EARL conference

Time Series Visualisation in R.

WORKSHOP 5.

Workshop:

github.com/MangoTheCat/ts-vis-workshop

Chapter 1

Introduction to the workshop

1.1 Aims and scope

If you have seen ambient temperature plots, electrocardiograms, or stock market fluctuations, you have already seen time series visualisations. Why not be able to create your own?

Time series is a rather distinct concept in analytics, and data scientists often find it hard to get started. Most textbooks focus on modelling, which may require some degree of mathematical rigour. This short course aims at introducing the concept through creating and examining plots, taking advantage of the exceptional plotting capabilities of the R language. At the end of the course, participants will be able to create time series objects from their data, plot them in various ways, thus adding a very powerful tool in their exploratory data analysis toolkit.

1.2 R packages

The following is a list of R packages we will need for this workshop.

```r
library(dplyr)
library(ggplot2)
library(forecast)
library(zoo)
library(xts)
library(lattice)
library(tsibble)
library(feasts)
library(purrr)
library(imputeTS)
library(fable)
```

In many cases, we will additionally use the "`::`" notation to refer to the package a function is coming from, for the readers' convenience.

For several tasks in this workshop we use the R package `{forecast}`, which provides methods and tools for displaying and analysing univariate time series forecasts. Although this package is now retired in favour of the `{fable}` package, it's still maintained and frequently encountered in existing code. A reference to the equivalent functions of both packages will be given, where appropriate.

1.3 Workshop data

For this workshop we will use a synthetic dataset representing sales of a retail business.

```
z <- readRDS("sales_EARL.rds")
```

This is how the dataset looks like:

```
head(z)
#>             value  online in_store electronics cosmetics  toys clothing
#> 1 sales_2015_01 237.980   87.293       58.260    80.326 1.719   58.182
#> 2 sales_2015_02 217.770   85.586       57.972    89.014 3.250   49.312
#> 3 sales_2015_03 226.641   86.902       51.610   101.889 3.091   57.799
#> 4 sales_2015_04 227.654   89.266       53.611   106.459 3.055   58.984
#> 5 sales_2015_05 238.254   87.219       55.177   108.781 4.873   50.614
#> 6 sales_2015_06 258.113   85.798       66.895   116.092 3.791   52.401
#>      food   total
#> 1 126.786 325.273
#> 2 103.809 303.356
#> 3  99.154 313.543
#> 4  94.811 316.920
#> 5 106.028 325.473
#> 6 104.732 343.911
```

Disclaimer: The dataset contains entirely *synthetic* data, and is not subject to copyright, confidentiality agreement, or any other restrictions.

This dataset contains the total sales of a retail store (column `total`) broken down to:

- `online` sales, and
- `in_store` sales

The total sales are also broken down by the type of goods into:

- `electronics`
- `cosmetics`
- `toys`
- `clothing`
- `food`

These variables were recorded once per month over the course of 6 years and 3 months, between January 2015 and March 2022.

Let's check if the total sum of `electronics`, `cosmetics`, `toys`, `clothing`, and `food` matches the `total` column.

```
z %>%
  mutate(tot_goods = rowSums(across(electronics:food))) %>%
  transmute(tot_diff = round(total - tot_goods, 2)) %>%
  summary()
#>     tot_diff
#>  Min.   :0
#>  1st Qu.:0
#>  Median :0
#>  Mean   :0
#>  3rd Qu.:0
#>  Max.   :0
```

Check if the sum of `online` and `in_store` sales matches the `total`, by modifying the `mutate()` call above.

```
z %>%
  mutate(tot_mode = online + in_store) %>%
  transmute(tot_diff = round(total - tot_mode, 2)) %>%
  summary()
#>      tot_diff
#>   Min.   :0
#>   1st Qu.:0
#>   Median :0
#>   Mean   :0
#>   3rd Qu.:0
#>   Max.   :0
```

Now we are ready to begin our analysis.

Chapter 2
Timeless data

For this chapter we will temporarily ignore the temporal aspect of the data, and view them as “timeless” data.

2.1 Describing a single variable

Suppose you were asked to analyse a small dataset, without any further instructions on the exact objective of the analysis. Let’s say that the `total` column of our data frame (total sales) represents this dataset.

```
xt <- z$total
xt %>% head()
#> [1] 325.273 303.356 313.543 316.920 325.473 343.911
```

The dataset is just a numeric vector. You can start analysing the data by obtaining an idea about the average and the range:

```
avg <- mean(xt)
summary(xt)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   241.4   321.9   340.2   334.7   349.0   392.6
```

For measuring the variability there are several metrics, such as the standard deviation, the interquartile range (IQR), or the coefficient of variation:

```
std <- sd(xt)
iqr <- IQR(xt)
c(std, IQR(xt), 100 * std/avg)
#> [1] 23.935519 27.117500  7.152231
```

Find any observations in the dataset that are likely to be outliers. You can use the quantities `avg`, `std`, and `iqr` we have already computed.

1. Use the heuristic of three standard deviations about the mean as your decision criterion.

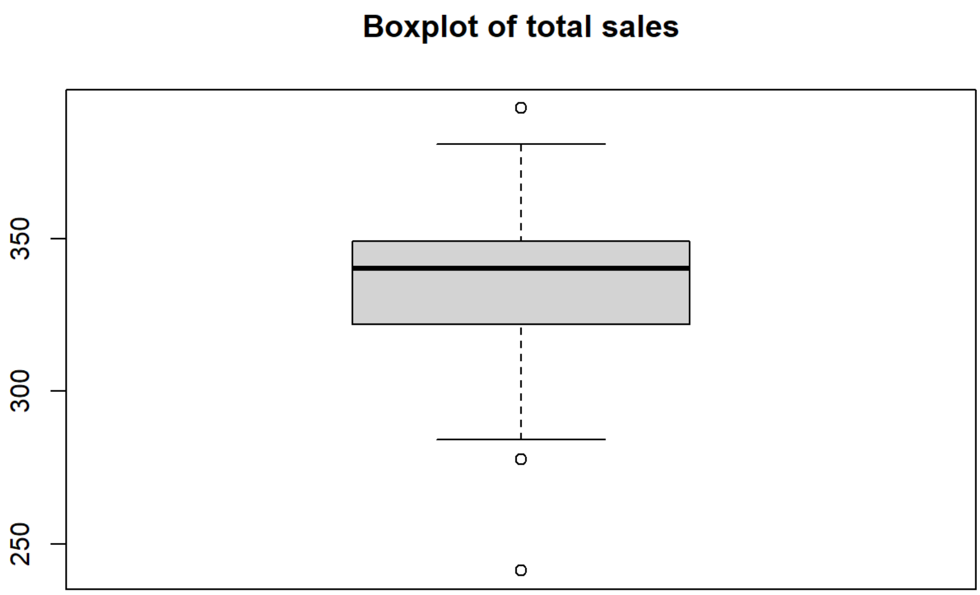
2. Change your criterion to be the interval [Q1 - IQR, Q3 + IQR], where Q1 and Q3 are the 25% and the 75% quantiles. Does the result change?

```
data.frame(xt = xt) %>%
  mutate(
    id = row_number(),
    is_out1 = !between(xt, avg - 3 * std, avg + 3 * std),
    is_out2 = !between(xt,
                       quantile(xt, 0.25) - 1.5 * iqr,
                       quantile(xt, 0.75) + 1.5 * iqr)
    ) %>%
  filter(is_out1 | is_out2)
#>        xt id is_out1 is_out2
#> 1 241.385 12    TRUE    TRUE
#> 2 277.659 48   FALSE    TRUE
#> 3 392.581 87   FALSE    TRUE
```

2.2 Visualising a single variable

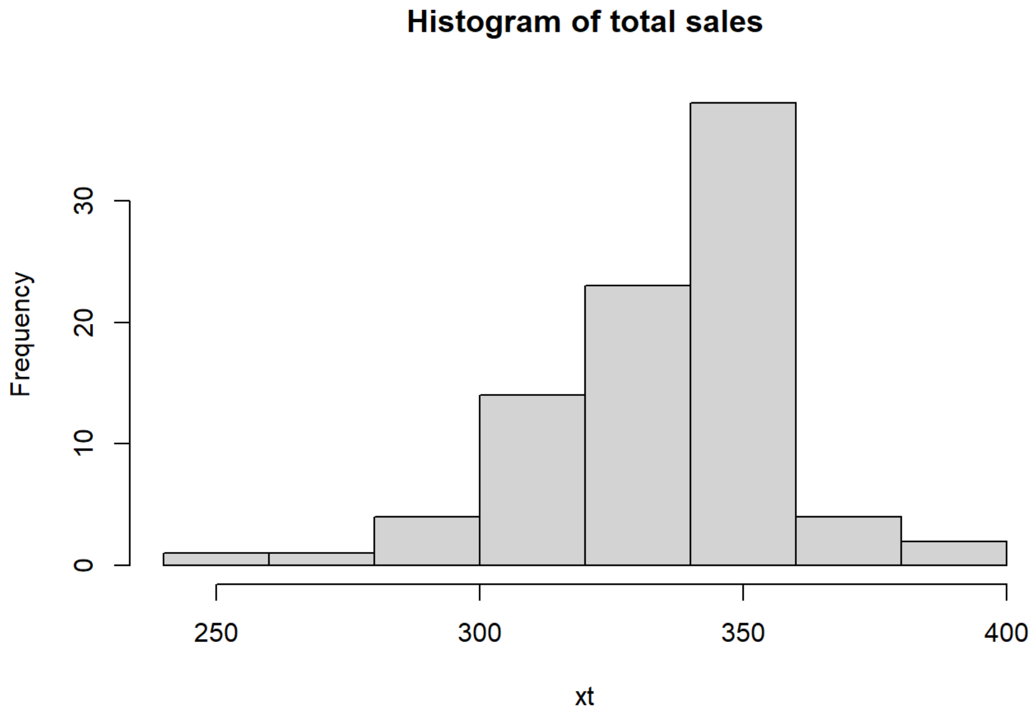
To visualise the summary and spot possible outliers at the same time, the simplest option is the boxplot:

```
boxplot(xt, main = "Boxplot of total sales")
```



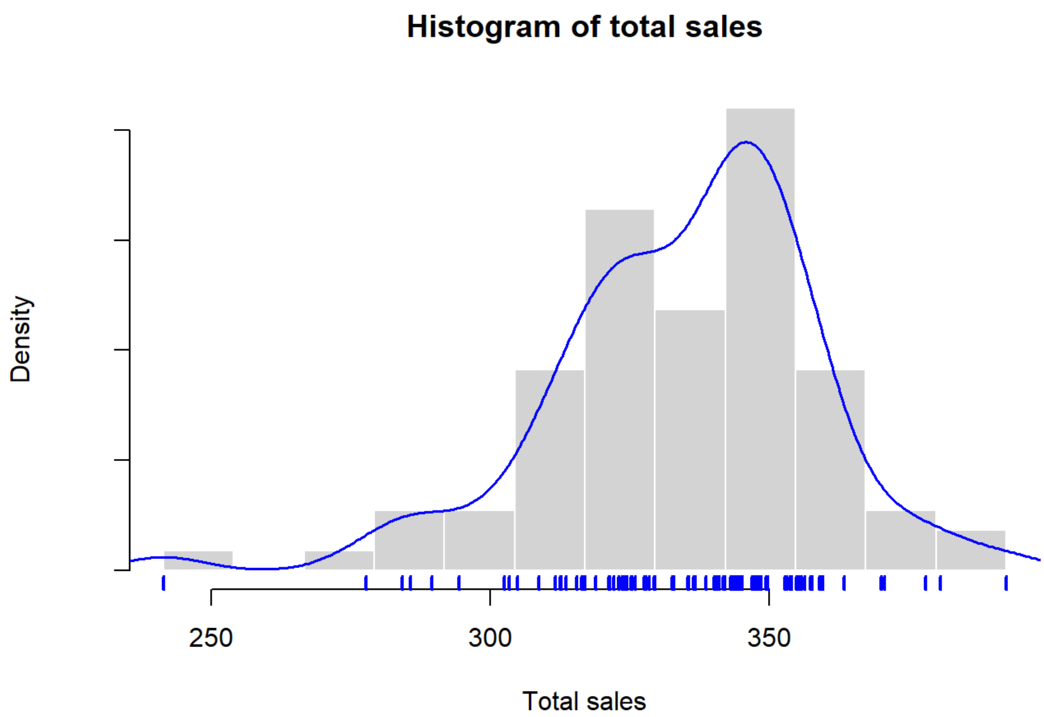
While boxplots are great for comparing multiple variables, when analysing a single variable, the histogram provides a more detailed picture:

```
hist(xt, main = "Histogram of total sales")
```



We can add some more visual tools to a histogram, such as better binning, the density, and the rug plot at the bottom.

```
hist(xt, prob = TRUE, border = "white", yaxt = "n",
     breaks = seq(min(xt), max(xt), length.out = 1 + 12),
     main = "Histogram of total sales", xlab = "Total sales")
axis(side = 2, labels = FALSE)
lines(density(xt), lwd = 1.5, col = "blue")
rug(xt, lwd = 2.0, col = "blue", ticksize = 0.025)
```



These tasks are part of **descriptive statistics:** We produce plots, summaries, and various metrics in order to answer questions such as:

- How do the data look like?
- What is the average?
- What is the spread?
- How are the data distributed?
- Are there any outliers?

There are ways to generalise these tasks to a multivariate setting. However, our ability to perform inference, prediction, and forecast tasks is very limited at this stage.

2.3 Inference with a single variable

We move from descriptive to **inferential statistics** when we want to use our sample for testing hypotheses and for drawing conclusions about the true effect we are measuring.

Suppose now that you receive new instructions about the small dataset you analysed in the previous section, asking you to test whether the true average of these values is greater than 330.

The **t-test** is a classic tool to determine if there is a statistically significant difference.

```
t.test(xt, mu = 330, alternative = "greater")
#> 
#>  One Sample t-test
#> 
#> data:  xt
#> t = 1.8152, df = 86, p-value = 0.03649
#> alternative hypothesis: true mean is greater than 330
#> 95 percent confidence interval:
#>  330.3911      Inf
#> sample estimates:
#> mean of x 
#>  334.6581
```

The Wilcoxon test is a common alternative to the t-test. It's *non-parametric*, so it's particularly recommended when the data are not normally distributed.

```
wilcox.test(xt, mu = 330, alternative = "greater")
#> 
#>  Wilcoxon signed rank test with continuity correction
#> 
#> data:  xt
#> V = 2500, p-value = 0.006605
#> alternative hypothesis: true location is greater than 330
```

The p-values tell us that we have enough evidence to support that the true average of our values is greater than 330.

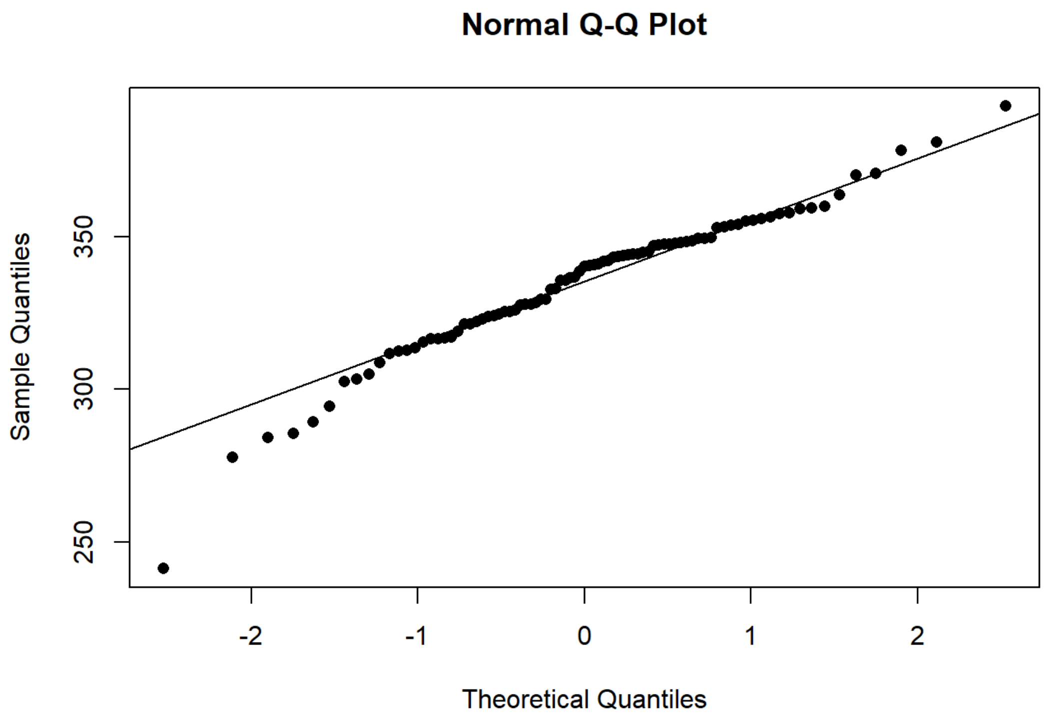
Repeat the same tests with the value 331 instead of 330. Do the results agree? If not, which test would you rather trust?

```
t.test(xt, mu = 331, alternative = "greater")$p.value
#> [1] 0.07881608
wilcox.test(xt, mu = 331, alternative = "greater")$p.value
#> [1] 0.0187527
```

2.4 Data normality

It's important to know if the data are normally distributed, as many inference and modelling methods depend on this assumption. We can visually check that with the socalled **Q–Q plot** (quantile-quantile plot):

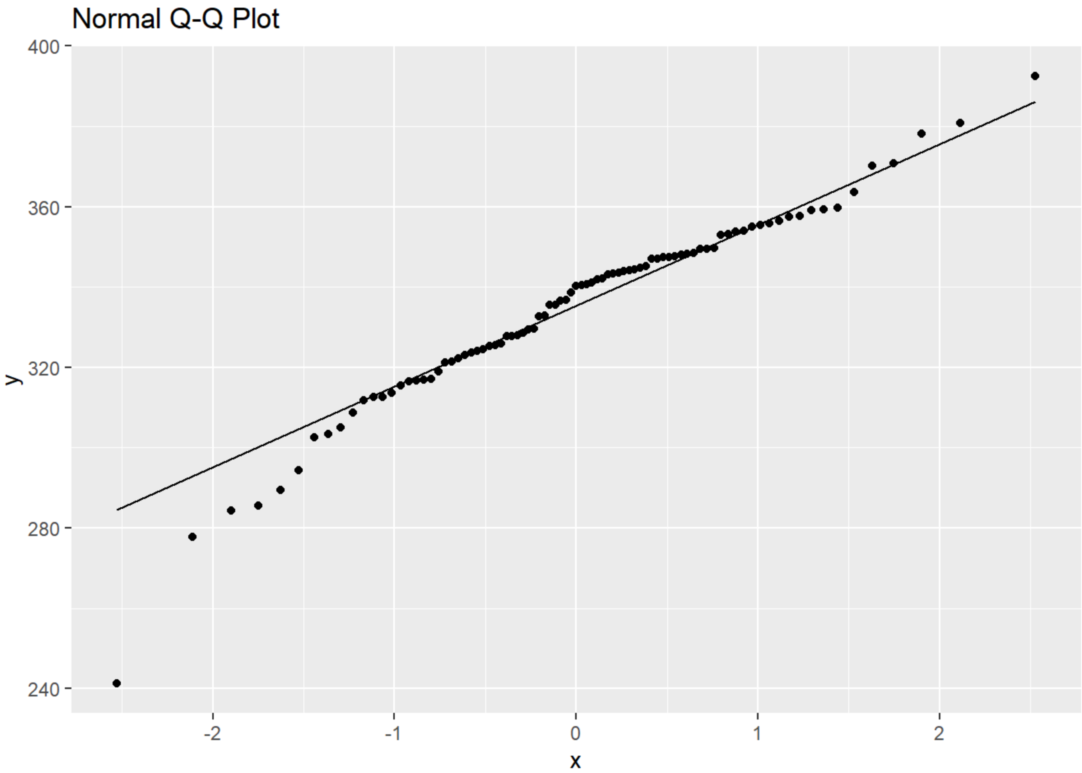
```
qqnorm(xt, pch = 16)
qqline(xt)
```



Using `{ggplot2}` requires a bit more coding:

```
data.frame(xt = xt) %>%
  ggplot(aes(sample = xt)) +
  stat_qq() +
  stat_qq_line() +
  ggtitle("Normal Q-Q Plot")
```

2.4 Data normality



Apply the Shapiro-Wilk normality test to the data using the `shapiro.test()` function. Are the data normally distributed according to the output?

The null hypothesis of the test is that the population is normally distributed. If the p-value is smaller than the alpha level 0.05, the null hypothesis is rejected, and there is evidence that the data are not normally distributed.

```
shapiro.test(xt)
#> 
#>  Shapiro-Wilk normality test
#> 
#> data:  xt
#> W = 0.95859, p-value = 0.007114
```

Chapter 3

Introduction to time series

When you collect your data today, then all the conclusions you make about your data are only relevant for today. But if you want to know how the data will look like tomorrow (or next month, or next year), you need to assume that everything related to the data will be the same as today. This is a static view of the world. We refer to this as “stationarity”.

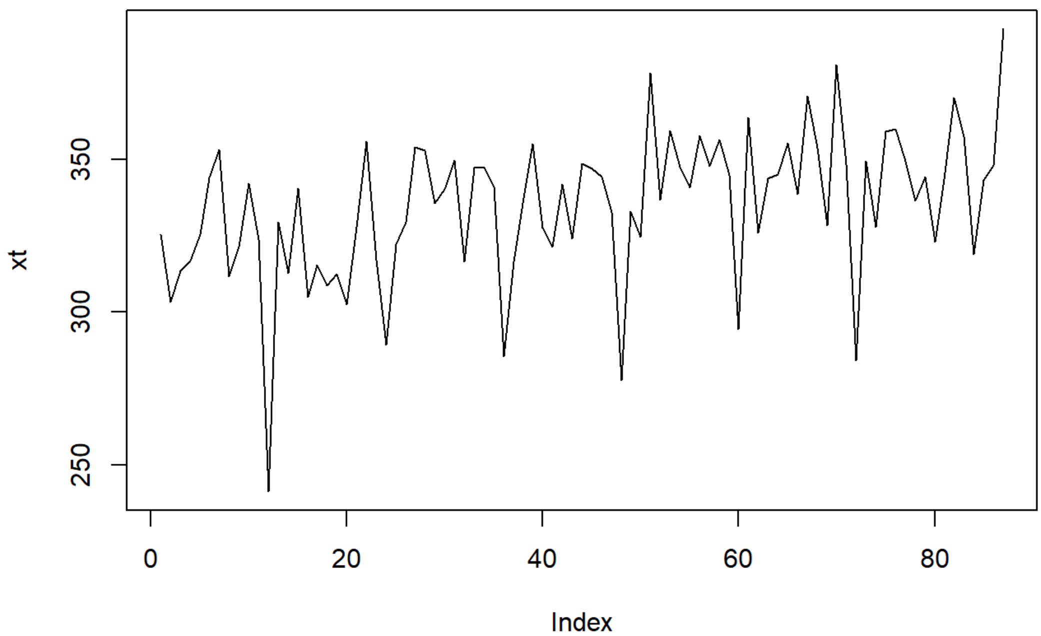
Instead, we need to find ways to capture the patterns and the dynamics of our data, and to follow them into the future. Then, the data will be able to guide us to more reliable forecasts.

3.1 The concept of time series

Let’s return to the small dataset we analysed in the previous chapter. Suppose now that you received newer information about the data: you know that they actually represent 87 recorded observations of the total sales of a retail store. In addition, you know that your measurements are time-ordered, and that they are monthly observations, so they are recorded in regular points in time.

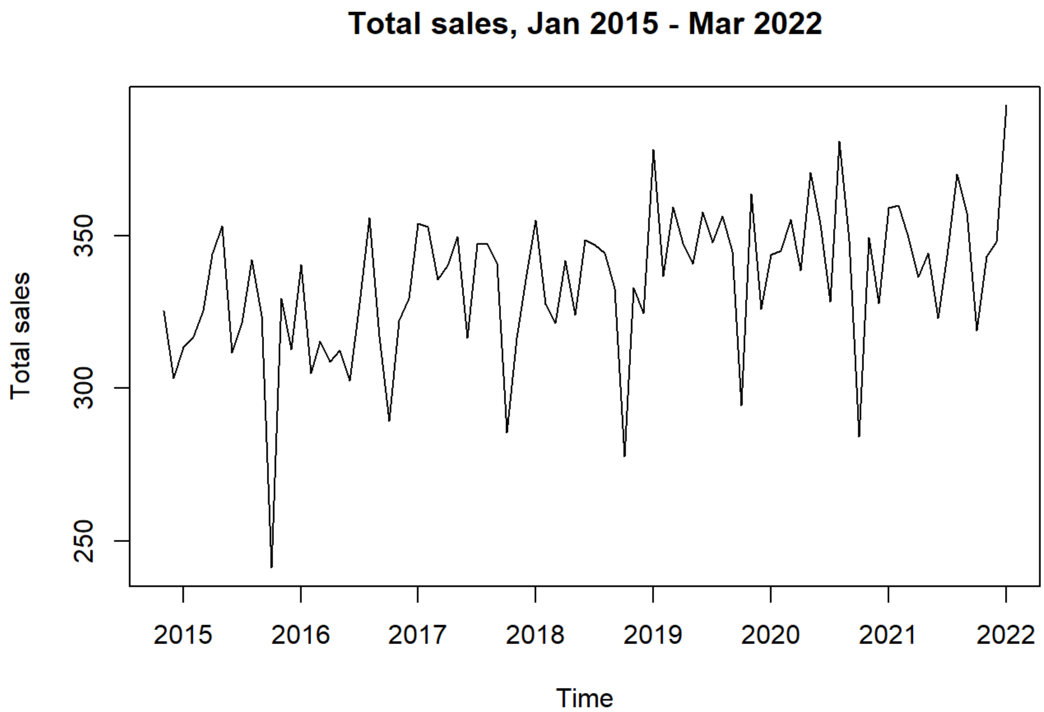
We can now make a plot using base R graphics, as follows:

```
plot(xt, type = "l")
```



Finally, if we also know that the sales data span from January 2015 to March 2022, we can create a proper x-axis label and finalise our very first time series plot:

```
plot(z$total, type = "l", xaxt = "n",
     xlab = "Time", ylab = "Total sales",
     main = "Total sales, Jan 2015 - Mar 2022")
axis(side = 1, at = 12*(0:7) + 3, labels = 2015:2022)
```



At first glance the data seem noisy and random, but by looking carefully we can observe a slight upward trend and a pattern in the monthly fluctuations. In the following, we will investigate if the data are completely random, or have at least one of the two important properties:

- Future values depend on past values (“autoregression”)
- Data follow seasonal patterns (“seasonality”)

This is the concept of time series: data with a time component, usually (but not necessarily) recorded at regular intervals, which depict the development of an effect over time, and possibly some hidden patterns related to a trend or a seasonality, plus some noise.

Can you think of any real-world examples of time series data?

3.2 Creating a time series object

There are special classes in R for time series data. The easiest way to create a time series object is with the `ts()` function:

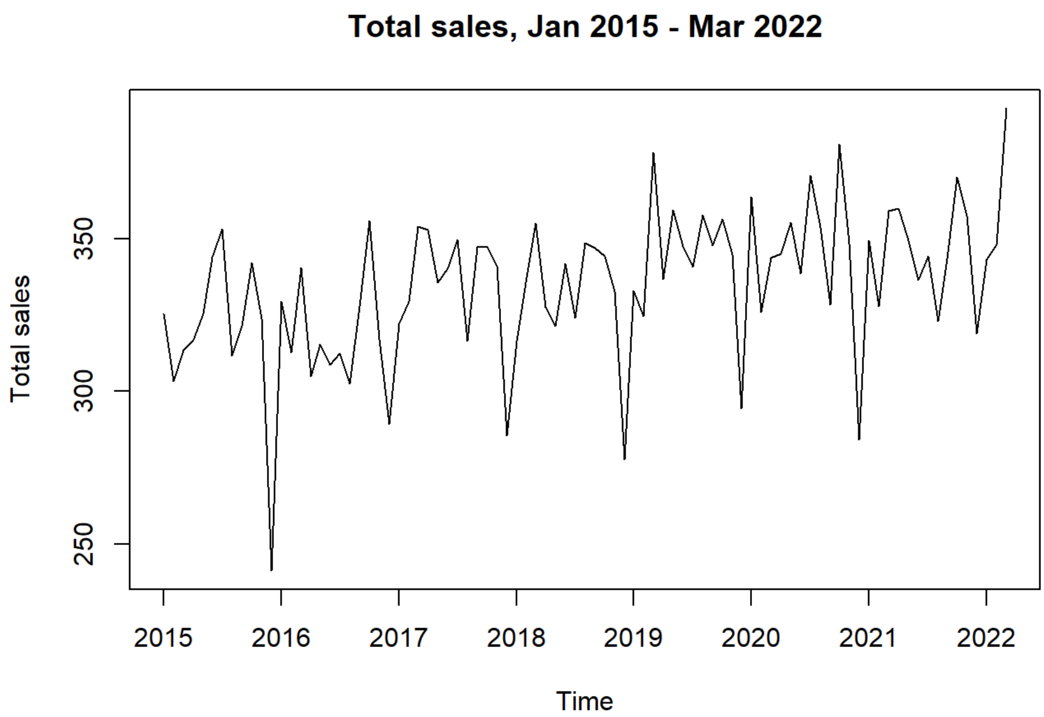
```
X <- z %>%
  select(online:total) %>%
  ts(start = c(2015, 1), frequency = 12)
head(X)
#>           online in_store electronics cosmetics  toys clothing
#> Jan 2015 237.980   87.293      58.260    80.326 1.719   58.182
#> Feb 2015 217.770   85.586      57.972    89.014 3.250   49.312
#> Mar 2015 226.641   86.902      51.610   101.889 3.091   57.799
#> Apr 2015 227.654   89.266      53.611   106.459 3.055   58.984
#> May 2015 238.254   87.219      55.177   108.781 4.873   50.614
#> Jun 2015 258.113   85.798      66.895   116.092 3.791   52.401
#>             food   total
#> Jan 2015 126.786 325.273
#> Feb 2015 103.809 303.356
#> Mar 2015  99.154 313.543
#> Apr 2015  94.811 316.920
#> May 2015 106.028 325.473
#> Jun 2015 104.732 343.911
```

Now we have an object containing multiple time series that share a common time component. We can extract individual components as we do with ordinary data frames.

```
Xt <- X[, "total"]  # Get total sales
class(Xt)
#> [1] "ts"
```

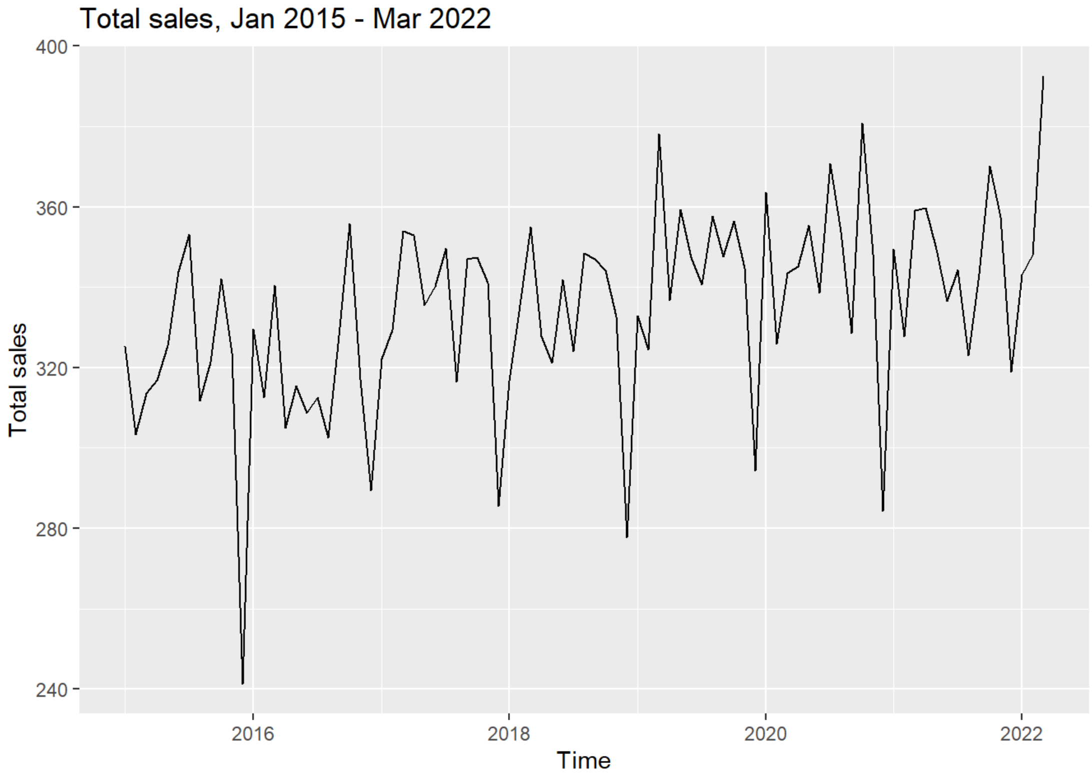
Once you have a time series object, you can plot it using the `plot()` function from base R:

```
title_ts <- "Total sales, Jan 2015 - Mar 2022"
base::plot(Xt, ylab = "Total sales", main = title_ts)
```



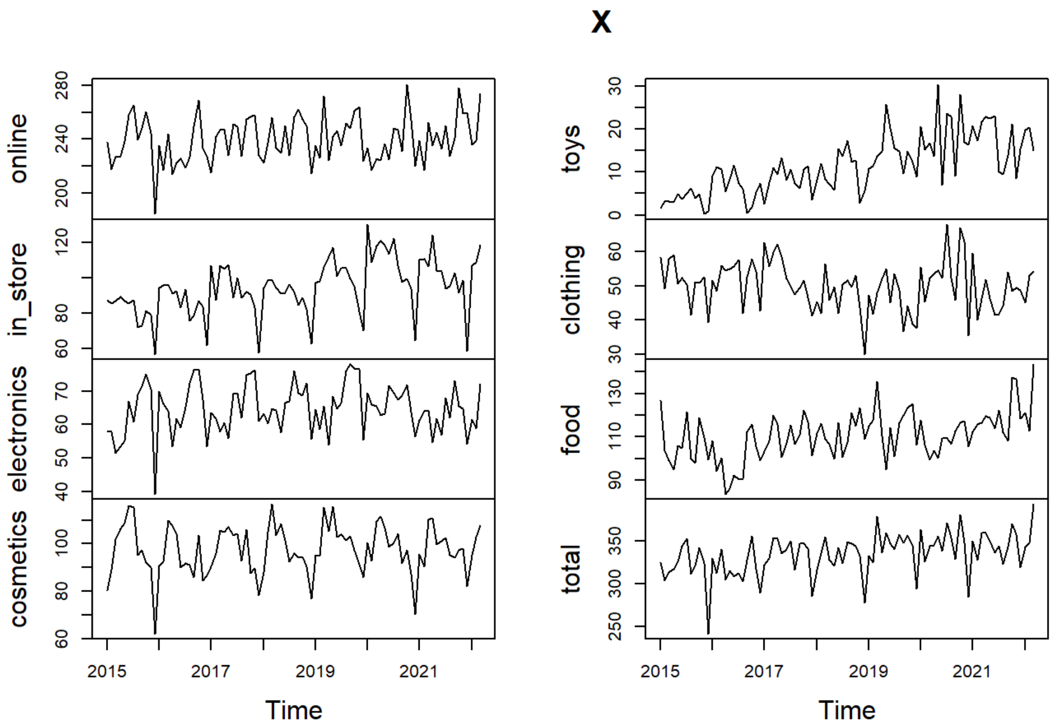
Notice how the x-axis label was automatically created. Using `{ggplot2}`:

```
ggplot2::autoplot(Xt, ylab = "Total sales") +
  ggtitle(title_ts)
```



For multivariate time series objects, `plot()` will split into separate panels:

```
plot(X)
```

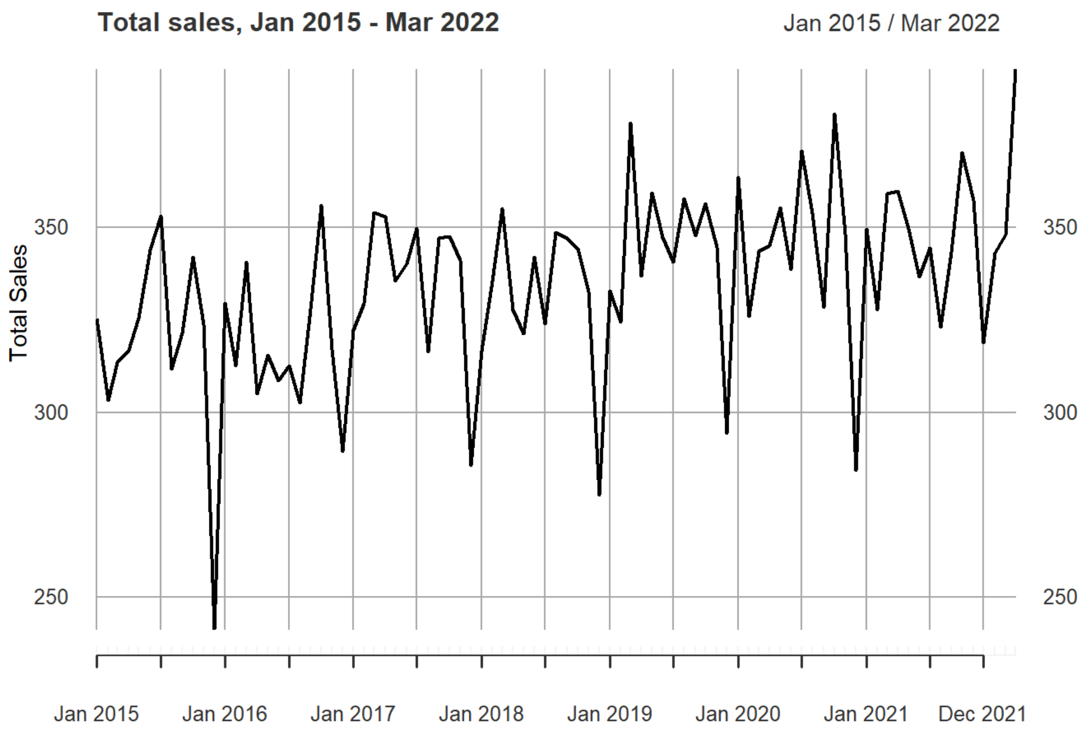


Another popular class for storing and manipulating time series data, with many additional features, is implemented in the `{zoo}` package:

```
zoo::zoo(X) %>% class()
#> [1] "zoo"
```

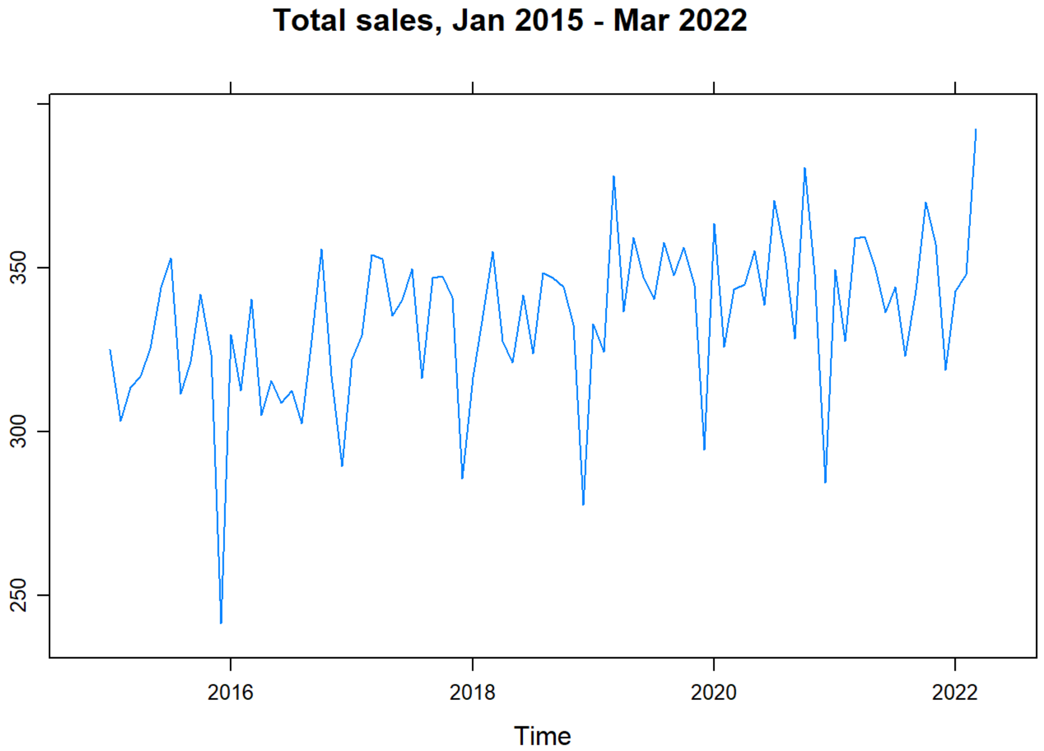
The `{xts}` package extends `{zoo}` and provides also interesting plotting capabilities:

```
plot(xts::as.xts(Xt), ylab = "Total Sales", main = title_ts)
```

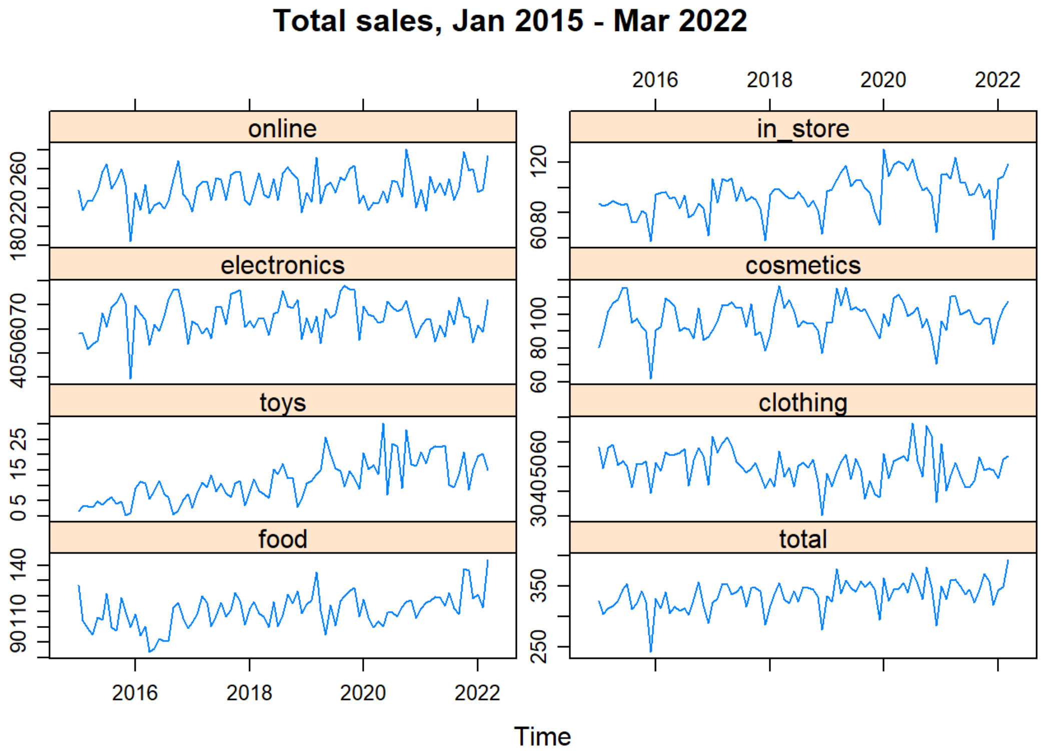


The `{lattice}` package (that comes pre-installed with base R as “recommended”), includes the `xyplot()` function that can plot one or multiple time series:

```
lattice::xyplot(Xt, main = title_ts)
```

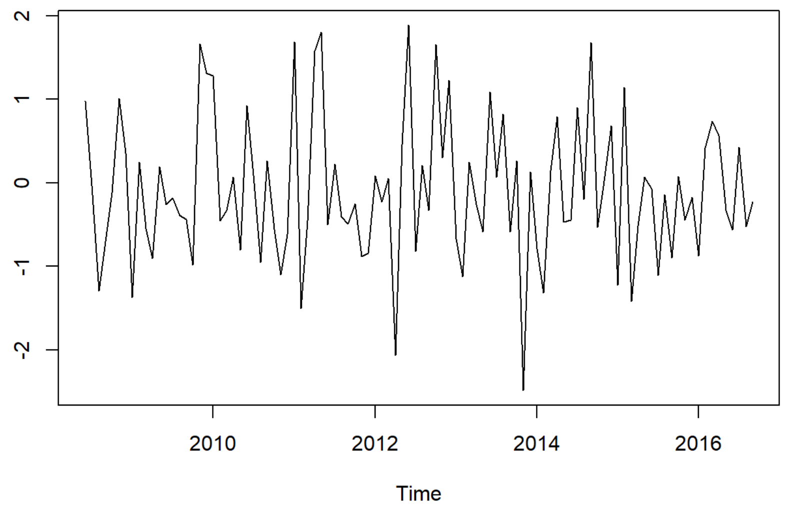



```r
lattice::xyplot(X, main = title_ts)
```



Create a time series with 100 random variates from the standard normal distribution using the `rnorm()` function, and plot them as a time series starting from June 2008. Use any plotting function you like. Do you notice any patterns in your data?

```
rnorm(100) %>%
  ts(start = c(2008, 6), frequency = 12) %>%
  plot()
```

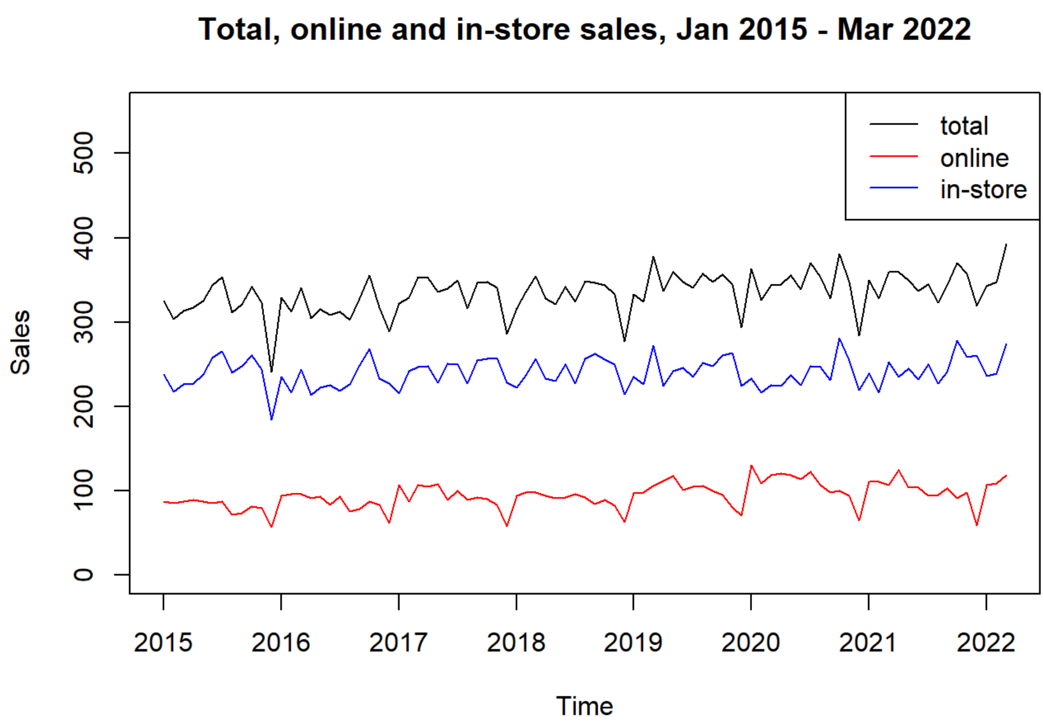


Would you argue that, given the data are random, there isn't ant useful information for producing a forecast?

3.3 Plotting multiple time series

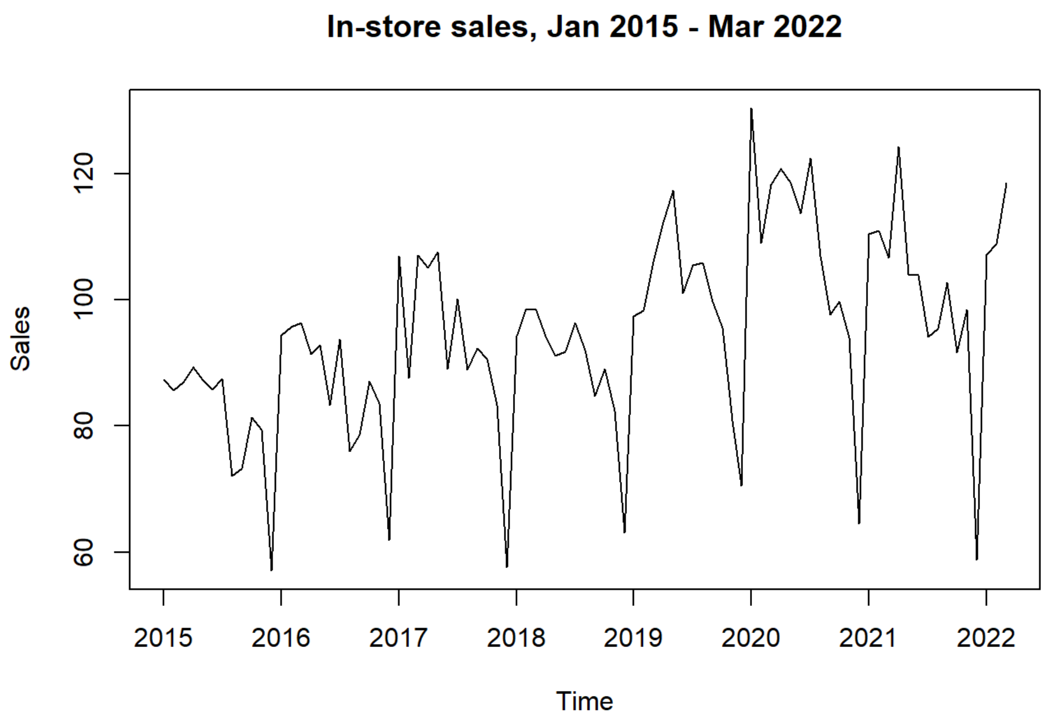
Plotting multiple time series on the same plot can be very helpful in spotting common patterns and differences between time series.

```
plot(X[, "total"], ylim = c(0, 550),
     xlab = "Time", ylab = "Sales",
     main = "Total, online and in-store sales, Jan 2015 - Mar 2022")
lines(X[, "in_store"], col = "red")
lines(X[, "online"], col = "blue")
legend("topright", col = c("black", "red", "blue"), lty = 1,
       c("total", "online", "in-store"))
```

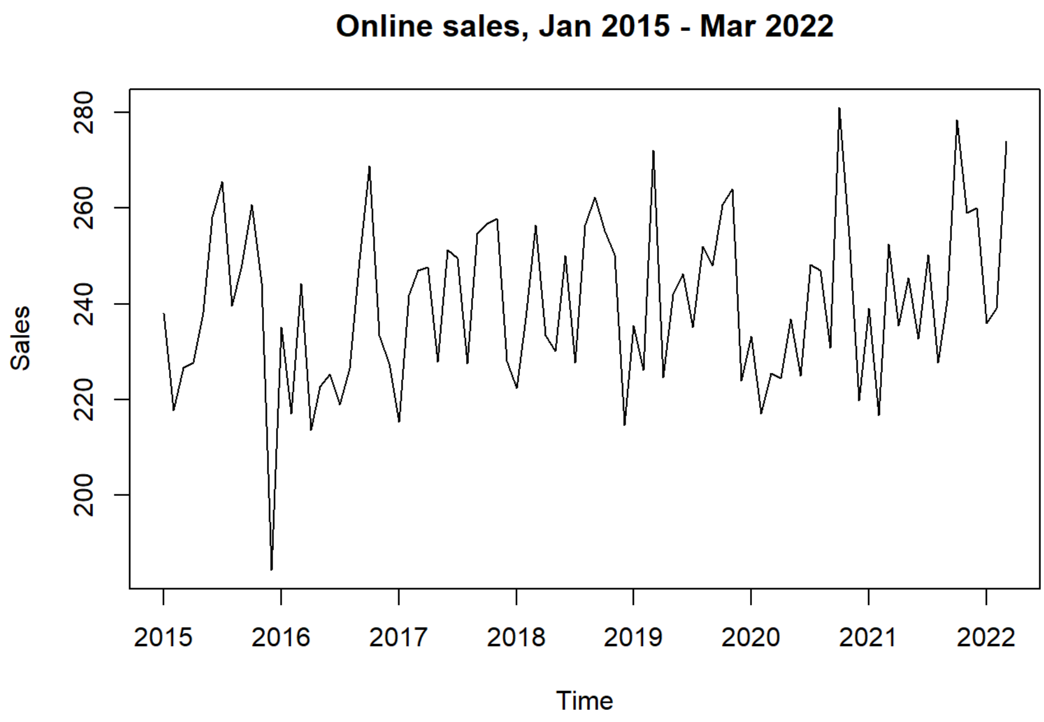


However, when we plot online and in-store sales separately, we are able to see the characteristics of these time series more clearly. Therefore, plotting multiple time series on the same plot can sometimes be misleading, as the all tend to look about “the same”.

```
X[, "in_store"] %>%
  plot(ylab = "Sales", main = "In-store sales, Jan 2015 - Mar 2022")
```




```
X[, "online"] %>%
  plot(ylab = "Sales", main = "Online sales, Jan 2015 - Mar 2022")
```



Try plotting the "electronics", "cosmetics", "toys", "clothing", "food" on the same plot, and separately. Which time series seems to stand out in terms of shape?

Chapter 4

Visual exploration of time series

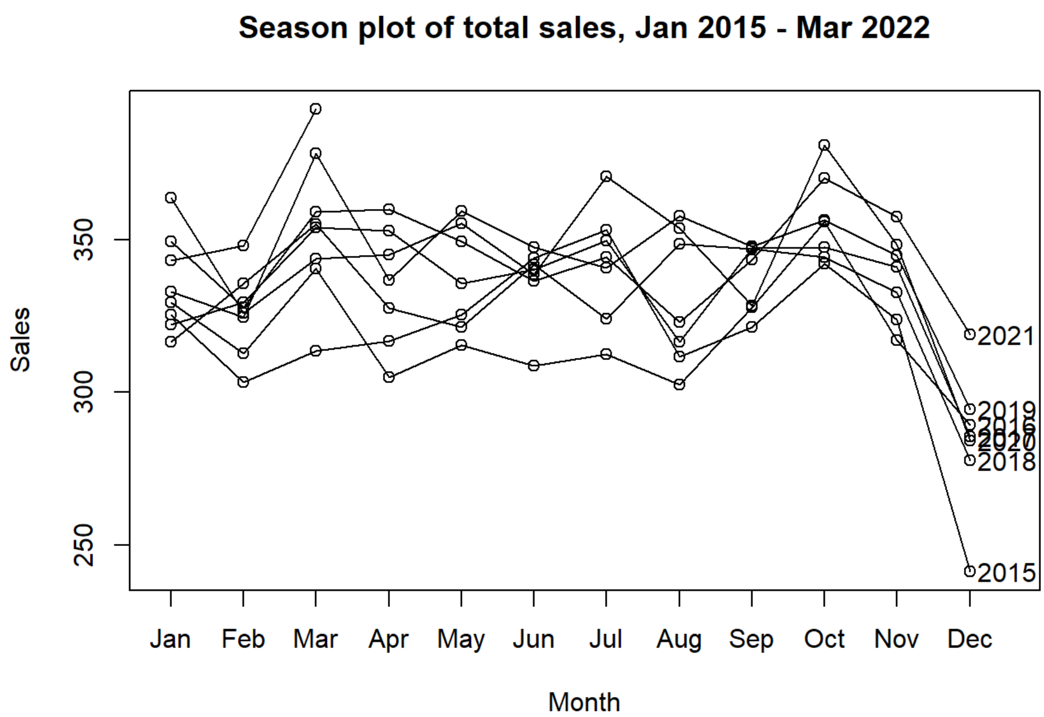
So far we've only visualised our time series as a whole. However, time series have some time-related components with their own corresponding plots, which can potentially be very insightful.

4.1 The season plot

The season plot is a time plot, except that each year is plotted with a separate line. So, we can compare different years in terms of magnitude as well as pattern.

The `seasonplot()` function from `{forecast}` provides a base R view:

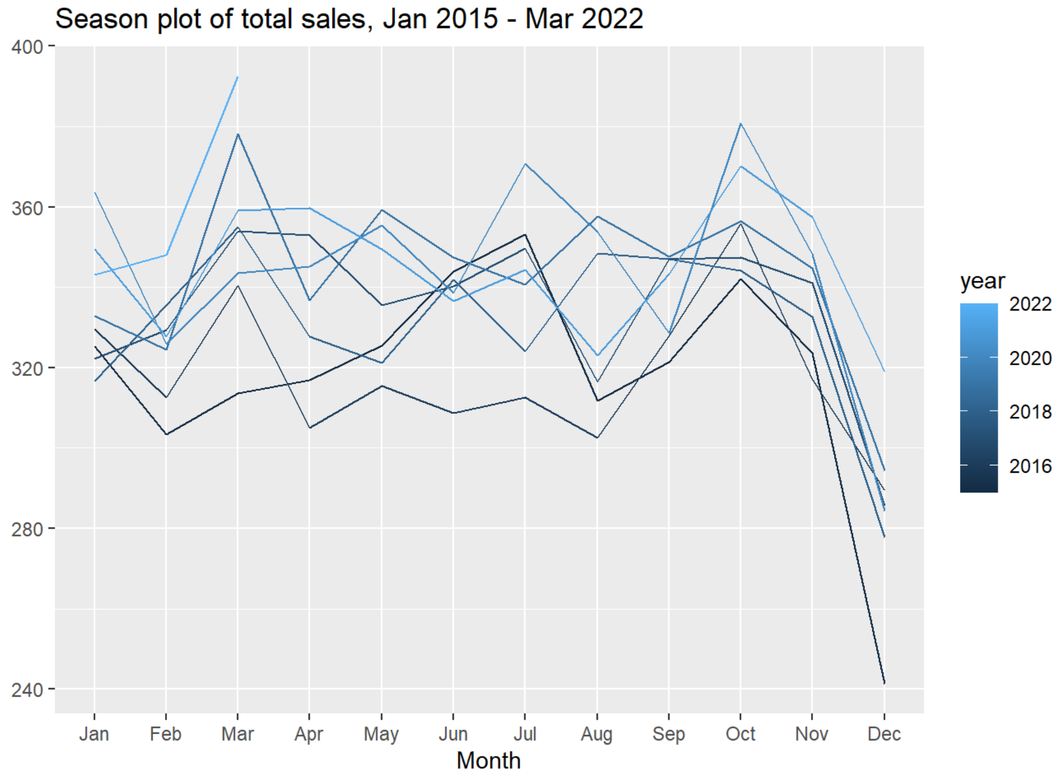
```
title_sp <- "Season plot of total sales, Jan 2015 - Mar 2022"
forecast::seasonplot(Xt, year.labels = TRUE, ylab = "Sales",
  main = title_sp)
```



The `ggseasonplot()` function from `{forecast}` provides a "gg"-version:

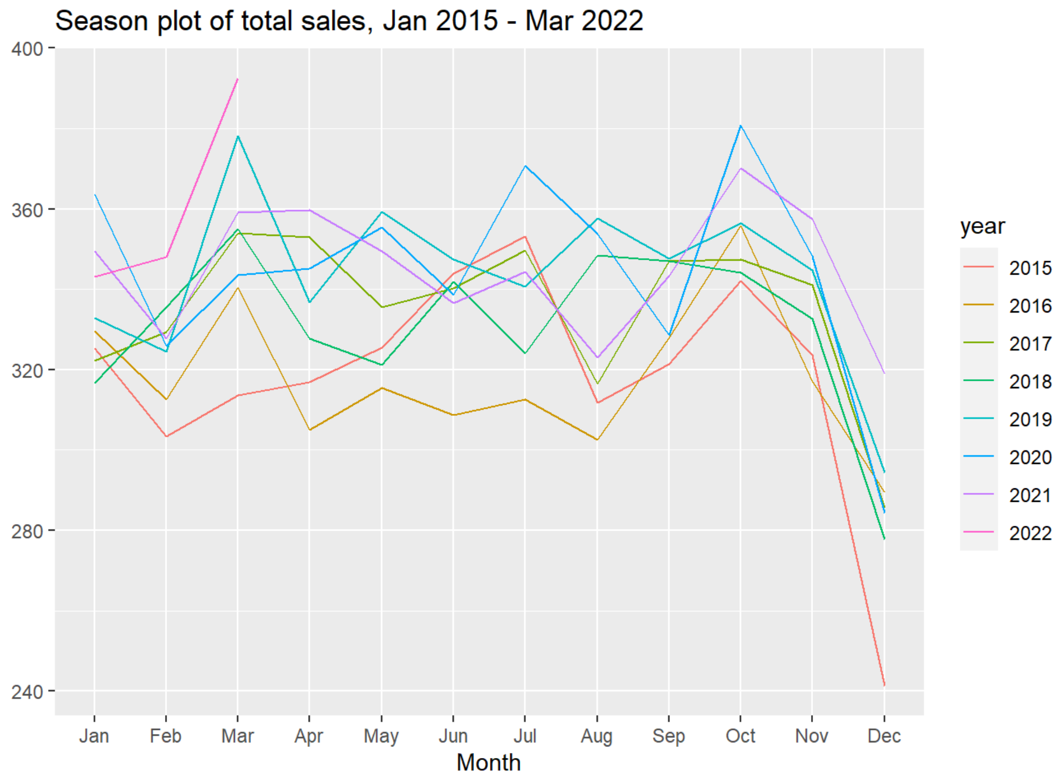
```
forecast::ggseasonplot(Xt, continuous = TRUE, main = title_sp)
```

4.1 The season plot



Setting `continuous = FALSE` will treat the years as a factor variable, instead of continuous:

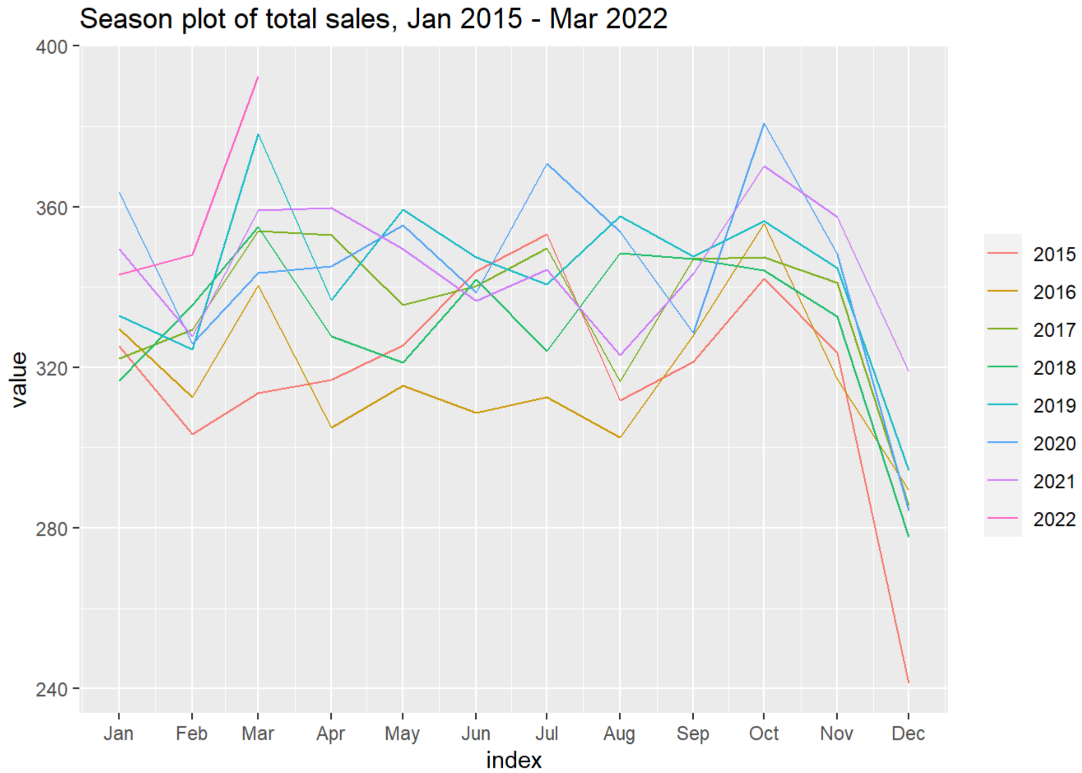
```
forecast::ggseasonplot(Xt, continuous = FALSE, main = title_sp)
```



The `gg_season()` from `{feasts}` implements the same plot as above, but the time series input must be of the `tsibble` class:

```
Xt_tsb <- tsibble::as_tsibble(Xt)
```

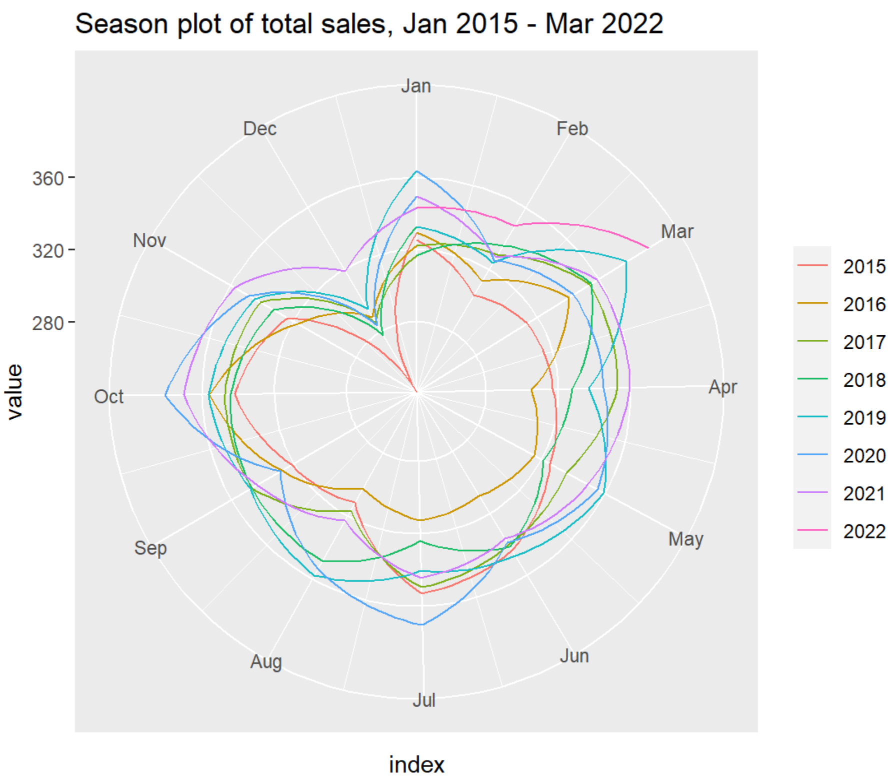
```
feasts::gg_season(Xt_tsb, y = value) +
  ggtitle(title_sp)
```



The `tsibble` class (time series tibble) provides a data infrastructure for “tidy” temporal data; for more information see: https://tsibble.tidyverts.org/.

Both the `ggseasonplot()` and the `gg_season()` function include a `polar` argument, which is set to `FALSE` by default. Setting it to `TRUE` creates the following:

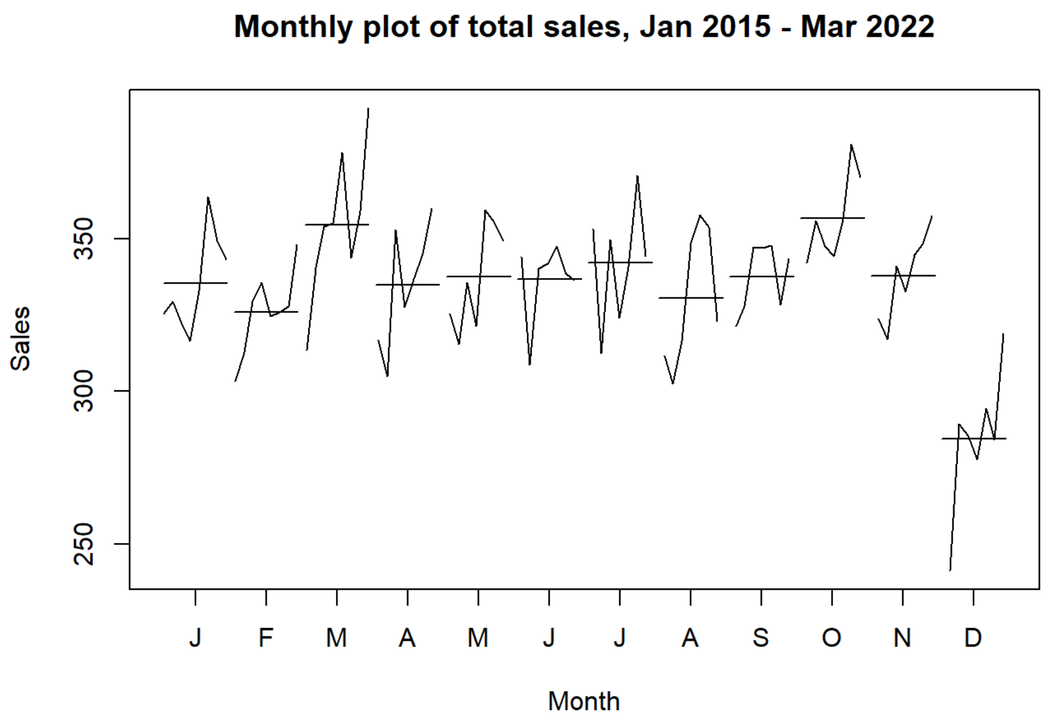
```
feasts::gg_season(Xt_tsb, y = value, polar = TRUE) +
  ggtitle(title_sp)
```



4.2 The month plot

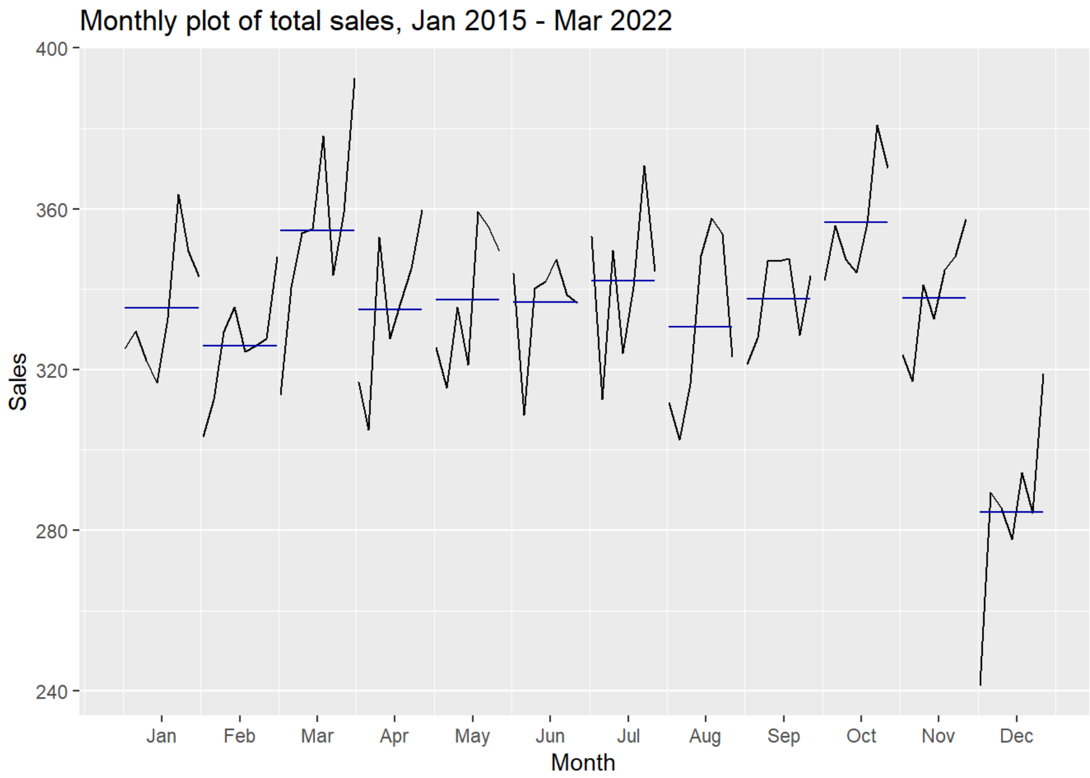
The month plot shows how values observed in the same month of the year change over time:

```
title_mp <- "Monthly plot of total sales, Jan 2015 - Mar 2022"
stats::monthplot(Xt, xlab = "Month", ylab = "Sales", main = title_mp)
```



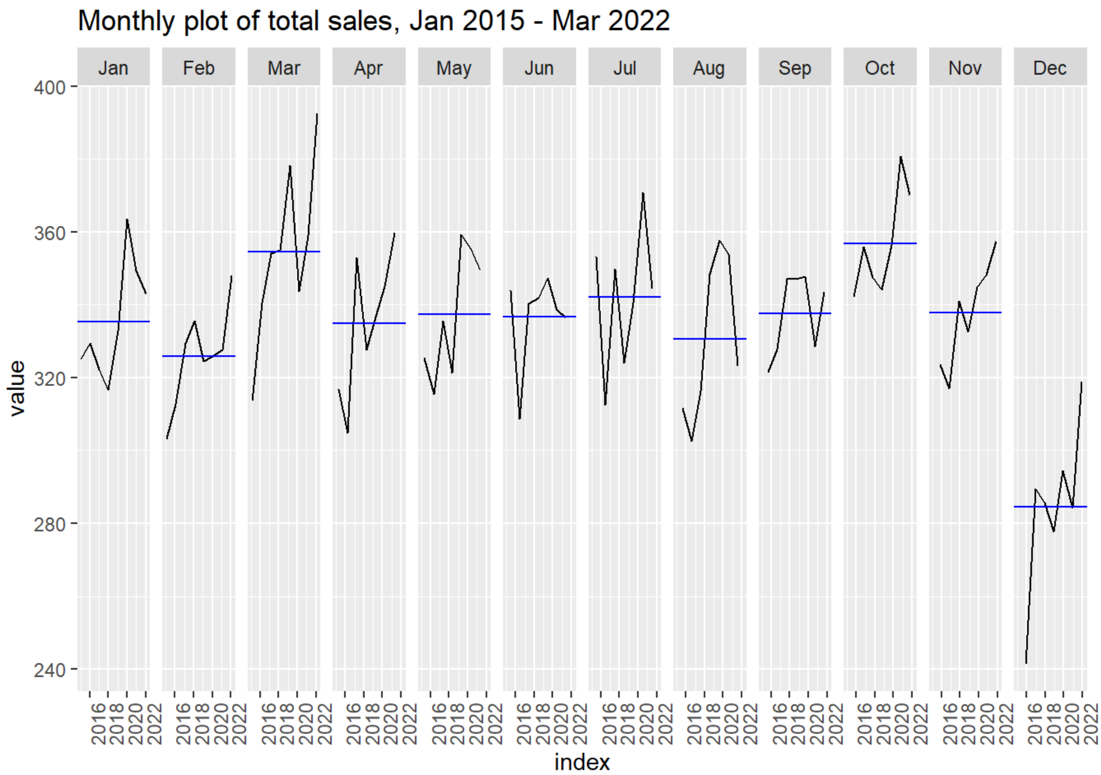
The `ggsubseriesplot()` function from `{forecast}` provides a “gg”-version:

```
forecast::ggsubseriesplot(Xt, ylab = "Sales", main = title_mp)
```



The `gg_subseries()` function from `{feasts}` will add faceting by month with a column grid:

```
feasts::gg_subseries(Xt_tsb, y = value) +
  ggtitle(title_mp)
```



The season plot and the month plot provide the same information arranged differently. Sometimes one of the two will reveal insights to the data that the other will fail to capture.

4.3 The lag plot

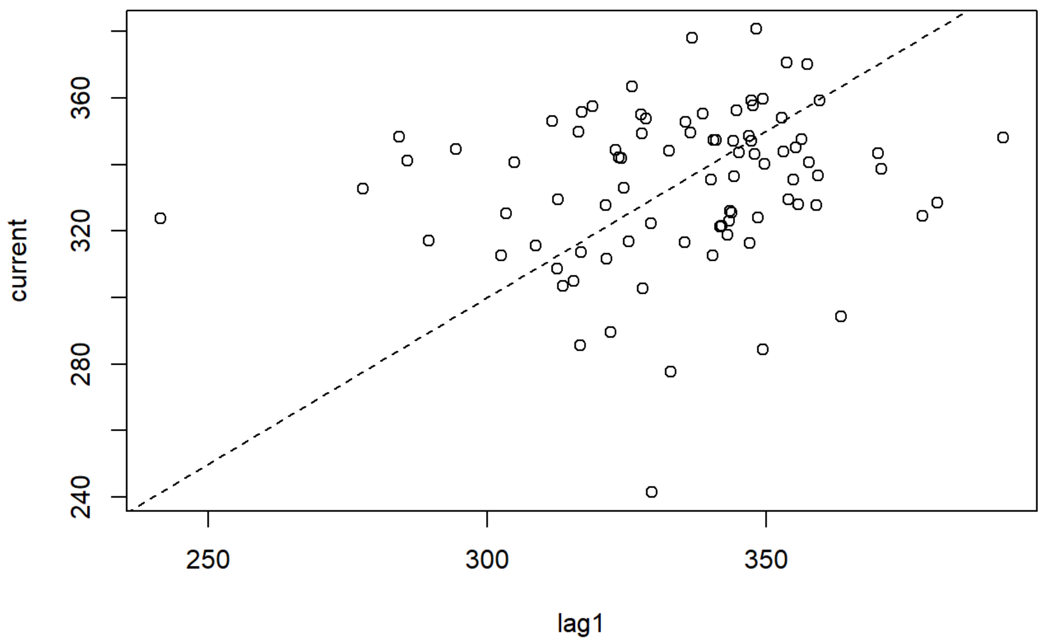
Suppose we suspect that the values we observed in one month are correlated with the values measured in the previous month(s). To investigate this hypothesis, we have to create pairs of observations: the values now vs. the past values. Note that the order is not important, as we are only looking for correlations.

One pair will be missing, since the first observation we have has no previous value to compare with. Let's see a simple example with 5 observations:

```
v <- c(2.3, 7.3, 19.8, 51.0, 129.2)
data.frame(lag1 = v[-1], current = v[-5])
#>    lag1 current
#> 1   7.3     2.3
#> 2  19.8     7.3
#> 3  51.0    19.8
#> 4 129.2    51.0
```

If we apply this idea to our 87 sales data, we can visualise the association with a scatterplot:

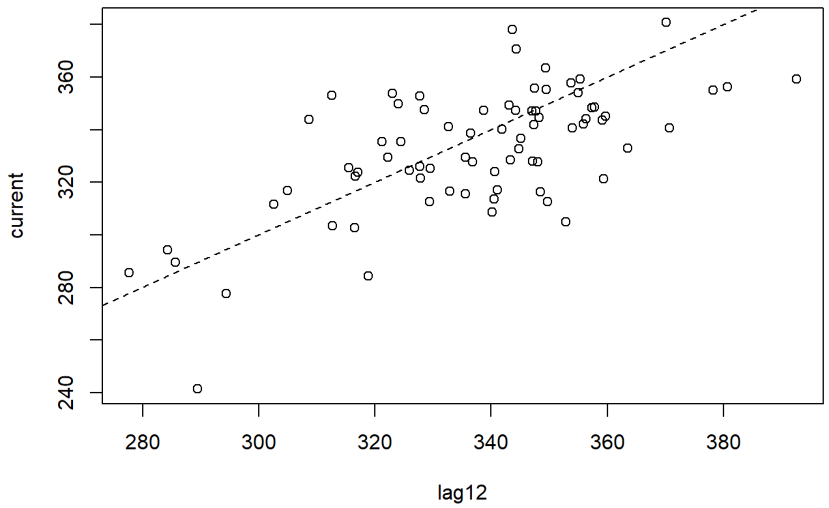
```
x <- c(Xt)
t1 <- x[-1]  # lag1
t2 <- x[-87]  # current
plot(t1, t2, xlab = "lag1", ylab = "current")
abline(a = 0, b = 1, lty = 2)
```



The plot above (lag plot with k = 1) shows a very weak correlation, if any. However, this is just the beginning: we can search for and discover correlations between months with a larger lag, so we need an infrastructure to investigate that.

Plot the association of the current values with the values observed 12 months ago, by creating the lag plot with k = 12. What is this telling you about your time series?

```
plot(x[-(1:12)], x[-(76:87)],
     xlab = "lag12", ylab = "current")
abline(a = 0, b = 1, lty = 2)
```

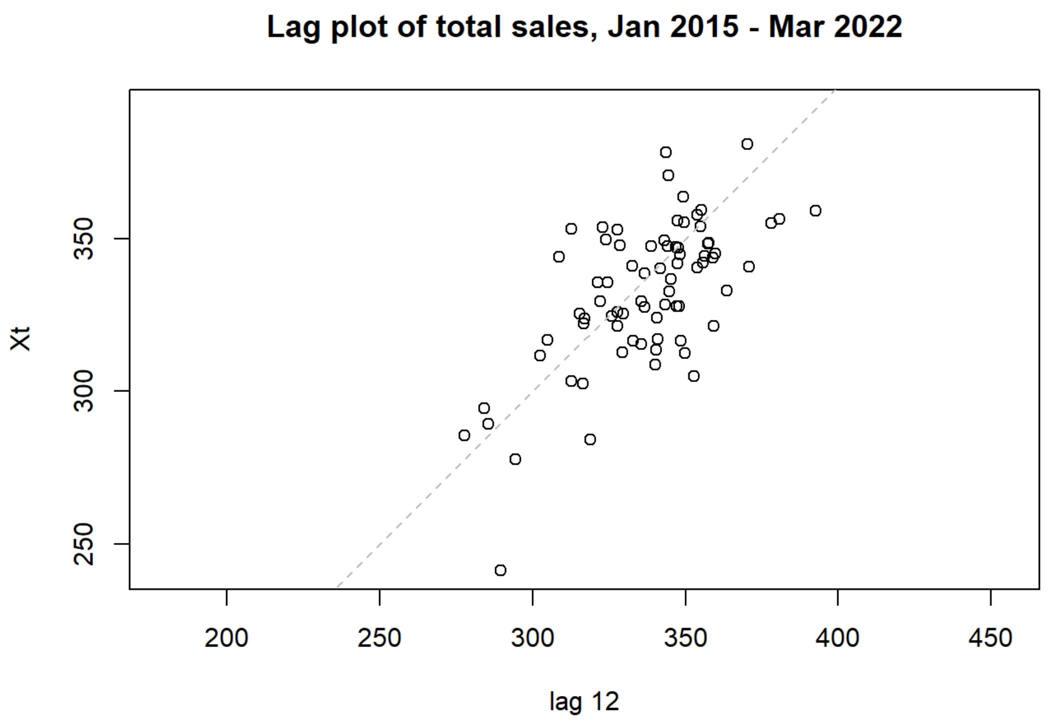


The `lag()` function helps us shift a time series in time, so we don't have to preform any subscripting as we did above.

```
stats::lag(Xt, k = 1)
#>          Jan     Feb     Mar     Apr     May     Jun     Jul     Aug
#> 2014
#> 2015 303.356 313.543 316.920 325.473 343.911 353.144 311.672 321.496
#> 2016 312.653 340.537 304.959 315.518 308.690 312.576 302.588 327.892
#> 2017 329.461 354.007 352.897 335.549 340.224 349.714 316.454 347.109
#> 2018 335.528 355.004 327.673 321.245 341.855 324.025 348.530 347.018
#> 2019 324.483 378.183 336.779 359.333 347.369 340.624 357.762 347.709
#> 2020 325.939 343.628 345.138 355.321 338.684 370.728 353.773 328.483
#> 2021 327.751 359.129 359.728 349.477 336.529 344.370 323.038 343.363
#> 2022 348.043 392.581
#>          Sep     Oct     Nov     Dec
#> 2014                         325.273
#> 2015 342.028 323.703 241.385 329.574
#> 2016 355.899 317.012 289.476 322.235
#> 2017 347.443 341.043 285.629 316.660
#> 2018 344.228 332.647 277.659 332.927
#> 2019 356.384 344.739 294.365 363.523
#> 2020 380.729 348.229 284.286 349.436
#> 2021 370.170 357.394 318.896 343.113
#> 2022
```

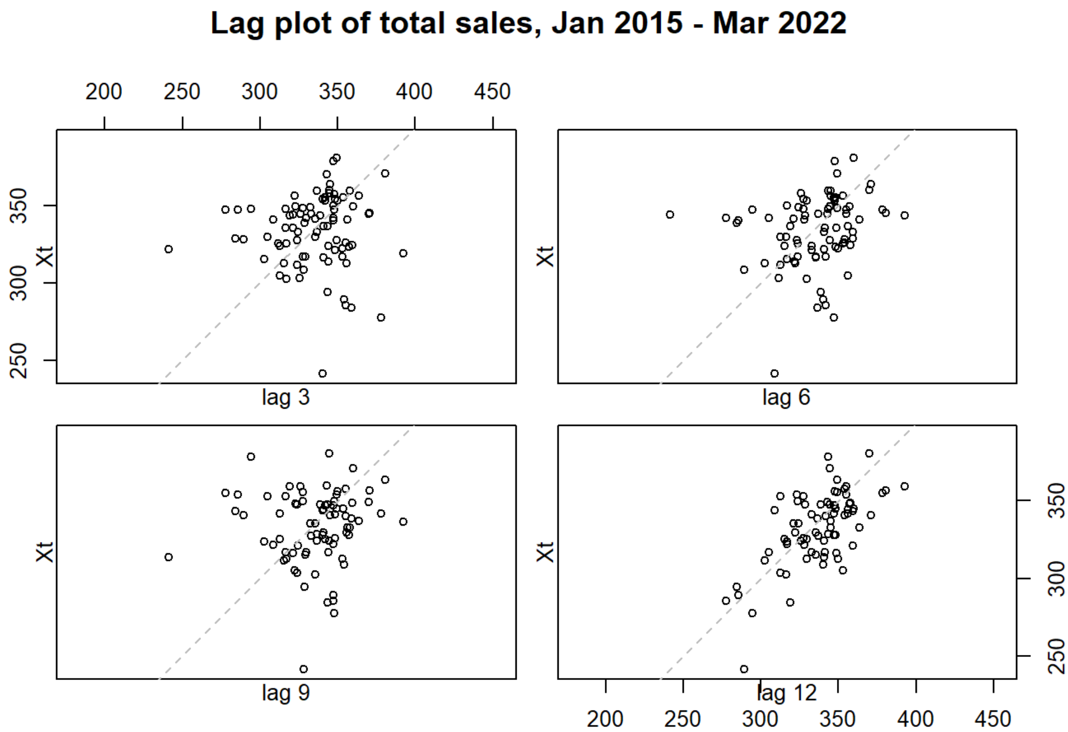
We can also create a lag plot using the `stats::lag.plot()` function:

```
title_lp <- "Lag plot of total sales, Jan 2015 - Mar 2022"
stats::lag.plot(Xt, set.lags = 12, do.lines = FALSE,
                labels = FALSE, main = title_lp)
```



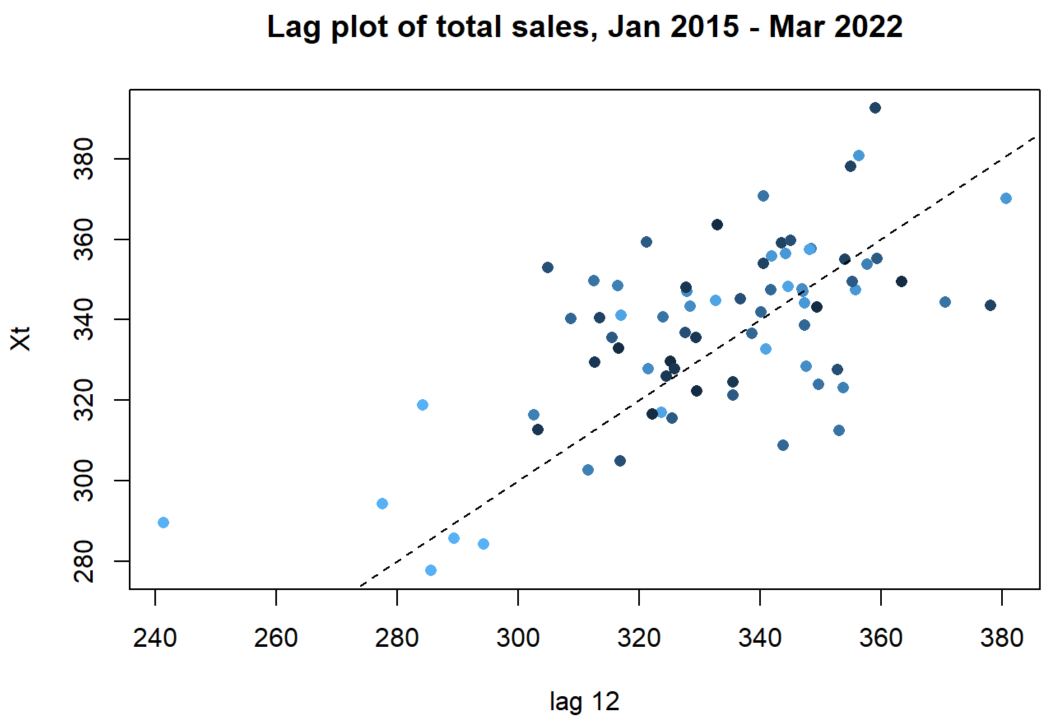
We can plot multiple lags on the same plot:

```
lag.plot(Xt, set.lags = c(3, 6, 9, 12), do.lines = FALSE,
         labels = FALSE, main = title_lp)
```



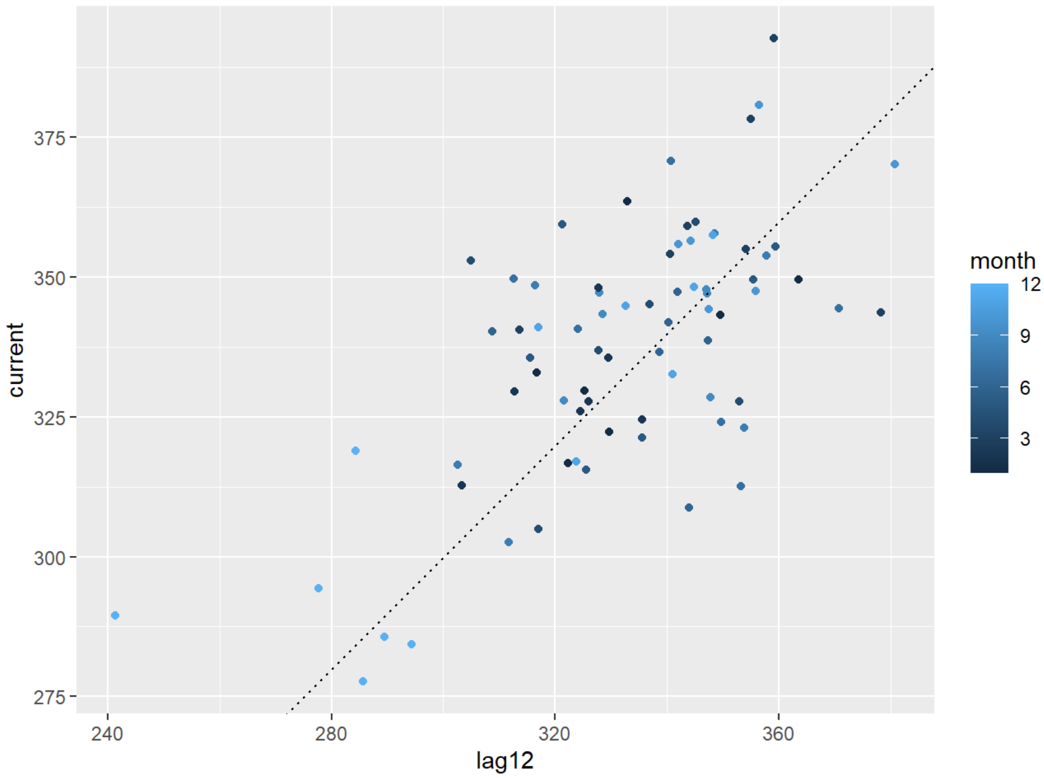
When we detect a noticeable correlation, it may be useful to further check if some specific months exhibit stronger correlation than others. Using the `colorRampPalette()` function from the `{grDevices}` package, we can create a colour gradient for representing months as integers, and plot the following using base R:

```
colour_palette <- colorRampPalette(colors = c("#132B43", "#56B1F7"))
plot(x[1:75], x[-(1:12)], xlab = "lag 12", ylab = "Xt",
     pch = 16, col = colour_palette(12), main = title_lp)
abline(a = 0, b = 1, lty = 2)
```



Using `{ggplot2}`:

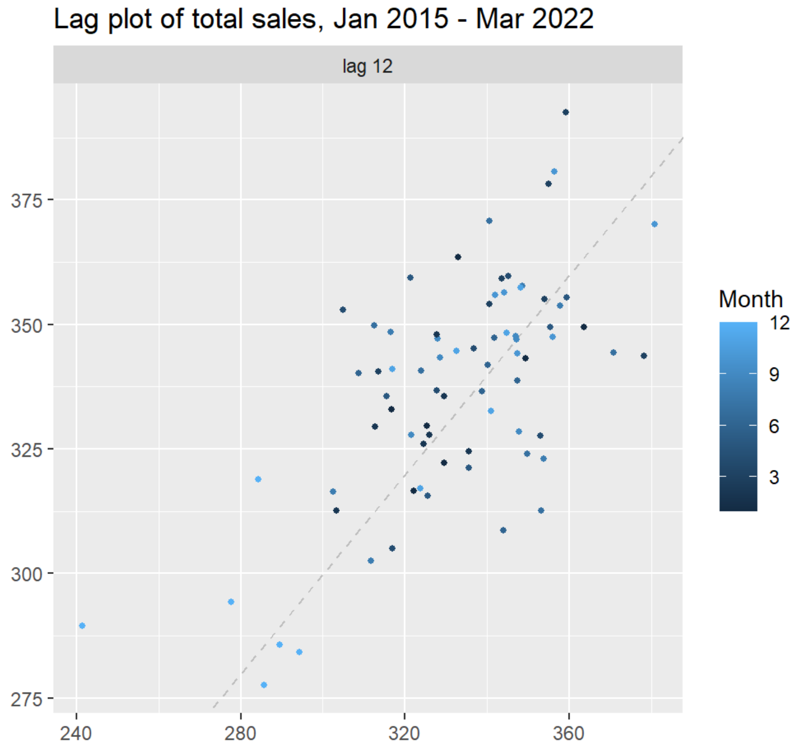
```
data.frame(lag12 = x, month = substr(z$value, 12, 13)) %>%
  mutate(month = as.numeric(month)) %>%
  slice(1:75) %>%
  mutate(current = x[-(1:12)]) %>%
  ggplot(aes(y = current, x = lag12, colour = month)) +
  coord_fixed() +
  geom_point() +
  geom_abline(slope = 1, linetype = 3)
```



The `{forecast}` package contains a convenient function `gglagplot()` that creates such plots without much hassle:

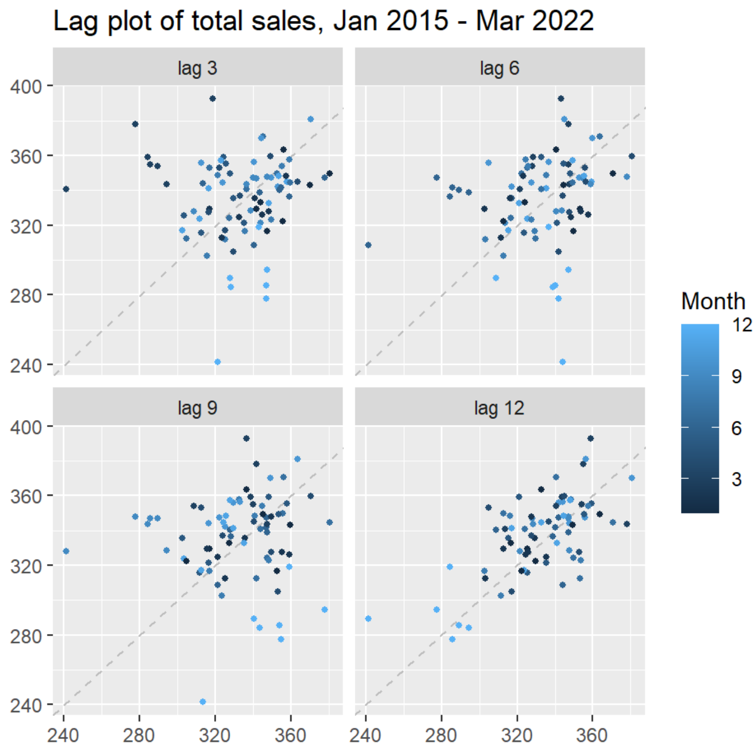
```
forecast::gglagplot(Xt, set.lags = 12, do.lines = FALSE,
                    continuous = TRUE, main = title_lp)
```

4.3 The lag plot



For multiple lag plots we modify the `set.lags` argument accordingly:

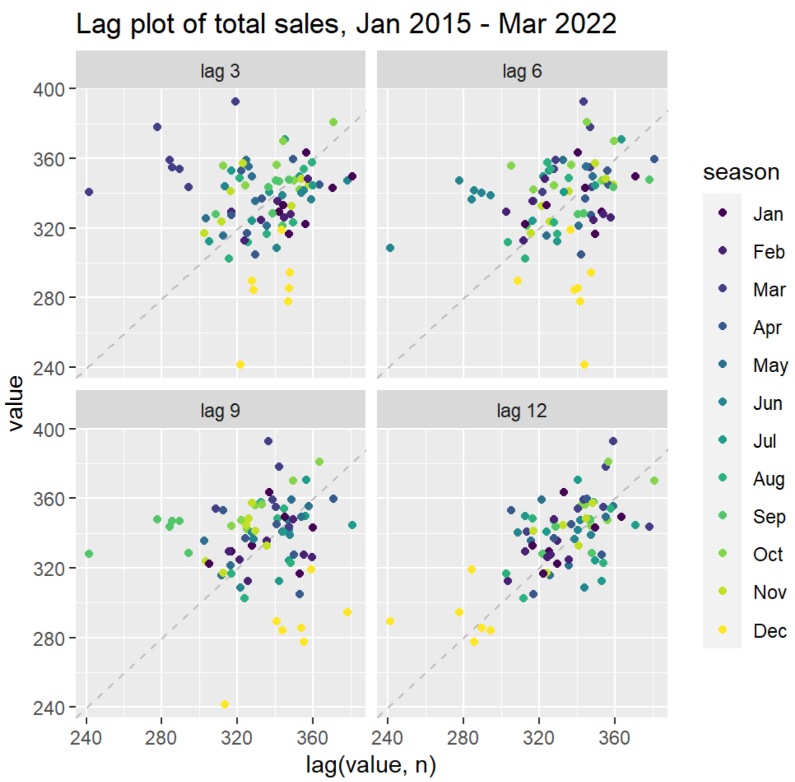
```
forecast::gglagplot(Xt, set.lags = c(3, 6, 9, 12), do.lines = FALSE,
                    continuous = TRUE, main = title_lp)
```



Notice how the positions of x and y have changed compared to the output of `stats::lag.plot()`. This only has to do with the different interpretation of the sign of the lag values `k`, which is not consistent across all R packages.

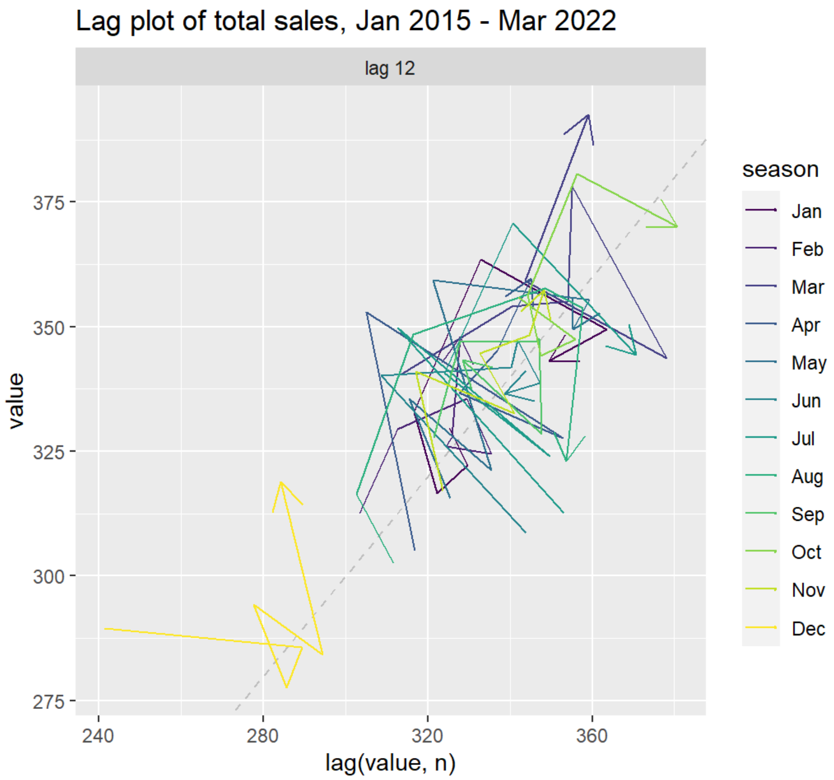
The `{feasts}` package contains the `gg_lag()` function, which replaces `gglagplot()` and uses the **viridis** colour palette instead of the the `{ggplot2}`'s default blue gradient.

```
feasts::gg_lag(Xt_tsb, y = value, lags = c(3, 6, 9, 12),
               geom = "point") +
  ggtitle(title_lp)
```



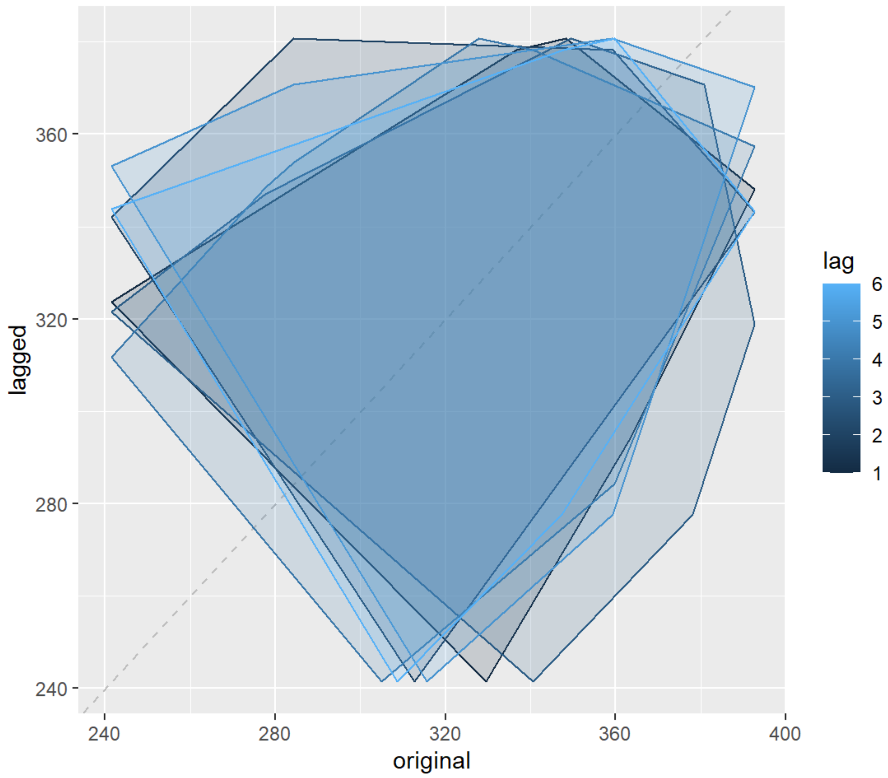
Adding `arrow = TRUE` can also show the arrow of time, which could also reveal a possible pattern:

```
feasts::gg_lag(Xt_tsb, y = value, lags = 12,
               geom = "path", arrow = TRUE) +
  ggtitle(title_lp)
```



The `gglagchull()` function from `{forecast}` will plot convex hulls of the lags, layered on a single plot. This helps visualise the change in "auto-dependence" as lags increase.

```
forecast::gglagchull(Xt, lags = 6)
```



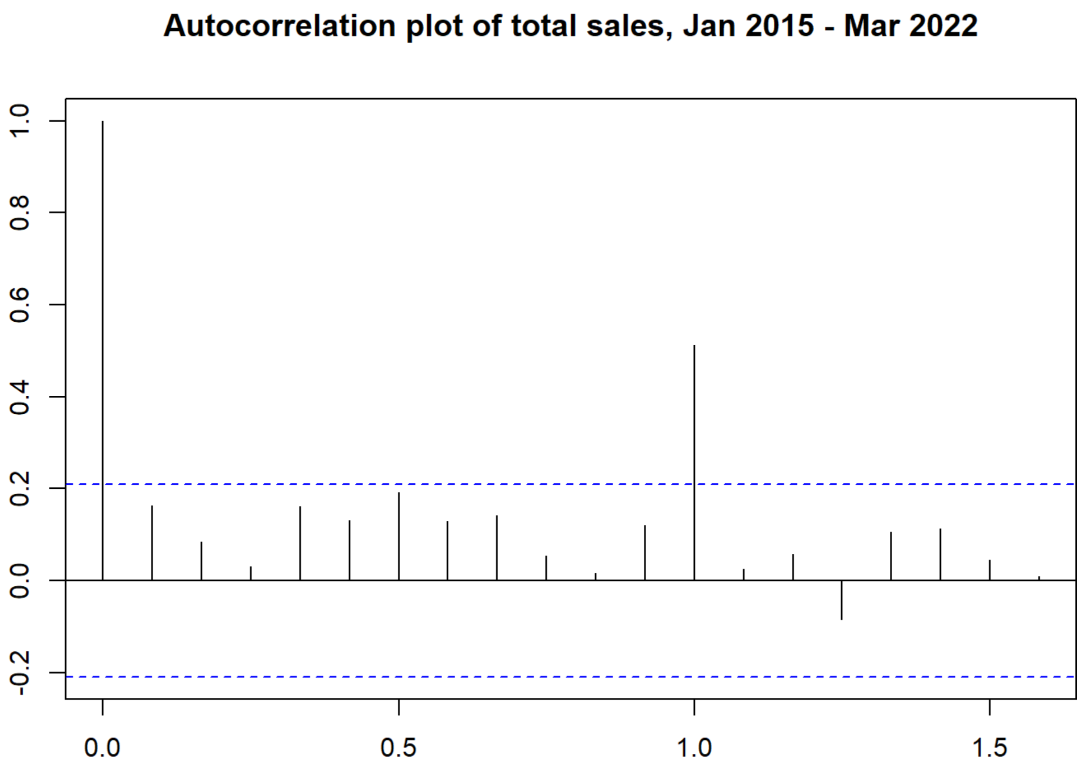
Finally, the `ts_lags()` function from `{TSstudio}` will create a **plotly** version of the lag plot.

4.4 The autocorrelation plots

The autocorrelation plot provides the infrastructure we need to investigate the correlation between current and previous values.

The `acf()` function will compute and plot the autocorrelation of a time series:

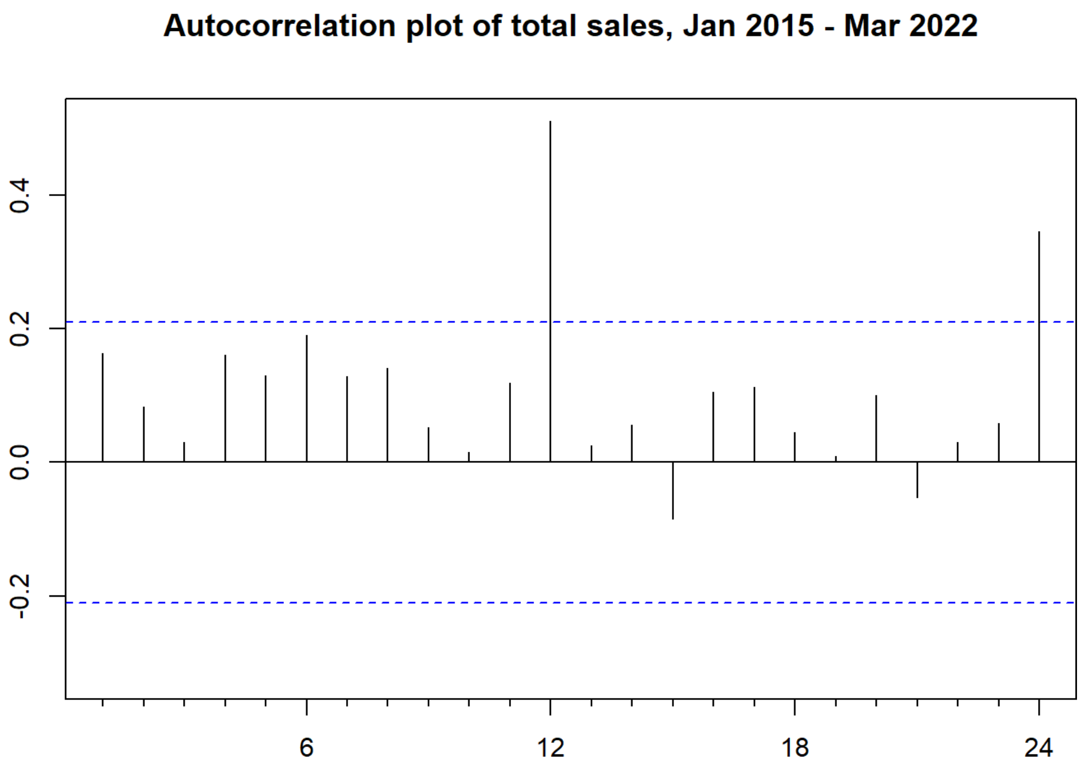
```
title_acf <- "Autocorrelation plot of total sales, Jan 2015 - Mar 2022"
par(mar = c(3, 3, 3, 0))
stats::acf(Xt, type = "correlation",
  main = title_acf)
```



The dotted blue lines show which correlations are (potentially) statistically significant.

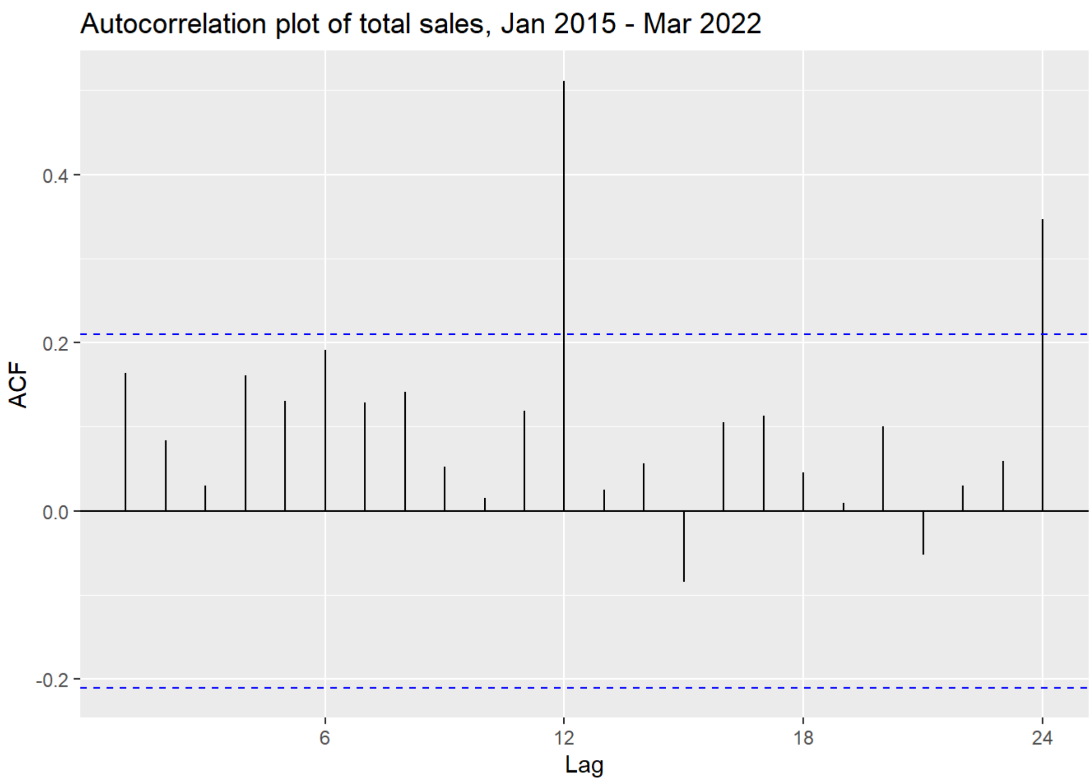
The correlation of the time series with itself is 1; this is not particularly helpful to visualise, so you may find that autocorrelation plots created by other R packages omit that. For example, the `Acf()` function from `{forecast}`:

```
par(mar = c(3, 3, 3, 0))
forecast::Acf(Xt, main = title_acf)
```



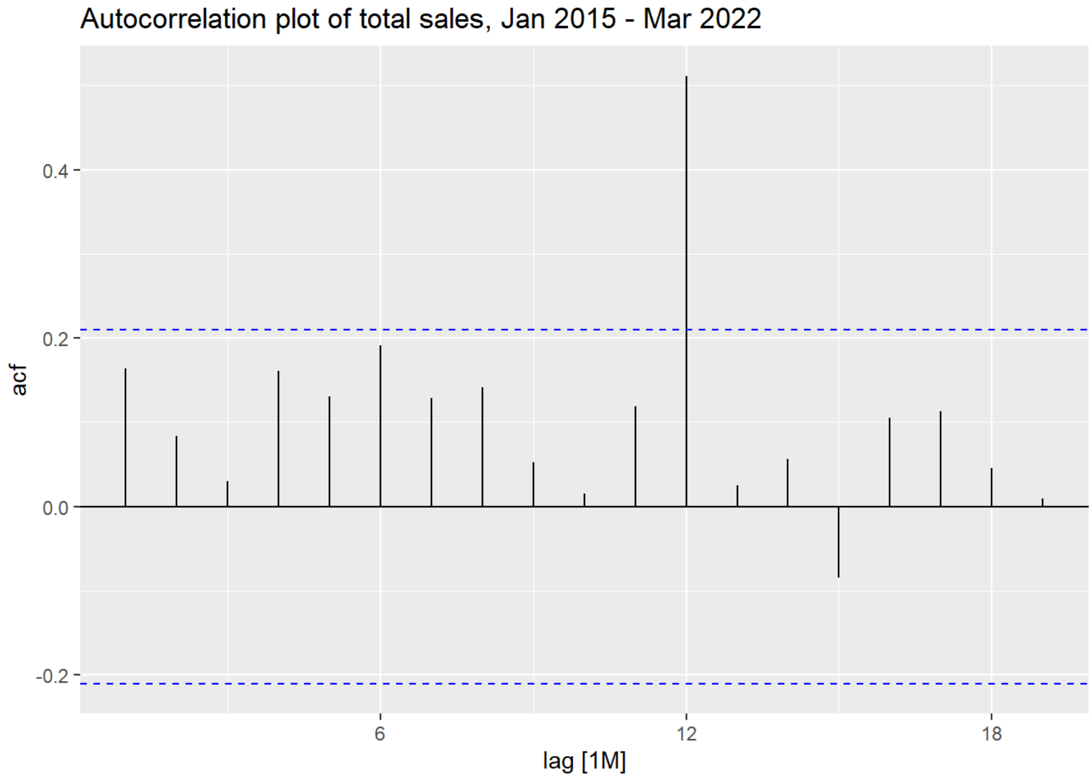
The `ggAcf()` function from `{forecast}` provides a "gg"-version:

```
forecast::ggAcf(Xt) +
  ggtitle(title_acf)
```



Once again, the (re)implementation in the `{feasts}` package requires the time series to be of a `tsibble` class, and splits the calculation and the plot into separate functions (the time window may also differ):

```
Xt_tsb %>%
  feasts::ACF(y = value) %>%
  feasts::autoplot() +
  ggtitle(title_acf)
```



Consider again a time series with random data simulated from the standard normal distribution.

```
ts_rand <- rnorm(100) %>%
  ts(start = c(2008, 6), frequency = 12)
```

Create the diagnostic plots we have seen in this chapter (add a last step `as_tsibble()` above, if needed). Can you notice any useful patterns?

Chapter 5

Seasonal decomposition

5.1 Introduction to time series components

One of the basic characteristics of a time series with huge forecast potential, is “seasonality”: if there is a pattern in the data repeated every year, then we have good reason to assume that the same pattern will be also present next year.

Let’s calculate the mean value of total sales per month:

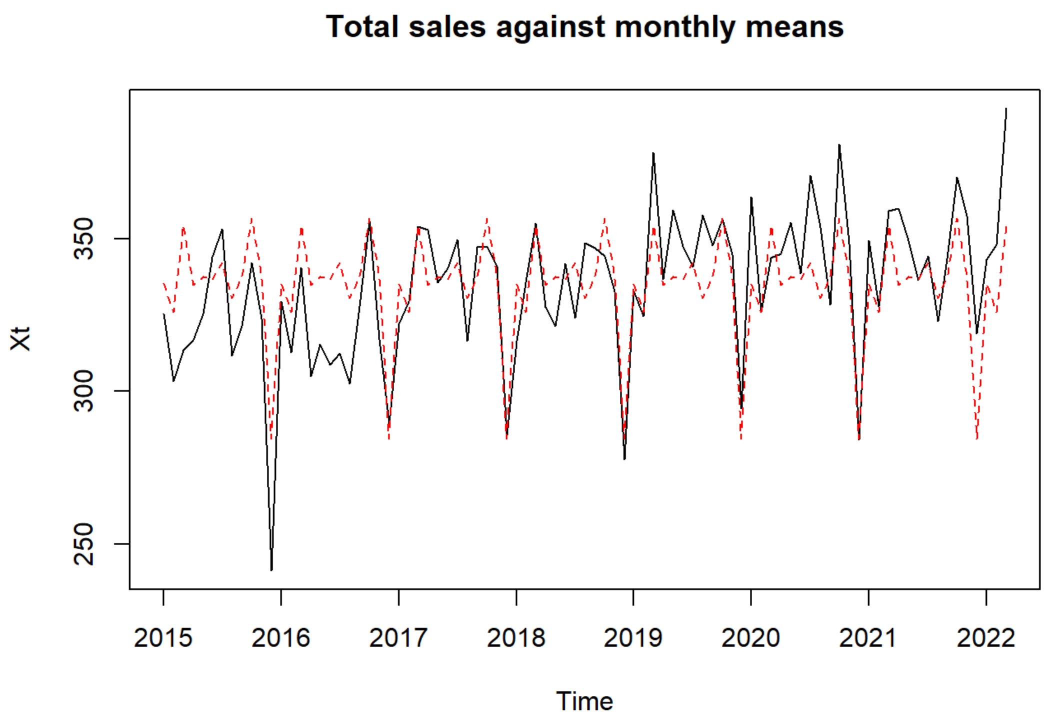
```
monthly_mean <- sapply(1:12, function(.x) mean(Xt[seq(.x, 87, 12)]))
data.frame(month = month.name, average = monthly_mean)
#>        month  average
#> 1    January 335.3426
#> 2   February 325.9017
#> 3      March 354.5765
#> 4      April 334.8706
#> 5        May 337.4166
#> 6       June 336.7517
#> 7       July 342.1687
#> 8     August 330.5453
#> 9  September 337.5814
#> 10   October 356.6973
#> 11  November 337.8239
#> 12  December 284.5280
```

Or, using tidyverse:

```
purrr::map(1:12, .f = ~ mean(Xt[seq(.x, 87, 12)])) %>% unlist()
#>  [1] 335.3426 325.9017 354.5765 334.8706 337.4166 336.7517 342.1687
#>  [8] 330.5453 337.5814 356.6973 337.8239 284.5280
```

Let’s visualise these means against our original data:

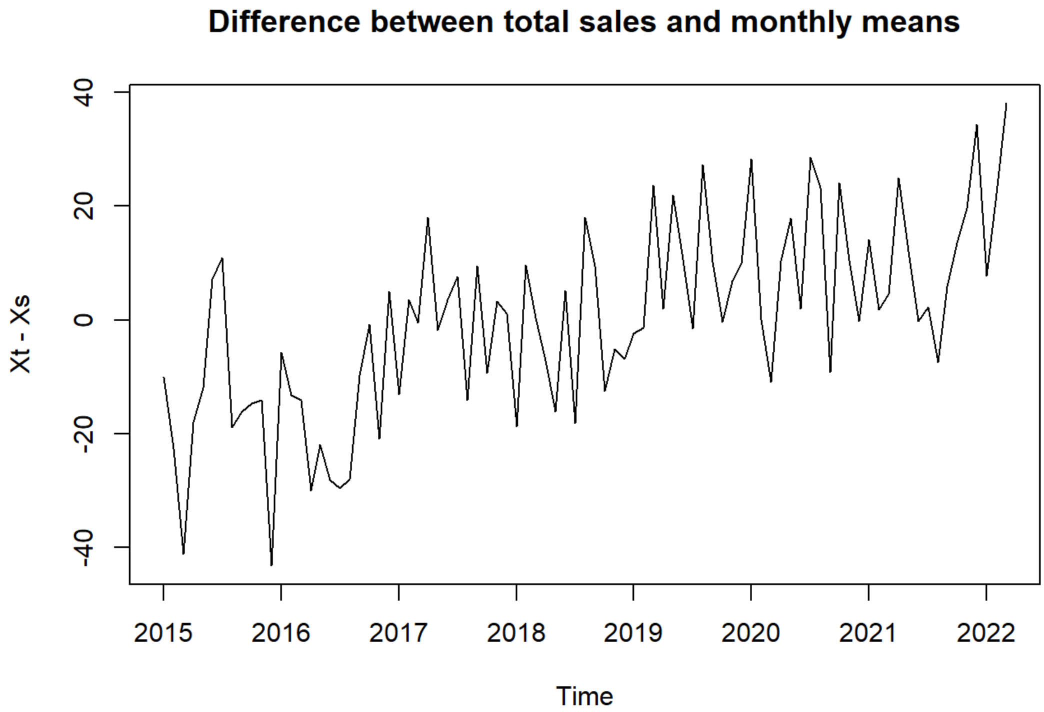
```
Xs <- ts(rep(monthly_mean, length.out = 87),
         start = c(2015, 1), frequency = 12)
plot(Xt, main = "Total sales against monthly means")
lines(Xs, lty = 2, col = "red")
```



Although the monthly means don't fully capture the actual values, we're not really far off. It seems that there is a strong seasonality in the data.

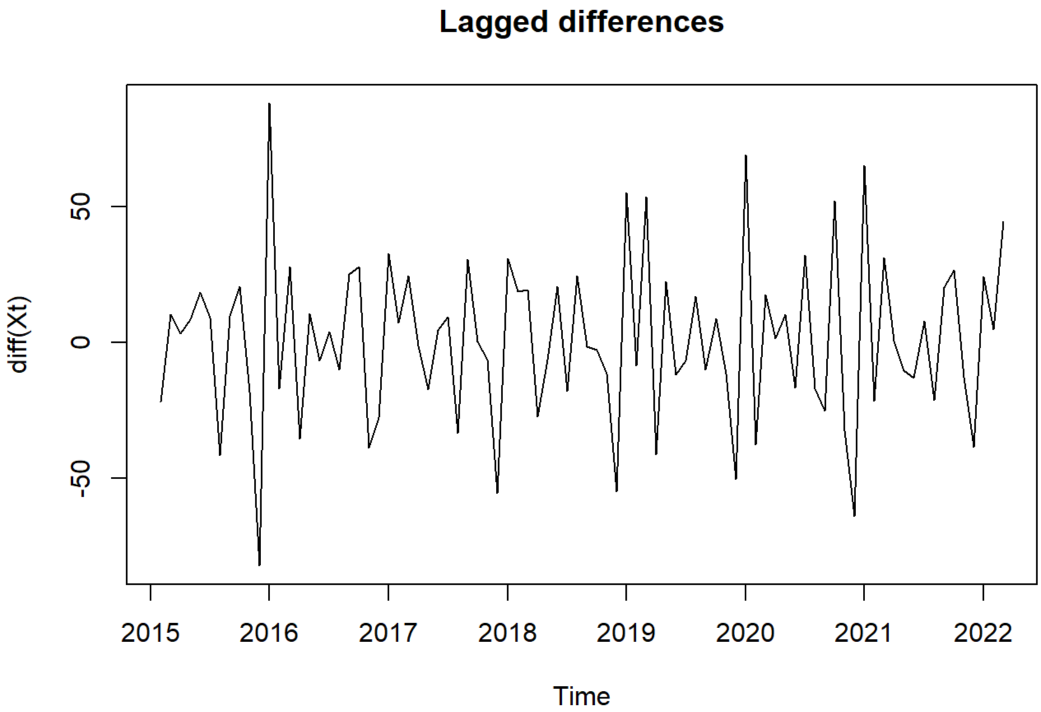
The difference between the two time series above could be thought of as a rough estimate of the trend, the temporal change over time:

```
plot(Xt - Xs,
     main = "Difference between total sales and monthly means")
```



In addition to the seasonality and the trend, there's also the "noise": a random fluctuation from month to month. We can get an idea of this fluctuations by plotting the (lagged) differences, i.e. the differences between the current and the previous values:

```
plot(diff(Xt), main = "Lagged differences")
```



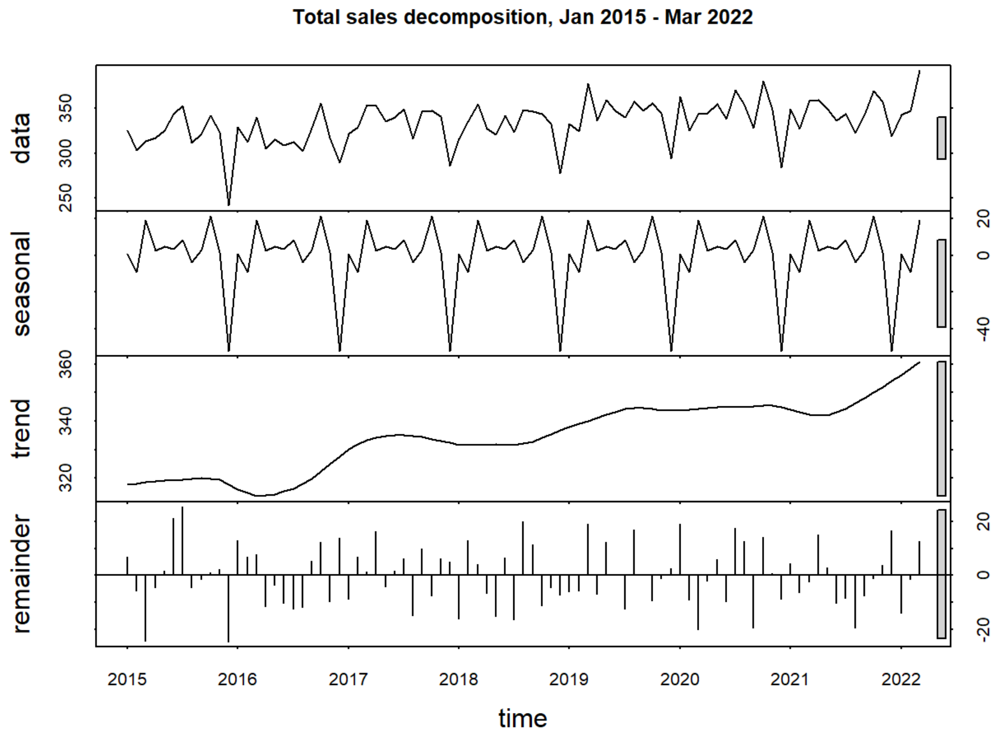
5.2 The STL decomposition

There are several formal methods to “decompose” a time series into a trend, a seasonal and a noise component. One of the most common is the “seasonal and trend decomposition using loess”, implemented in the `stl()` function:

```
stl_Xt <- stats::stl(Xt, s.window = "periodic")
stl_Xt$time.series[1:12, ]
#>          seasonal    trend   remainder
#>  [1,]   0.8236287 317.7217   6.7276314
#>  [2,]  -9.0889944 318.1862  -5.7411650
#>  [3,]  19.1140043 318.6506 -24.2215833
#>  [4,]   2.4849889 318.8986  -4.4635513
#>  [5,]   4.6473672 319.1465   1.6790869
#>  [6,]   3.4895860 319.2953  21.1261399
#>  [7,]   8.4136537 319.4440  25.2863440
#>  [8,]  -3.7604652 319.7644  -4.3319091
#>  [9,]   2.7249662 320.0847  -1.3137126
#> [10,]  21.2572259 319.8174   0.9533267
#> [11,]   1.8002126 319.5501   2.3526391
#> [12,] -51.9061718 317.7360 -24.4447884
```

We can plot the seasonal decomposition object (of the `stl` class):

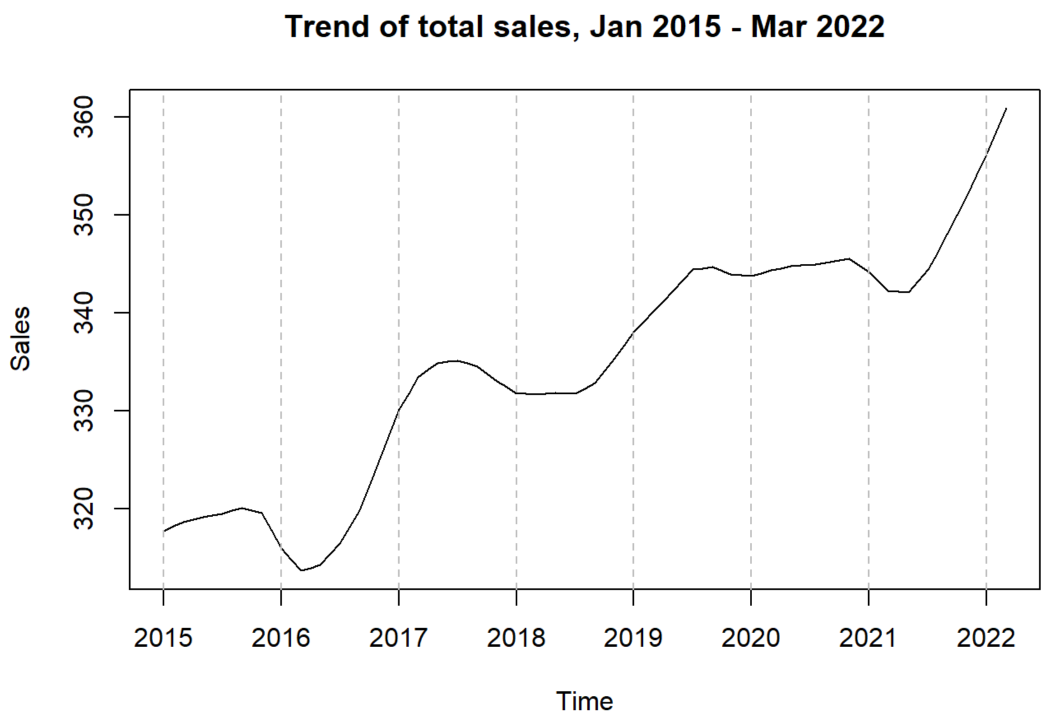
```
plot(stl_Xt,
  main = "Total sales decomposition, Jan 2015 - Mar 2022")
```



Let’s extract and plot the trend:

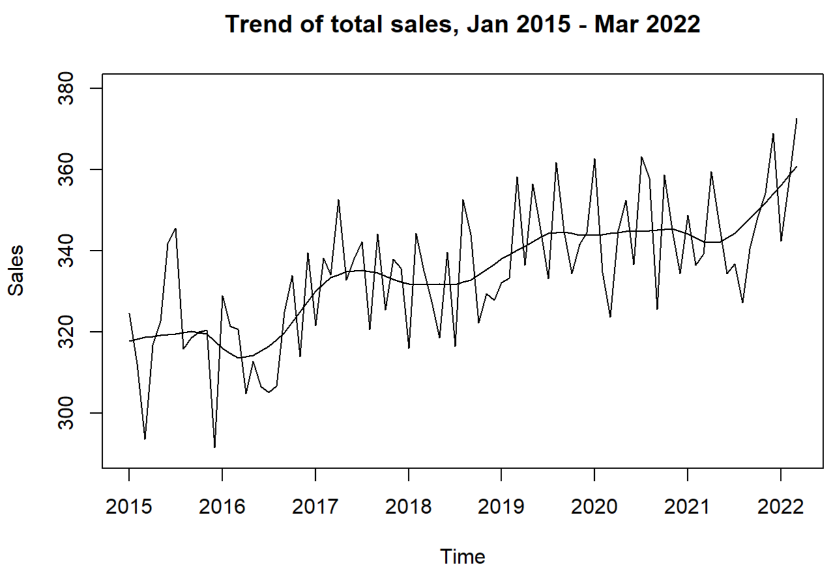
```
title_trend <- "Trend of total sales, Jan 2015 - Mar 2022"
plot(stl_Xt$time.series[, "trend"], ylab = "Sales", main = title_trend)
abline(v = 2015:2022, col = "grey", lty = 2)
```

5.2 The STL decomposition



Take the difference between total sales and monthly means, and lift it by the mean value of the entire time series. Plot this line over the trend plot we created above based on the `stl` decomposition. What do you observe?

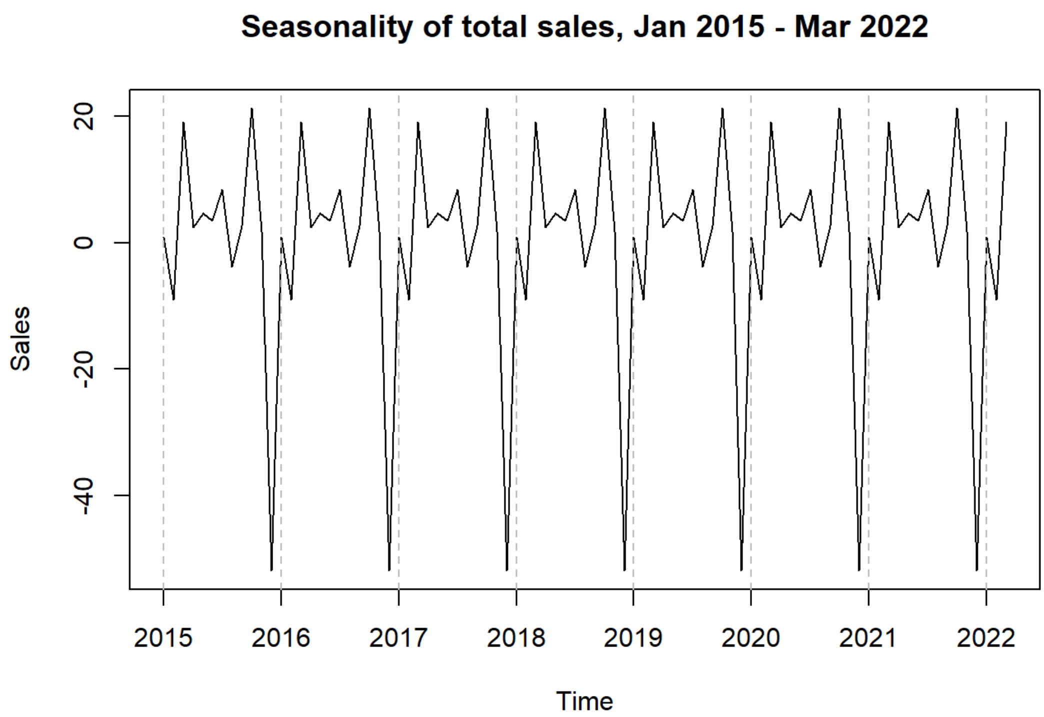
```
plot(stl_Xt$time.series[, "trend"], ylim = c(290, 380),
  ylab = "Sales", main = title_trend)
lines(mean(Xt) + Xt - Xs)
```



We can also extract and plot the seasonal component:

```
plot(stl_Xt$time.series[, "seasonal"], ylab = "Sales",
     main = "Seasonality of total sales, Jan 2015 - Mar 2022")
abline(v = 2015:2022, col = "grey", lty = 2)
```

5.2 The STL decomposition

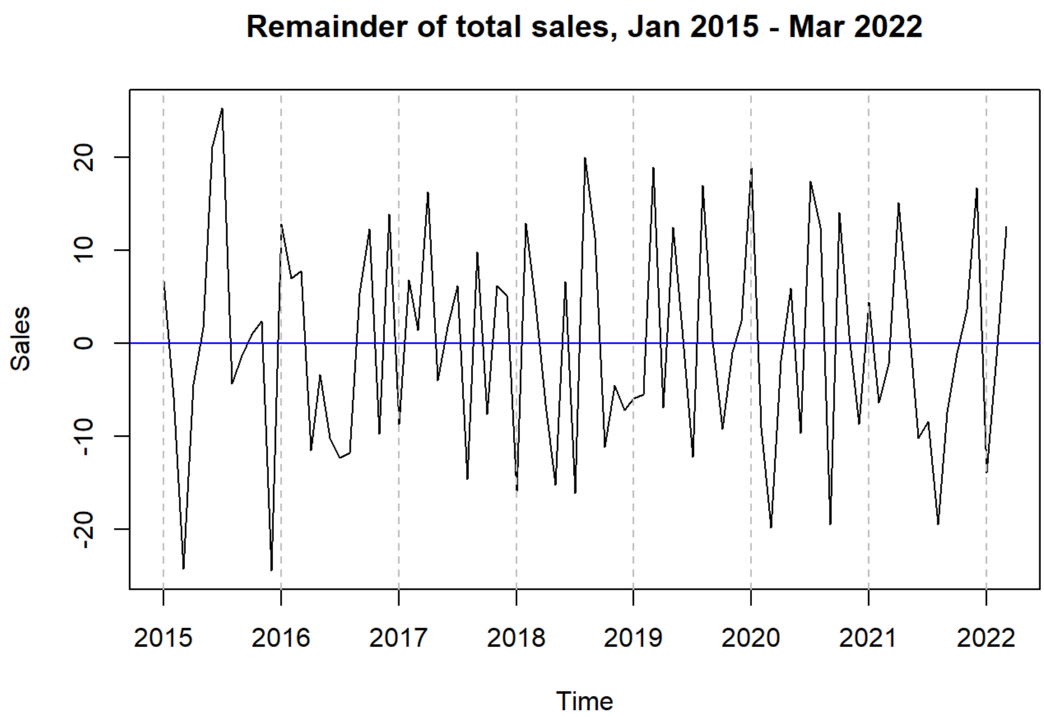


Take the monthly means we calculated earlier, and compare them with the seasonal component from the STL object (you will only need the first 12 of them). How close are the values?

```
data.frame(
  monthly_average = monthly_mean,
  lifted_seasonality = mean(Xt) +
    stl_Xt$time.series[1:12, "seasonal"])
#>    monthly_average lifted_seasonality
#> 1         335.3426           335.4817
#> 2         325.9017           325.5691
#> 3         354.5765           353.7721
#> 4         334.8706           337.1430
#> 5         337.4166           339.3054
#> 6         336.7517           338.1476
#> 7         342.1687           343.0717
#> 8         330.5453           330.8976
#> 9         337.5814           337.3830
#> 10        356.6973           355.9153
#> 11        337.8239           336.4583
#> 12        284.5280           282.7519
```

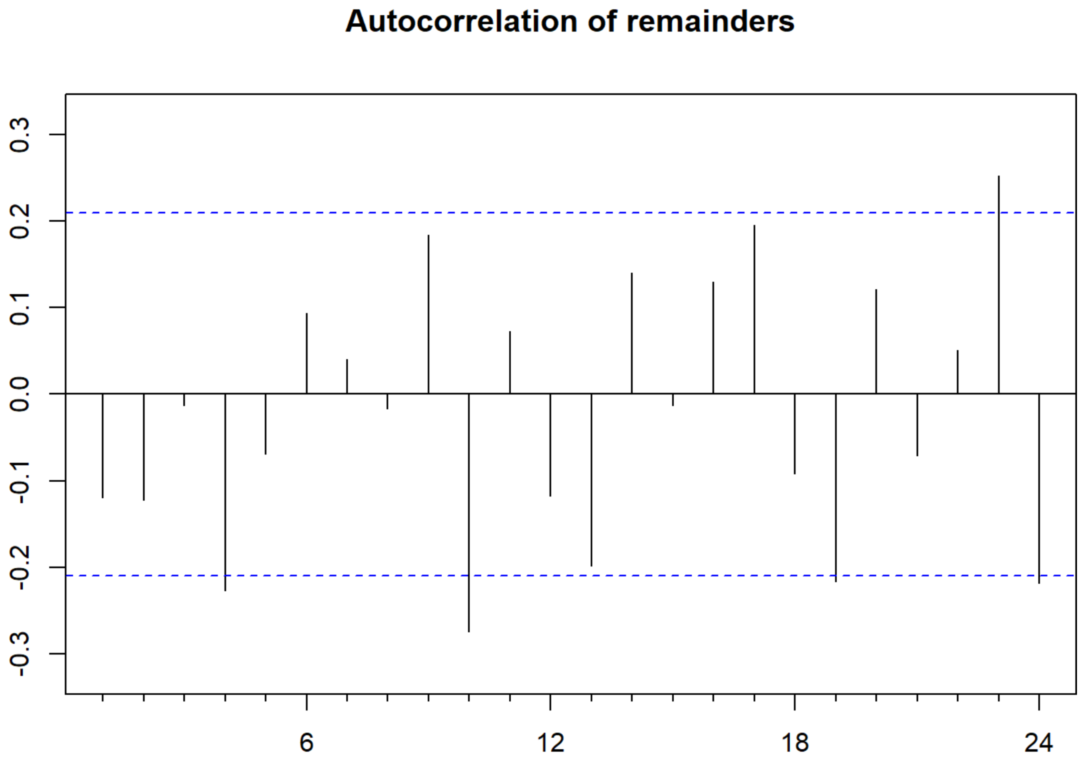
Finally, we can extract and plot the noise (the remainder component):

```
plot(stl_Xt$time.series[, "remainder"], ylab = "Sales",
     main = "Remainder of total sales, Jan 2015 - Mar 2022")
abline(v = 2015:2022, col = "grey", lty = 2)
abline(h = 0, col = "blue", lty = 1)
```



Remainders should not be autocorrelated. We can check that through a plot:

```
par(mar = c(3, 3, 3, 0))
forecast::Acf(stl_Xt$time.series[, "remainder"],
              main = "Autocorrelation of remainders")
```



Some spikes seem to go out of the boundaries that mark the statistical significance. The `Box.test()` function from the `{stats}` package implements the Ljung-Box test, which has as the null hypothesis that the residuals are independent.

```
Box.test(stl_Xt$time.series[, "remainder"],
         lag = 12, fitdf = 0, type = "Ljung")
#>
#>  Box-Ljung test
#>
#> data:  stl_Xt$time.series[, "remainder"]
#> X-squared = 21.878, df = 12, p-value = 0.03891
```

You can read more about STL decomposition here: https://otexts.com/fpp3/stl.html

Chapter 6

Introduction to modelling

Time series models can help us answer several different questions, such as how to:

- properly decompose our time series
- create smooth lines
- impute missing data
- forecast future values

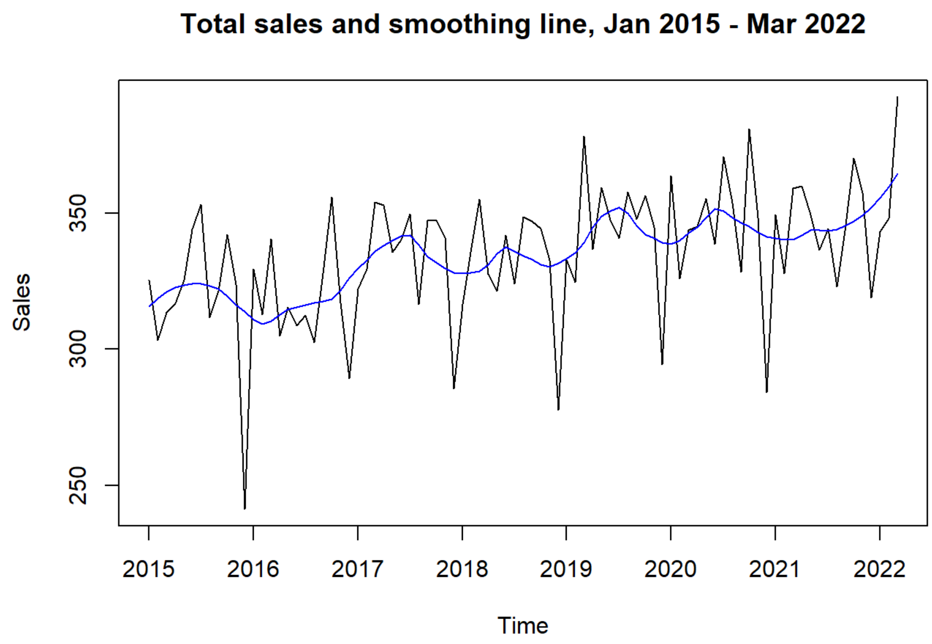
In the previous chapter we already discussed the time series decomposition problem. Here we will take a look at the other three.

6.1 Time series smoothing

Computing and plotting smooth lines for time series data is a huge topic, as there are many methods available.

We have already seen the loess (locally estimated scatterplot smoothing) method in the last chapter when we introduced the STL decomposition method. We can use this method to define a smooth line and eliminate the fluctuations of a time series, so we can reveal the trend:

```
X1 <- c(Xt)
X2 <- 1:length(X1)
X3 <- predict(loess(X1 ~ X2, span = 0.25))
plot(Xt, ylab = "Sales",
     main = "Total sales and smoothing line, Jan 2015 - Mar 2022")
lines(ts(X3, start = c(2015, 1), frequency = 12), col = "blue")
```



The parameter `span` controls the degree of smoothing.

A simpler idea is to calculate a “moving average”, namely a time series where the value at a point in time is the average of the actual observations around the point. Thus, the fluctuations get smoothed. The `rollmean()` function from `{zoo}` and the `ma()` function from `{forecast}` provide moving average estimates. The `{locfit}` package provides a very efficient alternative for local regression.

6.2 Missing data imputation

Missing data are inevitable in real life, so our time series tools should be capable of dealing with them.

Let’s create a function to induce missing values completely at random:

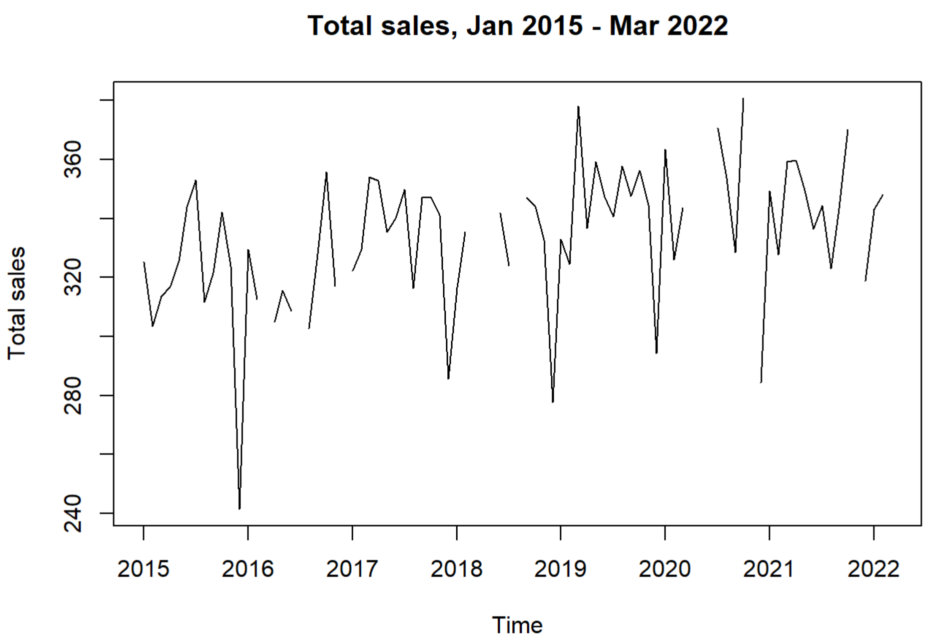
```
add_some_nas <- function(x, prop = 0.10) {
  num_nas <- floor(length(x) * prop)
  ind <- sample(1:length(x), num_nas, replace = TRUE) %>%
    sort()
  x[ind] <- NA
  return(x)
}
```

To replace 15% of our data with missing data, we can run the following:

```
add_some_nas(Xt, prop = 0.15)
#>          Jan     Feb     Mar     Apr     May     Jun     Jul     Aug
#> 2015 325.273 303.356 313.543 316.920 325.473      NA 353.144 311.672
#> 2016 329.574 312.653 340.537 304.959 315.518 308.690 312.576 302.588
#> 2017 322.235 329.461 354.007 352.897      NA 340.224 349.714 316.454
#> 2018 316.660 335.528 355.004 327.673      NA 341.855      NA 348.530
#> 2019 332.927 324.483 378.183 336.779 359.333 347.369      NA 357.762
#> 2020 363.523 325.939 343.628 345.138 355.321      NA 370.728 353.773
#> 2021 349.436 327.751 359.129 359.728 349.477 336.529 344.370 323.038
#> 2022 343.113      NA 392.581
#>          Sep     Oct     Nov     Dec
#> 2015 321.496      NA 323.703 241.385
#> 2016 327.892 355.899 317.012 289.476
#> 2017      NA 347.443 341.043 285.629
#> 2018      NA 344.228 332.647 277.659
#> 2019 347.709 356.384 344.739 294.365
#> 2020      NA      NA 348.229 284.286
#> 2021 343.363 370.170 357.394 318.896
#> 2022
```

If we plot a time series that contains missing values, we will notice some gaps.

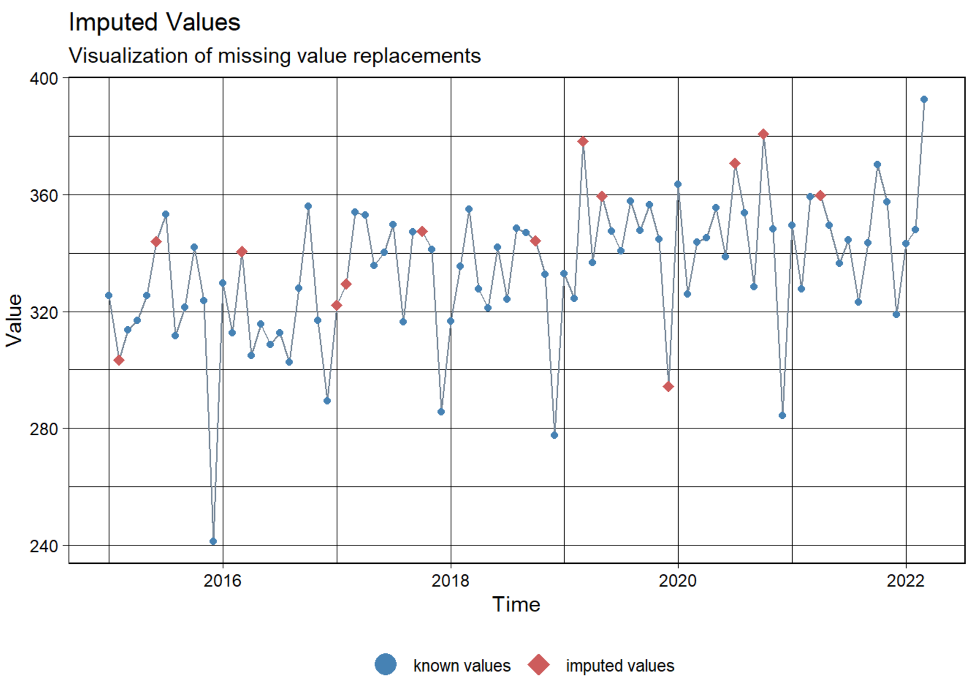
```
plot(add_some_nas(Xt, prop = 0.15), ylab = "Total sales",
     main = "Total sales, Jan 2015 - Mar 2022")
```



At first glance, it might be straightforward to fill in the gaps by connecting the lines. However, things are not so simple.

The `ggplot_na_imputations()` function from the `{imputeTS}` package provides a very nice plot of the known and missing values.

```
month_index <- seq(
  as.Date("2015-01-01"), as.Date("2022-03-31"), by = "month")
imputeTS::ggplot_na_imputations(
  x_with_na = add_some_nas(Xt, prop = 0.15),
  x_with_imputations = Xt,
  x_axis_labels = month_index)
```



If we run the above code multiple times we can see that there are cases where imputing the missing values simply by connecting the lines between the known values is not sufficient.

Missing data imputation methods for time series are quite different from those used in tabular data. The following list includes some common methods provided by the `{imputeTS}` package:

- `na_interpolation()`: Set the argument `option` to `"linear"` for linear interpolation using `approx` (default choice), `"spline"` for spline interpolation using `spline`, `"stine"` for Stineman interpolation using `stinterp`.
- `na_ma()`: Set the argument `weighting` to `"simple"` for simple Moving Average (SMA), `"linear"` for Linear Weighted Moving Average (LWMA), `"exponential"`

for Exponential Weighted Moving Average (EWMA) (default choice).

- `na_kalman()`: Set the argument `model` to `"auto.arima"` for using the state space representation of arima model (using `auto.arima`) (default choice), `"StructTS"` for using a structural model fitted by maximum likelihood (using `StructTS`)

Take the time series `Xt` and induce a proportion of missing values. Pick one or more methods from the list above and impute the missing values. Calculate the MAE (mean absolute error) of the estimate. Which method seems to work best?

```
Xm <- add_some_nas(Xt, prop = 0.15)
```

```
mae_calc <- function(na_est) {
  abs(c(Xt) - c(na_est)) %>% sum()
  }
```

```
list(
  na_interpolation(Xm, option = "linear"),
  na_interpolation(Xm, option = "spline"),
  na_interpolation(Xm, option = "stine"),
  na_ma(Xm, weighting = "simple"),
  na_ma(Xm, weighting = "linear"),
  na_ma(Xm, weighting = "exponential"),
  na_kalman(Xm, model = "auto.arima"),
  na_kalman(Xm, model = "StructTS")
) %>%
  sapply(mae_calc) %>% round(1)
#> [1] 195.7 203.7 196.4 212.9 214.8 217.2 182.1 155.1
```

6.3 Time series forecasting

Time series forecasting is an extremely broad topic, while new models are still being constantly developed. The two basic models one needs to begin with, are

- Exponential smoothing state space model
- Autoregressive integrated moving average (ARIMA)

In this section we will briefly explore the exponential smoothing model.

Suppose we train an exponential smoothing model on the data January 2015 - March 2021, and then test it on the subset April 2021 - March 2022. We can use the `window()` function from `{stats}` for time-based subsetting.

```
Xt_train <- window(Xt, end = c(2021, 3))
Xt_test <- window(Xt, start = c(2021, 4))
```

The `ets()` function from `{forecast}` will choose the model automatically:

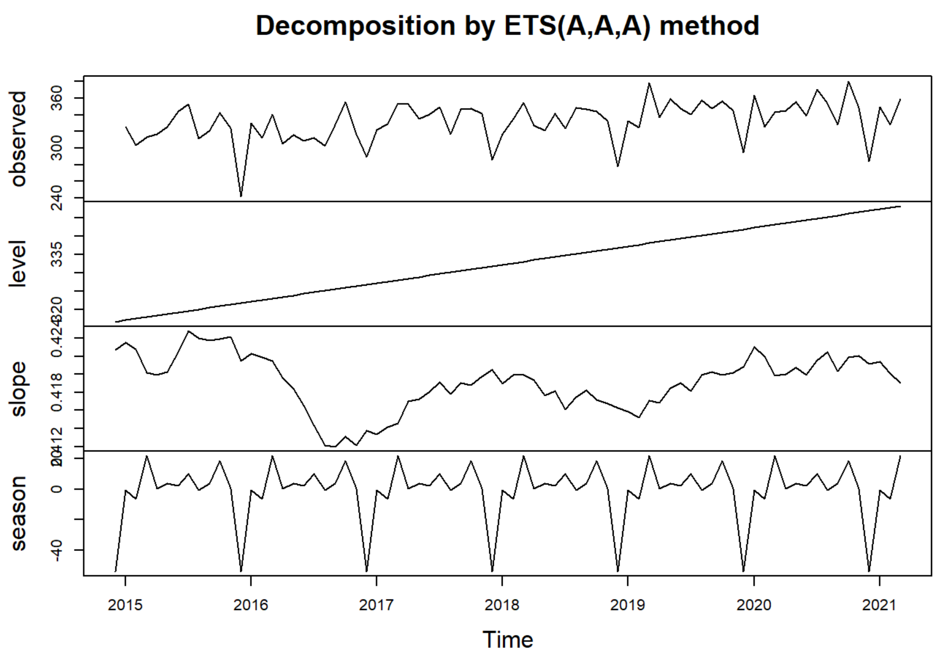
```
ets_Xt <- ets(Xt_train, model = "ZZZ")
ets_Xt
#> ETS(A,A,A)
#>
#> Call:
#>  ets(y = Xt_train, model = "ZZZ")
#>
#>   Smoothing parameters:
#>     alpha = 1e-04
#>     beta  = 1e-04
#>     gamma = 1e-04
#>
#>   Initial states:
#>     l = 316.7511
#>     b = 0.4227
#>     s = -53.8918 0.4148 18.7799 3.8304 -0.4896 9.8538
#>            2.4649 3.9086 0.4858 21.6898 -6.5814 -0.4652
#>
#>   sigma:  14.0509
#>
#>      AIC     AICc      BIC
#> 736.2180 746.9549 775.6153
```

The ETS implementation in the `{fable}` package has a slightly different syntax:

```
Xt_train %>%
  tsibble::as_tsibble() %>%
  model(fable::ETS(value)) %>%
  report()
#> Series: value
#> Model: ETS(A,A,A)
#>   Smoothing parameters:
#>     alpha = 0.0001000693
#>     beta  = 0.0001000486
#>     gamma = 0.000100127
#>
#>   Initial states:
#>      l[0]      b[0]      s[0]     s[-1]    s[-2]    s[-3]      s[-4]
#>  316.7511 0.4226941 -53.89184 0.4147673 18.77994 3.830359 -0.4895513
#>    s[-5]    s[-6]    s[-7]     s[-8]   s[-9]    s[-10]     s[-11]
#>  9.85383 2.464927 3.908609 0.4857569 21.6898 -6.581396 -0.4652033
#>
#>   sigma^2:  197.4271
#>
#>      AIC     AICc      BIC
#> 736.2180 746.9549 775.6153
```

We can plot our model as:

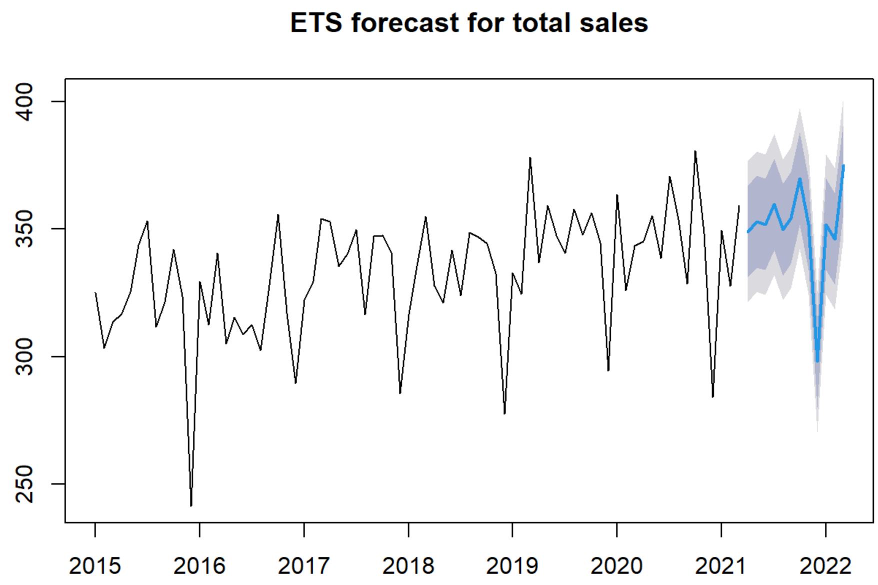
```
plot(ets_Xt)
```



Let's obtain and plot a forecast for the test set (12 months ahead of our data):

```
fc_ets_Xt <- forecast::forecast(ets_Xt, 12)
fc_ets_Xt
#>          Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
#> Apr 2021       349.0877 331.0808 367.0946 321.5485 376.6269
#> May 2021       352.9301 334.9232 370.9370 325.3909 380.4693
#> Jun 2021       351.9069 333.9000 369.9138 324.3677 379.4461
#> Jul 2021       359.7131 341.7062 377.7201 332.1739 387.2523
#> Aug 2021       349.7887 331.7818 367.7956 322.2495 377.3279
#> Sep 2021       354.5277 336.5208 372.5346 326.9885 382.0669
#> Oct 2021       369.8978 351.8908 387.9047 342.3585 397.4370
#> Nov 2021       351.9505 333.9435 369.9574 324.4112 379.4897
#> Dec 2021       298.0617 280.0548 316.0687 270.5225 325.6010
#> Jan 2022       351.9101 333.9031 369.9171 324.3708 379.4494
#> Feb 2022       346.2089 328.2019 364.2158 318.6696 373.7481
#> Mar 2022       374.8975 356.8905 392.9045 347.3582 402.4368
```

```
plot(fc_ets_Xt, main = "ETS forecast for total sales")
```



Our forecast is:

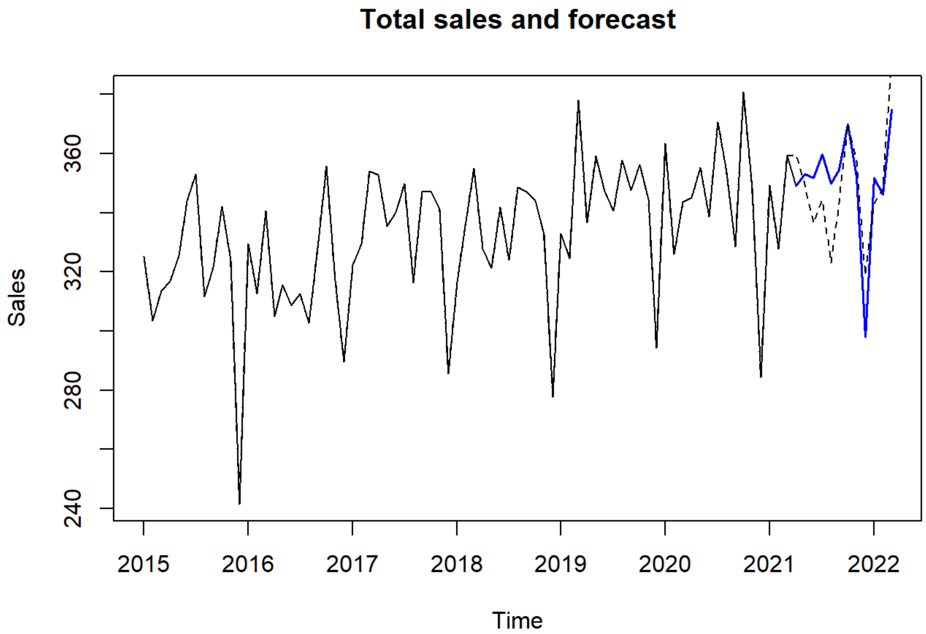
```
fc_mean_ets_Xt <- round(fc_ets_Xt$mean, digits = 2)
fc_mean_ets_Xt
#>         Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep
#> 2021                      349.09 352.93 351.91 359.71 349.79 354.53
#> 2022 351.91 346.21 374.90
#>         Oct    Nov    Dec
#> 2021 369.90 351.95 298.06
#> 2022
```

The mean absolute error per month of our model is:

```
abs(c(Xt_test) - c(fc_mean_ets_Xt)) %>% sum() / 12
#> [1] 11.466
```

If you have your own forecast that you want to plot along with the actual time series, you can do the following:

```
ref_ts <- Xt_train
fc_ts <- fc_mean_ets_Xt
Xt_full <- ts(c(Xt_train, fc_ts),
              start = start(ref_ts), frequency = frequency(ref_ts))
plot(Xt_full, ylab = "Sales",
     main = "Total sales and forecast")
lines(fc_ts, lwd = 1.5, col = "blue")
lines(Xt, lty = 2)
```



Let’s compare the ETS forecast with 3 baseline models:

- A constant estimate (the average observed value)
- The yearly average of January 2015 - March 2021
- The previous 12 months, April 2020 - March 2021

```
Xb1 <- window(Xt, end = c(2021, 3))
fc1 <- rep(mean(Xb1), 12)
abs(c(Xt_test) - c(fc1)) %>% sum() / 12
#> [1] 20.31573
```

```
Xb2 <- window(Xt, end = c(2021, 3))
fc2 <- sapply(1:12, function(.x) mean(Xb2[seq(.x, length(Xb2), 12)]))
abs(c(Xt_test) - c(fc2)) %>% sum() / 12
#> [1] 25.64376
```

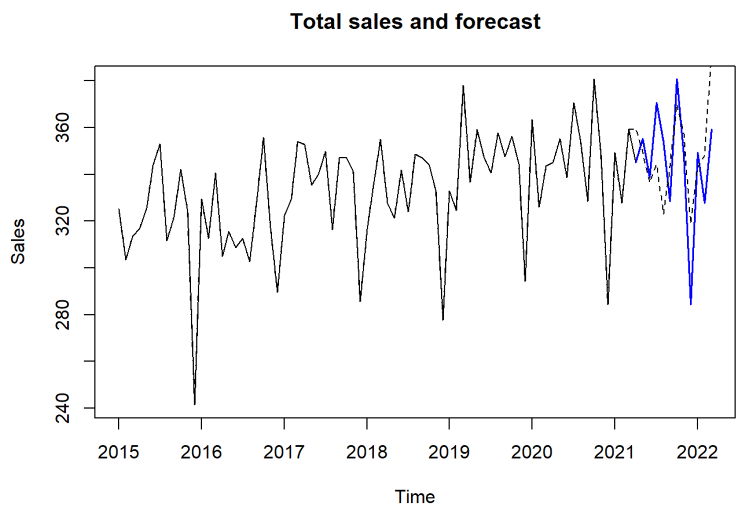
```
Xb3 <- window(Xt, start = c(2020, 4), end = c(2021, 3))
fc3 <- c(Xb3)
abs(c(Xt_test) - c(fc3)) %>% sum() / 12
#> [1] 17.41358
```

```
x0 <- c(Xt_test)
data.frame(
  actual = x0,
  baseline1 = fc1 - x0,
  baseline2 = fc2 - x0,
  baseline3 = fc3 - x0,
  ets_model = c(fc_mean_ets_Xt) - x0
) %>% round(1)
#>    actual baseline1 baseline2 baseline3 ets_model
#> 1   359.7     -27.3     -25.5     -14.6     -10.6
#> 2   349.5     -17.1     -26.7       5.8       3.5
#> 3   336.5      -4.1      12.6       2.2      15.4
#> 4   344.4     -12.0     -13.6      26.4      15.3
#> 5   323.0       9.3      12.4      30.7      26.8
#> 6   343.4     -11.0      -6.6     -14.9      11.2
#> 7   370.2     -37.8     -28.4      10.6      -0.3
#> 8   357.4     -25.0     -25.6      -9.2      -5.4
#> 9   318.9      13.5      17.7     -34.6     -20.8
#> 10  343.1     -10.7      11.3       6.3       8.8
#> 11  348.0     -15.7     -13.5     -20.3      -1.8
#> 12  392.6     -60.2    -113.8     -33.5     -17.7
```

Our ETS model outperforms the 3 baseline models.

Pick one of the 3 baseline models and plot it along with the true values.

```
ref_ts <- Xt_train
fc_ts <- ts(fc3, start = c(2021, 4), frequency = 12)
Xt_full <- ts(c(Xt_train, fc_ts),
              start = start(ref_ts), frequency =
         frequency(ref_ts))
plot(Xt_full, ylab = "Sales",
     main = "Total sales and forecast")
lines(fc_ts, lwd = 1.5, col = "blue")
lines(Xt, lty = 2)
```



Thank you!