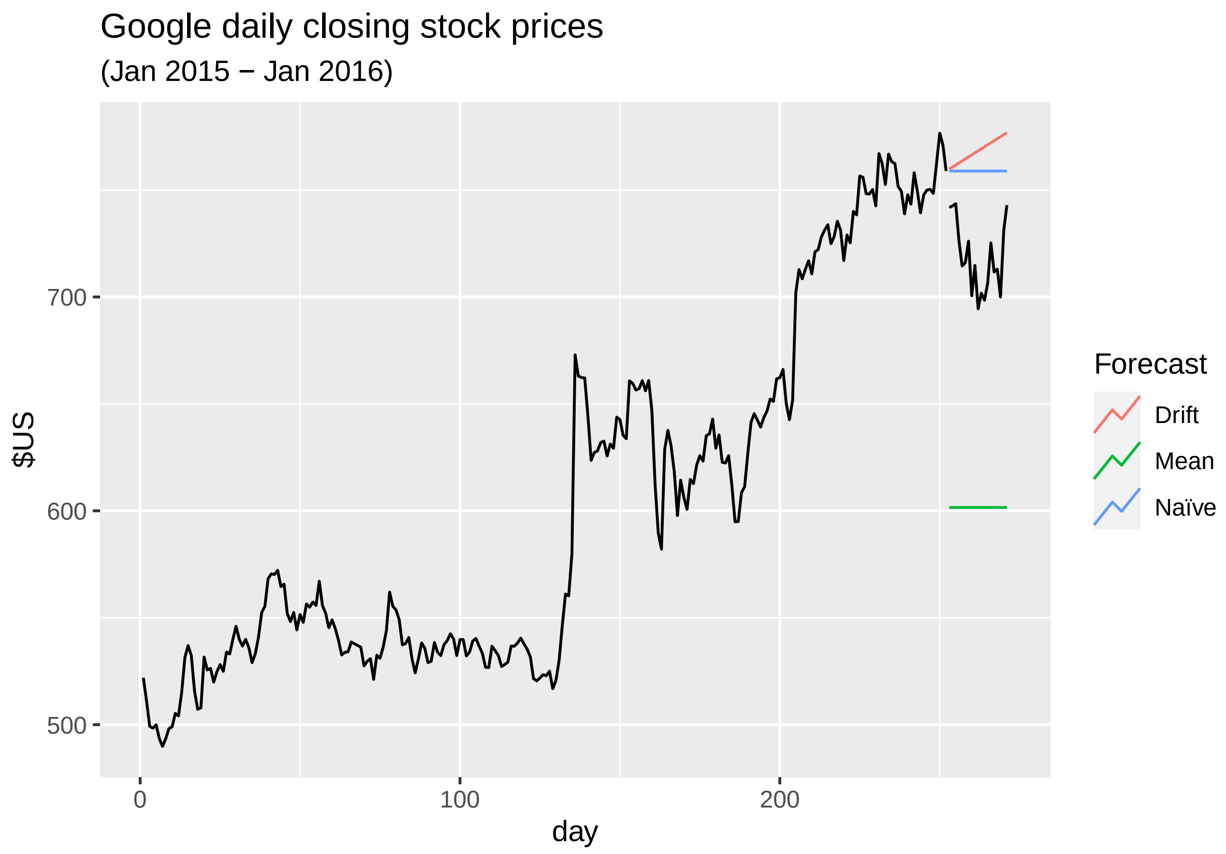
In this case, only the seasonal naïve forecasts are close to the observed values from 2007 onwards.

Example: Google's daily closing stock price

In Figure 5.8, the non-seasonal methods are applied to Google's daily closing stock price in 2015, and used to forecast one month ahead. Because stock prices are not observed every day, we first set up a new time index based on the trading days rather than calendar days.

```r
# Re-index based on trading days
google_stock <- gafa_stock |>
  filter(Symbol == "GOOG", year(Date) >= 2015) |>
  mutate(day = row_number()) |>
  update_tsibble(index = day, regular = TRUE)
# Filter the year of interest
google_2015 <- google_stock |> filter(year(Date) == 2015)
# Fit the models
google_fit <- google_2015 |>
  model(
    Mean = MEAN(Close),
    `Naïve` = NAIVE(Close),
    Drift = NAIVE(Close ~ drift())
  )
# Produce forecasts for the trading days in January 2016
google_jan_2016 <- google_stock |>
  filter(yearmonth(Date) == yearmonth("2016 Jan"))
google_fc <- google_fit |>
  forecast(new_data = google_jan_2016)
```

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



Sometimes one of these simple methods will be the best forecasting method available; but in many cases, these methods will serve as benchmarks rather than the method of choice. That is, any forecasting methods we develop will be compared to these simple methods to ensure that the new method is better than these simple alternatives. If not, the new method is not worth considering.

5.3 Fitted values and residuals

- $\hat{y}_{t|t-1}$ is the forecast of y_t based on the observations $y_1, ..., y_{t-1}$. It is expected value of the future value of y_t given everything all the information upto the period before $t-1$.
- We call these values as fitted values.
- Sometimes we drop the subscript: $\hat{y}_t \equiv \hat{y}_{t|t-1}$
- We call these values as fitted values and not forecasts because Often not true forecasts since many of the methods we use to generate these, we actually estimate the parameters are estimated using all the information on all data.
- *For example:*

(i) $\hat{y}_t = \bar{y}$ for average method.

(ii) $\hat{y}_t = y_{t-1} + \frac{y_T - y_1}{T-1}$ for drift method.

Examples

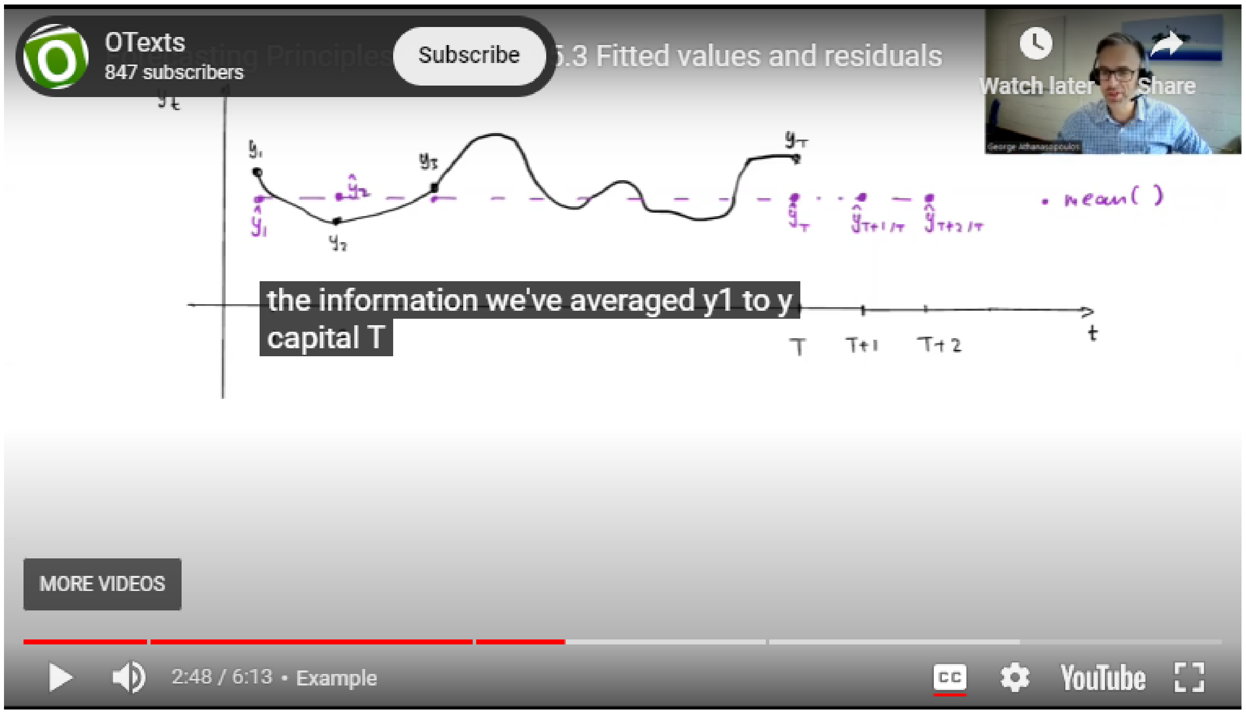


Figure 3: Mean Method

Here, we take the average from y_1 to y_T and all the fitted values lie on this average line. This is not really a true forecast.

Here, the fitted values are $\hat{y_{2|1}}$,$\hat{y_{3|2}}$, $\hat{y_{4|3}}$ etc.

Residuals in Forecasting

It is the difference between the observed value and its fitted value:

$$e_t = y_t - \hat{y}_{t|t-1}$$

Assumptions:

- Residuals e_t are uncorrelated. If they are not then then information left in residuals that should be used in computing forecasts.
- e_t have mean zero. If they don't then forecasts are biased.

Useful Properties (For distribution and prediction intervals):

- e_t have constant variance hence they are homoscedestic.
- e_t are normally distributed.

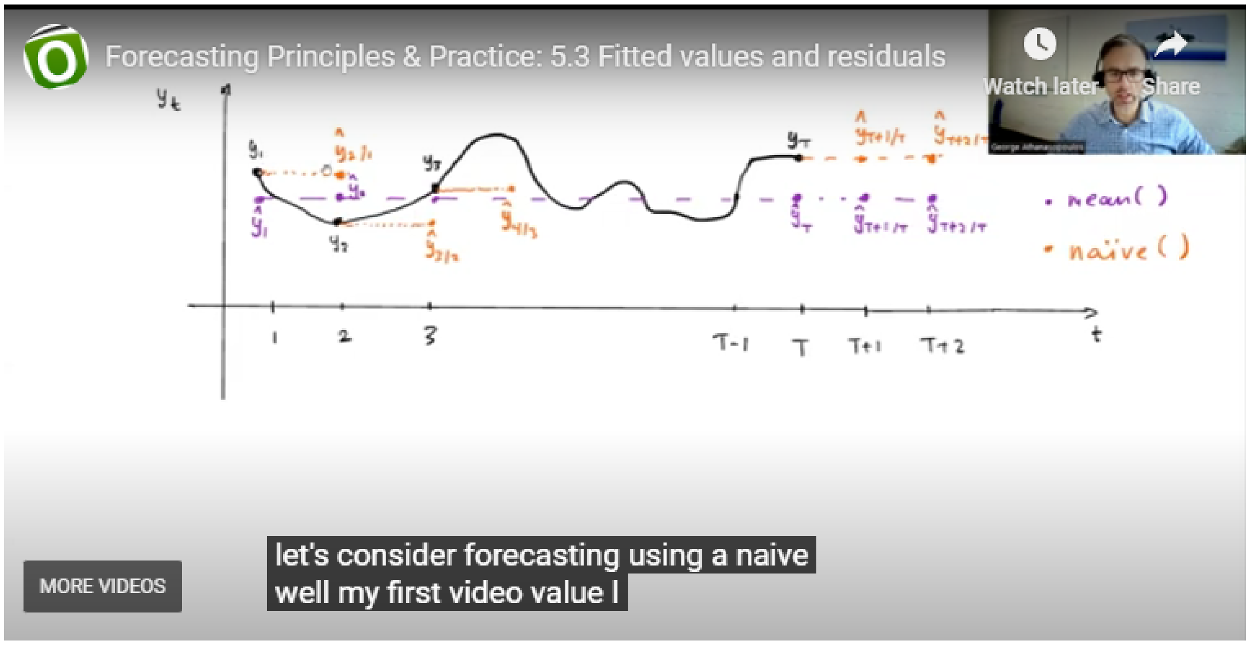

Forecasting Principles & Practice: 5.3 Fitted values and residuals
Watch later
Share
mean( )
naïve ( )
MORE VIDEOS
let's consider forecasting using a naive
well my first video value I

Figure 4: Naive Method

The fitted values and residuals from a model can be obtained using the augment() function. In the beer production example in Section 5.2, we saved the fitted models as beer_fit. So we can simply apply `augment()` to this object to compute the fitted values and residuals for all models.

```
augment(beer_fit)
```

```
## # A tsibble: 180 x 6 [1Q]
## # Key:       .model [3]
##    .model Quarter  Beer .fitted .resid .innov
##    <chr>    <qtr> <dbl>   <dbl>  <dbl>  <dbl>
##  1 Mean   1992 Q1   443    436.   6.55   6.55
##  2 Mean   1992 Q2   410    436. -26.4  -26.4
##  3 Mean   1992 Q3   420    436. -16.4  -16.4
##  4 Mean   1992 Q4   532    436.  95.6   95.6
##  5 Mean   1993 Q1   433    436.  -3.45  -3.45
##  6 Mean   1993 Q2   421    436. -15.4  -15.4
##  7 Mean   1993 Q3   410    436. -26.4  -26.4
##  8 Mean   1993 Q4   512    436.  75.6   75.6
##  9 Mean   1994 Q1   449    436.  12.6   12.6
## 10 Mean   1994 Q2   381    436. -55.4  -55.4
## # i 170 more rows
```

There are three new columns added to the original data:

- fitted contains the fitted values;
- resid contains the residuals;
- innov contains the “innovation residuals” which, in this case, are identical to the regular residuals.

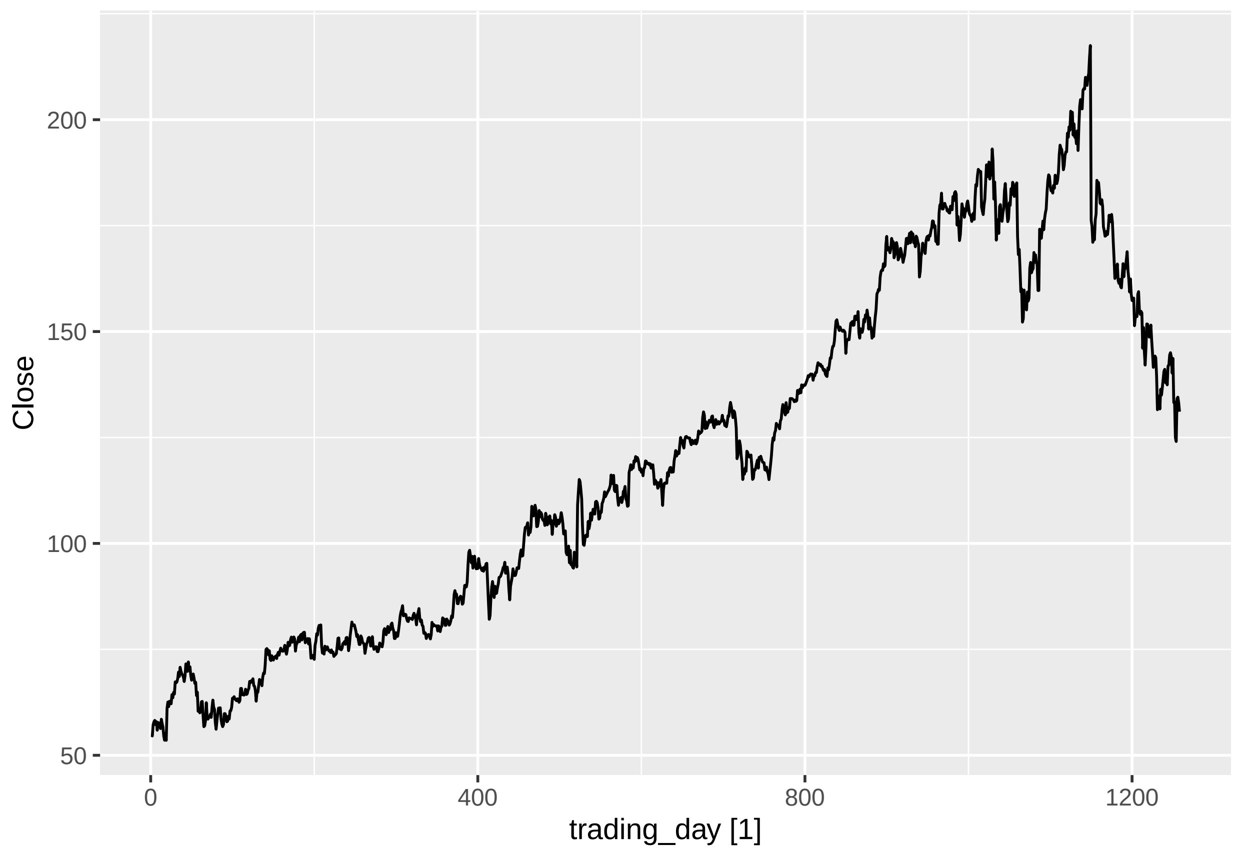
Residuals are useful in checking whether a model has adequately captured the information in the data. For this purpose, we use innovation residuals.

If patterns are observable in the innovation residuals, the model can probably be improved. We will look at some tools for exploring patterns in residuals in the next section.

5.4 Residual diagnostics

Example: Facebook closing stock price

```
fb_stock |> autoplot(Close)
```



Here, we are going to fit a model and check the residuals and look at the residuals of the model and check whether they satisfy the assumptions which we made.

```
fit <- fb_stock |> model(NAIVE(Close))
augment(fit)
```

```
## # A tsibble: 1,258 x 7 [1]
## # Key:       Symbol, .model [1]
##    Symbol .model       trading_day Close .fitted .resid .innov
##    <chr>  <chr>              <int> <dbl>   <dbl>  <dbl>  <dbl>
##  1 FB     NAIVE(Close)           1  54.7    NA   NA     NA
##  2 FB     NAIVE(Close)           2  54.6    54.7 -0.150 -0.150
##  3 FB     NAIVE(Close)           3  57.2    54.6  2.64   2.64
##  4 FB     NAIVE(Close)           4  57.9    57.2  0.720  0.720
##  5 FB     NAIVE(Close)           5  58.2    57.9  0.310  0.310
##  6 FB     NAIVE(Close)           6  57.2    58.2 -1.01  -1.01
##  7 FB     NAIVE(Close)           7  57.9    57.2  0.720  0.720
##  8 FB     NAIVE(Close)           8  55.9    57.9 -2.03  -2.03
##  9 FB     NAIVE(Close)           9  57.7    55.9  1.83   1.83
## 10 FB     NAIVE(Close)          10  57.6    57.7 -0.140 -0.140
## # i 1,248 more rows
```

Here we are using the NAIVE model. It is very common model to use for the stock price data. So, here we got the fitted and residuals values and innov values.

When we are testing then we use the .innov values column. Here, we have residual and innov column have the same values but but for different for certain types of models or we have done a transformation of the data before we fit a model. In this case we have not done any transformation and we are using a straight forward additive model so two columns are same.

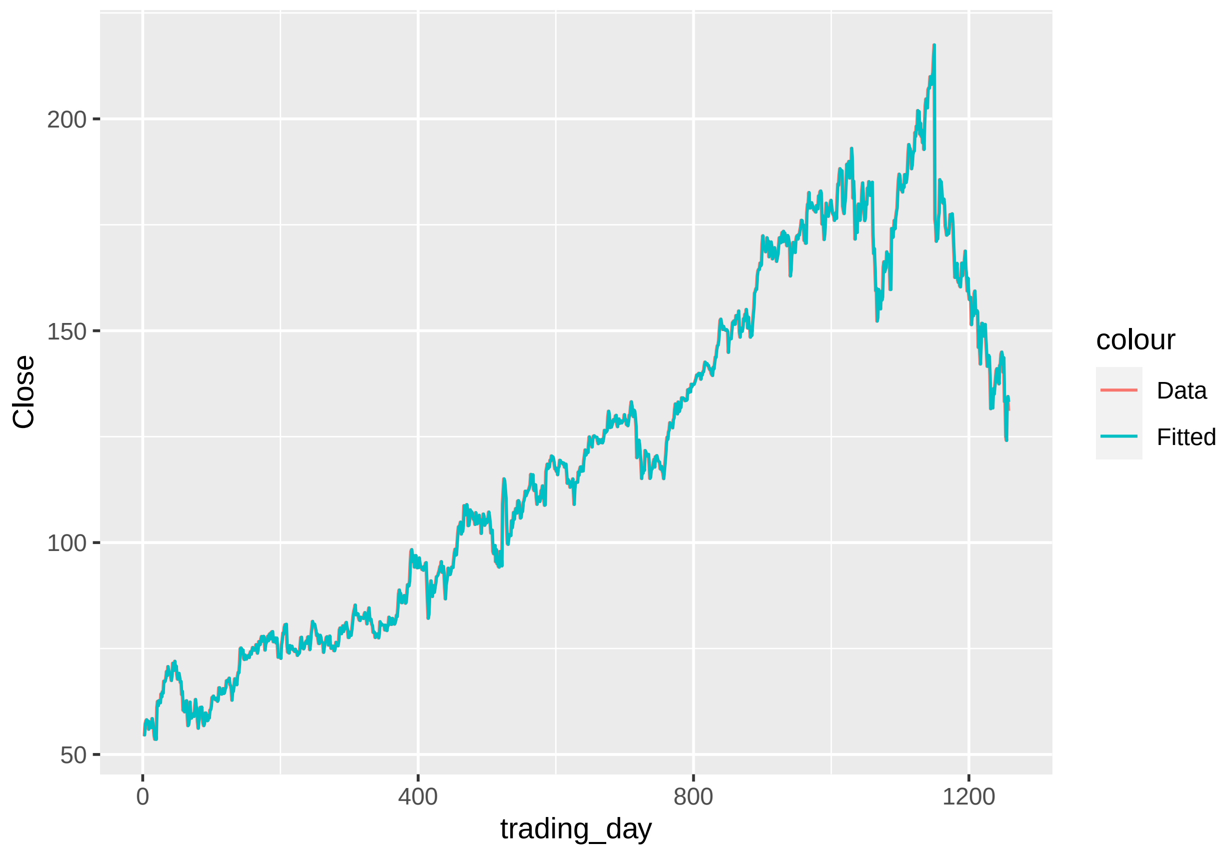
Here first value is missing because we can't get the forecast for the forecast using the NAIVE method because we don't know what is happening before that observation.

Here, fitted value means $\hat{y}_{t|t-1} = y_{t-1}$ and residual means $e_t == y_t - \hat{y}_{t|t-1} = y_t - y_{t-1}$. We can check in the above table.

Now first we plot the observations.

```r
augment(fit) |>
  ggplot(aes(x=trading_day))+
  geom_line(aes(y=Close, colour="Data"))+
  geom_line(aes(y=.fitted, colour="Fitted"))
```

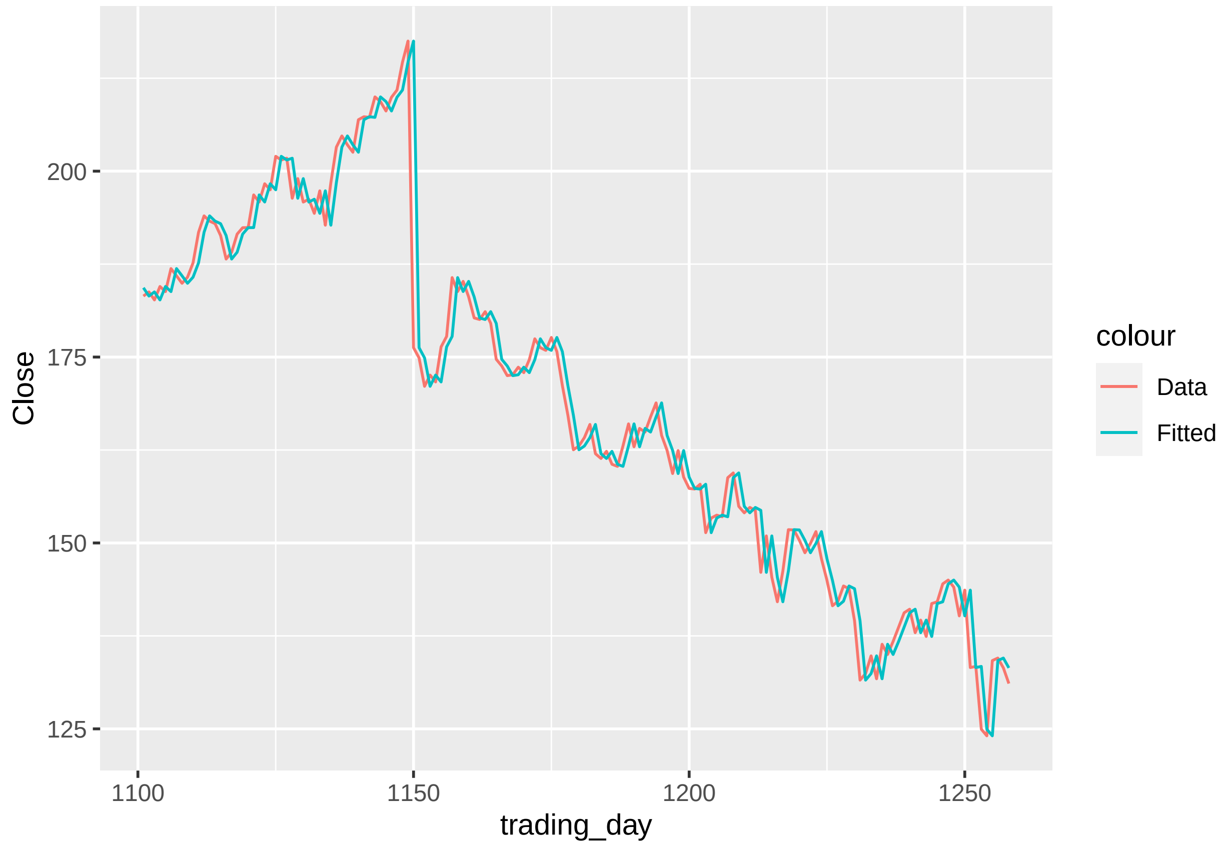
```
## Warning: Removed 1 row containing missing values (`geom_line()`).
```



They are very very close because they does not change day to day and so the observation yesterday is quite similar to the observation today and so these two plots are very similar.

If we take the vertical differences.

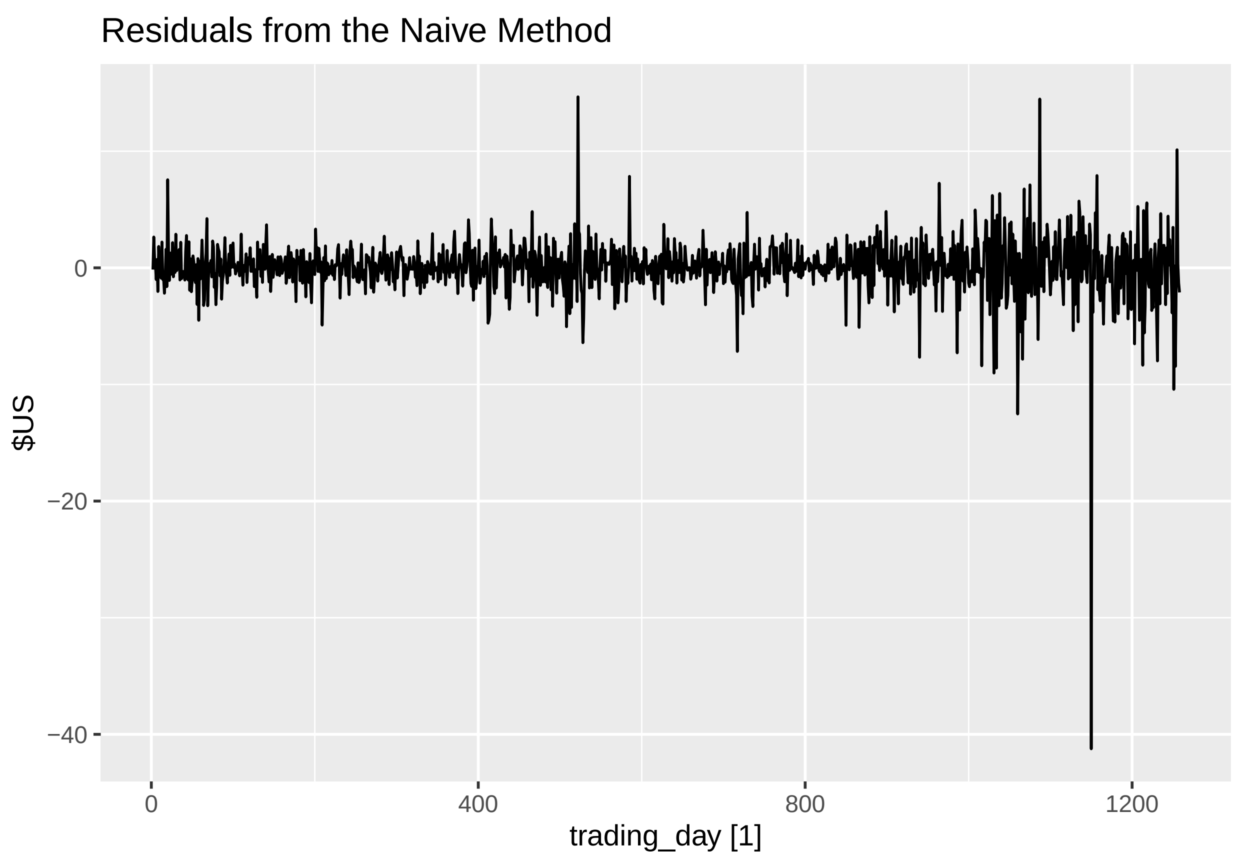
```r
augment(fit) |>
  filter(trading_day>1100) |>
  ggplot(aes(x=trading_day))+
  geom_line(aes(y=Close, colour="Data"))+
  geom_line(aes(y=.fitted, colour="Fitted"))
```



Now, check the residuals.

```r
augment(fit) |>
  autoplot(.resid)+
  labs(y="$US",
       title="Residuals from the Naive Method")
```

```
## Warning: Removed 1 row containing missing values (`geom_line()`).
```

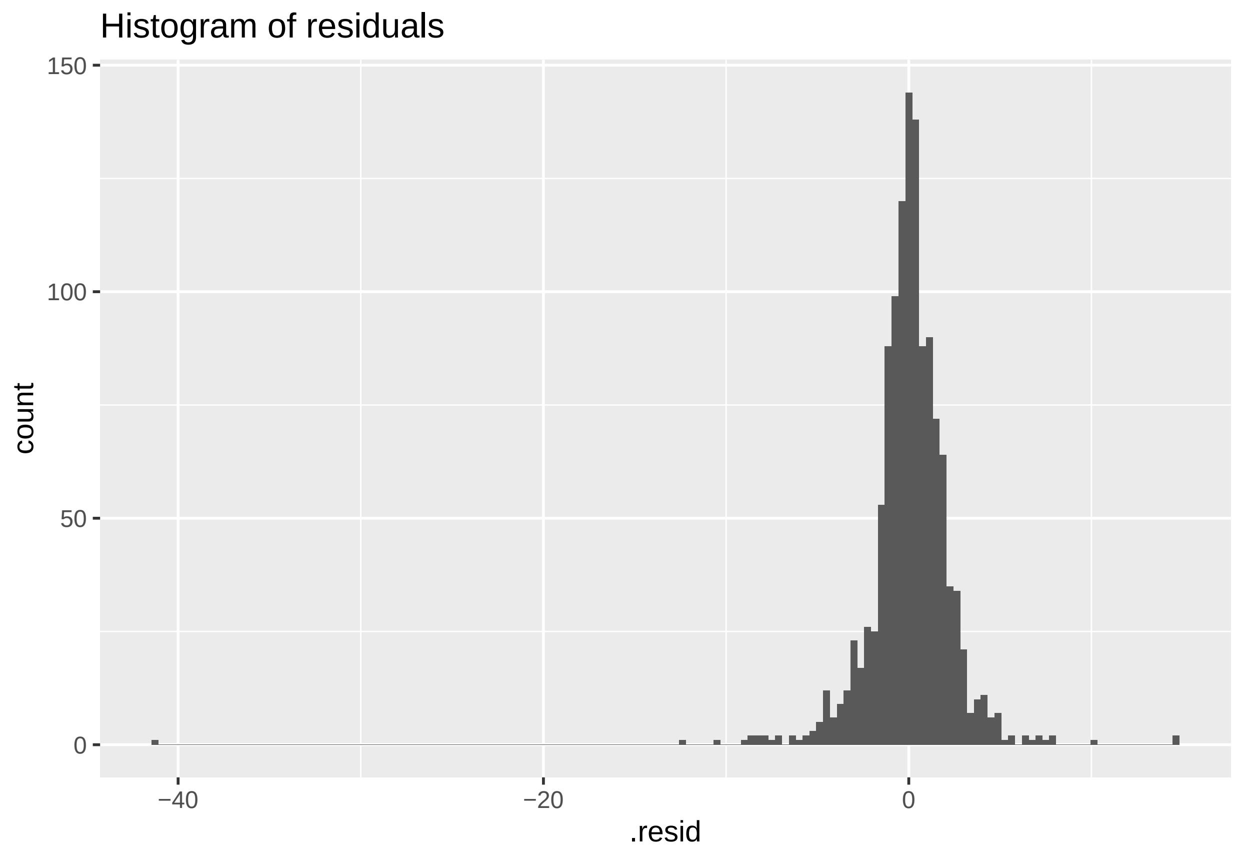


Here, we are getting spikes for large values in trading days and all these things suggest that our assumptions are not right here.

Now, make the histogram from the residuals.

```r
augment(fit) |>
  ggplot(aes(x=.resid)) +
  geom_histogram(bins=150)+
  labs(title = "Histogram of residuals")
```

```
## Warning: Removed 1 rows containing non-finite values (`stat_bin()`).
```

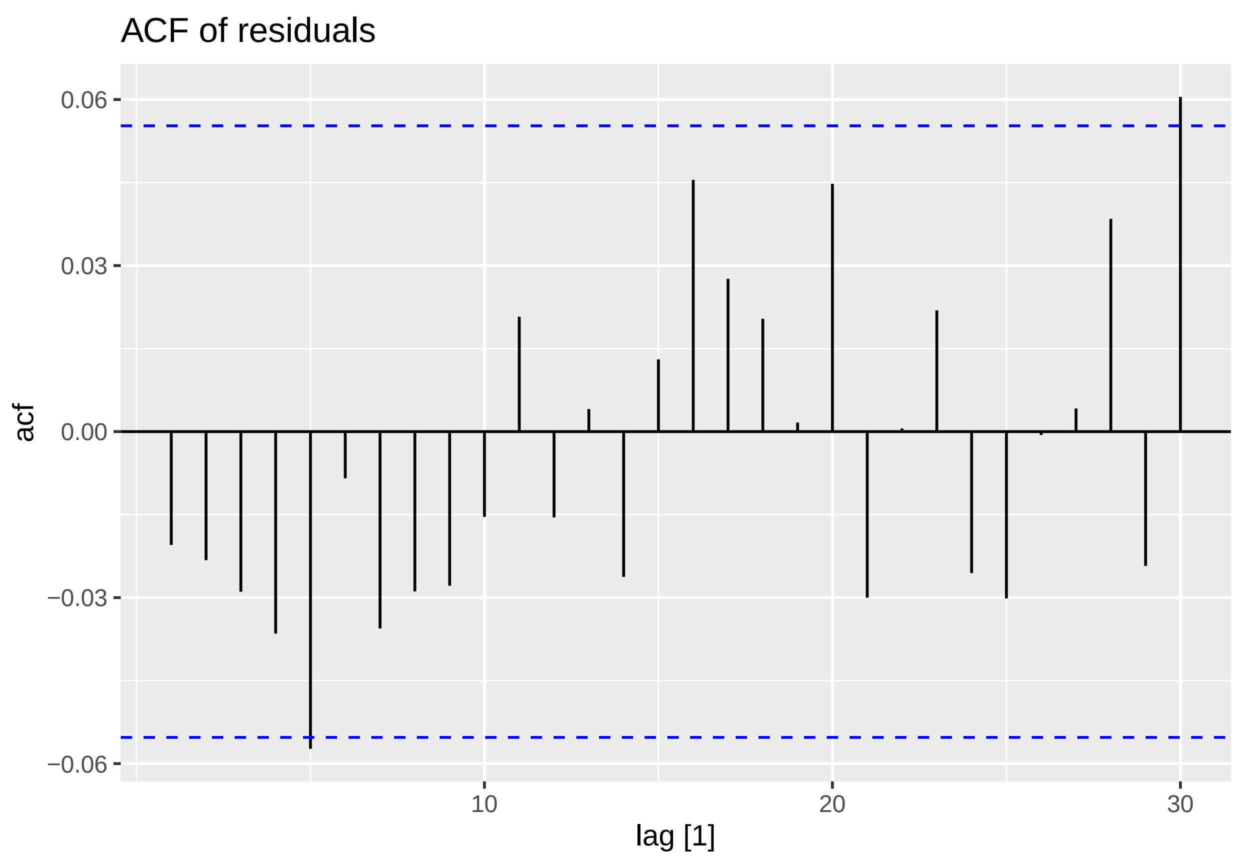


bins is the number of bars in the histogram. if we don't give the value to it then it takes the default value.

These residuals looks normal but it is not normal because it is having outliers and also it is little more peaked than what we should we expect.

Now, we look at the acf of the residuals.

```r
augment(fit) |>
  ACF(.resid) |>
  autoplot()+labs(title = "ACF of residuals")
```



Here, we get two spikes above the limit out of 30 spikes which is not unusual.

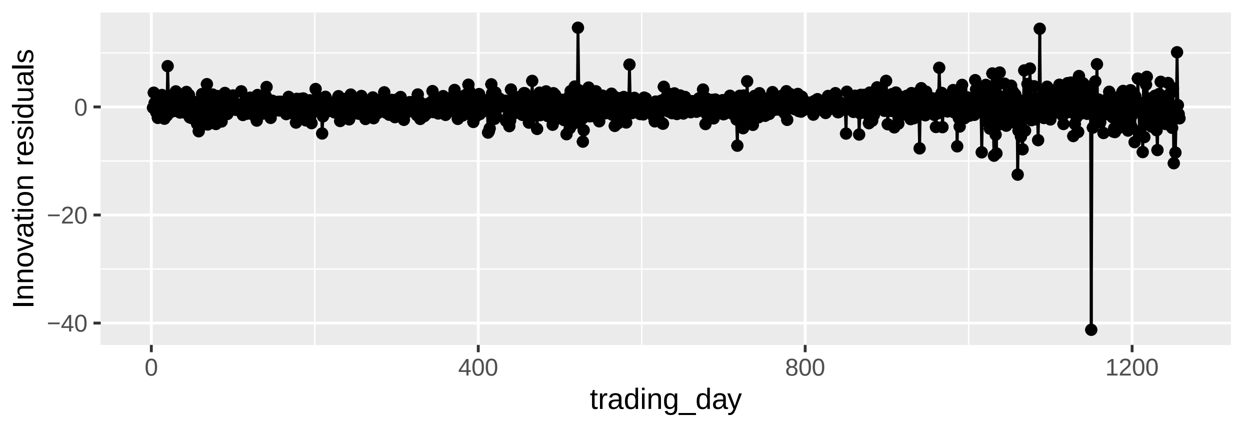
Now, we can plot all the graphs as:

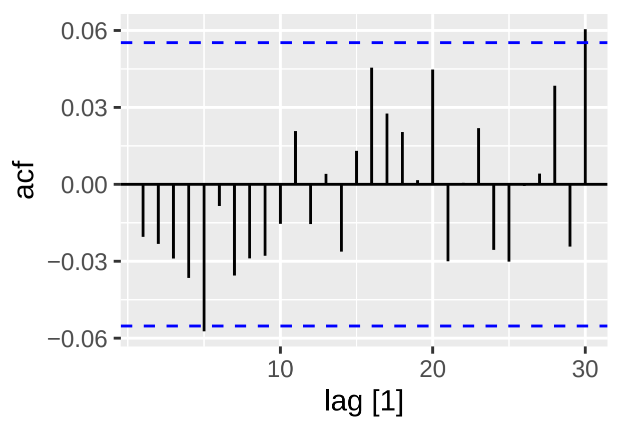
```
gg_tsresiduals(fit)
```

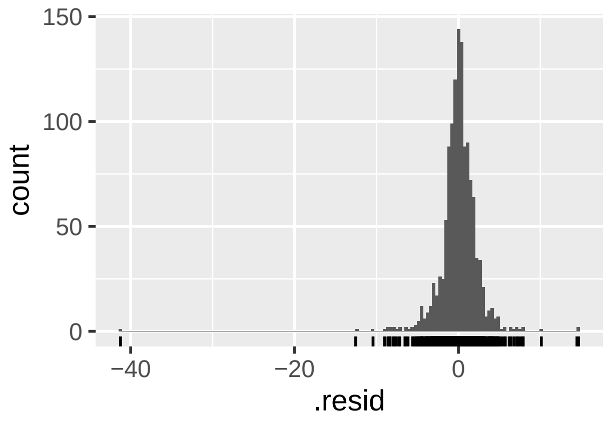
```
## Warning: Removed 1 row containing missing values (`geom_line()`).

## Warning: Removed 1 rows containing missing values (`geom_point()`).

## Warning: Removed 1 rows containing non-finite values (`stat_bin()`).
```







ACF of residuals

- We assume that residuals are white noise (uncorrelated, mean zero and constant variance). If they are not then there is information left in the residuals that should be used in computing forecasts and make the forecasts better.
- So a standard residual diagnostic is to check the ACF of the residuals of a forecasting method. Using autocorrelation, we are also checking whether lag 1 is significant or lag 2 is significant or lag 3 is significant etc.
- We `expect` these to look like white noise.

Portmanteau tests

The word Portmanteau refers to a suitcase which carries everything together. So this test combines a lot of things. So this test do all the correlations simultaneously.

r_k = autocorrelation of residuals at lag k

Consider a whole set of r_k values, and develop a test to see whether the set is significantly different from a zero set.

(1) Box-Pierce test

- It is developed by George Box (same in box-cot transformation) and his PhD student Pierce.
- The idea is if we squared all the correlations up to some upper limit say lag "l". If we multiply it by the number of observations then we get a thing that has a chi-squared distribution.

- We can compare it against the probabilities that we would expect from the chi-squared distribution to see whether the value of Q for our data set is bigger than what we would expect by chance. If it is a small Q then the sort of things which we expect by chance and so we can say well the test is insignificant. If it is a large Q bigger than what we would expect by chance then we say there is a significant autocorrelation is going on here. So Summary is:

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

where l is the max lag being considered and T is the number of observations.

- If each r_k close to zero then Q will be *small*.
- If some r_k values large (positive or negative), Q will be *large*.

After this test, Box with his another PhD student comes with new test which gives little more accurate result and that's the one we will use.

Ljung-Box test

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

where l is the max lag being considered and T is the number of observations.

- My preference (based on experiments): $l = 10$ for non-seasonal data, $l = 2m$ for seasonal data (where m is seasonal period, so for quarterly data, m=4 and for monthly data, m=12)
- Better performance, especially in small samples.
- Here, $(T+k)^{-1}$ is a weight and this test has value which also has a small chi-squared distribution.
- If the data or residuals are white noise (WN) then Q^* has χ^2 distribution with l degrees of freedom and we compute the probabilities of getting a value as large under the assumption of white noise residuals.
- lag=l

```
augment(fit) |>
  features(.resid, ljung_box,lag=10)
```

```
## # A tibble: 1 x 4
##   Symbol .model       lb_stat lb_pvalue
##   <chr>  <chr>          <dbl>     <dbl>
## 1 FB     NAIVE(Close)    12.1     0.276
```

Since it is non-seasonal data so we are using lag=10.

- Here, FB is the key.
- Here, "lb_stat" is Q^* and "lb_pvalue" is a probability of gettting a Q^* as bis as this if the residuals were really white noise and so we reject the hypothesis of white noise if this p-value is less than 0.05 (a arbitrary threshold) that represents very unlikely to occur by chance but here p-value is 0.276 so very likely to occur by chance so these residuals are not easily distinguished from white noise so we accept the white noise hypothesis.

So, when we fit a model and we are ready to do some forecasting then it is a good idea to check your residuals to make sure it satisfies the assumptions. In this case, autocorrelation is satisfied but not the normality of the residuals and that will affect the size of the prediction intervals much more than the point forecasts.

————————- Summary ————————

A good forecasting method will yield innovation residuals with the following properties:

(1) The innovation residuals are uncorrelated. If there are correlations between innovation residuals, then there is information left in the residuals which should be used in computing forecasts.

(2) The innovation residuals have zero mean. If they have a mean other than zero, then the forecasts are biased.

Any forecasting method that does not satisfy these properties can be improved. However, that does not mean that forecasting methods that satisfy these properties cannot be improved. It is possible to have several different forecasting methods for the same data set, all of which satisfy these properties. Checking these properties is important in order to see whether a method is using all of the available information, but it is not a good way to select a forecasting method.

If either of these properties is not satisfied, then the forecasting method can be modified to give better forecasts. Adjusting for bias is easy: if the residuals have mean m, then simply add m to all forecasts and the bias problem is solved. Fixing the correlation problem is harder, and we will not address it until Chapter 10.

In addition to these essential properties, it is useful (but not necessary) for the residuals to also have the following two properties.

(3) The innovation residuals have constant variance. This is known as "homoscedasticity".

(4) The innovation residuals are normally distributed.

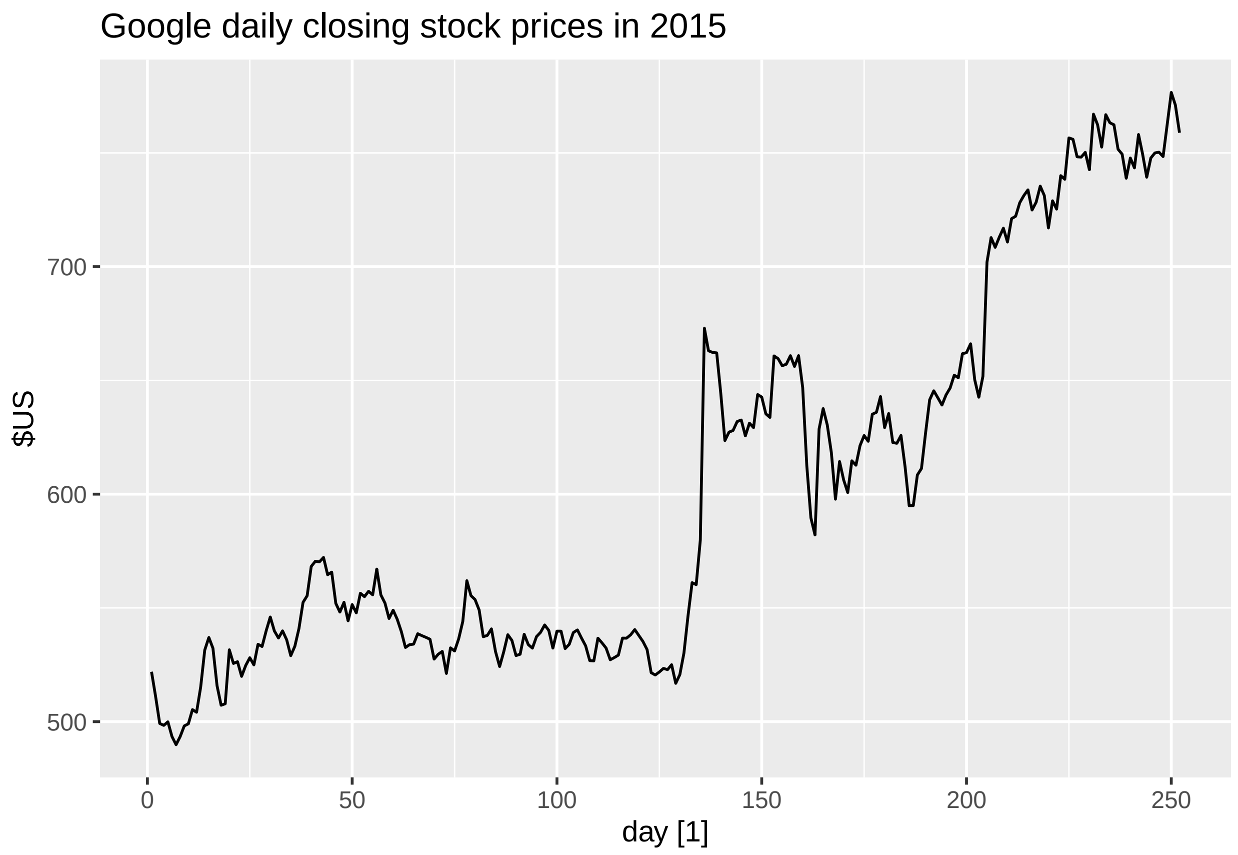
These two properties make the calculation of prediction intervals easier (see Section 5.5 for an example). However, a forecasting method that does not satisfy these properties cannot necessarily be improved. Sometimes applying a Box-Cox transformation may assist with these properties, but otherwise there is usually little that you can do to ensure that your innovation residuals have constant variance and a normal distribution. Instead, an alternative approach to obtaining prediction intervals is necessary. We will show how to deal with non-normal innovation residuals in Section 5.5.

Example: Forecasting Google daily closing stock prices

We will continue with the Google daily closing stock price example from Section 5.2. For stock market prices and indexes, the best forecasting method is often the naïve method. That is, each forecast is simply equal to the last observed value, or $\hat{y}_t = y_{t-1}$. Hence, the residuals are simply equal to the difference between consecutive observations:

The following graph shows the Google daily closing stock price for trading days during 2015. The large jump corresponds to 17 July 2015 when the price jumped 16% due to unexpectedly strong second quarter results.

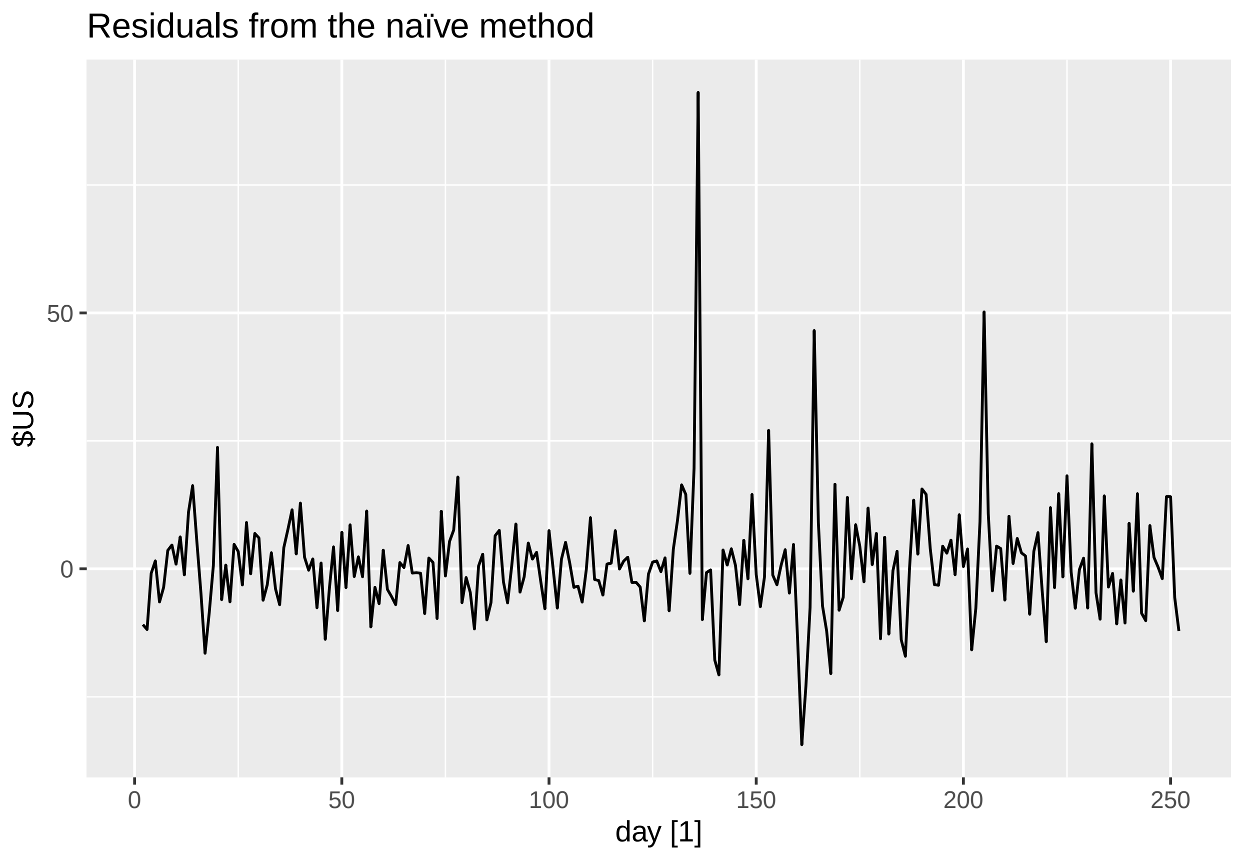
```r
autoplot(google_2015, Close) +
  labs(y = "$US",
       title = "Google daily closing stock prices in 2015")
```



The residuals obtained from forecasting this series using the naïve method are shown in Figure 5.10. The large positive residual is a result of the unexpected price jump in July.

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

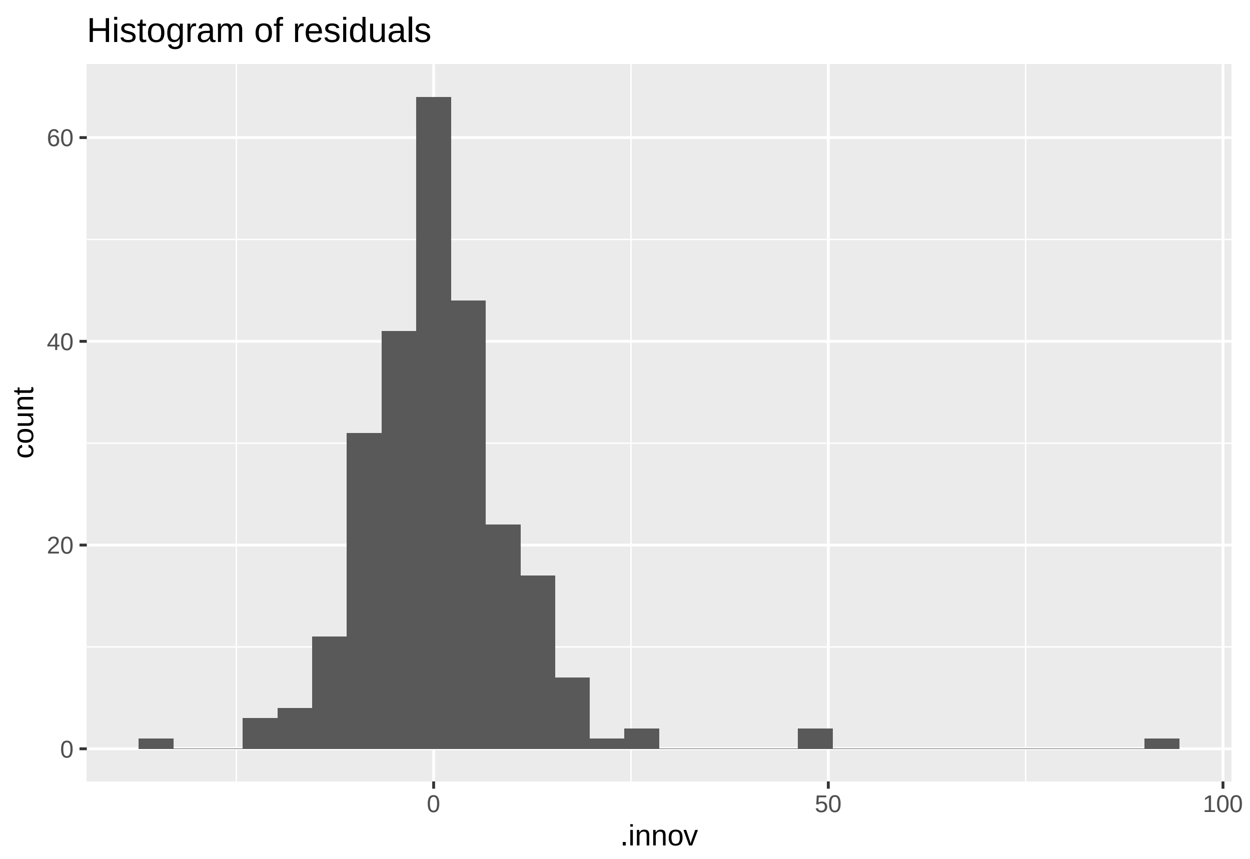
```
## Warning: Removed 1 row containing missing values (`geom_line()`).
```



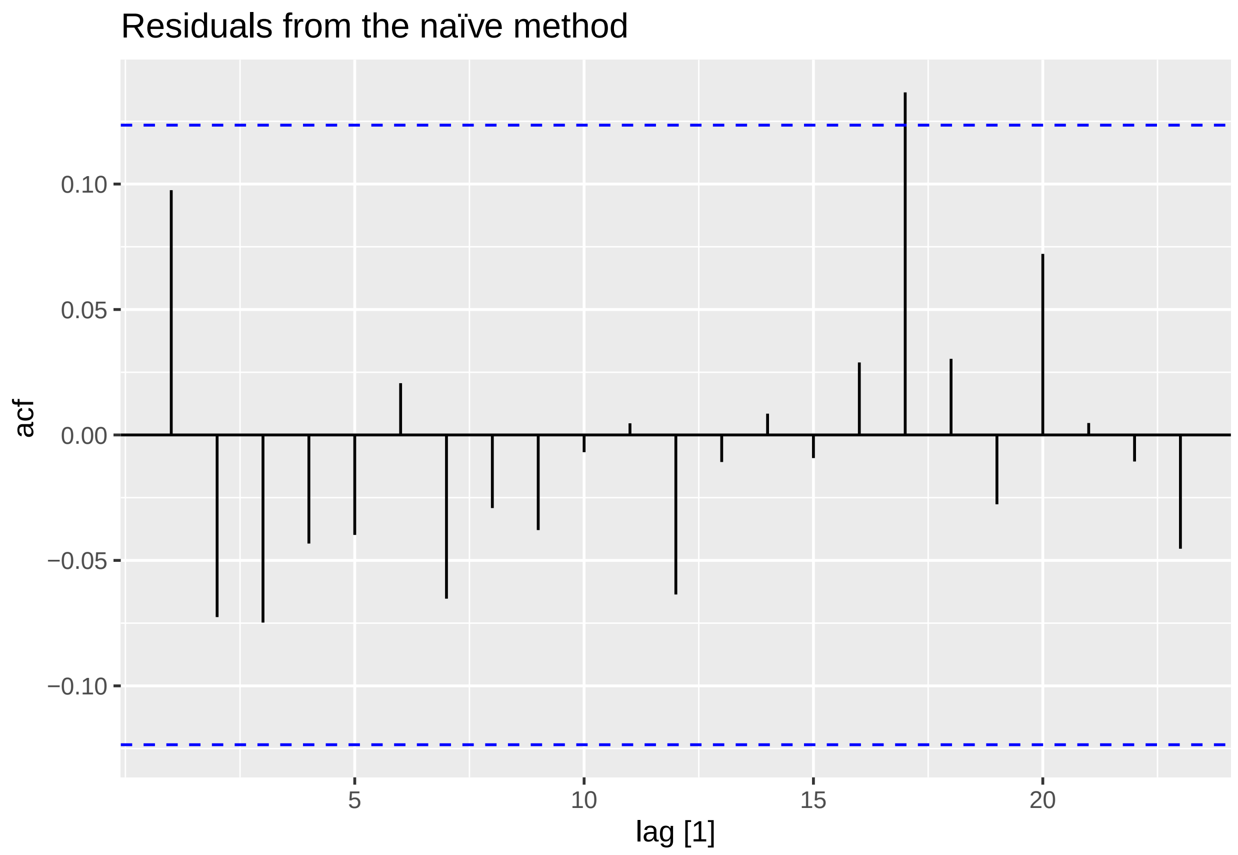
```r
aug |>
  ggplot(aes(x = .innov)) +
  geom_histogram() +
  labs(title = "Histogram of residuals")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 1 rows containing non-finite values (`stat_bin()`).
```



```
aug |>
  ACF(.innov) |>
  autoplot() +
  labs(title = "Residuals from the naïve method")
```



These graphs show that the naïve method produces forecasts that appear to account for all available information. The mean of the residuals is close to zero and there is no significant correlation in the residuals series. The time plot of the residuals shows that the variation of the residuals stays much the same across the historical data, apart from the one outlier, and therefore the residual variance can be treated as constant. This can also be seen on the histogram of the residuals. The histogram suggests that the residuals may not be normal — the right tail seems a little too long, even when we ignore the outlier. Consequently, forecasts from this method will probably be quite good, but prediction intervals that are computed assuming a normal distribution may be inaccurate.

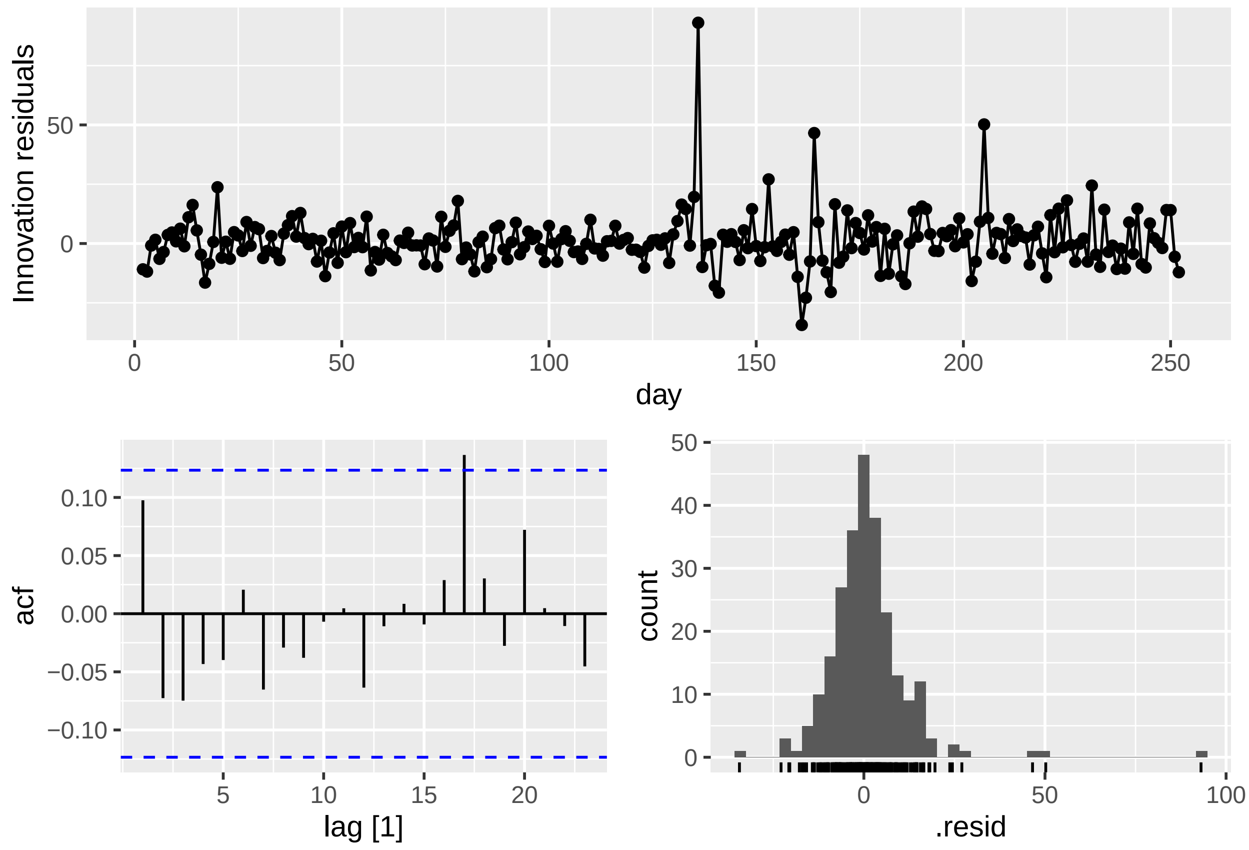
A convenient shortcut for producing these residual diagnostic graphs is the gg_tsresiduals() function, which will produce a time plot, ACF plot and histogram of the residuals.

```
google_2015 |>
  model(NAIVE(Close)) |>
  gg_tsresiduals()
```

```
## Warning: Removed 1 row containing missing values (`geom_line()`).
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```

```
## Warning: Removed 1 rows containing non-finite values (`stat_bin()`).
```



Portmanteau tests for autocorrelation

In addition to looking at the ACF plot, we can also do a more formal test for autocorrelation by considering a whole set of r_k values as a group, rather than treating each one separately.

Recall that r_k is the autocorrelation for lag k

When we look at the ACF plot to see whether each spike is within the required limits, we are implicitly carrying out multiple hypothesis tests, each one with a small probability of giving a false positive. When enough of these tests are done, it is likely that at least one will give a false positive, and so we may conclude that the residuals have some remaining autocorrelation, when in fact they do not.

In order to overcome this problem, we test whether the first l autocorrelations are significantly different from what would be expected from a white noise process. A test for a group of autocorrelations is called a portmanteau test, from a French word describing a suitcase or coat rack carrying several items of clothing.

One such test is the Box-Pierce test

If each r_k is close to zero, then Q will be small. If some r_k values are large (positive or negative), then Q will be large. We suggest using $l = 10$ 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 $l = T/5$

A related (and more accurate) test is the Ljung-Box test.

Again, large values of Q^* suggest that the autocorrelations do not come from a white noise series.

How large is too large? If the autocorrelations did come from a white noise series, then both Q and Q^* would have a χ^2 distribution with l degrees of freedom

```
aug |> features(.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
```

```
aug |> features(.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
```

For both Q and Q^*, the results are not significant (i.e., the p-values are relatively large). Thus, we can conclude that the residuals are not distinguishable from a white noise series.

An alternative simple approach that may be appropriate for forecasting the Google daily closing stock price is the drift method. The `tidy()` function shows the one estimated parameter, the drift coefficient, measuring the average daily change observed in the historical data.

```
fit <- google_2015 |> model(RW(Close ~ drift()))
tidy(fit)
```

```
## # 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
```

Applying the Ljung-Box test, we obtain the following result.

```
augment(fit) |> features(.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
```

As with the naïve method, the residuals from the drift method are indistinguishable from a white noise series.

5.5 Distributional forecasts and prediction intervals

So far we have talked about the point forecast, now we discuss about the probability distribution of the forecast.

- A point forecast $\hat{y}_{T+h|T}$ is (usually) the mean of the conditional probability distribution $y_{T+h}|y_1, y_2, ..., y_T$ but we are also interested in the whole probability distribution especially so that we can get the prediction intervals.
- Most time series models produce normally distributed forecasts provided the residuals of the model are normally distributed. If you have done the transformation before the modelling then they will be normally distributed on the transform scale and when we back transform them, we get a transform normal distribution.
- The forecast distribution describes the probability of observing any future value. So it gives a way to compute things like 95% prediction intervals which contain the true value with 95% probability.

Forecast Distributions:

Assuming residuals are normal, uncorrelated, $sd = \hat{\sigma}$ (standard deviation is constant that means it is homoscadestic) then we can derive the probability distributions for the 4 benchmark methods.

These assumptions are important because if these assumptions are not true then the following statements are not true is some way. But assuming that residuals are normal and there is no autocorrelation in them and there is constant variance then we can prove mathematically the following distributions:

(1) Mean: $y_{T+h|T} \sim N(\bar{y}, (1+\frac{1}{T})\hat{\sigma}^2)$

It does not depend on "h" and so same prediction interval for all future values.

(2) Naive: $y_{T+h|T} \sim N(y_T, h\hat{\sigma}^2)$

Here variance of the forecast distribution increases linearly with h.

(3) Seasonal Naive: $y_{T+h|T} \sim N(y_{T+h-m(k+1)}, (k+1)\hat{\sigma}^2)$

It is close to linear increase with "h" because "k" is integer part of "h".

(4) Drift: $y_{T+h|T} \sim N(y_T + \frac{h}{T-1}(y_T - y_1), h\frac{T+h}{T}\hat{\sigma}^2)$

Here also, variance of the forecast distribution increases linearly with h.

where k is the integer part of $\frac{h-1}{m}$ and h is the forecast horizon.

Note that when $h = 1$ and T is large (means lots of data) then these all give the same approximate forecast variance: $\hat{\sigma}^2$ and this is true for all forecasting models not only the above 4 models.

Prediction Intervals

Once we know the forecast distribution then we can get the prediction intervals and if the forecast distribution is normal as it usually is for the models, then we can simply write down an expression as the point forecast plus/minus some number times the forecast variance and that number depends on the level means how wide or what probability coverage you want your prediction interval to have. If it is 95% then it is 1.96.

Remember that a prediction interval gives a region within which we expect a true value to lie with a specific probability. It's always possible that the true value lie outside the given range but hopefully the probability coverage that you think you have got it is actually what happens in practice. So,

- A prediction interval gives a region within which we expect y_{T+h} to lie with a specified probability.
- Assuming the forecast errors are normally distributed, then a 95% PI(prediction interval) is:

$$\hat{y}_{T+h|T} \pm 1.96\hat{\sigma}^2$$

where $\hat{\sigma}_h$ is the standard deviation of the h-step distribution.

- When $h = 1$ then $\hat{\sigma}_h$ can be estimated from the residuals.

When we use fable then these things are computed automatically and we don't have to compute by hand.

Example

```
aus_production |>
  filter(!is.na(Bricks))  |>
  model(Seasonal_Naive=SNAIVE(Bricks)) |>
  forecast(h="5 years")
```

```
## # A fable: 20 x 4 [1Q]
## # Key:     .model [1]
##    .model         Quarter        Bricks .mean
##    <chr>            <qtr>        <dist> <dbl>
##  1 Seasonal_Naive 2005 Q3  N(428, 2336)   428
##  2 Seasonal_Naive 2005 Q4  N(397, 2336)   397
##  3 Seasonal_Naive 2006 Q1  N(355, 2336)   355
##  4 Seasonal_Naive 2006 Q2  N(435, 2336)   435
##  5 Seasonal_Naive 2006 Q3  N(428, 4672)   428
##  6 Seasonal_Naive 2006 Q4  N(397, 4672)   397
##  7 Seasonal_Naive 2007 Q1  N(355, 4672)   355
##  8 Seasonal_Naive 2007 Q2  N(435, 4672)   435
##  9 Seasonal_Naive 2007 Q3  N(428, 7008)   428
## 10 Seasonal_Naive 2007 Q4  N(397, 7008)   397
## 11 Seasonal_Naive 2008 Q1  N(355, 7008)   355
## 12 Seasonal_Naive 2008 Q2  N(435, 7008)   435
## 13 Seasonal_Naive 2008 Q3  N(428, 9343)   428
## 14 Seasonal_Naive 2008 Q4  N(397, 9343)   397
## 15 Seasonal_Naive 2009 Q1  N(355, 9343)   355
## 16 Seasonal_Naive 2009 Q2  N(435, 9343)   435
## 17 Seasonal_Naive 2009 Q3 N(428, 11679)   428
## 18 Seasonal_Naive 2009 Q4 N(397, 11679)   397
## 19 Seasonal_Naive 2010 Q1 N(355, 11679)   355
## 20 Seasonal_Naive 2010 Q2 N(435, 11679)   435
```

Here, seasonal naive distribution for bricks is computed using the previous formulas.

If you want to calculate the prediction interval then you can use the hilo() function.

```
aus_production |>
  filter(!is.na(Bricks)) |>
  model(Seasonal_naive = SNAIVE(Bricks)) |>
  forecast(h= "5 years") |>
  hilo(level=95) |>
  mutate(lower = `95%`$lower, upper=`95%`$upper)
```

```
## # A tsibble: 20 x 7 [1Q]
## # Key:       .model [1]
##    .model         Quarter        Bricks .mean                  `95%` lower upper
##    <chr>            <qtr>        <dist> <dbl>                 <hilo> <dbl> <dbl>
##  1 Seasonal_naive 2005 Q3  N(428, 2336)   428 [333.2737, 522.7263]95  333.  523.
##  2 Seasonal_naive 2005 Q4  N(397, 2336)   397 [302.2737, 491.7263]95  302.  492.
##  3 Seasonal_naive 2006 Q1  N(355, 2336)   355 [260.2737, 449.7263]95  260.  450.
##  4 Seasonal_naive 2006 Q2  N(435, 2336)   435 [340.2737, 529.7263]95  340.  530.
##  5 Seasonal_naive 2006 Q3  N(428, 4672)   428 [294.0368, 561.9632]95  294.  562.
##  6 Seasonal_naive 2006 Q4  N(397, 4672)   397 [263.0368, 530.9632]95  263.  531.
##  7 Seasonal_naive 2007 Q1  N(355, 4672)   355 [221.0368, 488.9632]95  221.  489.
##  8 Seasonal_naive 2007 Q2  N(435, 4672)   435 [301.0368, 568.9632]95  301.  569.
##  9 Seasonal_naive 2007 Q3  N(428, 7008)   428 [263.9292, 592.0708]95  264.  592.
## 10 Seasonal_naive 2007 Q4  N(397, 7008)   397 [232.9292, 561.0708]95  233.  561.
## 11 Seasonal_naive 2008 Q1  N(355, 7008)   355 [190.9292, 519.0708]95  191.  519.
## 12 Seasonal_naive 2008 Q2  N(435, 7008)   435 [270.9292, 599.0708]95  271.  599.
## 13 Seasonal_naive 2008 Q3  N(428, 9343)   428 [238.5474, 617.4526]95  239.  617.
## 14 Seasonal_naive 2008 Q4  N(397, 9343)   397 [207.5474, 586.4526]95  208.  586.
## 15 Seasonal_naive 2009 Q1  N(355, 9343)   355 [165.5474, 544.4526]95  166.  544.
## 16 Seasonal_naive 2009 Q2  N(435, 9343)   435 [245.5474, 624.4526]95  246.  624.
## 17 Seasonal_naive 2009 Q3 N(428, 11679)   428 [216.1855, 639.8145]95  216.  640.
## 18 Seasonal_naive 2009 Q4 N(397, 11679)   397 [185.1855, 608.8145]95  185.  609.
## 19 Seasonal_naive 2010 Q1 N(355, 11679)   355 [143.1855, 566.8145]95  143.  567.
## 20 Seasonal_naive 2010 Q2 N(435, 11679)   435 [223.1855, 646.8145]95  223.  647.
```

Summary

- Point forecasts are often useless without a measure of uncertainty (such as prediction intervals).
- Prediction intervals require a stochastic model(with random errors, etc).
- For most models, prediction intervals get wider as the forecast horizon increases. (For mean() model, it is not true but for other models, it is true)
- Use `level` argument to control coverage.
- Check residual assumptions before believing them.
- Prediction intervals are usually too narrow due to unaccounted uncertainty.

——— Summary ———

Forecast distributions

we express the uncertainty in our forecasts using a probability distribution. It describes the probability of observing possible future values using the fitted model. The point forecast is the mean of this distribution. Most time series models produce normally distributed forecasts — that is, we assume that the distribution of possible future values follows a normal distribution.

Prediction intervals

A prediction interval gives an interval within which we expect y_t to lie with a specified probability.

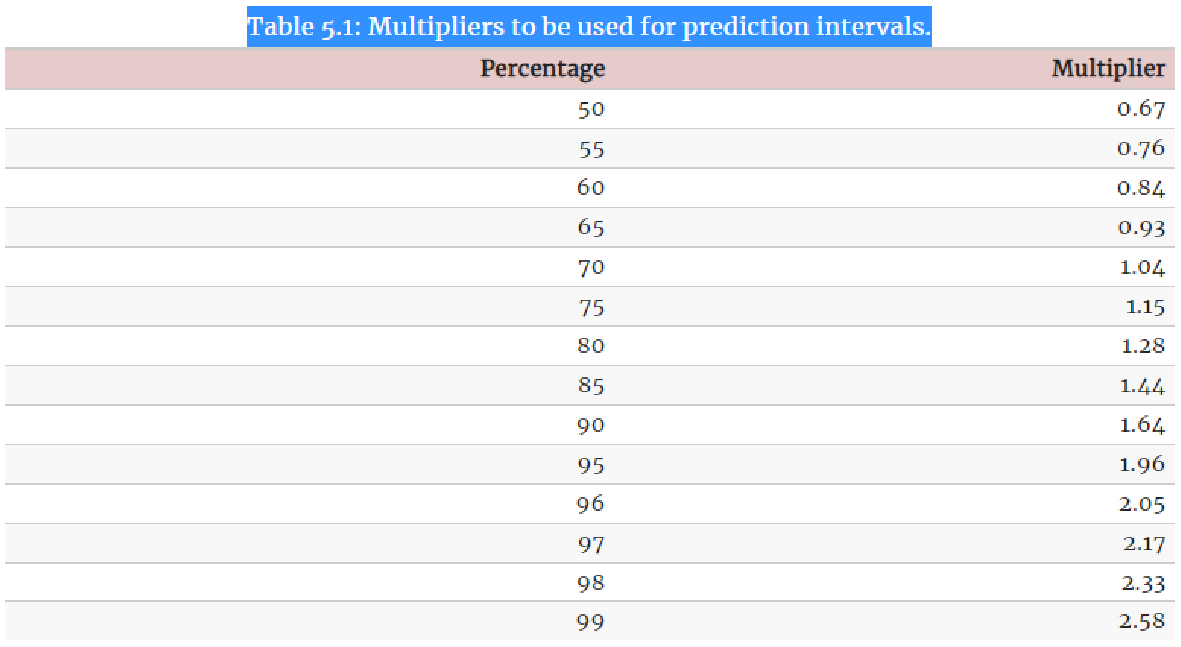


Table 5.1: Multipliers to be used for prediction intervals.

Percentage	Multiplier
50	0.67
55	0.76
60	0.84
65	0.93
70	1.04
75	1.15
80	1.28
85	1.44
90	1.64
95	1.96
96	2.05
97	2.17
98	2.33
99	2.58

Figure 5: Prediction Interval

The value of prediction intervals is that they express the uncertainty in the forecasts. If we only produce point forecasts, there is no way of telling how accurate the forecasts are. However, if we also produce prediction intervals, then it is clear how much uncertainty is associated with each forecast. For this reason, point forecasts can be of almost no value without the accompanying prediction intervals.

One-step prediction intervals

When forecasting one step ahead, the standard deviation of the forecast distribution can be estimated using the standard deviation of the residuals given by

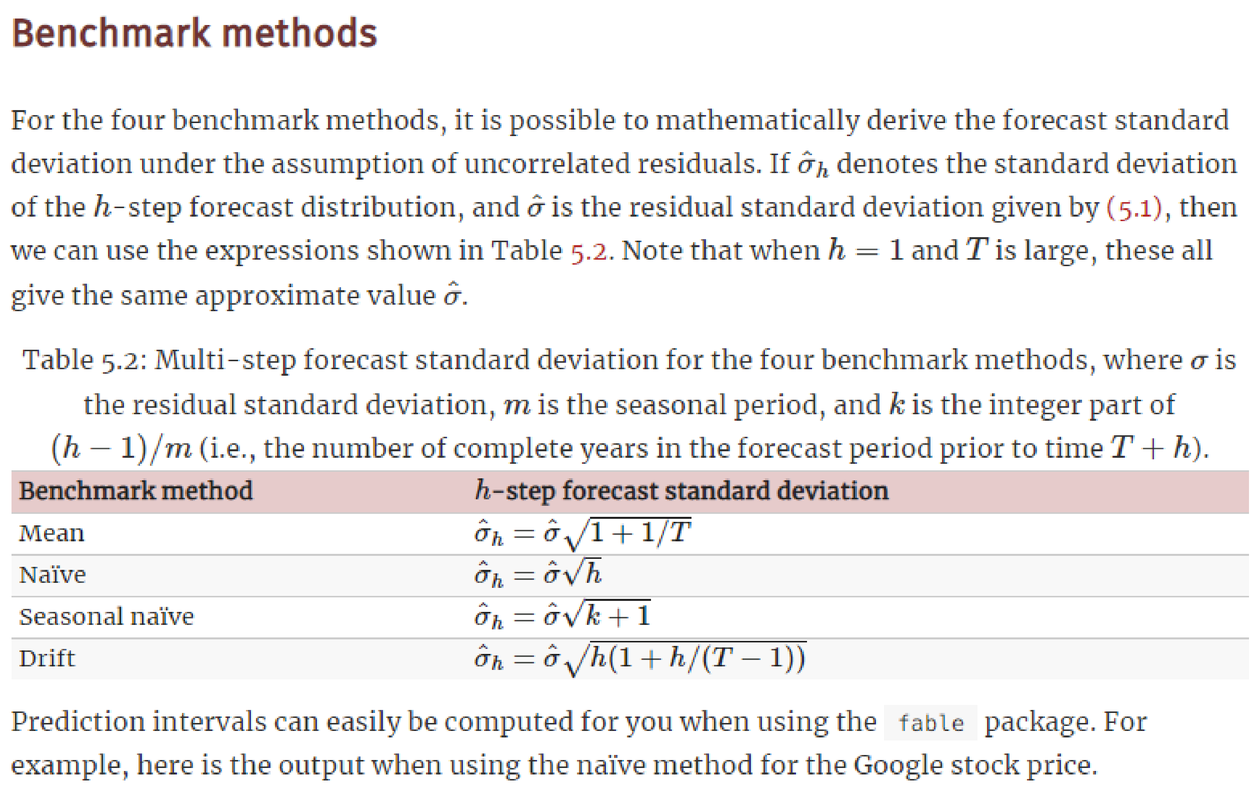
$$\hat{\sigma} = \sqrt{\frac{1}{T-K-M}\Sigma_{t=1}^{T} e_t^2}$$

where K is the number of parameters estimated in the forecasting method, and
M is the number of missing values in the residuals. (For example, M=1 for a naive forecast, because we can't forecast the first observation.)

Multi-step prediction interval

A common feature of prediction intervals is that they usually increase in length as the forecast horizon increases. The further ahead we forecast, the more uncertainty is associated with the forecast, and thus the wider the prediction intervals. That is,
σ_h usually increases with h (although there are some non-linear forecasting methods which do not have this property).

To produce a prediction interval, it is necessary to have an estimate of σ_h. As already noted, for one-step forecasts (h=1), Equation (5.1) provides a good estimate of the forecast standard deviation σ_1. For multi-step forecasts, a more complicated method of calculation is required. These calculations assume that the residuals are uncorrelated.



Benchmark methods

For the four benchmark methods, it is possible to mathematically derive the forecast standard deviation under the assumption of uncorrelated residuals. If $\hat{\sigma}_h$ denotes the standard deviation of the h-step forecast distribution, and $\hat{\sigma}$ is the residual standard deviation given by (5.1), then we can use the expressions shown in Table 5.2. Note that when $h = 1$ and T is large, these all give the same approximate value $\hat{\sigma}$.

Table 5.2: Multi-step forecast standard deviation for the four benchmark methods, where σ is the residual standard deviation, m is the seasonal period, and k is the integer part of $(h-1)/m$ (i.e., the number of complete years in the forecast period prior to time $T+h$).

Benchmark method	h-step forecast standard deviation
Mean	$\hat{\sigma}_h = \hat{\sigma}\sqrt{1+1/T}$
Naïve	$\hat{\sigma}_h = \hat{\sigma}\sqrt{h}$
Seasonal naïve	$\hat{\sigma}_h = \hat{\sigma}\sqrt{k+1}$
Drift	$\hat{\sigma}_h = \hat{\sigma}\sqrt{h(1+h/(T-1))}$

Prediction intervals can easily be computed for you when using the `fable` package. For example, here is the output when using the naïve method for the Google stock price.

Figure 6: Benchmarks Method

Prediction intervals can easily be computed for you when using the fable package. For example, here is the output when using the naïve method for the Google stock price.

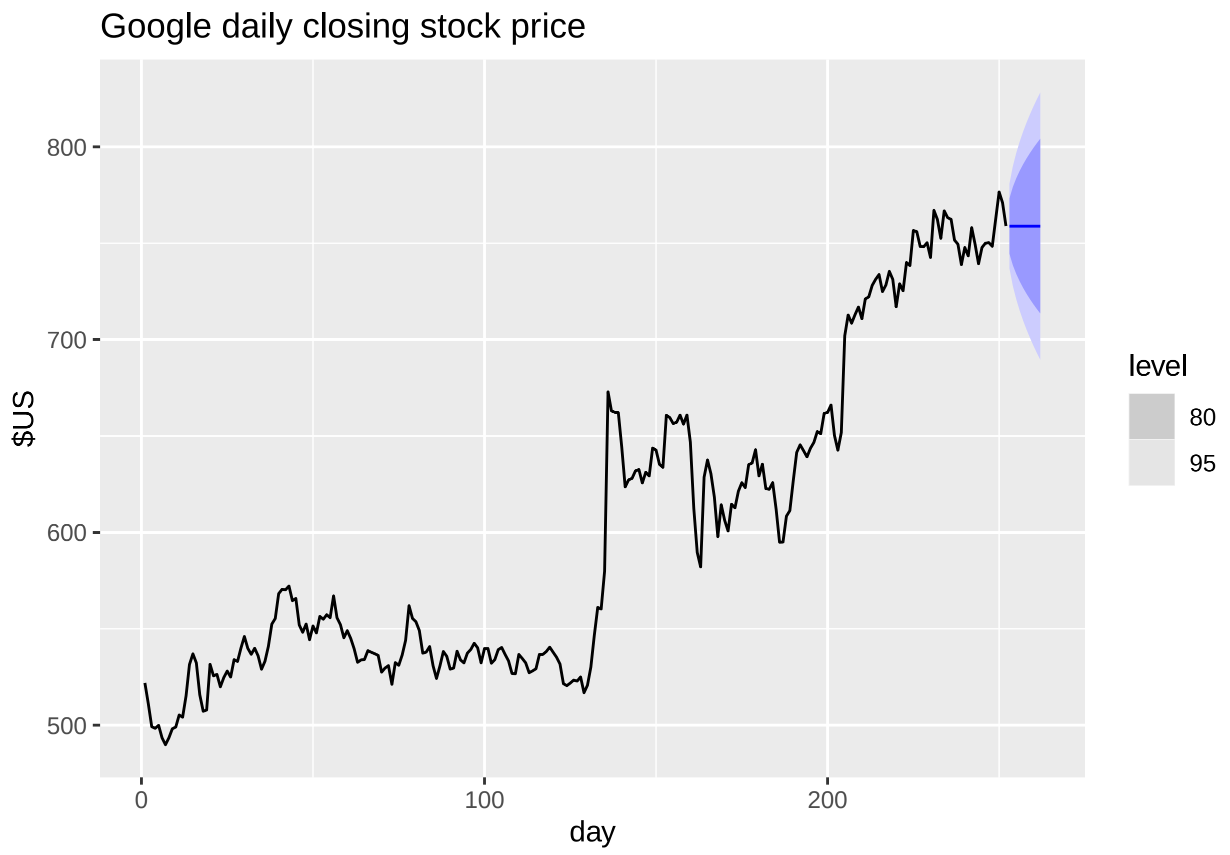
```
google_2015 |>
  model(NAIVE(Close)) |>
  forecast(h = 10) |>
  hilo()
```

```
## # 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  N(759, 125)  759. [744.5400, 773.2200]80
##  2 GOOG   NAIVE(Close)   254  N(759, 250)  759. [738.6001, 779.1599]80
##  3 GOOG   NAIVE(Close)   255  N(759, 376)  759. [734.0423, 783.7177]80
##  4 GOOG   NAIVE(Close)   256  N(759, 501)  759. [730.1999, 787.5601]80
##  5 GOOG   NAIVE(Close)   257  N(759, 626)  759. [726.8147, 790.9453]80
##  6 GOOG   NAIVE(Close)   258  N(759, 751)  759. [723.7543, 794.0058]80
##  7 GOOG   NAIVE(Close)   259  N(759, 876)  759. [720.9399, 796.8202]80
##  8 GOOG   NAIVE(Close)   260 N(759, 1002)  759. [718.3203, 799.4397]80
##  9 GOOG   NAIVE(Close)   261 N(759, 1127)  759. [715.8599, 801.9001]80
## 10 GOOG   NAIVE(Close)   262 N(759, 1252)  759. [713.5329, 804.2272]80
## # i 1 more variable: `95%` <hilo>
```

The hilo() function converts the forecast distributions into intervals. By default, 80% and 95% prediction intervals are returned, although other options are possible via the level argument.

When plotted, the prediction intervals are shown as shaded regions, with the strength of colour indicating the probability associated with the interval. Again, 80% and 95% intervals are shown by default, with other options available via the level argument.

```
google_2015 |>
  model(NAIVE(Close)) |>
  forecast(h = 10) |>
  autoplot(google_2015) +
  labs(title="Google daily closing stock price", y="$US" )
```



Prediction intervals from bootstrapped residuals

When a normal distribution for the residuals is an unreasonable assumption, one alternative is to use bootstrapping, which only assumes that the residuals are uncorrelated with constant variance. We will illustrate the procedure using a naïve forecasting method.

A one-step forecast error is defined as $e_t = y_t - \hat{y}_{t|t-1}$. For a naïve forecasting method, $\hat{y}_{t|t-1} = y_{t-1}$, so we can rewrite this as

$$y_t = y_{t-1} + e_t$$

Assuming future errors will be similar to past errors, when $t > T$ we can replace e_t by sampling from the collection of errors we have seen in the past (i.e., the residuals). So we can simulate the next observation of a time series using

$$y^*_{T+1} = y_T + e^*_{T+1}$$

where e^*_{T+1} is a randomly sampled error from the past, and y^*_{T+1} is the possible future value that would arise if that particular error value occurred. We use a * to indicate that this is not the observed y_{T+1} value, but one possible future that could occur. Adding the new simulated observation to our data set, we can repeat the process to obtain

$$y^*_{T+2} = y^*_{T+1} + e^*_{T+2}$$

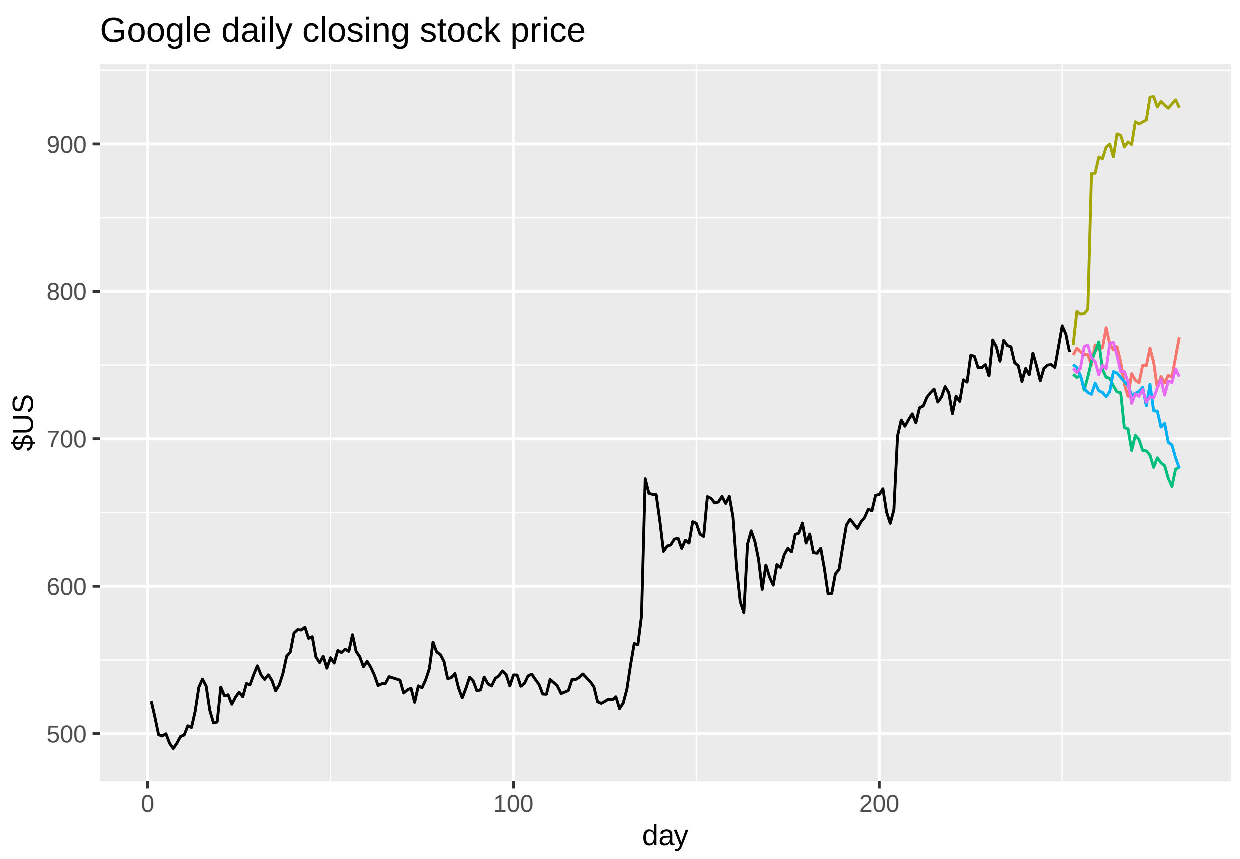
Doing this repeatedly, we obtain many possible futures. To see some of them, we can use the generate() function.

```
fit <- google_2015 |>
  model(NAIVE(Close))
sim <- fit |> generate(h = 30, times = 5, bootstrap = TRUE)
sim
```

```
## # A tsibble: 150 x 6 [1]
## # Key:       Symbol, .model, .rep [5]
##    Symbol .model       .rep    day .innov  .sim
##    <chr>  <chr>        <chr> <dbl>  <dbl> <dbl>
##  1 GOOG   NAIVE(Close) 1       253 -2.08   757.
##  2 GOOG   NAIVE(Close) 1       254  4.69   761.
##  3 GOOG   NAIVE(Close) 1       255 -2.49   759.
##  4 GOOG   NAIVE(Close) 1       256 -1.82   757.
##  5 GOOG   NAIVE(Close) 1       257 -0.188  757.
##  6 GOOG   NAIVE(Close) 1       258 -6.95   750.
##  7 GOOG   NAIVE(Close) 1       259 13.6    764.
##  8 GOOG   NAIVE(Close) 1       260 -3.09   761.
##  9 GOOG   NAIVE(Close) 1       261  1.19   762.
## 10 GOOG   NAIVE(Close) 1       262 13.6    775.
## # i 140 more rows
```

Here we have generated five possible sample paths for the next 30 trading days. The `.rep` variable provides a new key for the tsibble. The plot below shows the five sample paths along with the historical data.

```
google_2015 |>
  ggplot(aes(x = day)) +
  geom_line(aes(y = Close)) +
  geom_line(aes(y = .sim, colour = as.factor(.rep)),
    data = sim) +
  labs(title="Google daily closing stock price", y="$US" ) +
  guides(colour = "none")
```



Then we can compute prediction intervals by calculating percentiles of the future sample paths for each forecast horizon. The result is called a bootstrapped prediction interval. The name "bootstrap" is a reference to pulling ourselves up by our bootstraps, because the process allows us to measure future uncertainty by only using the historical data.

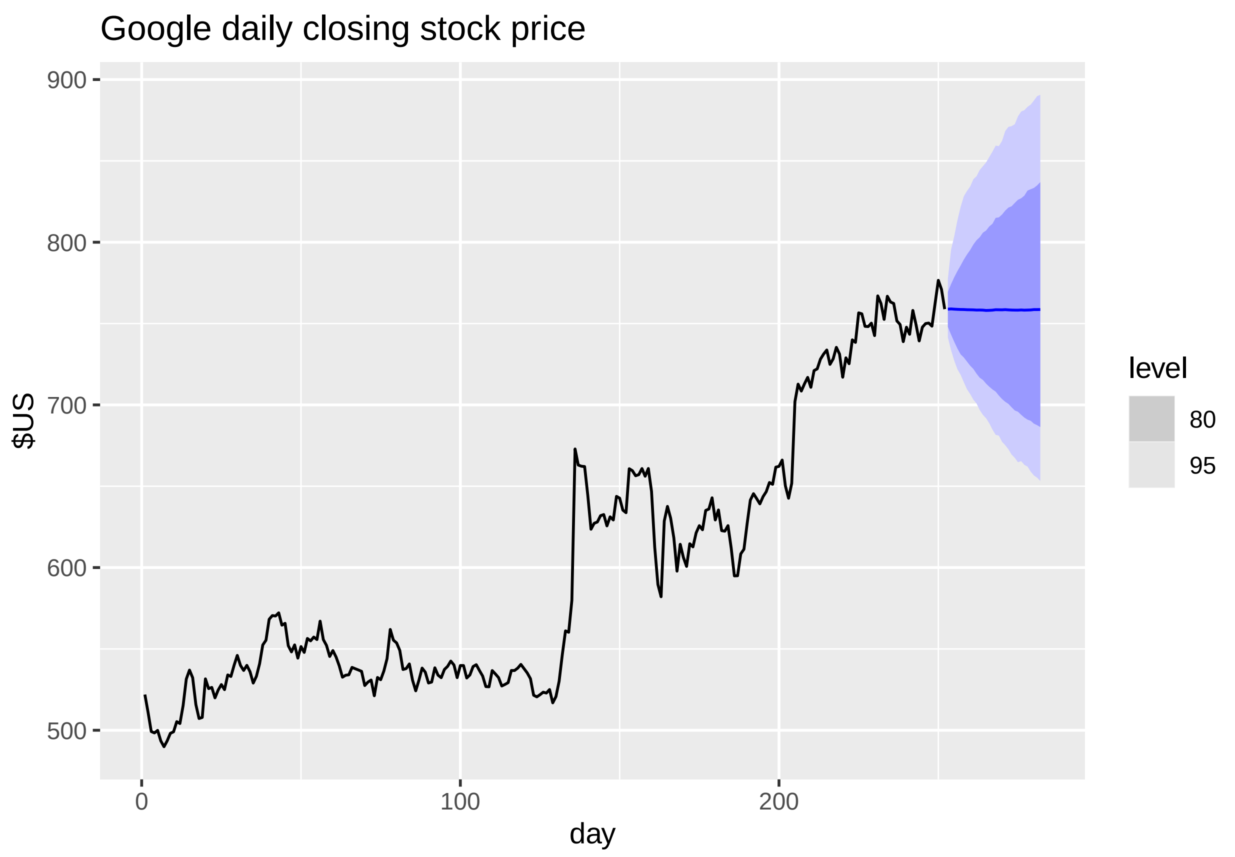
This is all built into the forecast() function so you do not need to call generate() directly.

```r
fc <- fit |> forecast(h = 30, bootstrap = TRUE)
fc
```

```
## # A fable: 30 x 5 [1]
## # Key:     Symbol, .model [1]
##    Symbol .model          day        Close .mean
##    <chr>  <chr>         <dbl>       <dist> <dbl>
##  1 GOOG   NAIVE(Close)    253 sample[5000]  759.
##  2 GOOG   NAIVE(Close)    254 sample[5000]  759.
##  3 GOOG   NAIVE(Close)    255 sample[5000]  759.
##  4 GOOG   NAIVE(Close)    256 sample[5000]  759.
##  5 GOOG   NAIVE(Close)    257 sample[5000]  759.
##  6 GOOG   NAIVE(Close)    258 sample[5000]  759.
##  7 GOOG   NAIVE(Close)    259 sample[5000]  759.
##  8 GOOG   NAIVE(Close)    260 sample[5000]  758.
##  9 GOOG   NAIVE(Close)    261 sample[5000]  758.
## 10 GOOG   NAIVE(Close)    262 sample[5000]  758.
## # i 20 more rows
```

Notice that the forecast distribution is now represented as a simulation with 5000 sample paths. Because there is no normality assumption, the prediction intervals are not symmetric. The `.mean` column is the mean of the bootstrap samples, so it may be slightly different from the results obtained without a bootstrap.

```
autoplot(fc, google_2015) +
  labs(title="Google daily closing stock price", y="$US" )
```



The number of samples can be controlled using the times argument for forecast(). For example, intervals based on 1000 bootstrap samples can be sampled with:

```
google_2015 |>
  model(NAIVE(Close)) |>
  forecast(h = 10, bootstrap = TRUE, times = 1000) |>
  hilo()
```

```
## # 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. [748.0890, 769.8420]80
##  2 GOOG   NAIVE(Close)   254 sample[1000]  759. [742.6333, 774.9262]80
##  3 GOOG   NAIVE(Close)   255 sample[1000]  760. [737.7440, 782.4061]80
##  4 GOOG   NAIVE(Close)   256 sample[1000]  760. [734.6269, 786.0770]80
##  5 GOOG   NAIVE(Close)   257 sample[1000]  760. [731.8557, 789.7621]80
##  6 GOOG   NAIVE(Close)   258 sample[1000]  760. [728.1954, 793.3272]80
##  7 GOOG   NAIVE(Close)   259 sample[1000]  760. [725.3305, 796.3682]80
##  8 GOOG   NAIVE(Close)   260 sample[1000]  759. [723.6868, 798.3543]80
##  9 GOOG   NAIVE(Close)   261 sample[1000]  759. [721.6053, 801.8498]80
## 10 GOOG   NAIVE(Close)   262 sample[1000]  759. [718.8027, 806.4425]80
## # i 1 more variable: `95%` <hilo>
```

5.6 Forecasting using transformations

Mathematical Transformations:

- If the data show different variation at different levels of series, then a transformation can be useful.
- Denote original observations as $y_1, y_2, ..., y_n$ and transformed observations as $w_1, w_2, ..., w_n$ then the Box-cox transformation can be defined as:

$w_t = \log(y_t)\ if\ \lambda = 0$ and

$w_t = \frac{sign(y_t)|y_t|^{\lambda-1}}{\lambda}\ if\ \lambda \neq 0$

- Natural logarithm, particularly useful because they are more interpretable: changes in a log values are *relative(percent) changes on the original scale.*
- Transformations are useful mainly for two reasons. One: needs to stabilize the variance and other is that if we want to stay on the positive scale.
- Once we take a transformation, we need to take a forecast and we need to back transform to the original scale as we are interested in that. So, we must *reverse the transformation* or *back transform* to obtain forecasts on the original scale. The reverse Box-Cox transformations are given by:

$y_t = \exp(w_t)\ if\ \lambda = 0$

$y_t = sign(\lambda(w_t) + 1)|\lambda(w_t) + 1)|^{1/\lambda}\ if\ \lambda \neq 0$

But it creates a problem.

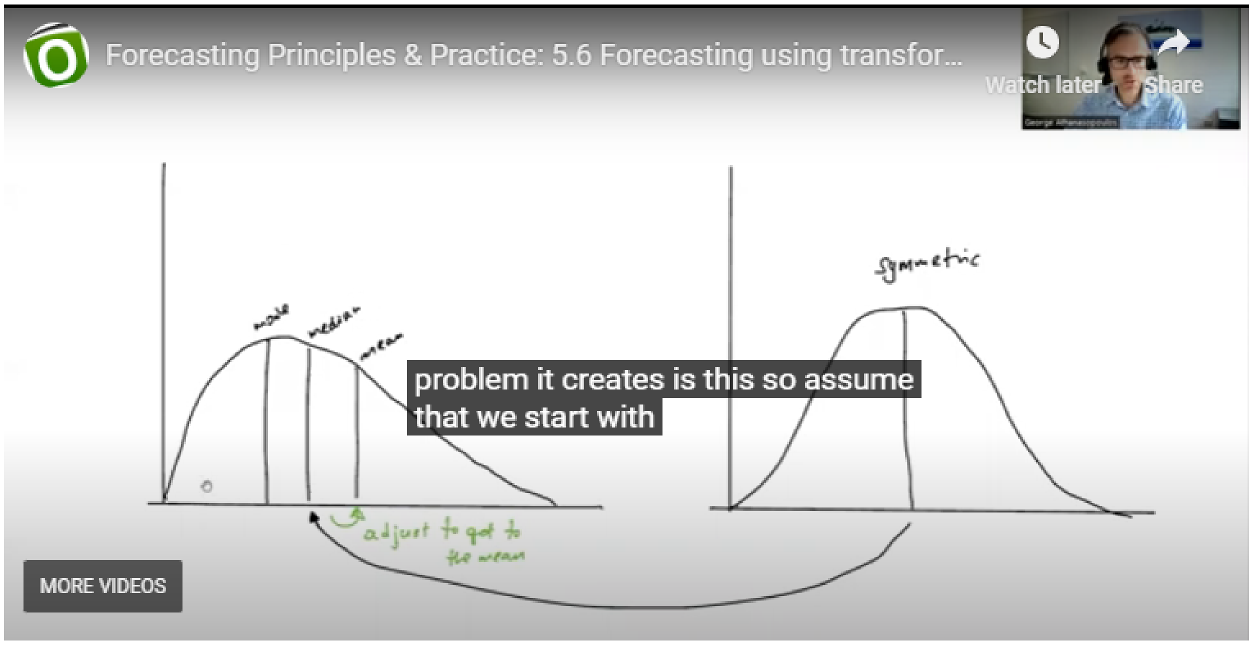

Forecasting Principles & Practice: 5.6 Forecasting using transfor...
Watch later
Share
mode
median
mean
symmetric
problem it creates is this so assume that we start with
adjust to get to the mean
MORE VIDEOS

Figure 7: Transformation Issue

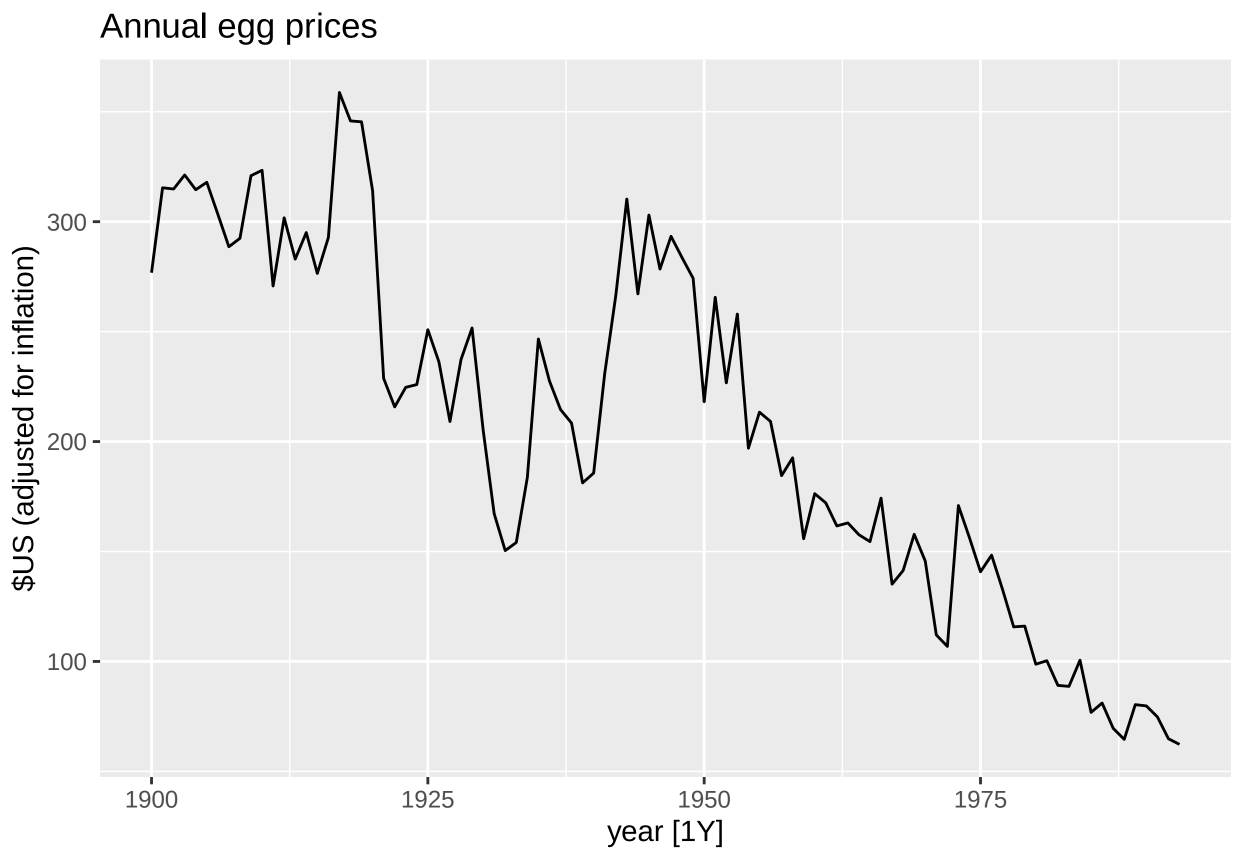
Here, from after taking the transformation, we get the right figure from the left figure. We take a log transformation of the positive skew distribution and convert to the symmetric distribution on the right side. Now, we also have to do back transform to go into the original scale. The challenge here is that going back any quantile like median will map directly into the distribution. So, median on the right side should be mapped to median on the left side. In fact, any quantile will do that.

The problem here is that for our forecast, we need the mean and not the median and hence to actually get the mean, we need to make some adjustment on the left side.

Example

```
eggs <- prices |>
  filter(!is.na(eggs)) |>
  select(eggs)
eggs |> autoplot()+
  labs(title = "Annual egg prices", y= "$US (adjusted for inflation)")
```

```
## Plot variable not specified, automatically selected `.vars = eggs`
```



There are few interesting features on this plot:

(1) There a couple of times when egg prices are increased rapidly like in world war 1 and 2. And also decreased much in the period of great depression between 1925 and 1950 when egg prices are at minimal level.

(2) After 1950, egg prices are getting cheaper and some point it crosses the x-axis and we don't want it and so we need log transformation.

Modelling with transformations

Transformations used in the left of the formulav i.e. log(eggs) below in code will be automatically back-transformed. To model log-transformed egg prices, we could use:

```
fit <- eggs |>
  model(RW(log(eggs)~ drift()))
fit
```

```
## # A mable: 1 x 1
##   `RW(log(eggs) ~ drift())`
##                     <model>
## 1              <RW w/ drift>
```

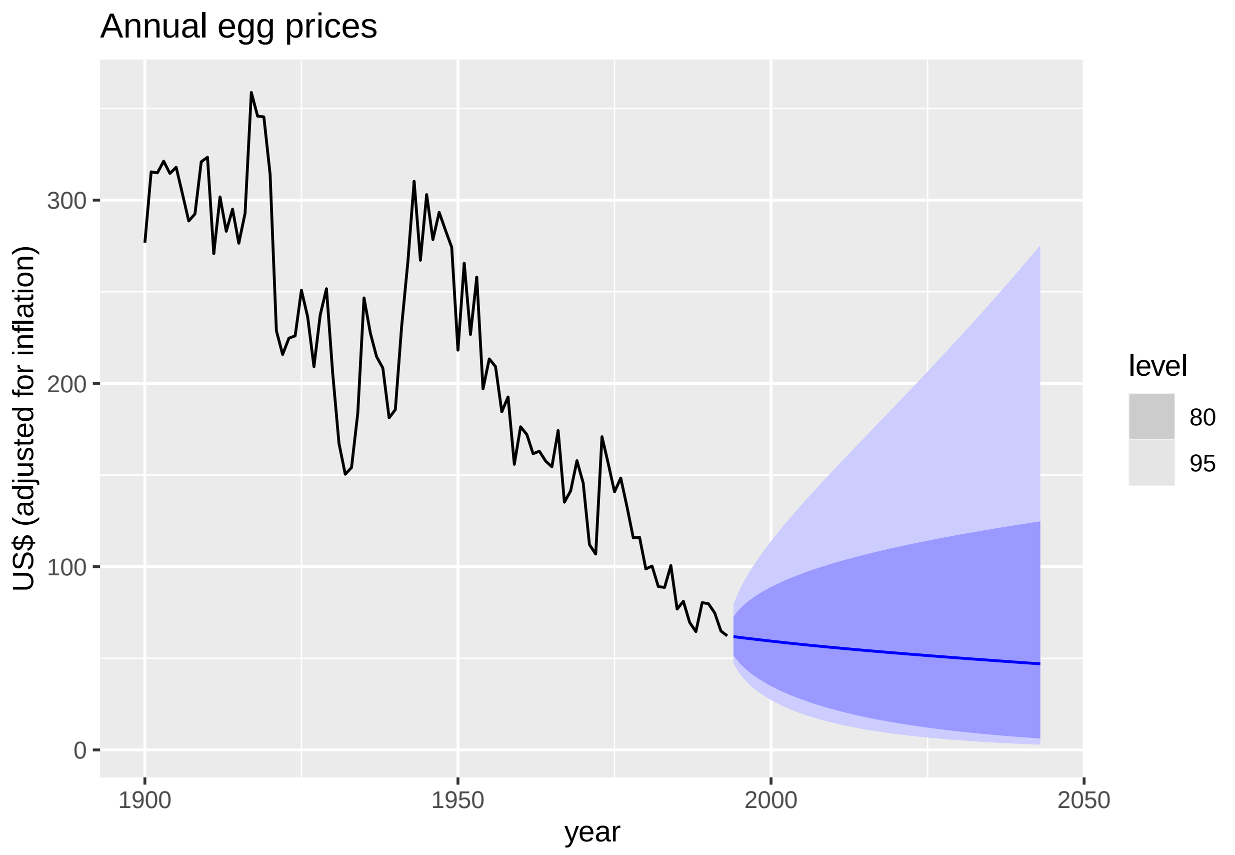
```
fc <- fit |>
  forecast(h=50)
fc
```

```
## # A fable: 50 x 4 [1Y]
## # Key:     .model [1]
##    .model                   year            eggs .mean
##    <chr>                   <dbl>          <dist> <dbl>
##  1 RW(log(eggs) ~ drift())  1994 t(N(4.1, 0.018))  61.8
##  2 RW(log(eggs) ~ drift())  1995 t(N(4.1, 0.036))  61.4
##  3 RW(log(eggs) ~ drift())  1996 t(N(4.1, 0.055))  61.0
##  4 RW(log(eggs) ~ drift())  1997 t(N(4.1, 0.074))  60.6
##  5 RW(log(eggs) ~ drift())  1998 t(N(4.1, 0.093))  60.2
##  6 RW(log(eggs) ~ drift())  1999    t(N(4, 0.11))  59.8
##  7 RW(log(eggs) ~ drift())  2000    t(N(4, 0.13))  59.4
##  8 RW(log(eggs) ~ drift())  2001    t(N(4, 0.15))  59.0
##  9 RW(log(eggs) ~ drift())  2002    t(N(4, 0.18))  58.6
## 10 RW(log(eggs) ~ drift())  2003     t(N(4, 0.2))  58.3
## # i 40 more rows
```

Here, we get a transformed distribution t() and transformation we use is log transformation and it returns the mean of the forecast distribution on the original scale in .mean column. So, it makes the back transformation and do the adjustment.

Let's look it graphically:

```
fc |> autoplot(eggs) +
  labs(title="Annual egg prices", y=  "US$ (adjusted for inflation)")
```



Here, log transformation has worked quite well. The blue line is actually the mean.

Bias Adjustment

- Back-transformed point forecasts are medians.
- Back-transformed prediction interval(PI) have the correct coverage.

Back-transformed means

We have transformed X to f(X) and now we have to back-transform it to Y.

Let X (transform variable) have means μ and variance σ^2

Let $f(X)$ be back-transformed function and $Y = f(X)$

Taylor series expansion about μ (we have taylor series approximation for quadratic polynomial)

$Y = f(X) \approx f(\mu) + (X - \mu)f'(\mu)(linear\ trend) + \frac{1}{2}(X - \mu)^2 f''(\mu)(quadratic\ trend)$.

The beauty of this expression is we can take the expected value as:

$$E[Y] = E[f(X)] \approx f(\mu) + \frac{1}{2}\sigma^2 f''(\mu)$$

Now, what happens in Box-Cox transformation:

$y_t = \exp(w_t)\ if\ \lambda = 0$ and

$y_t = (\lambda w_t + 1)^{1/\lambda}\ if\ \lambda \neq 0$

$f(x) = \exp(x)\ if\ \lambda = 0$ and

$f(x) = (\lambda x + 1)^{1/\lambda}$ $if\ \lambda \neq 0$

$f''(x) = \exp(x)$ $if\ \lambda = 0$ and

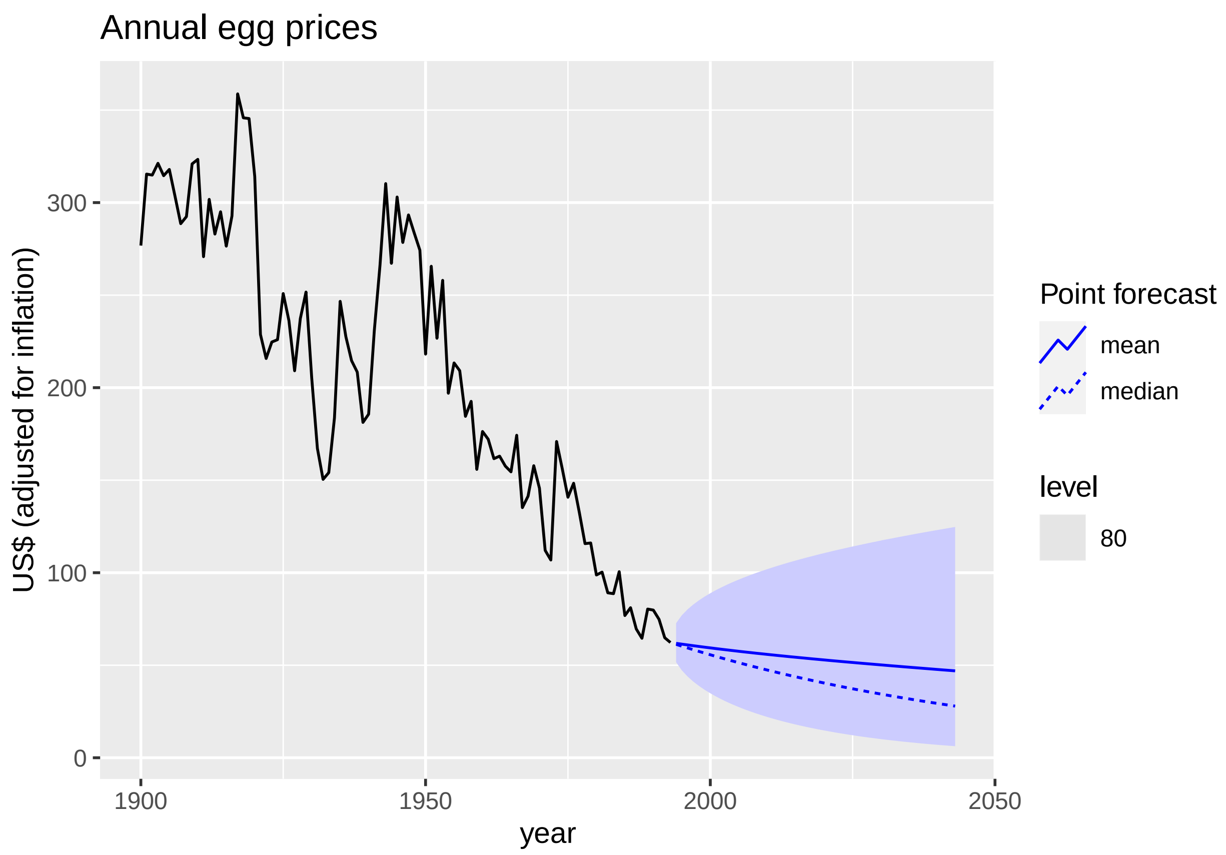
$f''(x) = (1-\lambda)(\lambda x + 1)^{1/\lambda - 2}$ $if\ \lambda \neq 0$

Now,

$E[Y] \approx e^{\mu}[1 + \frac{\sigma^2}{2}]$ $if\ \lambda = 0$ and

$E[Y] \approx (\lambda\mu + 1)^{1/\lambda}[1 + \frac{\sigma^2(1-\lambda)}{2(\lambda\mu+1)^2}]$ $if\ \lambda = 0$

```r
fc |>
  autoplot(eggs, level = 80, point_forecast=lst(mean,median))+
  labs(title = "Annual egg prices", y= "US$ (adjusted for inflation)")
```



The dashed line in Figure 5.17 shows the forecast medians while the solid line shows the forecast means. Notice how the skewed forecast distribution pulls up the forecast distribution's mean; this is a result of the added term from the bias adjustment.

Bias-adjusted forecast means are automatically computed in the fable package. The forecast median (the point forecast prior to bias adjustment) can be obtained using the median() function on the distribution column.

Summary

Some common transformations which can be used when modelling were discussed in Section 3.1. When forecasting from a model with transformations, we first produce forecasts of the transformed data. Then, we need to reverse the transformation (or back-transform) to obtain forecasts on the original scale.

The fable package will automatically back-transform the forecasts whenever a transformation has been used in the model definition. The back-transformed forecast distribution is then a “transformed Normal” distribution.

Prediction intervals with transformations

If a transformation has been used, then the prediction interval is first computed on the transformed scale, and the end points are back-transformed to give a prediction interval on the original scale. This approach preserves the probability coverage of the prediction interval, although it will no longer be symmetric around the point forecast.

The back-transformation of prediction intervals is done automatically when using the fable package, provided you have used a transformation in the model formula.

Transformations sometimes make little difference to the point forecasts but have a large effect on prediction intervals.

Bias adjustments

One issue with using mathematical transformations such as Box-Cox transformations is that the back-transformed point forecast will not be the mean of the forecast distribution. In fact, it will usually be the median of the forecast distribution (assuming that the distribution on the transformed space is symmetric). For many purposes, this is acceptable, although the mean is usually preferable. For example, you may wish to add up sales forecasts from various regions to form a forecast for the whole country. But medians do not add up, whereas means do.

The difference between the simple back-transformed forecast given by (5.2) and the mean given by (5.3) is called the bias. When we use the mean, rather than the median, we say the point forecasts have been bias-adjusted.

5.7 Forecasting with decomposition

We can decompose the original data into seasonal component and rest of the components as:

$y_t = \hat{S}_t + \hat{A}_t$ (Additive decomposition)

where $\hat{S}_t$ is the seasonal component and $\hat{A}_t$ is the seasonally adjusted component and $\hat{A}_t = \hat{T}_t + \hat{R}_t$. If a multiplicative decomposition is used then $y_t = \hat{S}_t \times \hat{A}_t$ where $\hat{A}_t = \hat{T}_t \times \hat{R}_t$.

We have forecasted a seasonal component using seasonal naive method because seasonal component does not changed much over time and so forecast for the next year should be pretty similar to what we have seen this year.

For seasonally adjusted component, we can use any non-seasonal time series method.

To forecast the seasonally adjusted component, any non-seasonal forecasting method may be used. For example, the drift method, or Holt’s method (discussed in Chapter 8), or a non-seasonal ARIMA model (discussed in Chapter 9), may be used.

So,

- Forecast $\hat{S}_t$ using SNAIVE() method.
- Forecast $\hat{A}_t$ using non-seasonal time series method.
- Combine forecasts of $\hat{S}_t$ and $\hat{A}_t$ to get forecasts of original data.

Example: US Retail Employment

```r
us_retail_employment <- us_employment |>
  filter(year(Month) >= 1990, Title=="Retail Trade") |>
  select(-Series_ID)
us_retail_employment
```

```
## # A tsibble: 357 x 3 [1M]
##       Month Title        Employed
##       <mth> <chr>           <dbl>
##  1 1990 Jan Retail Trade   13256.
##  2 1990 Feb Retail Trade   12966.
##  3 1990 Mar Retail Trade   12938.
##  4 1990 Apr Retail Trade   13012.
##  5 1990 May Retail Trade   13108.
##  6 1990 Jun Retail Trade   13183.
##  7 1990 Jul Retail Trade   13170.
##  8 1990 Aug Retail Trade   13160.
##  9 1990 Sep Retail Trade   13113.
## 10 1990 Oct Retail Trade   13185.
## # i 347 more rows
```

First, we do the STL decomposition.

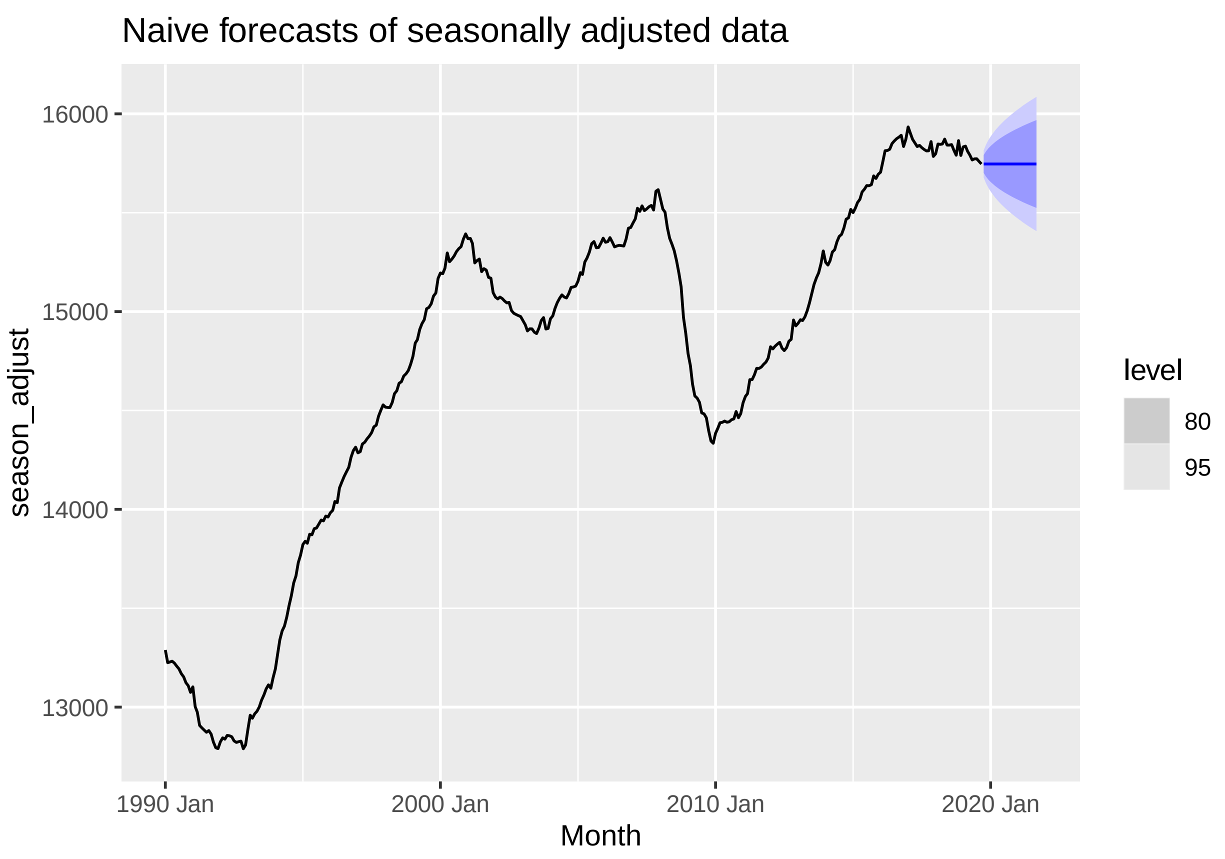
```r
dcmp <- us_retail_employment |>
  model(STL(Employed)) |>
  components() |>
  select(-.model)
dcmp
```

```
## # A tsibble: 357 x 6 [1M]
##       Month Employed  trend season_year remainder season_adjust
##       <mth>    <dbl>  <dbl>       <dbl>     <dbl>         <dbl>
##  1 1990 Jan   13256. 13288.      -33.0      0.836        13289.
##  2 1990 Feb   12966. 13269.     -258.     -44.6          13224.
##  3 1990 Mar   12938. 13250.     -290.     -22.1          13228.
##  4 1990 Apr   13012. 13231.     -220.       1.05         13232.
##  5 1990 May   13108. 13211.     -114.      11.3          13223.
##  6 1990 Jun   13183. 13192.      -24.3     15.5          13207.
##  7 1990 Jul   13170. 13172.      -23.2     21.6          13193.
##  8 1990 Aug   13160. 13151.       -9.52    17.8          13169.
##  9 1990 Sep   13113. 13131.      -39.5     22.0          13153.
## 10 1990 Oct   13185. 13110.       61.6     13.2          13124.
## # i 347 more rows
```

Here, `season_year` and `season_adjust` columns add up to the original data values. So we forecast `season_year` and `season_adjust` columns and then add these together.

Let's forecast `season_adjust`:

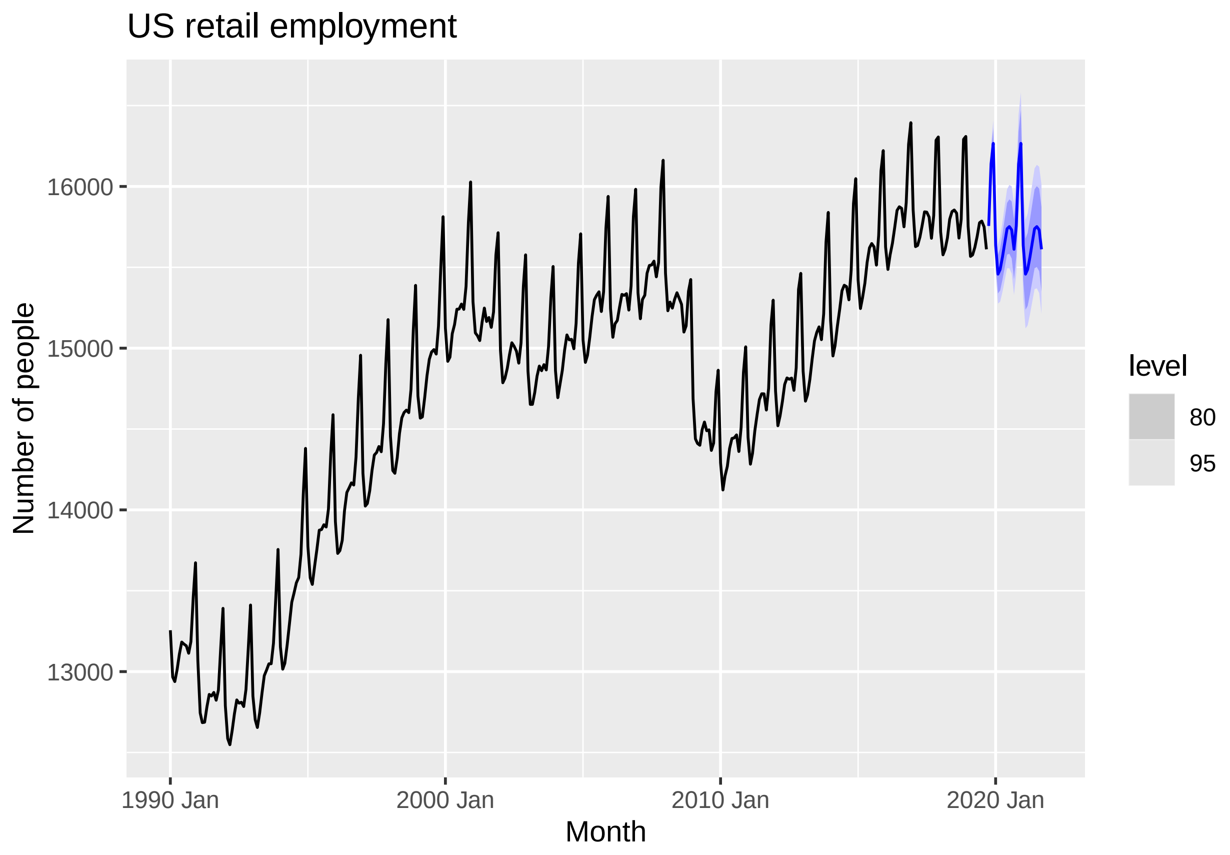
```r
dcmp |>
  model(NAIVE(season_adjust)) |>
  forecast() |>
  autoplot(dcmp) +
  labs(title = "Naive forecasts of seasonally adjusted data")
```



Seasonal component is forecasted using seasonal_naive() method and then we add up two things together.

We can do all these things in one step as:

```
fit_dcmp <- us_retail_employment |>
  model(stlf = decomposition_model(
    STL(Employed ~ trend(window = 7), robust = TRUE),
    NAIVE(season_adjust)
  ))
fit_dcmp |>
  forecast() |>
  autoplot(us_retail_employment)+
  labs(y = "Number of people",
       title = "US retail employment")
```



Here, seasonal_naive() method for the seasonal component is by default in the code.We can put it as third argument.

The prediction intervals shown in this graph are constructed in the same way as the point forecasts. That is, the upper and lower limits of the prediction intervals on the seasonally adjusted data are "reseasonalised" by adding in the forecasts of the seasonal component.

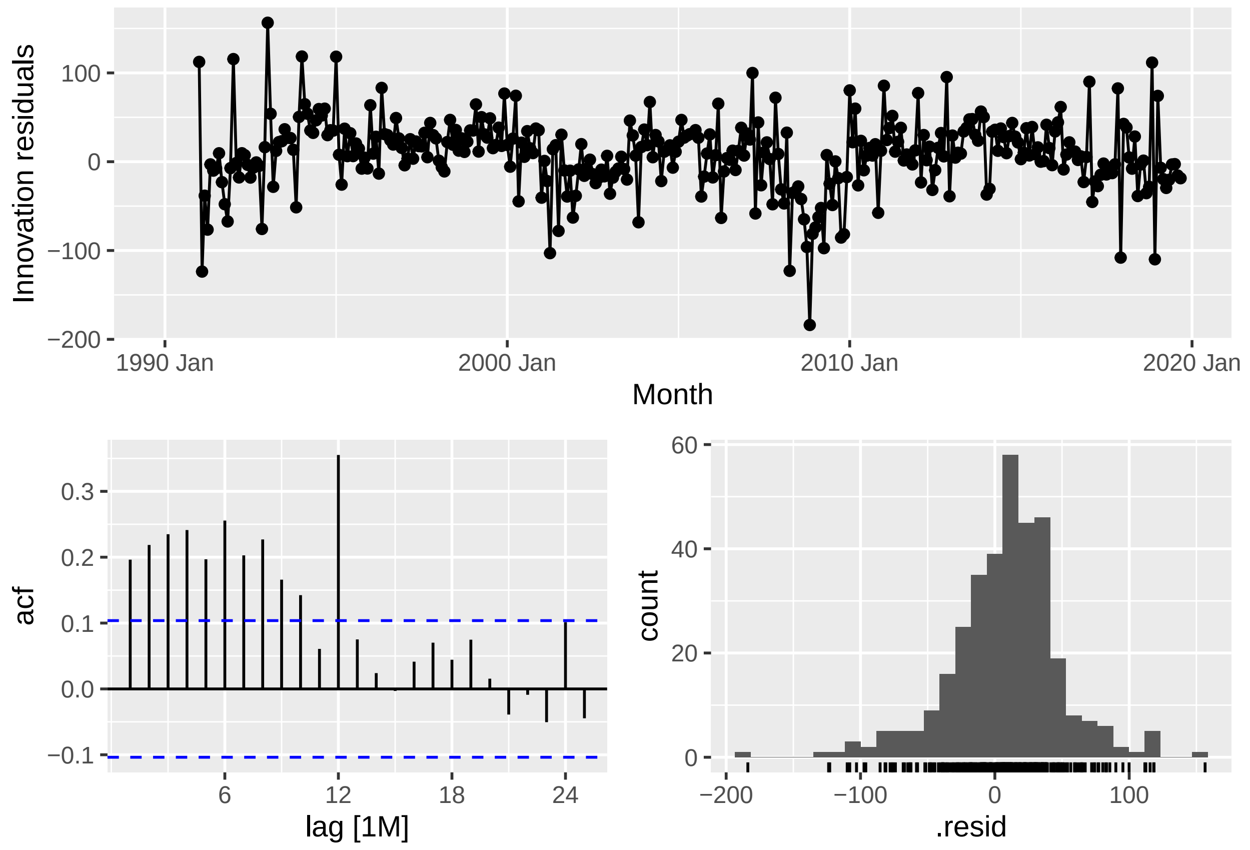
The ACF of the residuals, shown in Figure 5.20, displays significant autocorrelations. These are due to the naïve method not capturing the changing trend in the seasonally adjusted series.

```
fit_dcmp |> gg_tsresiduals()
```

```
## Warning: Removed 12 rows containing missing values (`geom_line()`).

## Warning: Removed 12 rows containing missing values (`geom_point()`).

## Warning: Removed 12 rows containing non-finite values (`stat_bin()`).
```



`decomposition_model()` creates a decomposition model

- You must provide a method for the forecasting the season_adjust series.
- A seasonal naive method is usually by default for the seasonal component.
- The variances from both the seasonally adjusted and seasonal forecasts are combined.

5.8 Evaluating point forecast accuracy

Training and test sets

The traditional way of doing this is to take the original data and split it into a training set and a test set. Basically we use a training set for the estimation and the test set for the evaluation. Now you can imagine that I have a highly complex model that fits some data series very well so you can imagine that we have got sqiggles in the data and we have a high order polynomial that can fit very well but then when we extrapolate then it might flop and might not forecast well.

Overfitting a model to data is just as bad as failing to identify systematic pattern in the data. We want to avoid overfitting.

Just to reiterate test data should not be used for any estimation and we assume that we have never seen this data. It's the future that we are trying to forecast and we can't use this. This is what will happen in real practice. We don't know what tomorrow looks like.

So,

- A model which fits the training data well will not necessarily forecast well.
- A perfect fit can always be obtained by using a model with enough parameters.

- Over-fitting a model to data is just as bad as failing to identify a systematic pattern in the data.
- The test set must not be used for any aspect of model development or calculation of forecasts
- Forecast accuracy is based only on test set.

Forecast "error": the difference between an observed value and its forecast.

$$e_{T+h} = y_{T+h} - \hat{y}_{T+h|h}$$

where the training data is given by $y_1, ..., y_T$

- Unlike residuals, forecast errors on the test set involve multi-step forecasts.
- These are true forecast errors as the test data is not used in computing $\hat{y}_{T+h|T}$.

Measure of forecast accuracy

Example

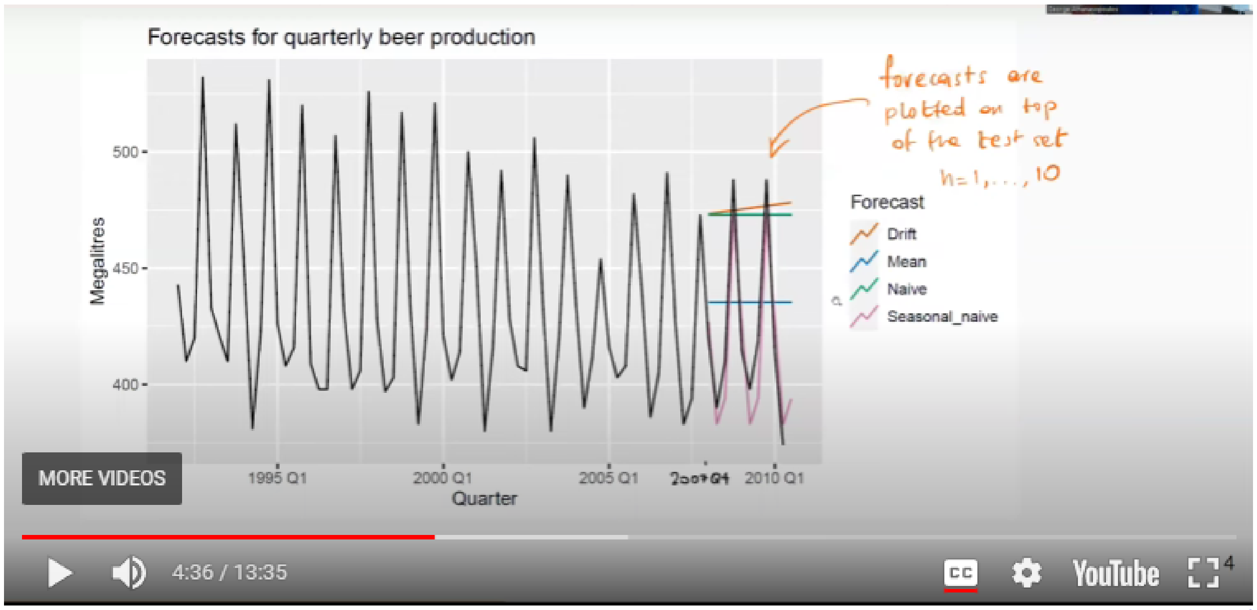


Figure 8: Example

Measure of forecast accuracy

$y_{T+h} = (T+h)^{th}$ observation, $h = 1, 2, ..., H$

$\hat{y}_{T+h|T}$: its forecast based on data up to time T.

$e_{T+h} = y_{T+h} - \hat{y}_{T+h|T}$

Forecast error $= e_{T+h} = y_{T+h} - \hat{y}_{T+h|T}$

Mean Error= mean(e_{T+h}) [mean(e_{T+h}) represents the biasedness over our forecast and we should not have the biased forecast and so we don't pay lots of attention and we pay more attention to actual accuracy and so we consider the following accuracy measures]

(1) *MAE(Mean Absolute Error)*= mean($|e_{T+h}|$)

(2) *MSE(Mean Squared Error)*= mean(e_{T+h}^2)

(3) *MAPE(Mean Absolute Percentage Error)=* $100\times$ mean($\frac{|e_{T+h}|}{|y_{T+h}|}$)

(4) *RMSE(Root Mean Squared Error)=* $\sqrt{mean(e^2_{T+h})}$ (unit is sqaure roor(unit^2) =unit)

- MAE,MSE,RMSE are all scale dependent. (For one time series it might be fine but for many time series it might be problematic)
- MAPE is scale independent but is only sensible if $y_t >> 0$ for all t, and y has a natural zero. (It is probably the most widely used error measure and it is also easily interpretable error measure) but it is problematic because in the formula, we are dividing by t_{T+h} and if it is zero then it will be undefined and if it is close to zero then it is also problematic because it skews the error measure and if it is far away from zero then it will not skews the result.

To overcome the disadvantage of MAPE, Hyndman and Koehler proposed the following:

- For non-seasonal time series, scale errors using naive forecasts:

$$q_j = \frac{e_j}{\frac{1}{T-1}\Sigma_{t=2}^T|y_t - y_{t-1}|}$$

- For seasonal time series, scale forecast errors using seasonal naive forecasts:

$$q_j = \frac{e_j}{\frac{1}{T-m}\Sigma_{t=m+1}^T|y_t - y_{t-m}|}$$

These are the scaled errors and Now,

MASE(Mean Absolute Scaled Error)= $mean(|q_j|)$

RMSSE(Root Mean Squared Scaled Error)= $\sqrt{mean(q_j^2)}$

where

$q_j^2 = \frac{e_j^2}{\frac{1}{T-m}\Sigma_{t=m+1}^T(y_t - y_{t-m})^2}$

This is for seasonal data and we set $m = 1$ for non-seasonal data.

Now, one of the disadvantage of MASE is it is not widely used as much MAPE and it is not easily interpretable. It is not a percentage, it is a forecast error relative to the in-sample naive forecast mean absolute error, so value greater than 1 means we are doing the worse than what in-sample naive does and a value below than 1 means we are doing better than in-sample naive does.

Application

```r
recent_production <- aus_production |>
  filter(year(Quarter) >= 1992)
train <- recent_production |>
  filter(year(Quarter)<=2007)
beer_fit <- train |>
  model(
    Mean=MEAN(Beer),
    Naive=NAIVE(Beer),
    Seasonal_naive=SNAIVE(Beer),
    Drift = RW(Beer ~drift())
    )
beer_fc <- beer_fit |>
  forecast(h=10)
beer_fc
```

```
## # A fable: 40 x 4 [1Q]
## # Key:     .model [4]
##    .model Quarter         Beer .mean
##    <chr>    <qtr>       <dist> <dbl>
##  1 Mean   2008 Q1 N(435, 1964)  435.
##  2 Mean   2008 Q2 N(435, 1964)  435.
##  3 Mean   2008 Q3 N(435, 1964)  435.
##  4 Mean   2008 Q4 N(435, 1964)  435.
##  5 Mean   2009 Q1 N(435, 1964)  435.
##  6 Mean   2009 Q2 N(435, 1964)  435.
##  7 Mean   2009 Q3 N(435, 1964)  435.
##  8 Mean   2009 Q4 N(435, 1964)  435.
##  9 Mean   2010 Q1 N(435, 1964)  435.
## 10 Mean   2010 Q2 N(435, 1964)  435.
## # i 30 more rows
```

Accuracy Measures

```
accuracy(beer_fit) |>
  arrange(.model) |>
  select(.model,.type,RMSE,MAE,MAPE,MASE,RMSSE)
```

```
## # A tibble: 4 x 7
##   .model         .type     RMSE   MAE  MAPE  MASE RMSSE
##   <chr>          <chr>    <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Drift          Training  65.3  54.8 12.2   3.83  3.89
## 2 Mean           Training  43.6  35.2  7.89  2.46  2.60
## 3 Naive          Training  65.3  54.7 12.2   3.83  3.89
## 4 Seasonal_naive Training  16.8  14.3  3.31  1     1
```

These error measures are calculated over the training set. Here seasonal naive gives the better results.

```
accuracy(beer_fc, recent_production) |>
  arrange(.model) |>
  select(.model,.type,RMSE,MAE,MAPE,MASE,RMSSE)
```

```
## # A tibble: 4 x 7
##   .model         .type  RMSE   MAE  MAPE  MASE RMSSE
##   <chr>          <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Drift          Test   64.9  58.9 14.6  4.12  3.87
## 2 Mean           Test   38.4  34.8  8.28 2.44  2.29
## 3 Naive          Test   62.7  57.4 14.2  4.01  3.74
## 4 Seasonal_naive Test   14.3  13.4  3.17 0.937 0.853
```

These error measures are calculated over the test set. Here seasonal naive gives the better results.

——————— Summary ———————-

Training and test sets

It is important to evaluate forecast accuracy using genuine forecasts. Consequently, the size of the residuals is not a reliable indication of how large true forecast errors are likely to be. The accuracy of forecasts can only be determined by considering how well a model performs on new data that were not used when fitting the model.

When choosing models, it is common practice to separate the available data into two portions, training and test data, where the training data is used to estimate any parameters of a forecasting method and the test data is used to evaluate its accuracy. Because the test data is not used in determining the forecasts, it should provide a reliable indication of how well the model is likely to forecast on new data.

The size of the test set is typically about 20% of the total sample, although this value depends on how long the sample is and how far ahead you want to forecast. The test set should ideally be at least as large as the maximum forecast horizon required. The following points should be noted.

- A model which fits the training data well will not necessarily forecast well.
- A perfect fit can always be obtained by using a model with enough parameters.
- Over-fitting a model to data is just as bad as failing to identify a systematic pattern in the data.

Some references describe the test set as the "hold-out set" because these data are "held out" of the data used for fitting. Other references call the training set the "in-sample data" and the test set the "out-of-sample data". We prefer to use "training data" and "test data" in this book.

Functions to subset a time series

The `filter()` function is useful when extracting a portion of a time series, such as we need when creating training and test sets. When splitting data into evaluation sets, filtering the index of the data is particularly useful. For example,

```
aus_production |> filter(year(Quarter) >= 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
```

extracts all data from 1995 onward. Equivalently,

```
aus_production |> filter_index("1995 Q1" ~ .)
```

```
## # 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
```

can be used.

Another useful function is `slice()`, which allows the use of indices to choose a subset from each group. For example,

```
aus_production |>
  slice(n()-19:0)
```

```
## # 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
```

extracts the last 20 observations (5 years).

Slice also works with groups, making it possible to subset observations from each key. For example,

```
aus_retail |>
  group_by(State, Industry) |>
  slice(1:12)
```

```
## # 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
```

will subset the first year of data from each time series in the data.

Forecast errors

A forecast "error" is the difference between an observed value and its forecast. Here "error" does not mean a mistake, it means the unpredictable part of an observation.

Note that forecast errors are different from residuals in two ways. First, residuals are calculated on the training set while forecast errors are calculated on the test set. Second, residuals are based on one-step forecasts while forecast errors can involve multi-step forecasts.

We can measure forecast accuracy by summarising the forecast errors in different ways.

Scale-dependent errors

The forecast errors are on the same scale as the data. Accuracy measures that are based only on e_t are therefore scale-dependent and cannot be used to make comparisons between series that involve different units.

The two most commonly used scale-dependent measures are based on the absolute errors or squared errors.

When comparing forecast methods applied to a single time series, or to several time series with the same units, the MAE is popular as it is easy to both understand and compute. A forecast method that minimises the MAE will lead to forecasts of the median, while minimising the RMSE will lead to forecasts of the mean. Consequently, the RMSE is also widely used, despite being more difficult to interpret.

Percentage errors

The percentage error is given by $p_t = 100e_t/y_t$. Percentage errors have the advantage of being unit-free, and so are frequently used to compare forecast performances between data sets. The most commonly used measure is MAPE.

Measures based on percentage errors have the disadvantage of being infinite or undefined if $y_t = 0$ for any t in the period of interest, and having extreme values if any y_t is close to zero. Another problem with percentage errors that is often overlooked is that they assume the unit of measurement has a meaningful zero.4 For example, a percentage error makes no sense when measuring the accuracy of temperature forecasts on either the Fahrenheit or Celsius scales, because temperature has an arbitrary zero point.

They also have the disadvantage that they put a heavier penalty on negative errors than on positive errors. This observation led to the use of the so-called "symmetric" MAPE (sMAPE) proposed by Armstrong (1978, p. 348), which was used in the M3 forecasting competition. It is defined by

$$sMAPE = mean(200|y_t - \hat{y}_t|/(y_t + \hat{y}_t)$$

However, if y_t is close to zero, $\hat{y}_t$ is also likely to be close to zero. Thus, the measure still involves division by a number close to zero, making the calculation unstable. Also, the value of sMAPE can be negative, so it is not really a measure of "absolute percentage errors" at all.

Hyndman & Koehler (2006) recommend that the sMAPE not be used. It is included here only because it is widely used, although we will not use it in this book

Scaled errors

Scaled errors were proposed by Hyndman & Koehler (2006) as an alternative to using percentage errors when comparing forecast accuracy across series with different units. They proposed scaling the errors based on the training MAE from a simple forecast method.

For a non-seasonal time series, a useful way to define a scaled error uses naïve forecasts.Because the numerator and denominator both involve values on the scale of the original data, q_j is independent of the scale of the data. A scaled error is less than one if it arises from a better forecast than the average one-step naïve forecast computed on the training data. Conversely, it is greater than one if the forecast is worse than the average one-step naïve forecast computed on the training data.

For seasonal time series, a scaled error can be defined using seasonal naïve forecasts.

Examples

```
recent_production <- aus_production |>
  filter(year(Quarter) >= 1992)
beer_train <- recent_production |>
  filter(year(Quarter) <= 2007)

beer_fit <- beer_train |>
  model(
    Mean = MEAN(Beer),
    `Naïve` = NAIVE(Beer),
    `Seasonal naïve` = SNAIVE(Beer),
    Drift = RW(Beer ~ drift())
  )

beer_fc <- beer_fit |>
  forecast(h = 10)

beer_fc |>
  autoplot(
    aus_production |> filter(year(Quarter) >= 1992),
    level = NULL
  ) +
  labs(
    y = "Megalitres",
    title = "Forecasts for quarterly beer production"
  ) +
  guides(colour = guide_legend(title = "Forecast"))
```

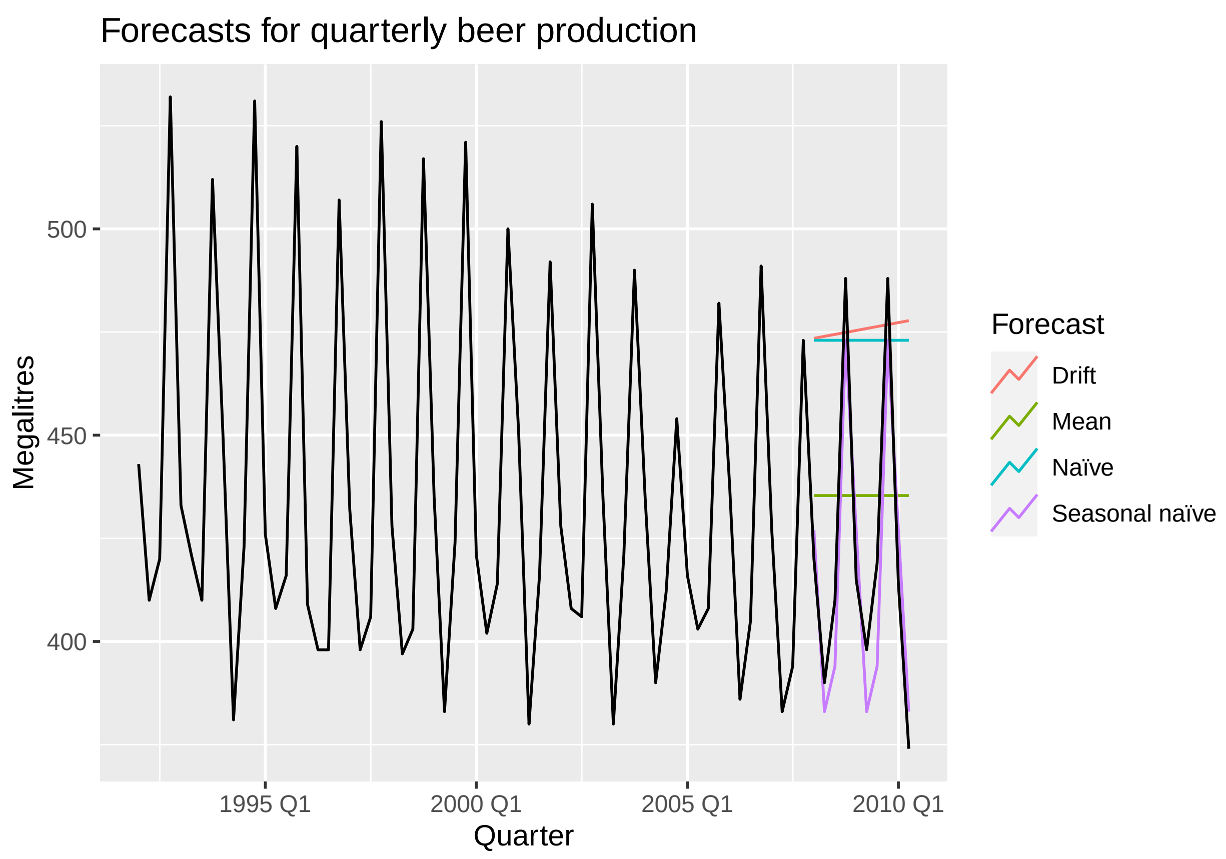


Figure 5.21 shows four forecast methods applied to the quarterly Australian beer production using data only to the end of 2007. The actual values for the period 2008–2010 are also shown. We compute the forecast accuracy measures for this period.

```
accuracy(beer_fc, recent_production)
```

```
## # A tibble: 4 x 10
##   .model         .type    ME  RMSE   MAE    MPE  MAPE  MASE RMSSE    ACF1
##   <chr>          <chr> <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>   <dbl>
## 1 Drift          Test  -54.0  64.9  58.9 -13.6  14.6  4.12  3.87  -0.0741
## 2 Mean           Test  -13.8  38.4  34.8  -3.97  8.28 2.44  2.29  -0.0691
## 3 Naïve          Test  -51.4  62.7  57.4 -13.0  14.2  4.01  3.74  -0.0691
## 4 Seasonal naïve Test    5.2  14.3  13.4   1.15  3.17 0.937 0.853  0.132
```

The `accuracy()` function will automatically extract the relevant periods from the data (recent_production in this example) to match the forecasts when computing the various accuracy measures.

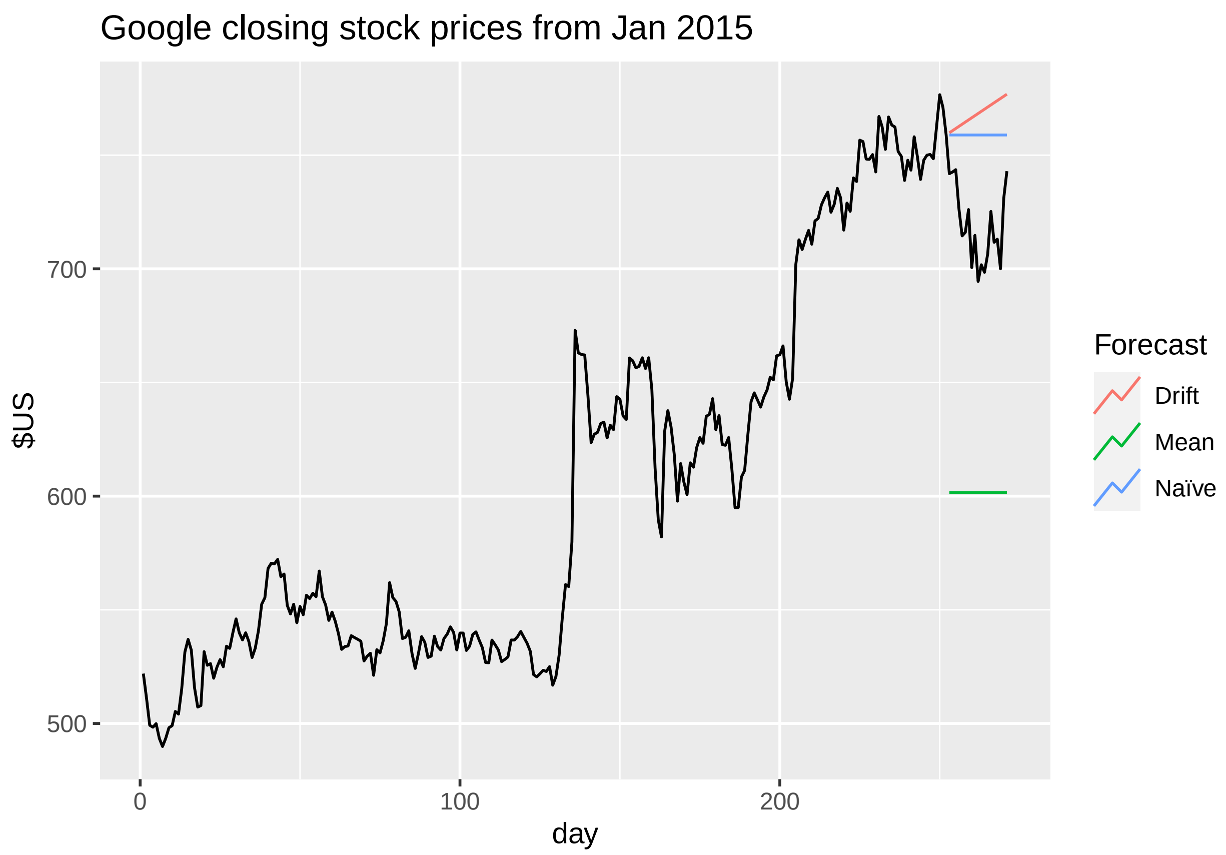
It is obvious from the graph that the seasonal naïve method is best for these data, although it can still be improved, as we will discover later. Sometimes, different accuracy measures will lead to different results as to which forecast method is best. However, in this case, all of the results point to the seasonal naïve method as the best of these four methods for this data set.

To take a non-seasonal example, consider the Google stock price. The following graph shows the closing stock prices from 2015, along with forecasts for January 2016 obtained from three different methods.

```
google_fit <- google_2015 |>
  model(
    Mean = MEAN(Close),
    `Naïve` = NAIVE(Close),
    Drift = RW(Close ~ drift())
  )

google_fc <- google_fit |>
  forecast(google_jan_2016)
```

```
google_fc |>
  autoplot(bind_rows(google_2015, google_jan_2016),
    level = NULL) +
  labs(y = "$US",
      title = "Google closing stock prices from Jan 2015") +
  guides(colour = guide_legend(title = "Forecast"))
```



```
accuracy(google_fc, google_stock)
```

```
## # A tibble: 3 x 11
##   .model Symbol .type    ME  RMSE   MAE   MPE  MAPE  MASE RMSSE  ACF1
##   <chr>  <chr>  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Drift  GOOG   Test  -49.8  53.1  49.8 -6.99  6.99  6.99  4.74 0.604
## 2 Mean   GOOG   Test  117.  118.  117.  16.2  16.2  16.4  10.5  0.496
## 3 Naïve  GOOG   Test  -40.4  43.4  40.4 -5.67  5.67  5.67  3.88 0.496
```

Here, the best method is the naïve method (regardless of which accuracy measure is used).

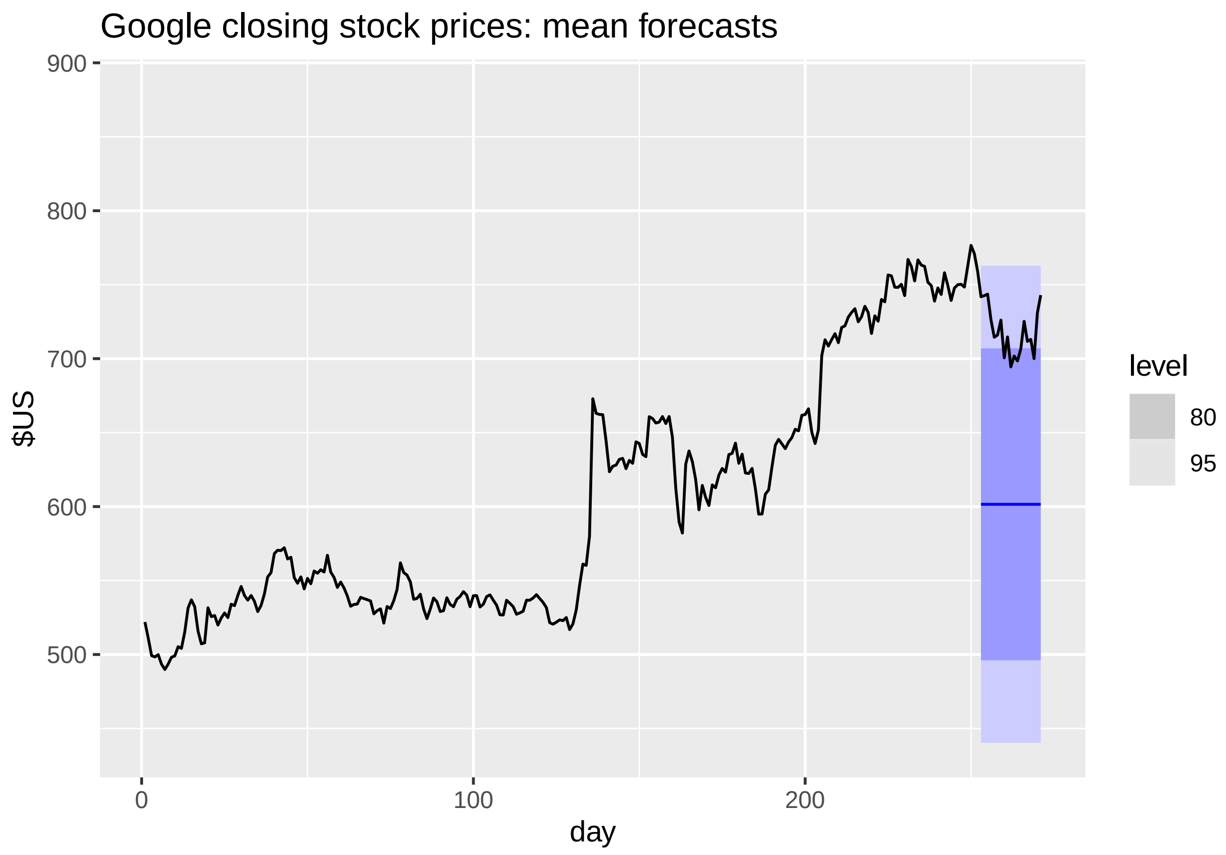
5.9 Evaluating distributional forecast accuracy

```
google_stock <- gafa_stock |>
  filter(Symbol=='GOOG',year(Date)>= 2015) |>
  mutate(day=row_number()) |>
  update_tsibble(index=day,regular=TRUE)
google_2015 <- google_stock |>
  filter(Symbol=='GOOG',year(Date)==2015)
google_jan_2016 <- google_stock |>
  filter(Symbol=='GOOG',yearmonth(Date) == yearmonth("2016 Jan"))
google_fit <- google_2015 |>
  model(
    Mean=MEAN(Close),
    Naive=NAIVE(Close),
    Drift=RW(Close~drift())
  )
google_fc <- google_fit |>
  forecast(google_jan_2016)
```

Example Here, we have used google stock data 2015 for training set and 2016 data for the test set. We fit three benchmark models that don't involve the seasonality because stock prices are not seasonal.

Now, we see the forecast:

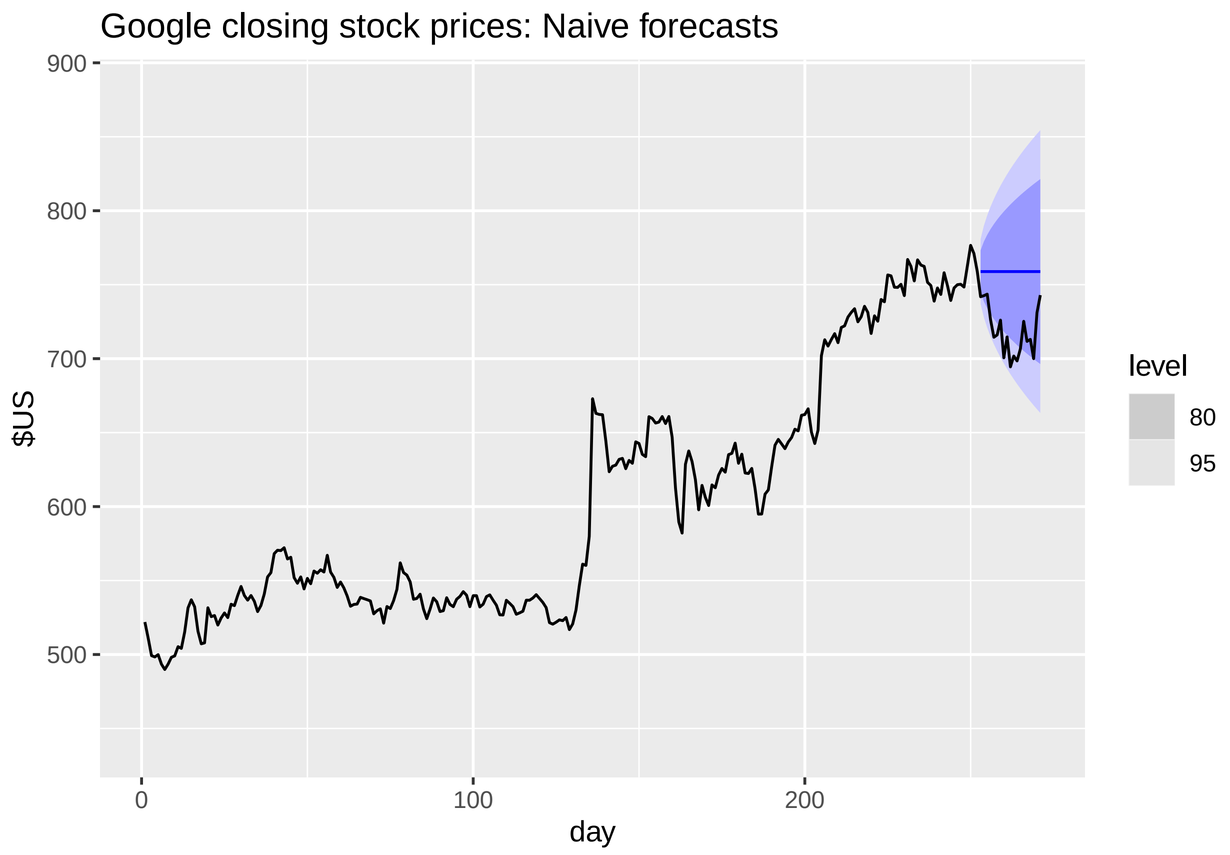
```
google_fc |>
  filter(.model == "Mean") |>
  autoplot(bind_rows(google_2015,google_jan_2016))+
  labs(y="$US",title = "Google closing stock prices: mean forecasts")+
  guides(colour=guide_legend(title = "Forecast"))+ylim(439,880)
```



Here we are using the mean model and using the train and test dataset. blue line is the mean. Here, mean model is not good. The black line is almost outside the 80% interval whereas in practice we expect it should be inside the 80% interval 80% of the time. It is never outside the 95% interval whereas in practice we expect it should be outside the 95% interval almost 5% of the time.

Now, we look at the naive forecasts.

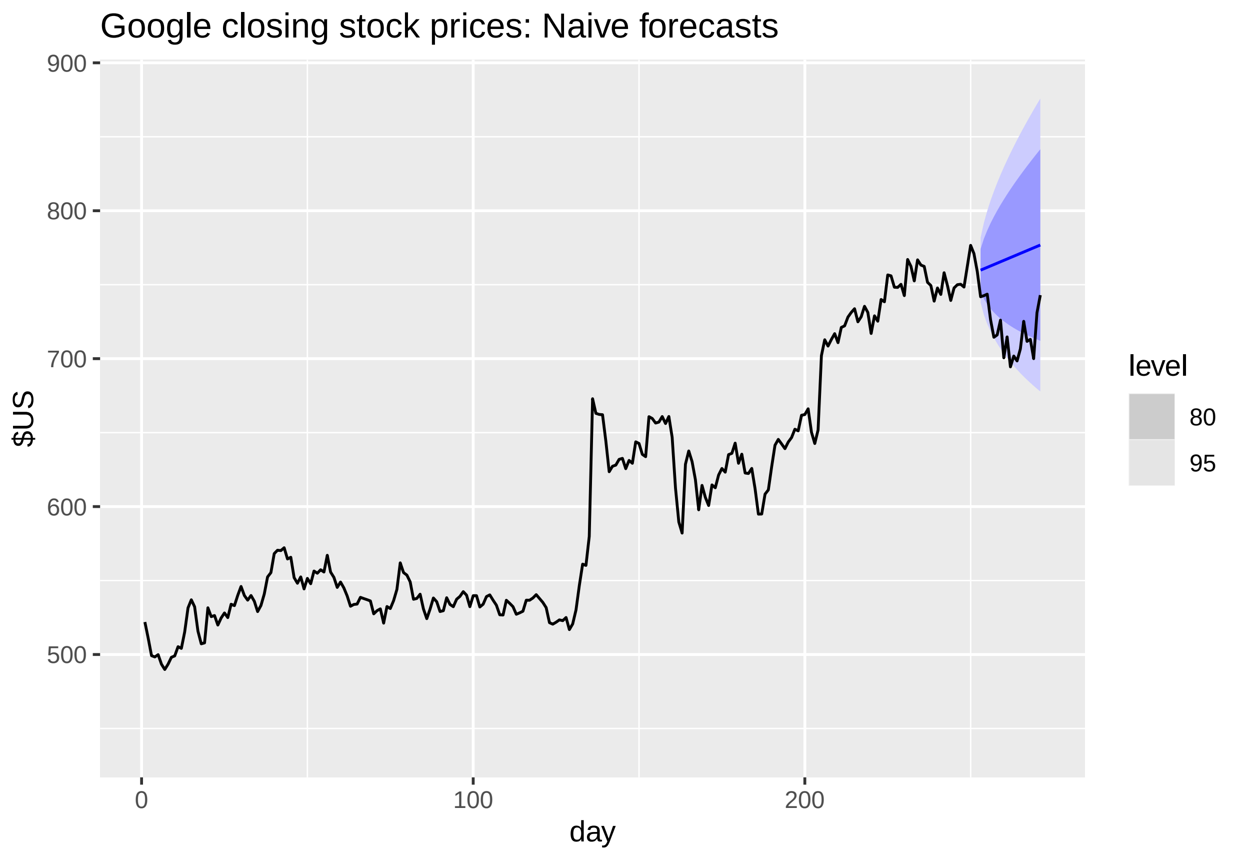
```
google_fc |>
  filter(.model=="Naive") |>
  autoplot(bind_rows(google_2015,google_jan_2016))+
  labs(y="$US",title = "Google closing stock prices: Naive forecasts")+
  guides(colour=guide_legend(title = "Forecast"))+ylim(439,880)
```



It looks little better. It is now within 80% region some of the time but not really enough of the time. None of them still go outside the 95% region.

Now, we look at the drift method.

```
google_fc |>
  filter(.model=="Drift") |>
  autoplot(bind_rows(google_2015,google_jan_2016))+
  labs(y="$US",title = "Google closing stock prices: Naive forecasts")+
  guides(colour=guide_legend(title = "Forecast"))+ylim(439,880)
```



It is little worse than the Naive method but not as bad as the mean method.

Particularly, when we have a lots of forecasts, we don't want to be looking at lots of graphs, we need a way of doing measuring the how accurate they are in some numerical fashion. so to do that we have to introduce quantile forecasts. When I plot a 80% interval, you can think it as a upper bound and lower bound like that. They are actually quantiles. 80% interval captures the 80% probability and so upper bound is at the 90th percentile and lower bound is at 10th percentile.

To do this, we use the quantiles score.

Evaluating quantile forecasts

$f_{p,t}$ = quantile forecast with probability p at time t

$y_t = $ observation at time t

$Expect\ Probability(y_t < f_{p,t}) = p$ [Proportion of time that y_t is less than $f_{p,t}$ is p]

If p is 0.9 so it's 90th percentile then we would expect proportion of times y_t is less than $f_{p,t}$ is 0.9.

If $p = 0.5$ then it is the median of the distribution and we would expect half of the time actual value to be below that and about half of the time above that.

Quantile score

$$Q_{p,t} = 2(1-p)|y_t - f_{p,t}|,\ if\ y_t < f_{p,t}$$

$$Q_{p,t} = 2p|y_t - f_{p,t}|,\ if\ y_t \geq f_{p,t}$$

- Low $Q_{p,t}$ is good.
- Multiplier of 2 often omitted, but useful for interpretation
- $Q_{p,t}$ like absolute error (weighted to account for likely excedance)

```
google_fc |>
  filter(.model=='Naive', Date=='2016-01-04') |>
  accuracy(google_stock,list(qs=quantile_score),probs=0.1)
```

```
## # A tibble: 1 x 4
##   .model Symbol .type    qs
##   <chr>  <chr>  <chr> <dbl>
## 1 Naive  GOOG   Test   4.86
```

```
google_fc |>
  filter(.model=='Naive', Date=='2016-01-04') |>
  accuracy(google_stock,list(qs=quantile_score),probs=0.9)
```

```
## # A tibble: 1 x 4
##   .model Symbol .type    qs
##   <chr>  <chr>  <chr> <dbl>
## 1 Naive  GOOG   Test   6.28
```

Here, for probability=0.9, quantile score is little worse.

Model is better at estimating the 0.1 quantile than it is estimating the 0.9 quantile and together these quantiles gives us the 80% interval.

If we are interested in evaluating an interval then we can actually use these two quantile scores in this way:

Winkler Score

For $100(1-\alpha)$ prediction interval: $[l_{\alpha,t}, u_{\alpha,t}]$.

$W_{\alpha,t} = \frac{Q_{\alpha/2,t}+Q_{(1-\alpha)/2,t}}{\alpha} =$

$= (u_{\alpha,t} - l_{\alpha,t}) + \frac{2}{\alpha}(l_{\alpha,t} - y_t)$ if $y_t < l_{\alpha,t}$

$= (u_{\alpha,t} - l_{\alpha,t})$ if $l_{\alpha,t} \leq y_t < u_{\alpha,t}$

$= (u_{\alpha,t} - l_{\alpha,t}) + \frac{2}{\alpha}(y_t - u_{\alpha,t})$ if $y_t > u_{\alpha,t}$

```
google_fc |>
  filter(.model=="Naive", Date == "2016-01-04") |>
  accuracy(google_stock,list(winkler=winkler_score),level=80)
```

```
## # A tibble: 1 x 4
##   .model Symbol .type winkler
##   <chr>  <chr>  <chr>   <dbl>
## 1 Naive  GOOG   Test     55.7
```

We can think of the winkler score as the width of interval plus the penalty if we miss the actual observation. So this is a tradeoff. If we don't want to make an interval as small as possible but if it's too small we are going to miss too often and then we are going to get lots of penalties. If we make interval too wide then we are not going to get penalized but we are going to know the width every time. Penalties are designed to perfectly balance the situation.

Here, we get the winkler score as 55.67 and according to formula, it is (quantile score for 0.1 + quantile score for 0.9)/2 = $(4.8 + 6.2)/2$

Continuous Ranked Probability Score

We are not just interested in certain prediction intervals but we are actually interested in the whole probability distribution and in that case we are interested in all the values of p. If we average the quantile score over every possible value of p between 0 and 1, we get what's called the continuous ranked probability score or the CRPS value.

```
google_fc |>
  accuracy(google_stock,list(crps=CRPS))
```

```
## # A tibble: 3 x 4
##   .model Symbol .type  crps
##   <chr>  <chr>  <chr> <dbl>
## 1 Drift  GOOG   Test   33.5
## 2 Mean   GOOG   Test   76.7
## 3 Naive  GOOG   Test   26.5
```

So, it is averaging over all the possible p values over all the possible prediction intervals that we could produce and it's telling us that the Naive method has the lowest value which is the best of all these methods. These values are in the units of data ie US$ here. To remove in point forecast we use scale errors and for the distribution forecast, we use the skill score.

Skill-free comparisons using skill scores

skill scores provide a forecast accuracy measure relative to some benchmark method (often the naive method).

$CRPS_SS_{Method} = \frac{CRPS_{Naive} - CRPS_{Method}}{CRPS_{Naive}}$

```
google_fc |>
  accuracy(google_stock,list(skill=skill_score(CRPS)))
```

```
## # A tibble: 3 x 4
##   .model Symbol .type  skill
##   <chr>  <chr>  <chr>  <dbl>
## 1 Drift  GOOG   Test  -0.266
## 2 Mean   GOOG   Test  -1.90
## 3 Naive  GOOG   Test   0
```

Inside skill_score(), we can use mean absolute error or percent error etc.

Here, we have to see the skill score for more positive values for better model.

Of course, the skill score for the naïve method is 0 because it can't improve on itself. The other two methods have larger CRPS values than naïve, so the skills scores are negative; the drift method is 26.6% worse than the naïve method.

The `skill_score()` function will always compute the CRPS for the appropriate benchmark forecasts, even if these are not included in the fable object. When the data are seasonal, the benchmark used is the seasonal naïve method rather than the naïve method. To ensure that the same training data are used for the benchmark forecasts, it is important that the data provided to the `accuracy()` function starts at the same time as the training data.

The `skill_score()` function can be used with any accuracy measure. For example, skill_score(MSE) provides a way of comparing MSE values across diverse series. However, it is important that the test set is large enough to allow reliable calculation of the error measure, especially in the denominator. For that reason, MASE or RMSSE are often preferable scale-free measures for point forecast accuracy.

Summary

The preceding measures all measure point forecast accuracy. When evaluating distributional forecasts, we need to use some other measures.

Quantile scores

Consider the Google stock price example from the previous section. Figure 5.23 shows an 80% prediction interval for the forecasts from the naïve method.

The lower limit of this prediction interval gives the 10th percentile (or 0.1 quantile) of the forecast distribution, so we would expect the actual value to lie below the lower limit about 10% of the time, and to lie above the lower limit about 90% of the time. When we compare the actual value to this percentile, we need to allow for the fact that it is more likely to be above than below.

More generally, suppose we are interested in the quantile forecast with probability p at future time$t$, and let this be denoted by $f_{p,t}$. That is, we expect the observation y_t to be less than $f_{p,t}$ with probability p. For example, the 10th percentile would be $f_{0.10,t}$

$Q_{p,t}$ is sometimes called the "pinball loss function" because a graph of it resembles the trajectory of a ball on a pinball table. The multiplier of 2 is often omitted, but including it makes the interpretation a little easier. A low value of $Q_{p,t}$ indicates a better estimate of the quantile.

The quantile score can be interpreted like an absolute error. In fact, when $p = 0.5$, the quantile score $Q_{0.5,t}$ is the same as the absolute error. For other values of p, the "error" $(y_t - f_{p,t})$ is weighted to take account of how likely it is to be positive or negative. If $p > 0.5, Q_{p,t}$ gives a heavier penalty when the observation is greater than the estimated quantile than when the observation is less than the estimated quantile. The reverse is true for $p < 0.5$.

Winkler Score

For observations that fall within the interval, the Winkler score is simply the length of the interval. Thus, low scores are associated with narrow intervals. However, if the observation falls outside the interval, the penalty applies, with the penalty proportional to how far the observation is outside the interval.

Continuous Ranked Probability Score

Often we are interested in the whole forecast distribution, rather than particular quantiles or prediction intervals. In that case, we can average the quantile scores over all values of p to obtain the Continuous Ranked Probability Score or CRPS (Gneiting & Katzfuss, 2014).

In the Google stock price example, we can compute the average CRPS value for all days in the test set. A CRPS value is a little like a weighted absolute error computed from the entire forecast distribution, where the weighting takes account of the probabilities.

Scale-free comparisons using skill scores

As with point forecasts, it is useful to be able to compare the distributional forecast accuracy of several methods across series on different scales. For point forecasts, we used scaled errors for that purpose. Another approach is to use skill scores. These can be used for both point forecast accuracy and distributional forecast accuracy.

With skill scores, we compute a forecast accuracy measure relative to some benchmark method.

5.10 Time series cross-validation

In traditional evaluation, we divide the data into training and test set.

In time series cross-validation, we actually set up lots of training sets and lots of test sets by subdividing the data at different points. We grow the training set one observation at a time and we see how model is doing for test set one step ahead. We can do the same thing for two steps ahead, 3 steps ahead and so on for different horizons. This is know as time series cross validation and sometimes known as "evaluation on a rolling forecasting origin". The forecast accuracy is averaged over all test sets.

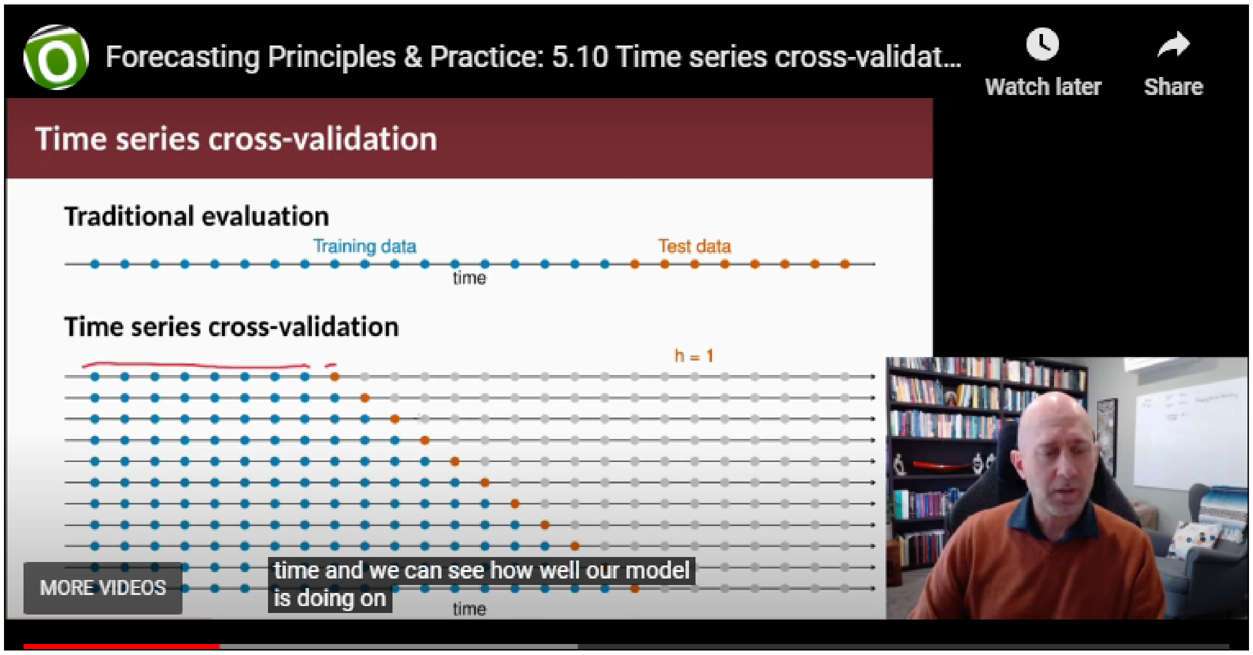

Forecasting Principles & Practice: 5.10 Time series cross-validat...
Watch later
Share
Time series cross-validation
Traditional evaluation
Training data
Test data
time
Time series cross-validation
h = 1
MORE VIDEOS
time and we can see how well our model
is doing on
time

Figure 9: Cross Validation

Stretch with a minimum length of 3, growing by 1 each step.

```
fb_stretch <- fb_stock |>
  stretch_tsibble(.init=3, .step=1) |>
  filter(.id != max(.id))
fb_stretch
```

```
## # A tsibble: 790,650 x 10 [1]
## # Key:       .id, Symbol [1,255]
##    Symbol Date        Open  High   Low Close Adj_Close  Volume trading_day   .id
##    <chr>  <date>     <dbl> <dbl> <dbl> <dbl>     <dbl>   <dbl>       <int> <int>
##  1 FB     2014-01-02  54.8  55.2  54.2  54.7      54.7 4.32e7           1     1
##  2 FB     2014-01-03  55.0  55.7  54.5  54.6      54.6 3.82e7           2     1
##  3 FB     2014-01-06  54.4  57.3  54.0  57.2      57.2 6.89e7           3     1
##  4 FB     2014-01-02  54.8  55.2  54.2  54.7      54.7 4.32e7           1     2
##  5 FB     2014-01-03  55.0  55.7  54.5  54.6      54.6 3.82e7           2     2
##  6 FB     2014-01-06  54.4  57.3  54.0  57.2      57.2 6.89e7           3     2
##  7 FB     2014-01-07  57.7  58.5  57.2  57.9      57.9 7.72e7           4     2
##  8 FB     2014-01-02  54.8  55.2  54.2  54.7      54.7 4.32e7           1     3
##  9 FB     2014-01-03  55.0  55.7  54.5  54.6      54.6 3.82e7           2     3
## 10 FB     2014-01-06  54.4  57.3  54.0  57.2      57.2 6.89e7           3     3
## # i 790,640 more rows
```

Here, ".init" tells the number of observations in our initial training set and we need a enough observation to fit a model. ".step" tells that how big you increase the training set each time. here, we are leaving max id observation for the test.

Here, we get a new column ".id" and here, initially we have id 1 and we have taken 3 observations, for id=2, we have taken 3+1=4 observations and so on.

Estimate RW w/ drift models for each window.

```
fit_cv <- fb_stretch |>
  model(RW(Close ~ drift()))
fit_cv
```

```
## # A mable: 1,255 x 3
## # Key:     .id, Symbol [1,255]
##      .id Symbol `RW(Close ~ drift())`
##    <int> <chr>                <model>
##  1     1 FB             <RW w/ drift>
##  2     2 FB             <RW w/ drift>
##  3     3 FB             <RW w/ drift>
##  4     4 FB             <RW w/ drift>
##  5     5 FB             <RW w/ drift>
##  6     6 FB             <RW w/ drift>
##  7     7 FB             <RW w/ drift>
##  8     8 FB             <RW w/ drift>
##  9     9 FB             <RW w/ drift>
## 10    10 FB             <RW w/ drift>
## # i 1,245 more rows
```

Produce one step ahead forecasts from all models.

```
fc_cv <- fit_cv |>
  forecast(h=1)
fc_cv
```

```
## # A fable: 1,255 x 6 [1]
## # Key:     .id, Symbol, .model [1,255]
##      .id Symbol .model              trading_day      Close .mean
##    <int> <chr>  <chr>                     <dbl>     <dist> <dbl>
##  1     1 FB     RW(Close ~ drift())           4 N(58, 5.8)  58.4
##  2     2 FB     RW(Close ~ drift())           5 N(59, 2.7)  59.0
##  3     3 FB     RW(Close ~ drift())           6 N(59, 1.9)  59.1
##  4     4 FB     RW(Close ~ drift())           7 N(58, 2.2)  57.7
##  5     5 FB     RW(Close ~ drift())           8 N(58, 1.7)  58.5
##  6     6 FB     RW(Close ~ drift())           9 N(56, 2.5)  56.1
##  7     7 FB     RW(Close ~ drift())          10 N(58, 2.5)  58.1
##  8     8 FB     RW(Close ~ drift())          11 N(58, 2.2)  57.9
##  9     9 FB     RW(Close ~ drift())          12   N(57, 2)  57.4
## 10    10 FB     RW(Close ~ drift())          13 N(56, 1.9)  56.4
## # i 1,245 more rows
```

```
# cross validated
fc_cv |> accuracy(fb_stock)
```

```
## # A tibble: 1 x 11
##   .model      Symbol .type      ME  RMSE   MAE     MPE  MAPE  MASE RMSSE    ACF1
##   <chr>       <chr>  <chr>   <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl>   <dbl>
## 1 RW(Close ~~ FB     Test  -0.0529  2.42  1.47 -0.0607  1.27  1.00  1.00 -0.0208
```

```
#Training set
fb_stock |>
```

```
  model(RW(Close ~ drift())) |>
  accuracy()
```

```
## # A tibble: 1 x 11
##   Symbol .model       .type    ME  RMSE   MAE      MPE  MAPE  MASE RMSSE    ACF1
##   <chr>  <chr>        <chr> <dbl> <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl>   <dbl>
## 1 FB     RW(Close ~ ~ Trai~     0  2.41  1.46 -0.00571  1.26 0.998  1.00 -0.0205
```

A good way to choose the best forecasting model is to find the model with the smallest RMSE computed using time series cross validation.

——————— Summary ————————

A more sophisticated version of training/test sets is time series cross-validation. In this procedure, there are a series of test sets, each consisting of a single observation. The corresponding training set consists only of observations that occurred prior to the observation that forms the test set. Thus, no future observations can be used in constructing the forecast. Since it is not possible to obtain a reliable forecast based on a small training set, the earliest observations are not considered as test sets.

The forecast accuracy is computed by averaging over the test sets. This procedure is sometimes known as "evaluation on a rolling forecasting origin" because the "origin" at which the forecast is based rolls forward in time.

With time series forecasting, one-step forecasts may not be as relevant as multi-step forecasts. In this case, the cross-validation procedure based on a rolling forecasting origin can be modified to allow multi-step errors to be used. Suppose that we are interested in models that produce good 4-step-ahead forecasts.

The stretch_tsibble() function is used to create many training sets. In the above example, we start with a training set of length .init=3, and increase the size of successive training sets by .step=1

The .id column provides a new key indicating the various training sets. The accuracy() function can be used to evaluate the forecast accuracy across the training sets.

As expected, the accuracy measures from the residuals are smaller, as the corresponding "forecasts" are based on a model fitted to the entire data set, rather than being true forecasts.

A good way to choose the best forecasting model is to find the model with the smallest RMSE computed using time series cross-validation.

Example: Forecast horizon accuracy with cross-validation

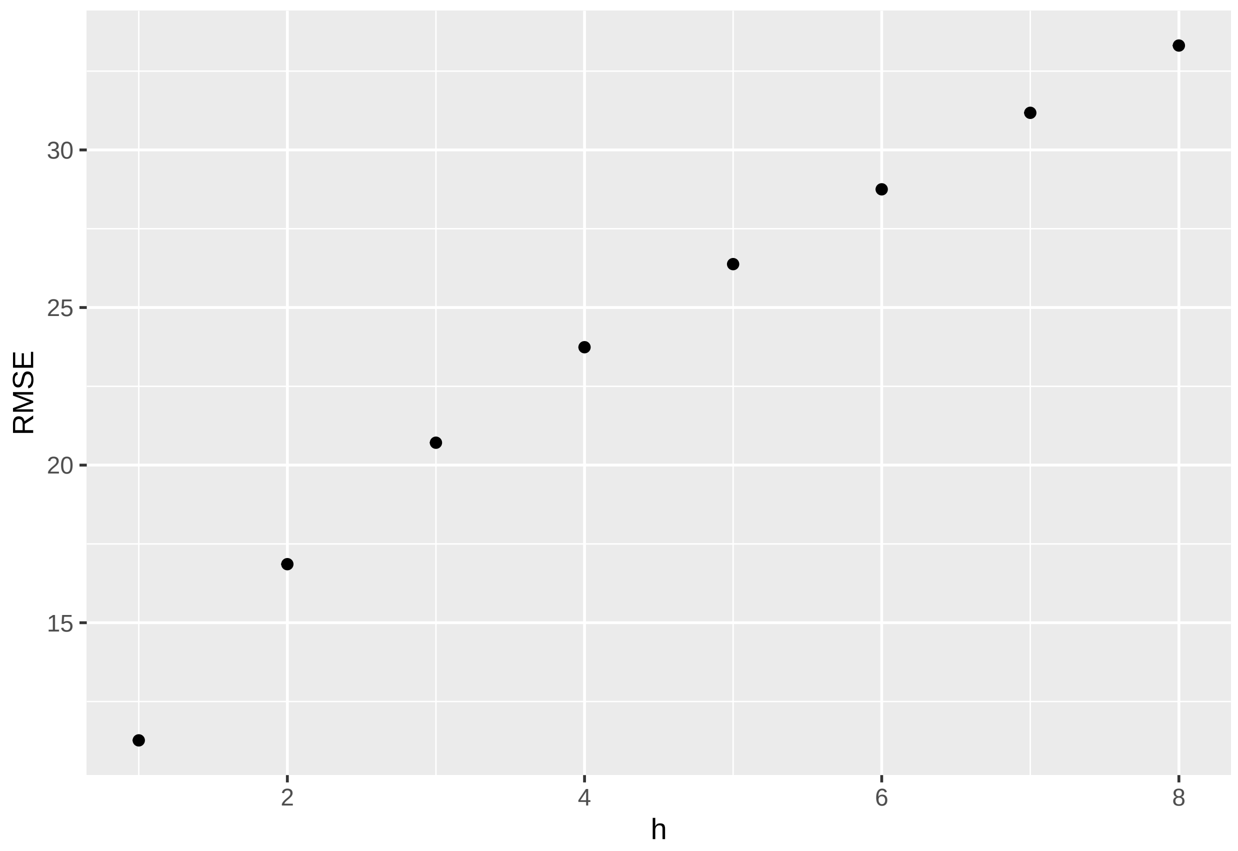
The google_2015 subset of the gafa_stock data, plotted in Figure 5.9, includes daily closing stock price of Google Inc from the NASDAQ exchange for all trading days in 2015.

The code below evaluates the forecasting performance of 1- to 8-step-ahead drift forecasts. The plot shows that the forecast error increases as the forecast horizon increases, as we would expect.

```
google_2015_tr <- google_2015 |>
  stretch_tsibble(.init = 3, .step = 1)
fc <- google_2015_tr |>
  model(RW(Close ~ drift())) |>
  forecast(h = 8) |>
  group_by(.id) |>
  mutate(h = row_number()) |>
  ungroup() |>
  as_fable(response = "Close", distribution = Close)
fc |>
  accuracy(google_2015, by = c("h", ".model")) |>
```

```r
ggplot(aes(x = h, y = RMSE)) +
geom_point()
```

```
## Warning: The future dataset is incomplete, incomplete out-of-sample data will be treated as missing.
## 8 observations are missing between 253 and 260
```


RMSE
h