Algoritma Academy: Time Series & Forecasting

Samuel Chan

Updated: January 31, 2019

Contents

Before you go ahead and run the codes in this coursebook, it's often a good idea to go through some initial setup. Under the *Libraries and Setup* tab you'll see some code to initialize our workspace, and the libraries we'll be using for the projects. You may want to make sure that the libraries are installed beforehand by referring back to the packages listed here. Under the *Training Focus* tab we'll outline the syllabus, identify the key objectives and set up expectations for each module.

Background

Algoritma

The following coursebook is produced by the team at Algoritma for its Data Science Academy workshops. The coursebook is intended for a restricted audience only, i.e. the individuals and organizations having received this coursebook directly from the training organization. It may not be reproduced, distributed, translated or adapted in any form outside these individuals and organizations without permission.

Algoritma is a data science education center based in Jakarta. We organize workshops and training programs to help working professionals and students gain mastery in various data science sub-fields: data visualization, machine learning, data modeling, statistical inference etc.

Libraries and Setup

We'll set-up caching for this notebook given how computationally expensive some of the code we will write can get.

```
knitr::opts_chunk$set(cache=TRUE)
options(scipen = 9999)
```

You will need to use `install.packages()` to install any packages that are not already downloaded onto your machine. You then load the package into your workspace using the `library()` function:

```
library(dplyr)
library(forecast)
library(lubridate)
library(TTR)
library(fpp)
library(xts)
```

Training Objectives

Decomposition of time series allows us to learn about the underlying seasonality, trend and random fluctuations in a systematic fashion. In the next 9 hours, we'll learn the methods to account for seasonality and trend, work with autocorrelation models and create industry-scale forecasts using modern tools and frameworks.

- **Working with Time Series**
- Additive Time Series
- Time Series Characteristics
- Log-Transformation
- **Decomposition**
- Adjusting for Trend & Seasonality
- SMA for non-seasonal data
- Two-sided SMA
- Tips and Techniques
- **More on Time Series**
- Understanding Lags
- Autocorrelation & Partial autocorrelation
- Stationary Time Series

- Augmented Dickey-Fuller Test
- **Forecasting Models**
- Exponential Smoothing
- Exponential Smoothing (Calculation)
- Plotting Forecasts
- Holt-Winters Exponential Smoothing
- SSE & Standard Errors
- Correlogram and Lags
- Confidence and Prediction Interval
- Tips and Techniques
- **Learn-by-building: Crime Forecast** Combine your data visualization skills with what you've learned about forecasting to produce an RMD report that analyze dataset of crimes in Chicago (2001-2017, by City of Chicago) and present your report.

Alternatively, edit the `wiki.R` script that I gave you to scrap some data relating to Wikipedia views on any subject that you are interested in (choose between the English or Bahasa Indonesia version of Wikipedia). Produce a well-formatted RMD report that outline your findings and forecasts. Use any forecasting or time series method and argue your case. Show at least 1 visualization to complement your writing.

Working with Time Series

Time Series Analysis describes a set of research problems where our observations are collected at regular time intervals and where we can assume correlations among successive observations. The principal idea is to learn from these past observations any inherent structures or patterns within the data, with the objective of generating future values for the series. The activity of generating future value is called "forecasting"[1].

Time series forecasting is a significant area of interest for numerous practical fields across business, finance, science, economics, politics, and engineering; Consequently, a lot of research effort has been invested in the development of forecasting models and two such models are the Holts-Winter model and ARIMA model. They are two of the most popular and frequently used models in the literature of time series[2] [3], and in the 3-day session we'll go through an array of problems to study both the theoretical and practical applications of these forecasting methods.

Before we venture any further on the subject, it is of particular importance that I remind you about the very essence of its definition. In the opening sentence I said "observations that are collected at regular intervals and where we can **assume correlations among successive observations**". Let's illustrate this point with an example.

[1] T. Raicharoen, C.Lursinsap, P. Sanguanbhoki, *Application of critical support vector machine to time series prediction*
[2] G.E.P. Box, G.Jenkins, *Time Series Analysis, Forecasting and Control*
[3] G.P. Zhang, *Time Series forecasting using a hybrid ARIMA and neural network model*

Supposed we collect some presidential election data in Indonesia from 2004 and we want to estimate voter turnout for *Pilpres 2019* using the data we've collected - if we believe that there is no correlation between voter turnout for the next presidential election and the empirical data, then the lack of assumed correlation will necessitate a different approach than a time series forecast. Political scientists who are less-aware will have committed a poor mis-judgement in their construction of their forecast. As an interesting aside, if we have looked at the turnout statistics for past General Election turnout in Malaysia we would have gathered the following time series data upon our forecasting model will be developed:

- **General Election 1982**: Voter turnout is 75.4% (4.3 million casted their vote)
- **General Election 1986**: Voter turnout is 74.3% (4.8 million casted their vote)
- **General Election 1990**: Voter turnout is 72.3% (5.7 million casted their vote)
- **General Election 1995**: Voter turnout is 68.3% (6.1 million casted their vote)
- **General Election 1999**: Voter turnout is 69.3% (6.6 million casted their vote)
- **General Election 2004**: Voter turnout is 73.9% (6.9 million casted their vote)

- **General Election 2008**: Voter turnout is 76.0% (8.1 million casted their vote)

Gathering only the above data, what would you predict the turnout size to be in the subsequent election (2013)?

Our forecast is likely going to be a poor estimate for the succeeding observation. In 2013, 11.3 million turns up to cast their votes, which is a big deviation from the past where we observe increments of +0.5 to +0.9 million. In 2018 (the most recent General Election that has concluded less than a week ago as I wrote this), 15 million casted their votes. Both in 2013 and 2018, more than 82% show up to cast their votes - again a significant deviation from empirical observations.

Time series data doesn't just apply to political campaigns - common application areas of time series include:

- `finance`: Technical trading strategies using daily share price fluctuation, or tracing the fluctuation of a current daily exchange rate, etc.

- `marketing`: Predicting global demand for a beverage using aggregated monthly sales data

- `economics`: Monthly statistics on unemployment, personal income tax, gov't expenditure records

- `socio-environmental`: Periodic records on hospital admissions, rainfall, air quality readings, seasonal influenza-associated deaths, energy demand forecasting
- `science`: ECG brain wave activity every 2^{-8} (0.004) seconds

Once you've identify an opportunity for forecasting (where past data is in fact a good indication of what may be in the future), R provides a very wide set of tools that will help us work with time series and that shall be the focus of this workshop.

Time Series in R

R has a rich set of functionality for analyzing and working with time series data and its open source community has invested substantial effort into giving it the infrastructure for representing, visualizing and forecasting time series data. The "class" we'll be working with is `ts` (stands for "time series"), and an object of class `ts` is base R's built-in way of storing and representing regularly spaced time series. It is a useful class to storing and working with annually, monthly, quarterly, daily, hourly data or even more regular intervals. The documentation of this function reads that a time series object *represent data which has been sampled at equispaced points in time*[4].

When we convert data into a time series object, we need to specify a value for the `frequency` parameter. This value indicates the number of observations in a natural period. Examples:

[4]R Documentation, Time Series Object

- We would use a value of 7 for frequency when the data are sampled daily and the natural time period we'd like to consider is a week

- We would use a value of 12 when the data are samped monthly and the natural time period is a year

Dive Deeper: - What would you use as a reasonable value for frequency if you are working with daily data and the natural time period is a year?

Let's take a look at how we can convert a generic data frame column into a univariate time series object. We'll read in some data which I've prepared in your directory. This is a dataset[5] that consists of 6 attributes representing various gas emissions contributed to Indonesia's atmosphere.

The data inspects the amount of these emissions starting from the year 1970 until the end of 2012. We'll take a look at the structure of the data we read in:

```
co2 <- read.csv("data_input/environment_1970f.csv")
str(co2)
```

```
## 'data.frame':    43 obs. of  7 variables:
##  $ year                                                                          : int  19
##  $ CO2.emissions..kt.                                                            : num  35
##  $ CO2.emissions..metric.tons.per.capita.                                        : num  0
##  $ Methane.emissions..kt.of.CO2.equivalent.                                      : num  1
##  $ Nitrous.oxide.emissions..thousand.metric.tons.of.CO2.equivalent.              : num  5
##  $ Other.greenhouse.gas.emissions..HFC..PFC.and.SF6..thousand.metric.tons.of.CO2.equivalent.: num  1
##  $ Total.greenhouse.gas.emissions..kt.of.CO2.equivalent.                         : num  3
```

The 2nd to 7th variables indicate respectively:

1. CO2 emissions (metric tons per capita):
   Carbon dioxide emissions are those stemming from the burning of fossil fuels and the manufacture of cement. They include carbon dioxide produced during consumption of solid, liquid, and gas fuels and gas flaring.

2. CO2 emissions (kt):
   Carbon dioxide emissions are those stemming from the burning of fossil fuels and the manufacture of cement. They include carbon dioxide produced during consumption of solid, liquid, and gas fuels and gas flaring.

3. Methane emissions (kt of CO2 equivalent):
   Methane emissions are those stemming from human activities such as agriculture and from industrial methane production.

4. Nitrous oxide emissions (thousand metric tons of CO2 equivalent):
   Nitrous oxide emissions are emissions from agricultural biomass burning, industrial activities, and livestock management.

5. Other greenhouse gas emissions, HFC, PFC and SF6 (thousand metric tons of CO2 equivalent):
   Other greenhouse gas emissions are by-product emissions of hydrofluorocarbons, perfluorocarbons, and sulfur hexafluoride.

6. Total greenhouse gas emissions (kt of CO2 equivalent)
   Total greenhouse gas emissions in kt of CO2 equivalent are composed of CO2 totals excluding short-cycle biomass burning (such as agricultural waste burning and Savannah burning) but including other biomass burning (such as forest fires, post-burn decay, peat fires and decay of drained peatlands), all anthropogenic CH4 sources, N2O sources and F-gases (HFCs, PFCs and SF6).

The data is yearly observations between 1970 and 2012:

[5]World Development Indicators, The World Bank DataBank. Retrieved 30/6/2017

```
range(co2$year)
```

```
## [1] 1970 2012
```

Our variable of interest is the carbon dioxide emissions per capita, so we'll use that as the value for the `data` parameter. Our `start` and `end` are 1970 and 2012 respectively, and we make 1 observation per unit of time so we'll pass that into our `frequency` parameter:

```
co2_ts <- ts(data=co2$CO2.emissions..metric.tons.per.capita.,
             start=range(co2$year)[1],
             end=range(co2$year)[2],
             frequency=1)
```

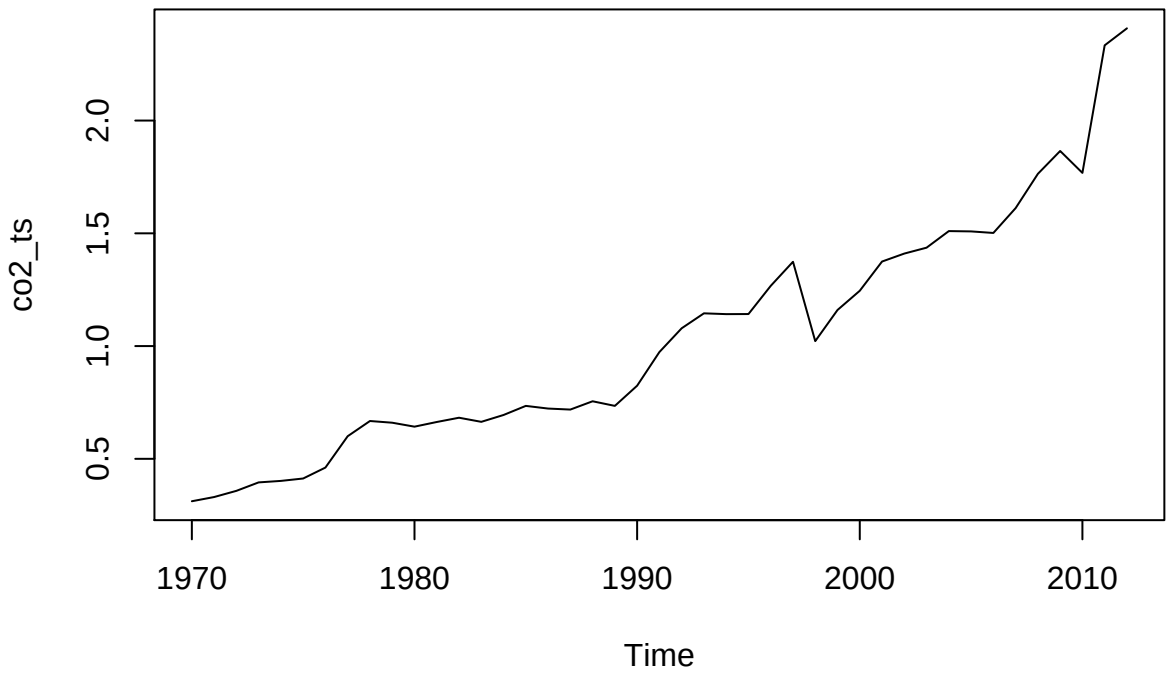
If you take a look at the `co2_ts` object we just created, we would see that it is now a Time Series object storing the values from `$CO2.emissions..metric.tons.per.capita.`:

```
co2_ts
```

```
## Time Series:
## Start = 1970
## End = 2012
## Frequency = 1
##  [1] 0.3119519 0.3306215 0.3580080 0.3954703 0.4021567 0.4128050 0.4612390
##  [8] 0.6002978 0.6677802 0.6601457 0.6426495 0.6634071 0.6822242 0.6640976
## [15] 0.6944021 0.7347680 0.7229173 0.7184145 0.7552095 0.7348064 0.8243417
## [22] 0.9735380 1.0788747 1.1452296 1.1416286 1.1420774 1.2669930 1.3738789
## [29] 1.0218517 1.1599925 1.2452416 1.3748183 1.4102338 1.4364045 1.5098982
## [36] 1.5084806 1.5015768 1.6118554 1.7638951 1.8651654 1.7679079 2.3335854
## [43] 2.4089201
```

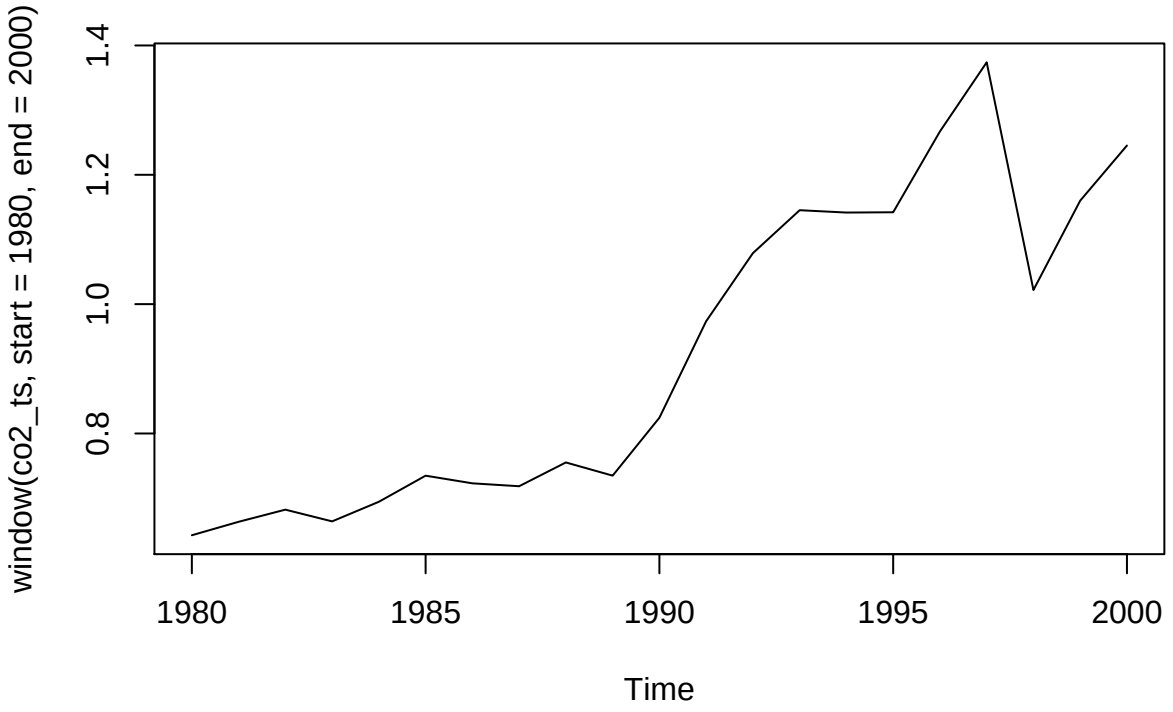
We can use plot on our time series:

```
plot(co2_ts)
```



We can subset a time series using the `window()` function and specify the `start` and `end` for the time window to be selected:

```
# subset the time series (June 2014 to December 2014)
plot(window(co2_ts, start=1980, end=2000))
```



What we just observe above is a time series with a clear trend but without any clear indication of a cyclic pattern - time series like this are generally less-interesting because there are no repeated observations for each period, making it impossible to study the underlying patterns.

In other words, you can't decompose the above time series any further, for each year there is exactly one observation (frequency = 1): and you can't extract any pattern of seasonality from a time series where frequency is 1.

Dive Deeper: `presidents` is a built-in dataset recording the quarterly Approval Ratings of US Presidents.

1. What is the class of `presidents`? If it is not a time series object, convert it to a time series object.
2. Plot the time series.
3. 2 hours and 8 minutes after John F.Kennedy's assassination, amid suspicions of a conspiracy against the government, Lyndon Johnson was sworn in as president. His presidential term is between the last quarter of 1963 (hint: `start=c(1963,4)`) to the last quarter of 1968. Create a time series plot for that time window using the appropriate `start` and `end` parameters for the `window()` function.
4. Was Lydon Johnson a popular president during his presidential term?
5. As a bonus exercise, give the plot an appropriate main title and axis labels (x and y).

Reference Answer: Lydon Johnson is arguably one of the least popular president in US history, with backlash stemming from "Frustration over Vietnam; too much federal spending and... taxation; no great public support for your Great Society programs; and ... public disenchantment with the civil rights programs". Famously, he "could scarcely travel anywhere without facing protests" as people protested and chanted "Hey, hey, LBJ, how many kids did you kill today?"[6]

[6]Lyndon Johnson left office as a deeply unpopular president. So why is he so admired today?

```
plot(window(presidents, start=c(1963,4), end=c(1968,4)))
```

Let's take a look at the following data instead. The following is monthly data starting from January 1946 corresponding to the number of births (thousands) per month in New York City:

```
births <- read.csv("data_input/nybirth.csv")
str(births)
```

```
## 'data.frame':    168 obs. of  2 variables:
##  $ date  : Factor w/ 168 levels "1946-01-01","1946-02-01",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ births: num  26.7 23.6 26.9 24.7 25.8 ...
```

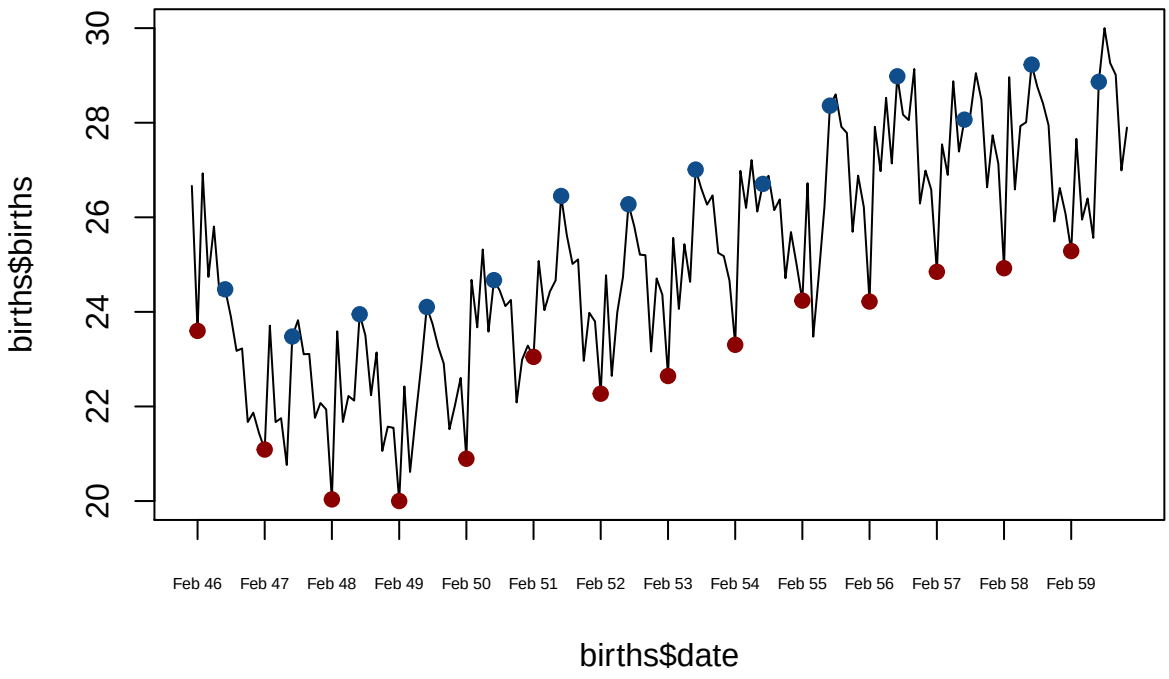
Dive Deeper: Referring to the code earlier, convert the above data frame to be a time series. Use the appropriate value for `frequency`, `start` and optionally, `end`. Plot the time series.

As you study your time series, you should notice that there are a considerable amount of "structural" pattern in your time series. We call these seasonal variation from year to year "seasonality": birth rate is lower near winter (Jan - Feb) and peaks in July (summer weather).

I marked the "February" months with a dark red points and the "July" months with a blue point to highlight that year to year variation. Your `plot` function should be called on a timeseries object and not on a dataframe like the code below as they are used for illustrative purposes only:

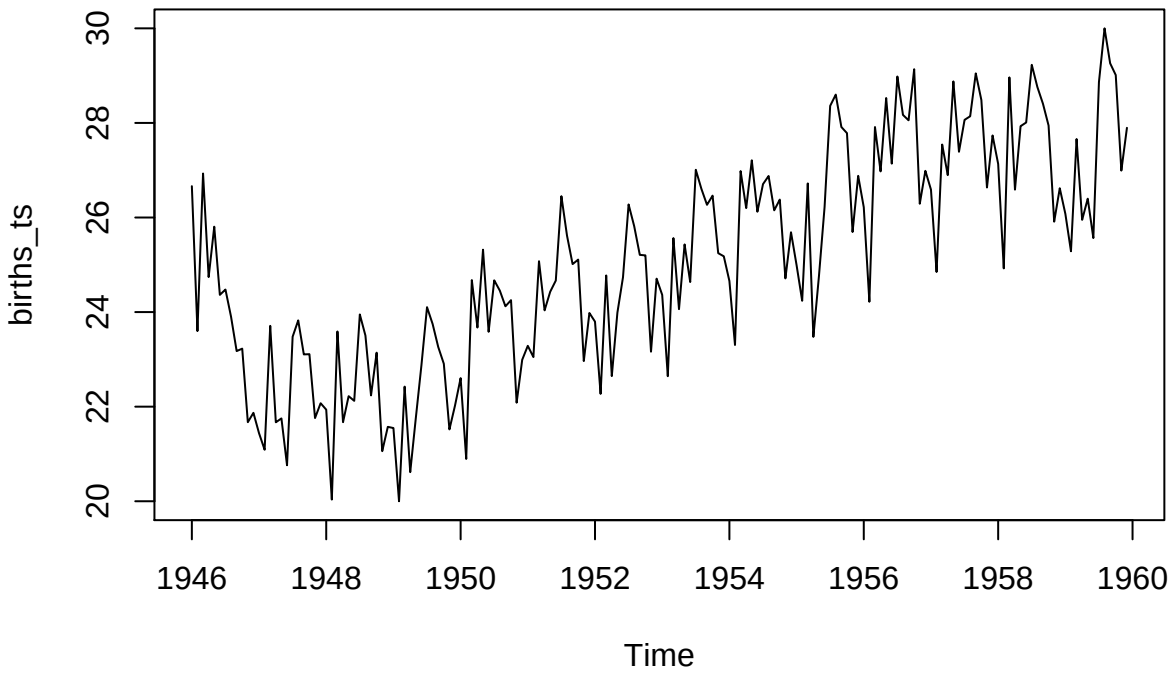
```
library(dplyr)
library(lubridate)
births <- births %>%
  mutate(date = as.Date(date), month = month(date))

plot(births$date, births$births, type="l", xaxt="n")
```

```
points(
  x =  births[births$month == 2, "date"],
  y = births[births$month == 2, "births"], col="darkred", pch=19)
points(
  x =  births[births$month == 7, "date"],
  y = births[births$month == 7, "births"], col="dodgerblue4", pch=19)
axis(1, births$date[seq(2, length(births$date), by = 12)], format(births$date[seq(2, length(births$date
```



The reference answer to the above exercise:

```
births <- read.csv("data_input/nybirth.csv")
births_ts <- ts(births$births, frequency = 12, start = c(1946,1))
plot(births_ts)
```



Trend, Seasonality, Cycles and Residuals

A simple method of describing a time series is a **classical decomposition**: a proposition that a series can be decomposed into three main elements:

- **Trend** (T_t): long term movements in the mean
- **Seasonal** (S_t): repeated seasonal fluctuations (think: calendar or business cycles)
- **Residuals** (E_t): Irregular components or random fluctuations not captured by the Trend and Seasonal The idea of classical decomposition is to create separate models for each of the three elements in the aim of describing our series either additively:

$X_t = T_t + S_t + E_t$

or multiplicatively: $X_t = T_t \cdot S_t \cdot E_t$

When we use the `decompose` function in R, we're performing the classical seasonal decomposition using moving averages (a concept we'll get into). We can optionally specify a type of the seasonal component by using the `type` parameter. By default, it assumes an additive model but this can be changed: `decompose(x, type="multiplicative")`

```
str(decompose(births_ts))
```

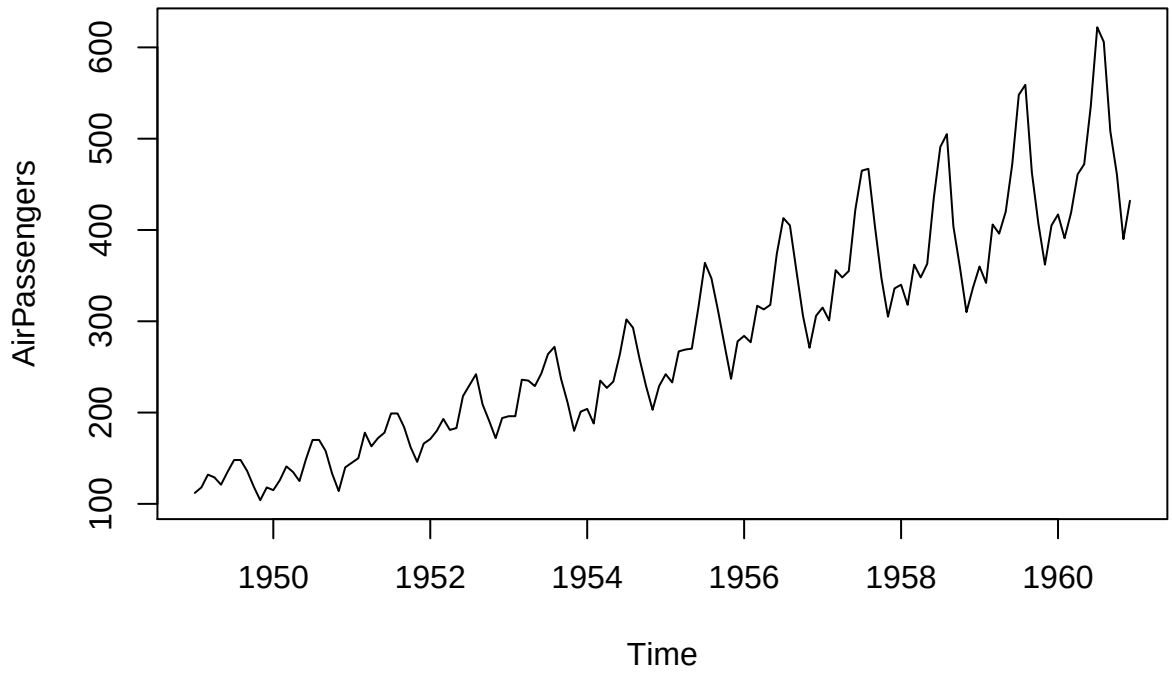
```
## List of 6
##  $ x       : Time-Series [1:168] from 1946 to 1960: 26.7 23.6 26.9 24.7 25.8 ...
##  $ seasonal: Time-Series [1:168] from 1946 to 1960: -0.677 -2.083 0.863 -0.802 0.252 ...
##  $ trend   : Time-Series [1:168] from 1946 to 1960: NA NA NA NA NA ...
```

```
##  $ random  : Time-Series [1:168] from 1946 to 1960: NA NA NA NA NA ...
##  $ figure  : num [1:12] -0.677 -2.083 0.863 -0.802 0.252 ...
##  $ type    : chr "additive"
##  - attr(*, "class")= chr "decomposed.ts"
```

Observe the plot of your time series again - an additive model seems a good fit for the time series since the seasonal variation appear to be rather constant across the observed period (x-axis). Note that a series with multiplicative effects can often be transformed into one with additive effect through a log transformation, a technique we've discussed in your Classification 1 modules.

Dive Deeper: Monthly Airline Passenger, 1949-1960 1. Look at the following plot of an airline passenger time series. Recall that an additive model is appropriate if the seasonal and random variation is constant. Do you think this time series is better described using an additive or multiplicative model?

```
data("AirPassengers")
plot(AirPassengers)
```



2. Is it possible to transform the air passengers data such that it can be described using an additive model?

```
# your code below:
```

Tip (Extra Intuition): If you needed help with (2) above, recall about the log-transformation we've learned in the chapter of Logistic Regression from the Classification 1 workshop. A series like the following seems multiplicative (the values increase by a magnitude of ~4.5 from the previous value). However, wrap them in `log()` and you'll transform the series into an additive one:

```
c(7.389056, 33.115451, 148.413159, 665.141633, 2980.957987)
```

```
## [1]    7.389056   33.115451  148.413159  665.141633 2980.957987
```

Now let's inspect the `seasonal` component and notice that the values in each month of the year is the same: this agrees with the definition of the decomposition process as well as our "visual" inspection earlier. In February, a maximum loss of -2.08 is applied to the additive equation while in July we see a maximum gain of 1.45 being applied to the additive equation, corresponding to the dips and peaks in these two months:

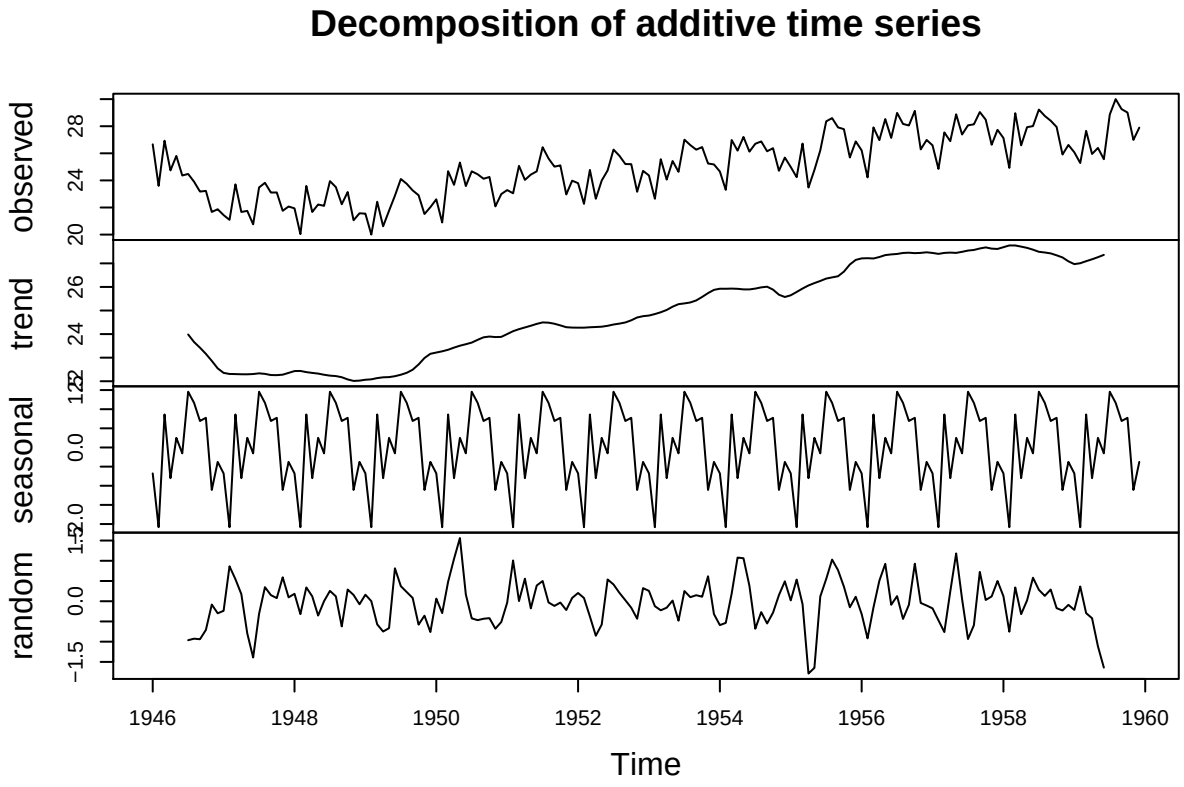
```
births_dc <- decompose(births_ts)
births_dc$seasonal
```

```
##             Jan        Feb        Mar        Apr        May        Jun
## 1946 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1947 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1948 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1949 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1950 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1951 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1952 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1953 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1954 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1955 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1956 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1957 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1958 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1959 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
##             Jul        Aug        Sep        Oct        Nov        Dec
## 1946  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1947  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1948  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1949  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1950  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1951  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1952  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1953  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1954  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1955  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1956  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1957  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1958  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1959  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
```

In fact if you plot the decomposition of time series you will see that the seasonal is constant among the months across the observed period.

- The first panel from top is the original, *observed* time series.
- The second panel plots the *trend* component, **using a moving average with a symmetric window with equal weights**. This means that for each point on the series, a new value is estimated by taking the average of the point itself +- 6 points (since frequency = 12)
- The third panel plots the *seasonal* component, with the figure being computed by taking the average for each time unit over all periods and then centering it around the mean
- The bottom-most panel the *error* component, which is determined by removing the trend and seasonal figure

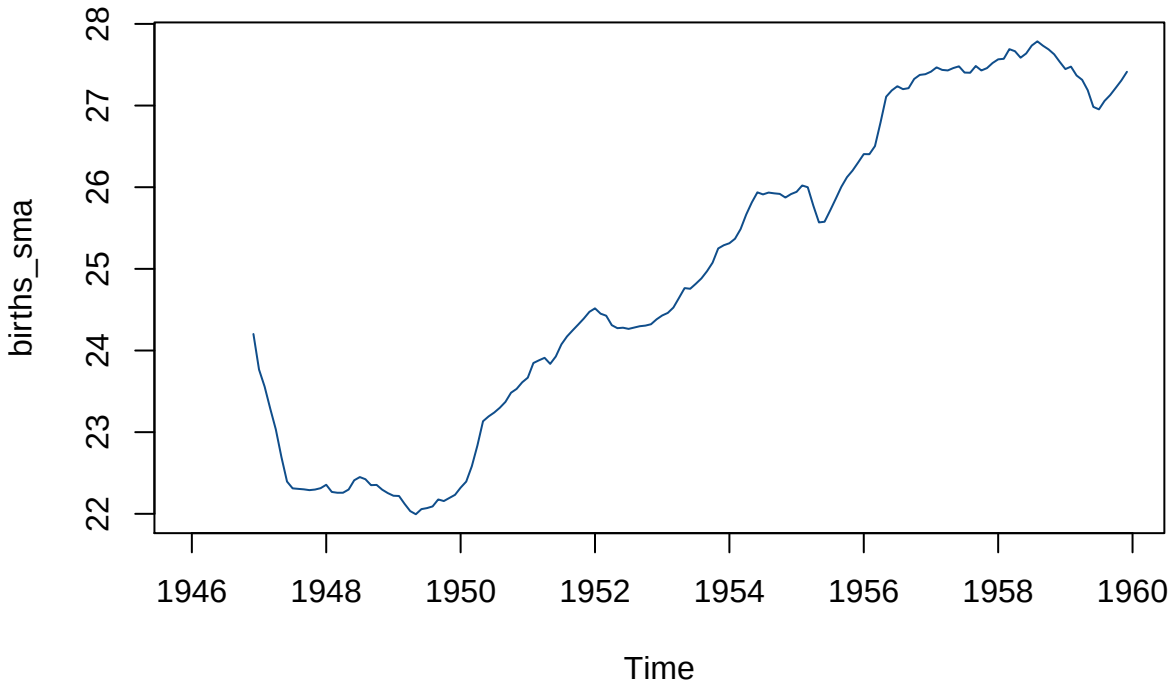
```
plot(births_dc)
```



Mathematical details of Time Series Decomposition

The `SMA` function in R calculates various moving averages (MA) of a series, but only do so by averaging over past n observations - which is **not** the moving average model that our classical decomposition uses. When you do a simple moving average on the original time series using `SMA`, you'll set `n` to be the number of past observations over which an average is computed.

```
library(TTR)
births_sma <- SMA(births_ts, n = frequency(births_ts))
plot.ts(births_sma, col="dodgerblue4")
```



So to solidify our understanding of the classical decomposition process, let's study the mathematical details behind `decompose()` by comparing our manual calculation to the one we obtain from `decompose`.

When we estimate the Trend component from a time series by using a moving average, recall that the specification is to have a symmetric window with equal weights. For a time series with 12 as the frequency value, the weight assigned to each observations within that considered window would be the same:

```
c(0.5, rep_len(1, 12-1), 0.5)/12
```

```
##  [1] 0.04166667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
##  [7] 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
## [13] 0.04166667
```

That applies to a frequency with an even number, so both sides to the point are equally considered. If frequency was an odd number, the calculation of weight is more straightforward:

```
frq <- 7
rep_len(1, frq)/frq
```

```
## [1] 0.1428571 0.1428571 0.1428571 0.1428571 0.1428571 0.1428571 0.1428571
```

Using the weights above, we can now calculate the moving average of observation #7. The first and last value is the considered window is assigned a weight of 0.04166667, and other points are given equal weight:

```
sum(
as.vector(births_ts[1])*0.04166667,
as.vector(births_ts[2:12])*0.08333333,
as.vector(births_ts[13])*0.04166667
)
```

```
## [1] 23.98433
```

To calculate the moving average of observation #8, we move the considered window by one unit of time to the right and then apply the weight similarly to the above:

```
sum(
as.vector(births_ts[2])*0.04166667,
as.vector(births_ts[3:13])*0.08333333,
as.vector(births_ts[14])*0.04166667
)
```

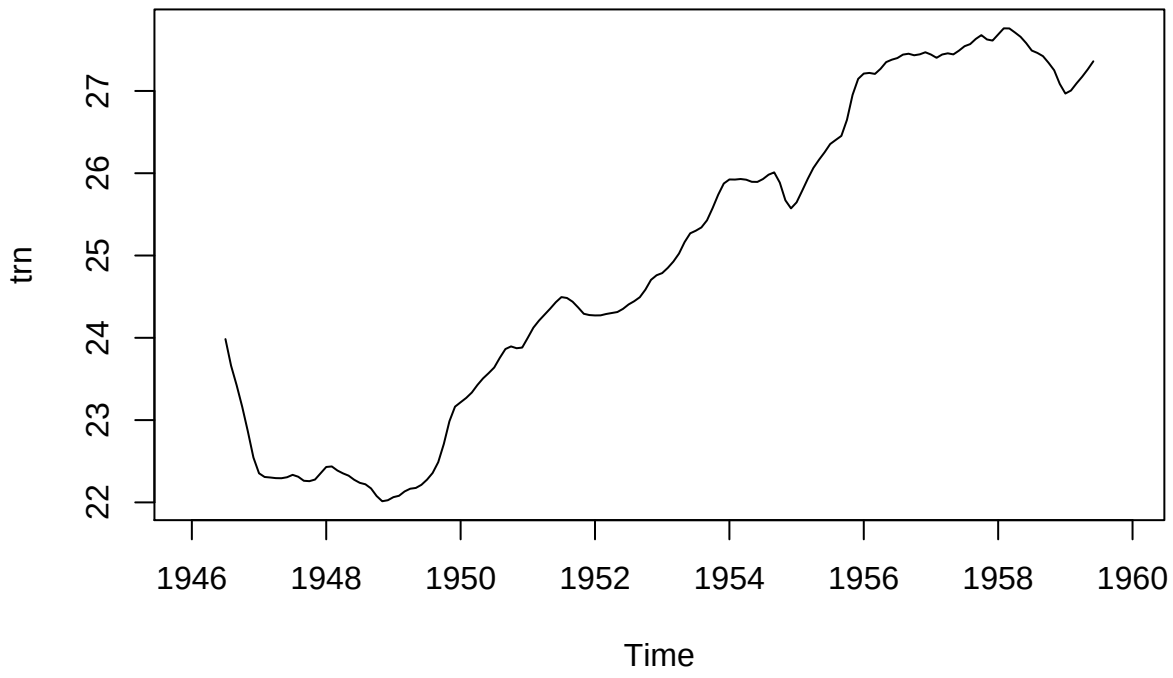
```
## [1] 23.66212
```

If that seems tedious, the good news is that we can make use of the `filter()` function which by default apply our weights / coefficients to both sides of the point. Observe that the result for observation 7 and 8 is the equivalent to our manual calculation above:

```
coef1 <- c(0.5, rep_len(1, frequency(births_ts)-1), 0.5)/frequency(births_ts)
trn <- stats::filter(births_ts, coef1)
trn
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul
## 1946       NA       NA       NA       NA       NA       NA 23.98433
## 1947 22.35350 22.30871 22.30258 22.29479 22.29354 22.30562 22.33483
## 1948 22.43038 22.43667 22.38721 22.35242 22.32458 22.27458 22.23754
## 1949 22.06375 22.08033 22.13317 22.16604 22.17542 22.21342 22.27625
## 1950 23.21663 23.26967 23.33492 23.42679 23.50638 23.57017 23.63888
## 1951 24.00083 24.12350 24.20917 24.28208 24.35450 24.43242 24.49496
## 1952 24.27204 24.27300 24.28942 24.30129 24.31325 24.35175 24.40558
## 1953 24.78646 24.84992 24.92692 25.02362 25.16308 25.26963 25.30154
## 1954 25.92446 25.92317 25.92967 25.92137 25.89567 25.89458 25.92963
## 1955 25.64612 25.78679 25.93192 26.06388 26.16329 26.25388 26.35471
## 1956 27.21104 27.21900 27.20700 27.26925 27.35050 27.37983 27.39975
## 1957 27.44221 27.40283 27.44300 27.45717 27.44429 27.48975 27.54354
## 1958 27.68642 27.76067 27.75963 27.71037 27.65783 27.58125 27.49075
## 1959 26.96858 27.00512 27.09250 27.17263 27.26208 27.36033       NA
##           Aug      Sep      Oct      Nov      Dec
## 1946 23.66213 23.42333 23.16112 22.86425 22.54521
## 1947 22.31167 22.26279 22.25796 22.27767 22.35400
## 1948 22.21988 22.16983 22.07721 22.01396 22.02604
## 1949 22.35750 22.48862 22.70992 22.98563 23.16346
## 1950 23.75713 23.86354 23.89533 23.87342 23.88150
## 1951 24.48379 24.43879 24.36829 24.29192 24.27642
## 1952 24.44475 24.49325 24.58517 24.70429 24.76017
## 1953 25.34125 25.42779 25.57588 25.73904 25.87513
## 1954 25.98246 26.01054 25.88617 25.67087 25.57312
## 1955 26.40496 26.45379 26.64933 26.95183 27.14683
## 1956 27.44150 27.45229 27.43354 27.44488 27.46996
## 1957 27.56933 27.63167 27.67804 27.62579 27.61212
## 1958 27.46183 27.42262 27.34175 27.25129 27.08558
## 1959       NA       NA       NA       NA       NA
```

```
plot(trn)
```



Compare the figures you've arrived at manually to the trend component output from the `decompose()` function. They are the same.

Now that we know how to arrive manually at the values of our Trend component, let's see how we arrive at the Seasonal component of the time series. Recall that the frequency of our time series is 12, and we will use that to create a sequence consisting of the multiples of 12. Informally, we are going to take the average of the values in this sequence later, add 1 to each value in the sequence and again record the average, then add 1 again - and repeat this 12 times so we end up with a series of length 12, each being the mean of the values in each iteration.

```
f <- frequency(births_ts)
index <- seq.int(1L, length(births_ts), by = f) - 1L
index
```

```
##  [1]   0  12  24  36  48  60  72  84  96 108 120 132 144 156
```

Formally, here's the code to do it:

```
prd <- length(births_ts)/frequency(births_ts)
# if multiplicative: birth_ts/births_dc$trend
detrend <- births_ts - births_dc$trend

# initialize figure to 0 0 0 0 0 ... 0 (12 times)
figure <- numeric(f)

for (i in 1L:f)
      figure[i] <- mean(detrend[index + i], na.rm = TRUE)
```

```r
# if multiplicative: figure / mean(figure)
figure <- figure - mean(figure)

ssnl <- ts(rep(figure, prd + 1)[seq_len(length(births_ts))], start = start(births_ts), frequency = f)
ssnl
```

```
##             Jan        Feb        Mar        Apr        May        Jun
## 1946 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1947 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1948 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1949 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1950 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1951 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1952 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1953 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1954 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1955 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1956 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1957 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1958 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
## 1959 -0.6771947 -2.0829607  0.8625232 -0.8016787  0.2516514 -0.1532556
##             Jul        Aug        Sep        Oct        Nov        Dec
## 1946  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1947  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1948  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1949  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1950  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1951  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1952  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1953  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1954  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1955  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1956  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1957  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1958  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
## 1959  1.4560457  1.1645938  0.6916162  0.7752444 -1.1097652 -0.3768197
```

Discussion: 1. Compare the above to the seasonal component that our decomposition function returned. Are they identical?

2. Using `births_ts`, `trn` and `ssnl`, can you estimate the random components in our time series? `trn` and `ssnl` are the components we calculated by hand, and we already confirmed in earlier exercises that they are the same as the outputs from the classical decomposition function built-in with R.

3. I've read the following data and save it to an object named `souvenir`. Create a time series using the data above and name it `souvenir_ts`.

```r
souvenir <- scan("data_input/fancy.dat")
souvenir_ts <- ts(souvenir, frequency = 12, start = c(1987,1))
```

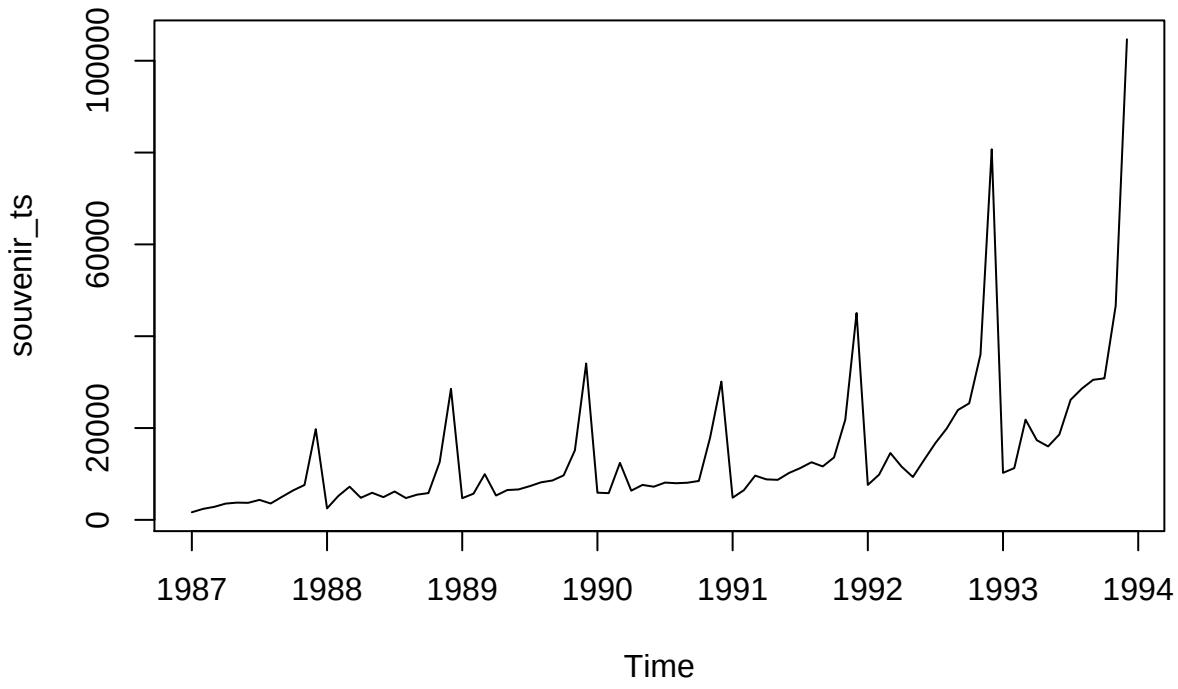
Plot the time series. Can you tell if an additive model is appropriate for this time series?

Write your solution for (Q3) below:

Seasonally Adjusting our time series

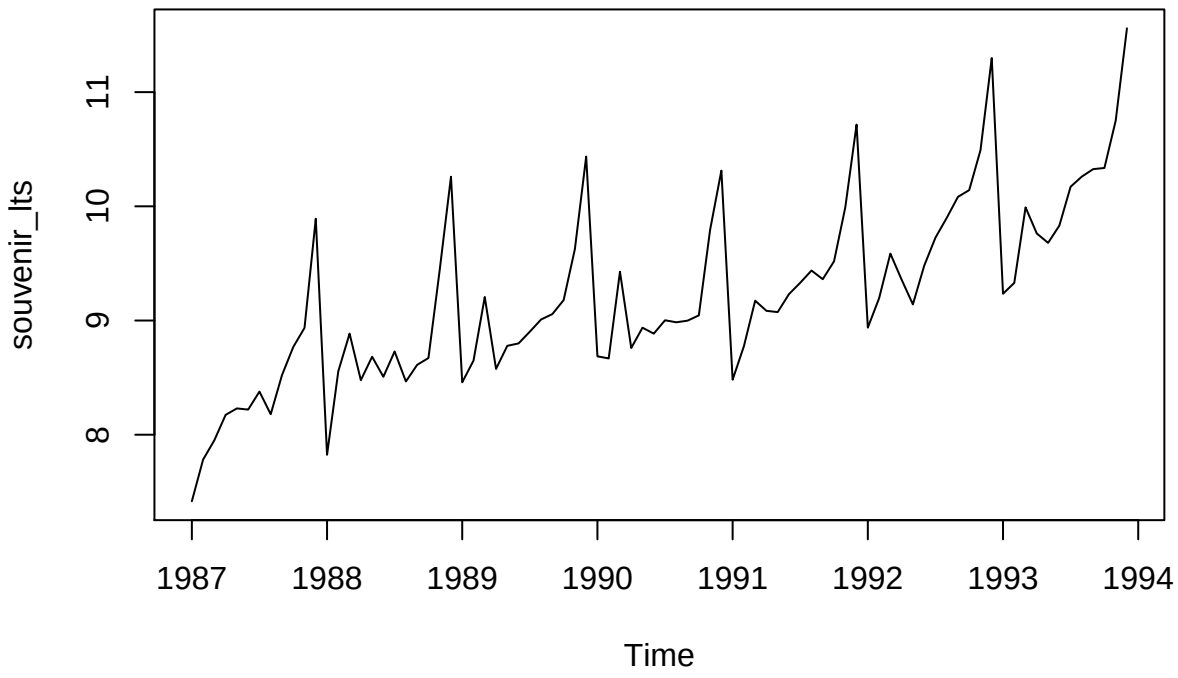
The following code plots a time series from our data, `fancy.dat`:

```
souvenir <- scan("data_input/fancy.dat", quiet = T)
souvenir_ts <- ts(souvenir, frequency = 12, start = c(1987,1))
plot.ts(souvenir_ts)
```



In this case it appears an additive model is not appropriate for describing this time series, since both **the size of the seasonal fluctuations and size of the random fluctuations seem to increase** with the level of the time series. Thus, we may need to transform the time series and get a transformed time series that can be described using an additive model:

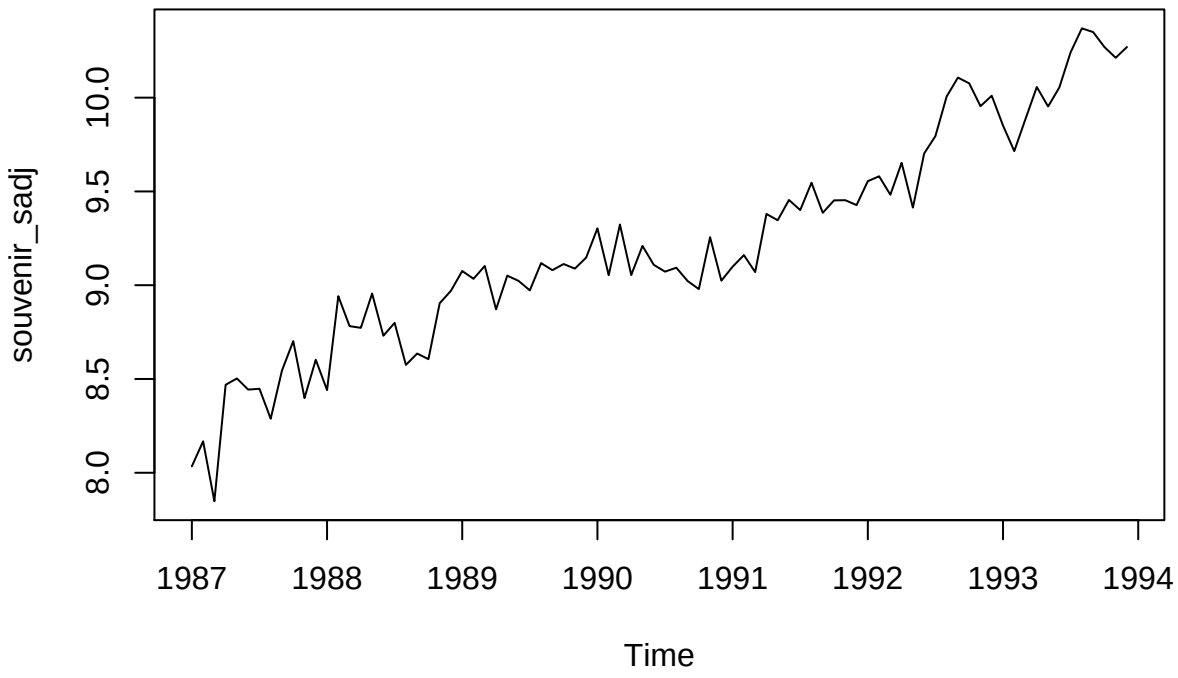
```
souvenir_lts <- log(souvenir_ts)
plot.ts(souvenir_lts)
```



Now observe that **the size of the seasonal fluctuations and random fluctuations in the log-transformed time series are roughly constant** and no longer dependent on the level of the time series. This log-transformed time series can better be described by an additive model. We still observe seasonal fluctuations in our data due to the seasonal components, but what if we observe the sales trend after having it adjusted for the seasonality effect? Would it still be generally upward-trending, or is our optimism unwarranted?

The following code first decompose the log-transformed time series, and then remove the seasonal component from the time series. We can see that, having adjusted for seasonality our sales have still seen an upward-rising trend:

```
souvenir_dc <- decompose(souvenir_lts)
souvenir_sadj <- souvenir_lts - souvenir_dc$seasonal
plot(souvenir_sadj)
```



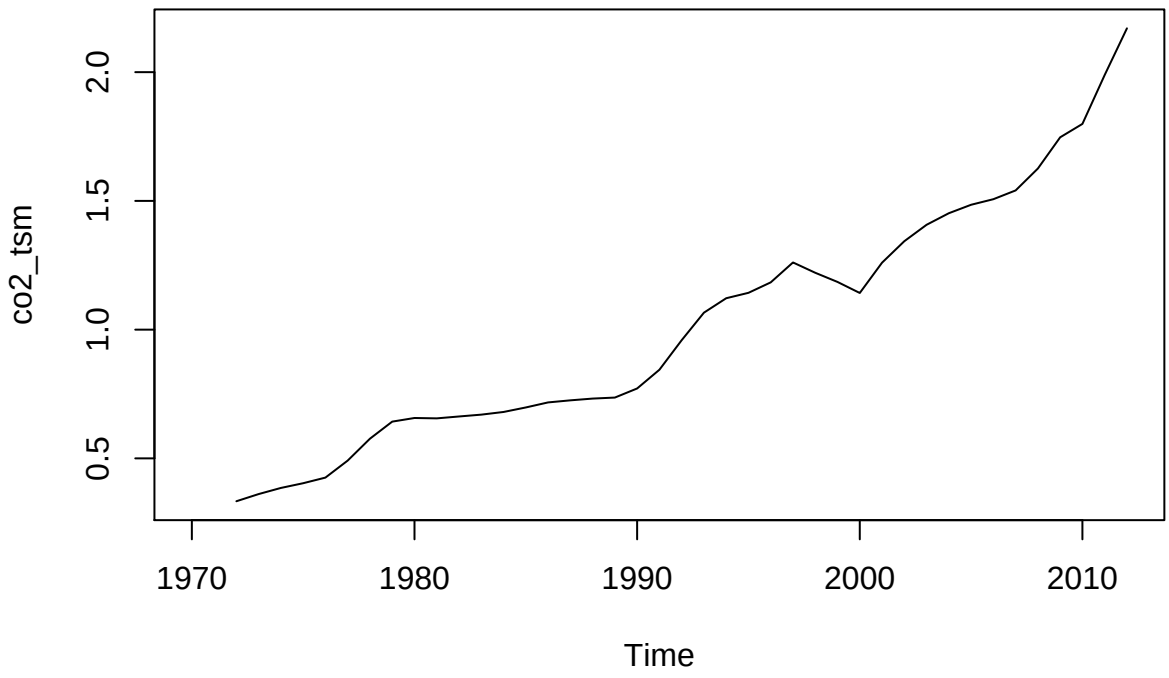
So far we've been working with time series data with a seasonality component. In the following sub-section, let's take a look at how we can work with non-seasonal data.

Decomposing Non-Seasonal Time Series

Decomposing a non-seasonal time series means separating the time series into the trend component and the irregular (random) component. To estimate the trend component of a non-seasonal time series that can be described using an additive model, it is common to use a smoothing method, such as calculating the simple moving average of the time series.

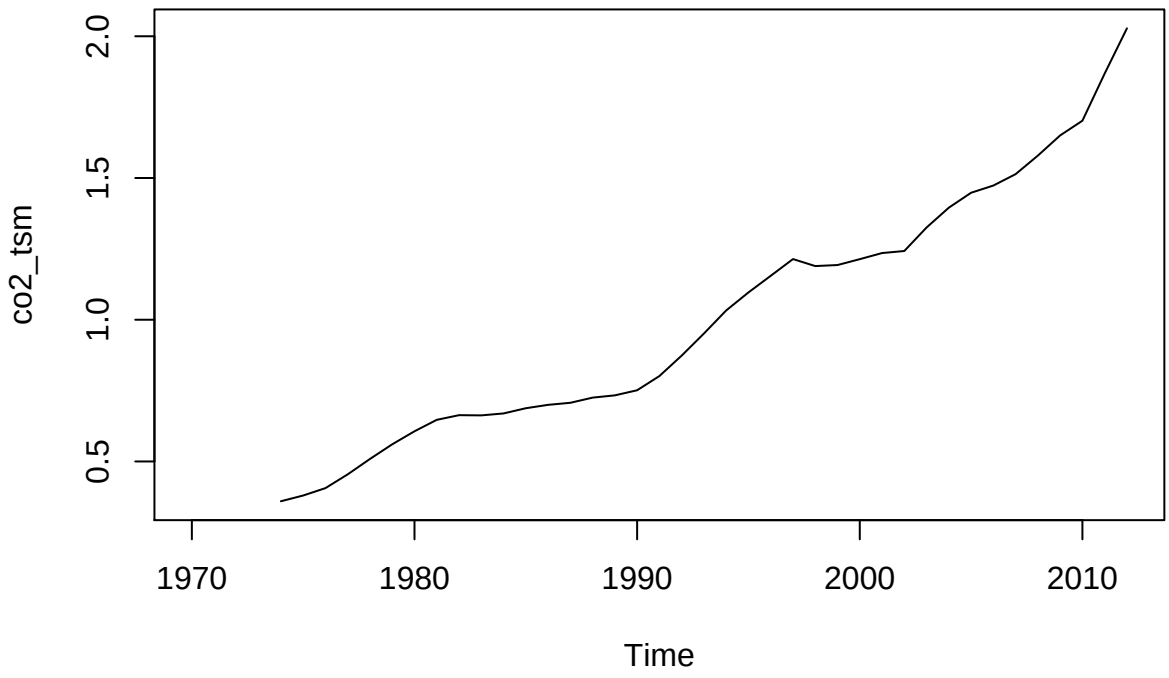
The `SMA()` function in the `TTR` package that we learn earlier can be used to smooth time series data using a simple moving average; Otherwise, you can also use the `filter()` function to apply a two-sided moving average. To see an example, consider the gas emissions in Indonesia data and observe how we first apply our `SMA()` function before plotting the time series:

```
library(TTR)
co2_tsm <- SMA(co2_ts, n=3)
plot(co2_tsm)
```



There still appears to be some random fluctuations with a simple moving average of order 3. To estimate the trend component more accurately, we may want to use smoothing with a higher order:

```
co2_tsm <- SMA(co2_ts, n=5)
plot(co2_tsm)
```



The data smoothed with a simple moving average of order 5 gives a clearer picture of the trend component, and we can see that the level of co2 emissions contributed to Indonesia's atmosphere have risen rather sharply in the more recent years (2005-2012). The increase was much sharper than in the past, say in the 20 years between 1970 to 1990.

As a reminder, recall that `SMA` calculates the arithmetic mean of the series over the past n observations. This is essentially a **one-sided moving average** and can be represented in the following form: $z_t = \frac{1}{k+1} \sum_{j=0}^{k} y_{t-j}$

Compare this to a two-sided moving average which takes the form of: $z_t = \frac{1}{2k+1} \sum_{j=-k}^{k} y_{t+j}$

Where k is the number of observations to consider in your moving average window and t is the observation we're replacing with a moving average.

Forecasts using Simple Moving Average

When we apply the simple moving average method we've just learned above on future values of our time series, we're essentially performing a forecasting using a rather simplistic method. The value of y_{t+1} is estimated by taking the average of the past 5 values $y_{t-4} + ... + y_t$, and then subsequently y_{t+2} is estimated by taking the average of $y_{t-3} + ... + y_{t+1}$ etc.

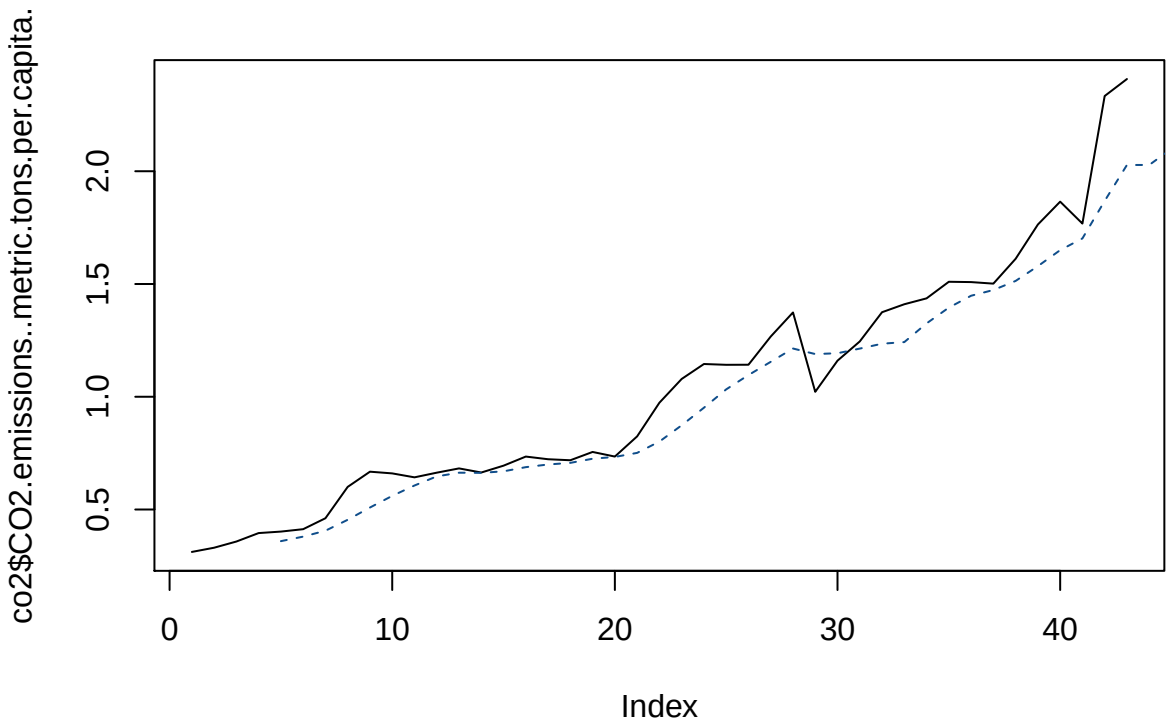
```
future1 <- mean(tail(co2_ts, 5))
future2 <- mean(tail(co2_ts, 4), future1)
future3 <- mean(tail(co2_ts, 3), future1, future2)
future1; future2;
```

```
## [1] 2.027895
```

```
## [1] 2.099375
```

And if we were to plot our observation, along with the simple moving average (+3 future points) then we have a forecast model (even though a simple one!):

```
plot(co2$CO2.emissions..metric.tons.per.capita., type="l")
co2_sma <- SMA(co2_ts, n=5)
lines(c(co2_sma, future1, future2, future3), col="dodgerblue4", lty=2)
```



Theoretical Considerations with a SMA Forecast

I get asked sometimes on when a simple moving average would suffice as a forecasting model, and I like to use that as an opportunity to help the student think about forecasting in general. If I present to you a constant mean model and tell you that the best estimate of tomorrow's oil prices is the average of oil prices throughout history (since the existence of the stock market), you would argue that it is unlikely to be a good estimate because a lot of the history should have been ignored in the forecast as they play no role in the oil prices of tomorrow. What we just described is sometimes the **mean model** or the **constant model**[7]. Oversimplistic as it may sound, that is a reasonable model if the values we're predicting are independently and identically distributed.

Intuitively, you've shown an understanding that some of the very-distant past isn't likely to help in our particular forecasting problem, and what matters tomorrow can better be estimated using a window of the more recent past - that is the basis of a SMA model. This model works best on time series with an

[7]Answer for Dive Deeper, because it's an additive model your random components are `births_ts - trn - ssnl`; Say it's a multiplicative model you would use `births_ts / trn / ssnl`

almost-constant or slowly (but non-systematically) changing mean because of the underlying calculation (refer to above subsection). If the mean is almost constant, choosing n to be the total number of observations, hence using the full dataset, will give us the best estimate of a future value. Naturally, a longer observation period will average out the fluctuations and reduce the effect of variability.

Why do we assume that the changes are non-systematic? Because if there was any systematic changes in our mean, it would have been attributed to the seasonal or trend component and we wouldn't have flatly used the mean as a forecast.

Another reasonable strategy you may propose is to weigh our observations such that recent observations have a stronger say in determining the future value than observations in the distant past. This brings us to the exponential smoothing method of forecasting in the next chapter.

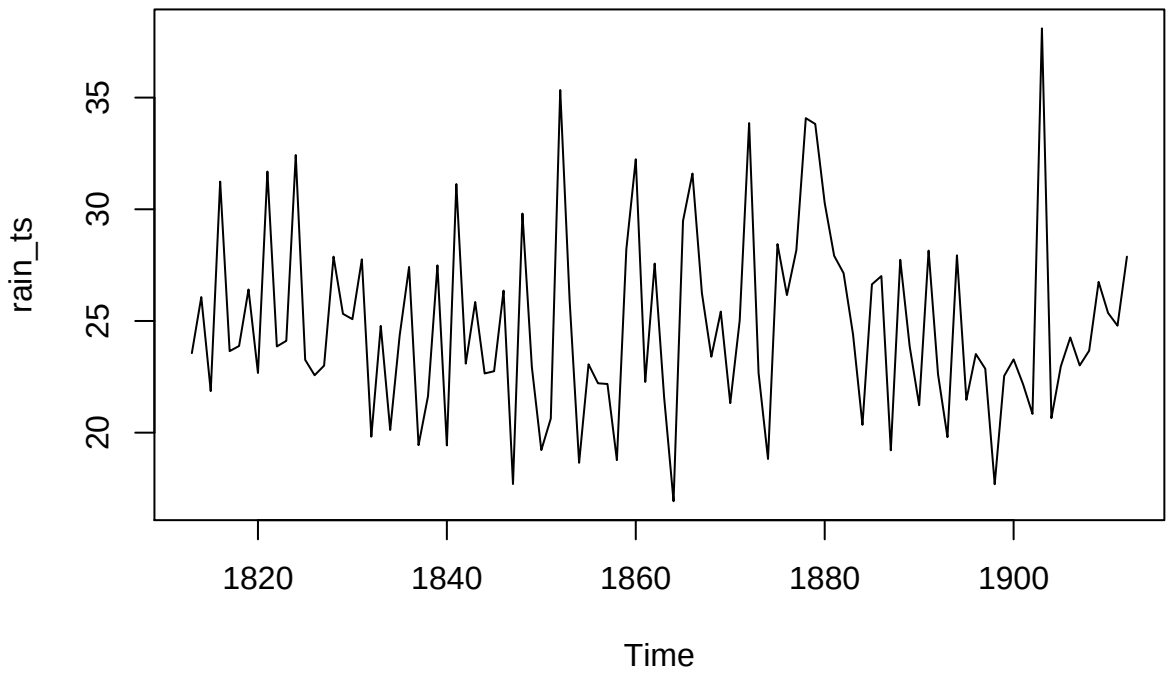
Forecasts using Exponential Smoothing

When we have a time series that can be described using an additive model with an almost-constant mean level and no seasonality, you can also use simple exponential smoothing to make short-term forecasts. Exponential smoothing is named such because it "smooths" the weight of our past values as it estimate the value at each point in the time series. The degree of smoothing is controlled by the parameter α, which takes a value of range 0 to 1. When alpha is equal to 1, only the most recent last observation is used as it tries to estimate the next successive value in the series.

The following data `precip1.dat`[8] contains a series of total annual rainfall in inches for London, from 1813-1912 (original data from Hipel and McLeod, 1994).

```
rain <- scan("data_input/precip1.dat", skip=1)
rain_ts <- ts(rain, start=c(1813))
plot(rain_ts)
```

[8]Duke University, Statistical forecasting: notes on regression and time series analysis



Observe that the mean hovers around 25 inches rather constantly throughout the 100 years and the random fluctuations also seem to be roughly constant in size, so it's probably appropriate to describe the data using an additive model. We've seen how a SMA would have worked on the above time series but here we'll see if a simple exponential smoothing algorithm is up to the task.

Comparison to SMA

Conceptually, where a single moving average assigns equal weight to n number of past observations, exponential smoothing assigns *exponentially decreasing weights* as the observation gets older and commonly consider the full data. Through the assignment of weights, we can then set our forecast to be more or less influenced by recent observations as compared to older observations.

In most cases an alpha parameter smaller than 0.40 is often effective. However, one may perform a grid search of the parameter space, with = 0.1 to = 0.9, with increments of 0.1. Then the best alpha has the smallest Mean Absolute Error (MA Error).

Comparison to the naive method & average method

Using the naive method, all forecasts for the future are equal to the last observed value of the series: $\hat{y}_{T+h|T} = y_T$

Because the naive method assumes that the most current observation is the only important one and all previous observations provide no information for the future, this can be thought of as a weighted average where all the weight is given to the last observation.

Using the average method, all observations are assumed to have equal important and given equal weight when generating forecasts.

We often want something between these two extremes, and one approach would be to assign heavier weights to more recent observations than to the distant ones. Using this approach, weights decrease exponentially leading to the smallest weights being assigned to the oldest observations:
$\hat{y}_{T+1|T} = \alpha y_T + \alpha(1-\alpha)y_{T-1} + \alpha(1-\alpha)^2 y_{T-2} + ...,$

The one-step-ahead forecast for time *T+1* is a weighted average of all the observations in the series $y_1 ... y_T$. The rate at which the weights decrease is controlled by the parameter alpha.

For any α between 0 and 1 (not including 0), the weights attached to the observations decrease exponentially as we go back in time, hence the name "exponential smoothing". If α is small (i.e, close to 0), more weight is given to observations from the more distant past. If α is large (i.e, close to 1), more weight is given to the more recent observations. At the extreme case where $\alpha = 1$, $\hat{y}_{T+1|T} = y_T$ and forecasts are equal to the naive forecasts. In other words: - $\alpha = 1$: past values have no influence over forecasts (under-smoothing)
- $\alpha \approx 0$: past values have equal influence over forecasts (over-smoothing)

Recall that the formula for the simple exponential smoothing: $\hat{y}_{T+1|T} = \alpha y_T + \alpha(1-\alpha)y_{T-1} + \alpha(1-\alpha)^2 y_{T-2} + ...,$

SES as an adaptive learning process

If we vectorize the original equation, and use L_t to denote the "level" at time t we can represent the level updating equation as: $L_t = \alpha Y_t + (1-a)L_{t-1} = aY_t + L_{t-1} - \alpha L_{t-1}$

Which is equivalent to:
$F_{t+1} = L_t = L_{t-1} + \alpha(Y_t - L_{t-1})$
$F_{t+1} = L_t = F_t + \alpha(Y_t - F_t)$
$F_{t+1} = L_t = F_t + \alpha * e_t$

This last form is simple and useful: it means that to forecast the next time period we update the previous forecast by an amount that depends on the error of the previous forecast. Another great benefit is that while our forecasts use all previous values, we really only need to store the most recent forecast and its error.

Let's see how exponential smoothing works

The function I use here is `ets()`, which accepts an optional `model` specification and an alpha value of 0.2. The `model` is specified through a three-character string, where:
- First letter denotes the error type ("A", "M" or "Z")
- Second letter denotes the trend type ("N"",A",M" or "Z")
- Third letter denotes the seasonality ("N"",A",M" or "Z")
- Where N indicates "none", "A" indicates additive, "M" indicates multiplicative and "Z" for automatic selection

Let's create an exponential smoothing that only consider the error ignoring any trend and seasonality in our time series:

```
library(forecast)
co2_ets <- ets(co2_ts, model="ANN", alpha=0.2)
co2_ets$fitted
```

```
## Time Series:
## Start = 1970
## End = 2012
## Frequency = 1
##  [1] 0.4290483 0.4056290 0.3906275 0.3841036 0.3863769 0.3895329 0.3941873
##  [8] 0.4075976 0.4461377 0.4904662 0.5244021 0.5480516 0.5711227 0.5933430
## [15] 0.6074939 0.6248755 0.6468540 0.6620667 0.6733363 0.6897109 0.6987300
```

```
## [22] 0.7238523 0.7737895 0.8348065 0.8968911 0.9458386 0.9850864 1.0414677
## [29] 1.1079499 1.0907303 1.1045827 1.1327145 1.1811353 1.2269550 1.2688449
## [36] 1.3170555 1.3553406 1.3845878 1.4300413 1.4968121 1.5704827 1.6099678
## [43] 1.7546913
```

```
co2_ets$residual
```

```
## Time Series:
## Start = 1970
## End = 2012
## Frequency = 1
##  [1] -0.11709643 -0.07500754 -0.03261950  0.01136666  0.01577974
##  [6]  0.02327215  0.06705166  0.19270012  0.22164253  0.16967954
## [11]  0.11824747  0.11535555  0.11110153  0.07075465  0.08690815
## [16]  0.10989249  0.07606323  0.05634783  0.08187320  0.04509546
## [21]  0.12561166  0.24968568  0.30508525  0.31042305  0.24473744
## [26]  0.19623883  0.28190663  0.33241119 -0.08609825  0.06926220
## [31]  0.14065889  0.24210384  0.22909853  0.20944949  0.24105334
## [36]  0.19142506  0.14623622  0.22726760  0.33385382  0.36835332
## [41]  0.19742513  0.72361760  0.65422886
```

Observe that we can confirm the formula $F_{t+1} = F_t + \alpha E_t$:

```
co2_ets$fitted[1] + 0.2 * co2_ets$residual[1]
```

```
## [1] 0.405629
```

```
co2_ets$fitted[2] + 0.2 * co2_ets$residual[2]
```

```
## [1] 0.3906275
```

```
co2_ets$fitted[3] + 0.2 * co2_ets$residual[3]
```

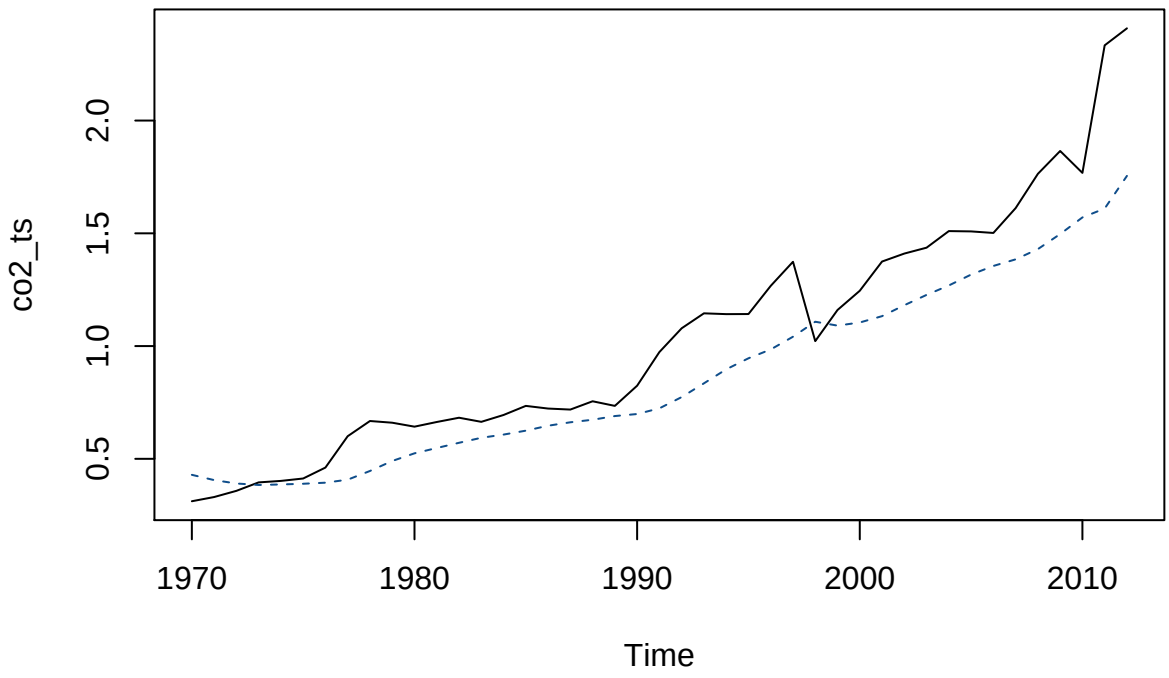
```
## [1] 0.3841036
```

```
co2_ets$fitted[2:4]
```

```
## [1] 0.4056290 0.3906275 0.3841036
```

Let's plot the time series along with our forecast to see whether the above model adequately describes the variation in our time series (it wouldn't because we use "N" to denote the trend component, hence the model has not considered a trend component yet):

```
plot(co2_ts)
lines(co2_ets$fitted, lty=2, col="dodgerblue4")
```



It seems like our earlier suspicion was right - it didn't seem to adequately "capture" the trend in our data.

Discussion: It looks like `model="ANN"` isn't the right model specification for our time series. There exist an additive trend component that wasn't considered. How would you change the following code?

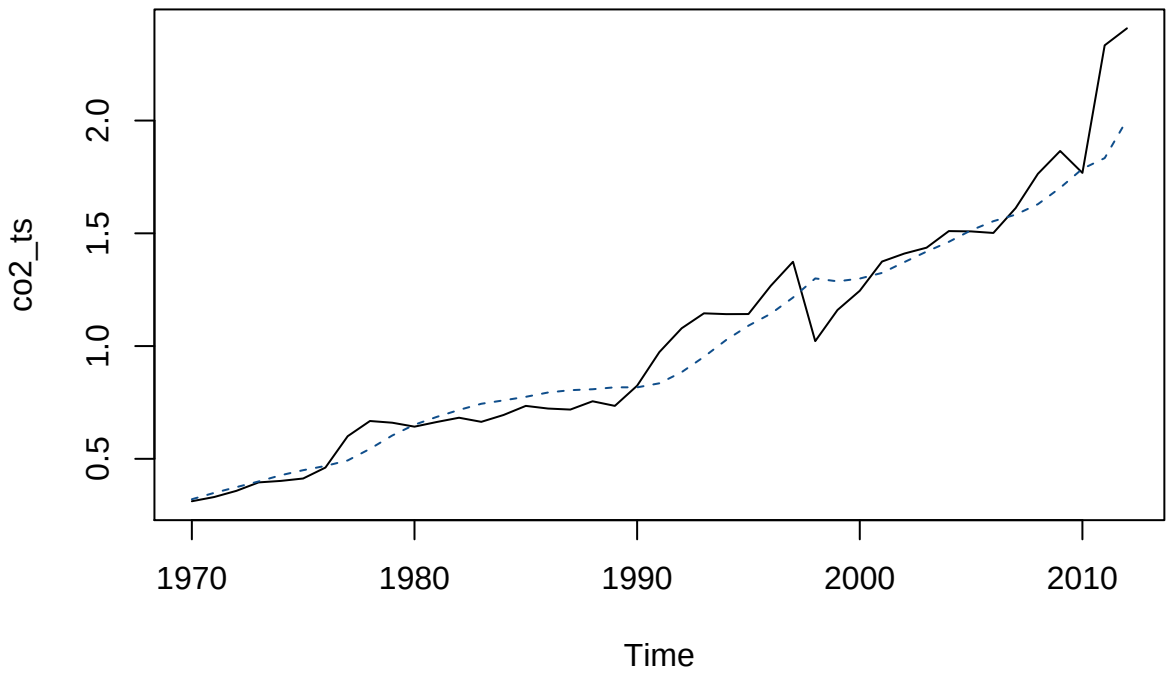
```
co2_ets<- ets(____, model="_____", alpha=0.2)
```

To see if your code works, plot the time series along with your forecasting line and compare to the original plot (the one that systematically under-predicts the value of CO2 in Indonesia). Is your model an improvement?

```
### plot your new exponential smoothing model below and compare to the plot we created above
```

Reference answer: The forecast under-predicted most of the value because the trend component in our time series wasn't considered. Let's change the `model` specification by including an additive trend type:

```
co2_ettrend <- ets(co2_ts, model="AAN", alpha=0.2)
plot(co2_ts)
lines(co2_ettrend$fitted, lty=2, col="dodgerblue4")
```



Another way we can fit a simple exponential smoothing predictive model is by using R's built-in `HoltWinters()` function, setting `FALSE` for the parameters `beta` and `gamma` in the function call. The `beta` and `gamma` parameters are used for Holt's exponential smoothing, or Holt-Winters exponential smoothing, which we will discuss in great details later in later sections of the coursebook. For now, let's call `HoltWinters()` on our co2 emission time series.

The `HoltWinters()` function will return a list variable that contains several named elements, which we will discuss:

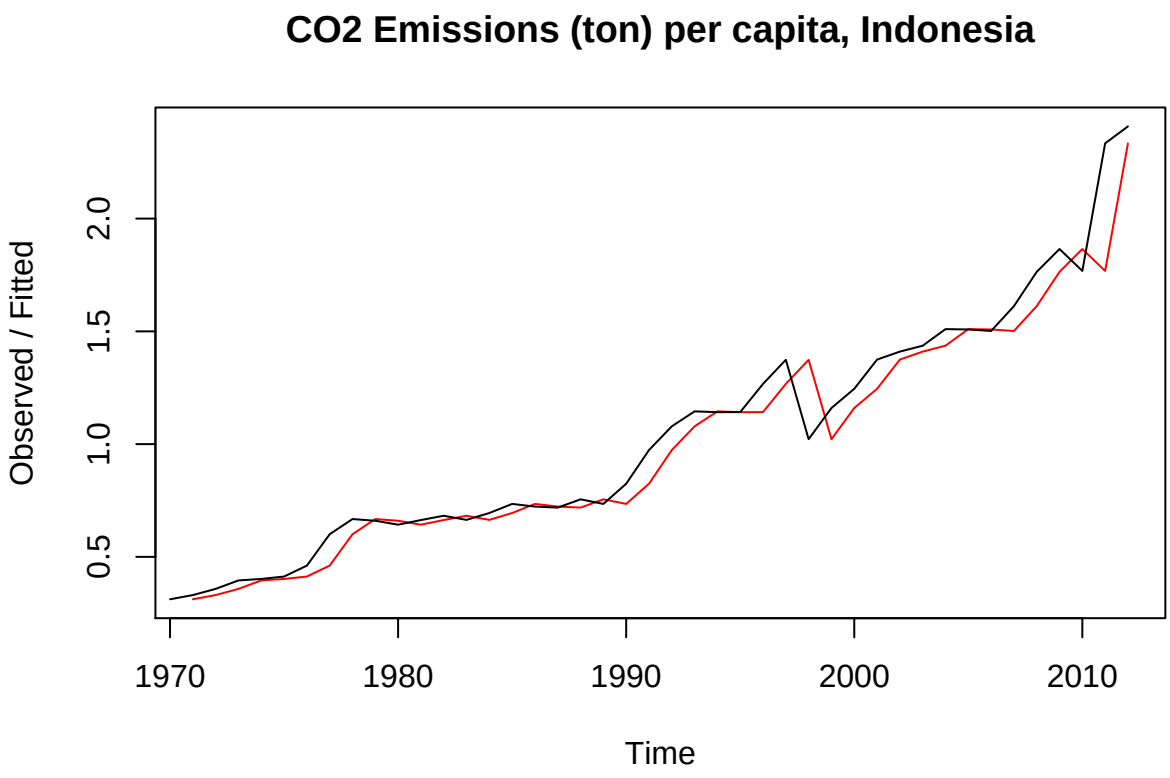
```
co2_hw <- HoltWinters(co2_ts, beta=F, gamma=F)
co2_hw
```

```
## Holt-Winters exponential smoothing without trend and without seasonal component.
##
## Call:
## HoltWinters(x = co2_ts, beta = F, gamma = F)
##
## Smoothing parameters:
##  alpha: 0.9999546
##  beta : FALSE
##  gamma: FALSE
##
## Coefficients:
##       [,1]
## a 2.408917
```

Notice that as we set `beta=FALSE` the time series is assumed to have no trend; `gamma=FALSE` indicates that the time series should be smoothed without consideration of a seasonal component.

Since there is a prior knowledge that our time series does in fact contain a trend component, if you remove `beta=F` from the function call and run the plot below again, you'll see that we obtain a slightly better estimate. For now, understand that the `beta` represents the coefficient for the trend component in our time series - we'll look at this with greater attention later.

```
plot(co2_hw, main="CO2 Emissions (ton) per capita, Indonesia")
```



I've created another exponential smoothing model on `rain_ts`, and store it under the variable `rain_hw`. Print `rain_hw` and compare it to the output from printing `co2_hw`.

```
rain_hw <- HoltWinters(rain_ts, beta=F, gamma=F)
```

Discussion: Compare the output and pay special attention to the estimated coefficient of the alpha parameter. Recall from earlier sections (line ~470) that:

> If α is small (i.e, close to 0), more weight is given to observations from the more distant past. If α is large (i.e, close to 1), more weight is given to the more recent observations.

Between the CO2 emission and rainfall time series, which has a alpha that is closer to 0? An alpha close to 0 indicates that the forecasts are based on both recent observations and less recent observations while an alpha close to 1 indicates that more weight is given to the more recent observations.

Reference answer:

```
co2_hw$alpha
```

```
## [1] 0.9999546
```

```
rain_hw$alpha
```

```
## [1] 0.02412151
```

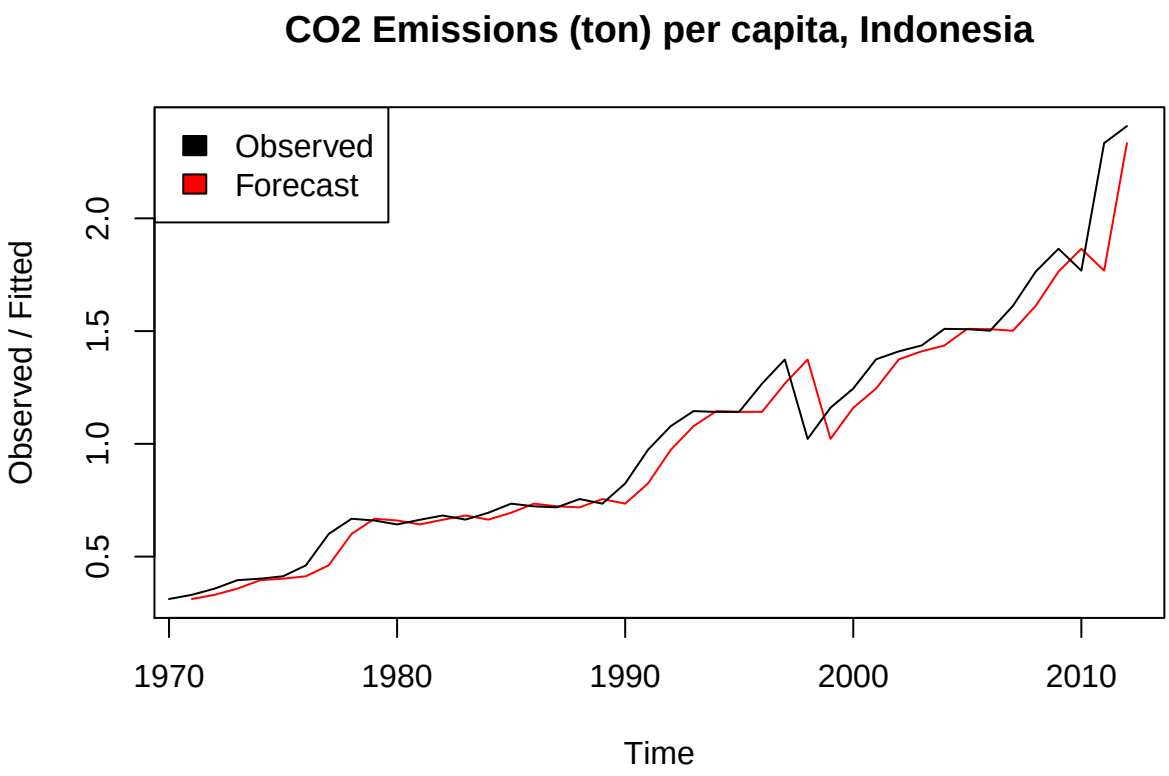
By default, `HoltWinters()` performs its smoothing for the same time period covered by our original time series. Because our original time series for Indonesia's CO2 emissions is from 1970 to 2012, so the forecasts are also for 1970 to 2012. The forecasts made by `HoltWinters()` are stored in a named element called "fitted":

```
head(co2_hw$fitted, 10)
```

```
## Time Series:
## Start = 1971
## End = 1980
## Frequency = 1
##           xhat     level
## 1971 0.3119519 0.3119519
## 1972 0.3306206 0.3306206
## 1973 0.3580067 0.3580067
## 1974 0.3954686 0.3954686
## 1975 0.4021564 0.4021564
## 1976 0.4128045 0.4128045
## 1977 0.4612368 0.4612368
## 1978 0.6002915 0.6002915
## 1979 0.6677771 0.6677771
## 1980 0.6601461 0.6601461
```

As we've seen previously, we can also plot the original time series against the forecasts:

```
plot(co2_hw, main="CO2 Emissions (ton) per capita, Indonesia")
legend("topleft", legend = c("Observed", "Forecast"), fill=1:2)
```



The forecasts is represented by the red line, and we see that it is smoother than the time series of the original

data. As a measure of the accuracy of the forecasts, we can calculate the **sum of squared errors** for the in-sample forecast errors, which is stored in the named element `SSE`:

```
co2_hw$SSE
```

```
## [1] 0.6627841
```

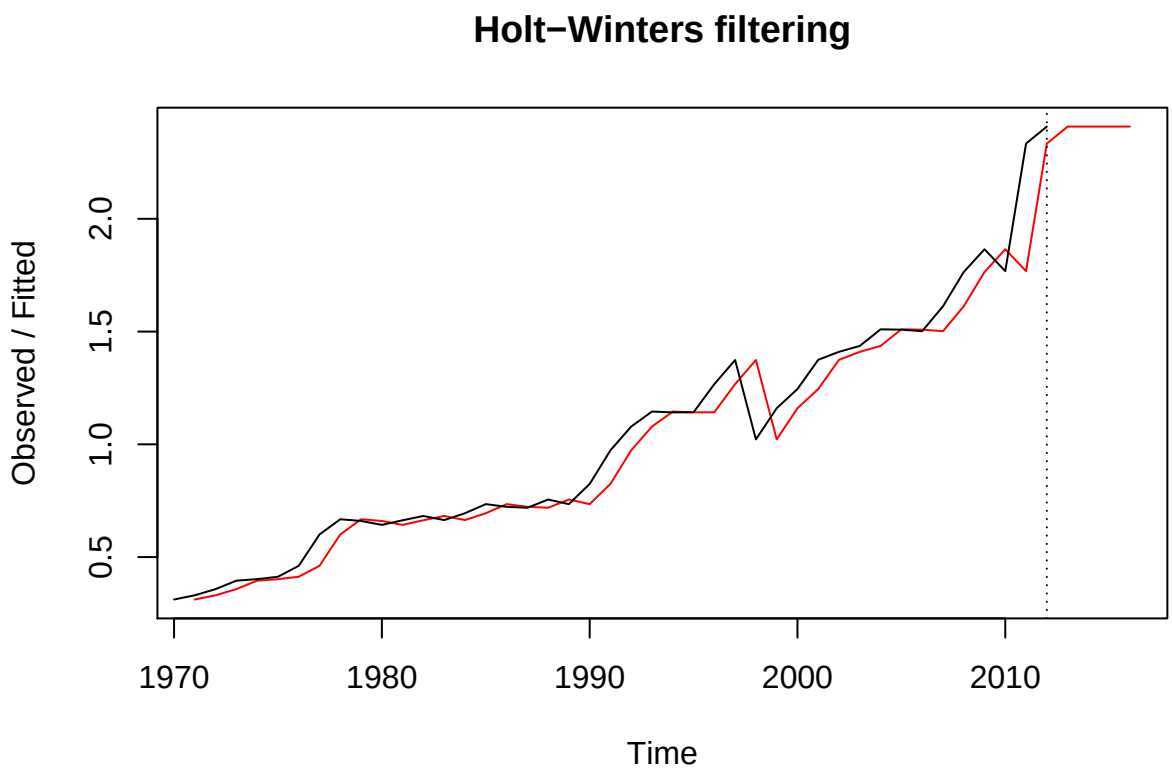
So we mentioned earlier that by default `HoltWinters()` makes forecasts for the time period covered by the original data, but to make forecasts for further time points we can use the `forecast()` function in the `forecast` package, or using the generic built-in `predict()` function.

```
co2_fut <- predict(co2_hw, n.ahead=4)
co2_fut
```

```
## Time Series:
## Start = 2013
## End = 2016
## Frequency = 1
##           fit
## [1,] 2.408917
## [2,] 2.408917
## [3,] 2.408917
## [4,] 2.408917
```

We can then use `plot()`, which accepts an additional `predicted.values` parameter to produce a chart of the original time series along with the predicted values (`n.ahead=4`):

```
plot(co2_hw, co2_fut)
```



Alternatively, using the `forecast` package (the point forecast we'll get will be exactly the same as the generic `predict()` method above):

```
library(forecast)
# forecast on 8 more further points (1913-1920, 8 more years)
co2_hwf <- forecast(co2_hw, h=4)
co2_hwf
```

```
##      Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
## 2013       2.408917 2.259399 2.558434 2.180250 2.637584
## 2014       2.408917 2.197472 2.620361 2.085540 2.732293
## 2015       2.408917 2.149953 2.667881 2.012866 2.804968
## 2016       2.408917 2.109892 2.707941 1.951598 2.866235
```

Recall that earlier in the lesson, I mentioned that a simple exponential smoothing without consideration of any trend or seasonality is only really suitable for a time series that (no surprises) have no trend or seasonality! Since the forecast is "told" that there is no trend and seasonal component, then it follows that the prediction of future values follows a "flat" forecast function.

While unnecessary, we can verify that the future forecasted value does in fact follow the same exponential smoothing formula we've learned about in the last few sections:

```
alp <- co2_hwf$model$alpha
round(
  alp * co2_hw$x[43] +
  alp * (1-alp) * co2_hw$x[42]
,4)
```

```
## [1] 2.4089
```

That turns out to be 2.4089, so the point forecast computed using the `forecast()` model is the same as our manual calculation using the exponential smoothing formula.

The forecast error, or sum of squared errors, are stored in the named element `residuals`:

```
round(co2_hwf$residuals,3)
```

```
## Time Series:
## Start = 1970
## End = 2012
## Frequency = 1
##  [1]     NA  0.019  0.027  0.037  0.007  0.011  0.048  0.139  0.067 -0.008
## [11] -0.017  0.021  0.019 -0.018  0.030  0.040 -0.012 -0.005  0.037 -0.020
## [21]  0.090  0.149  0.105  0.066 -0.004  0.000  0.125  0.107 -0.352  0.138
## [31]  0.085  0.130  0.035  0.026  0.073 -0.001 -0.007  0.110  0.152  0.101
## [41] -0.097  0.566  0.075
```

Let's manually verify that the sum of squared error if in fact the value returned from `$SSE`:

```
sum(as.numeric(co2_hwf$residuals)^2, na.rm=T)
```

```
## [1] 0.6627841
```

```
co2_hw$SSE
```

```
## [1] 0.6627841
```

We've learned, perhaps throughout the Machine Learning Specialization, that our forecast is rarely perfect and for the most part is estimated with some errors. We saw above that these errors can be accessed with `co2_hwf$residuals`, but if you want some confirmation of these values you could also have applied the

exponential smoothing formula manually and find the difference between these values and the corresponding observed values (actual):

```
round(co2_ts[2:8] - (co2_ts[1:7] * co2_hwf$model$alpha),3)
```

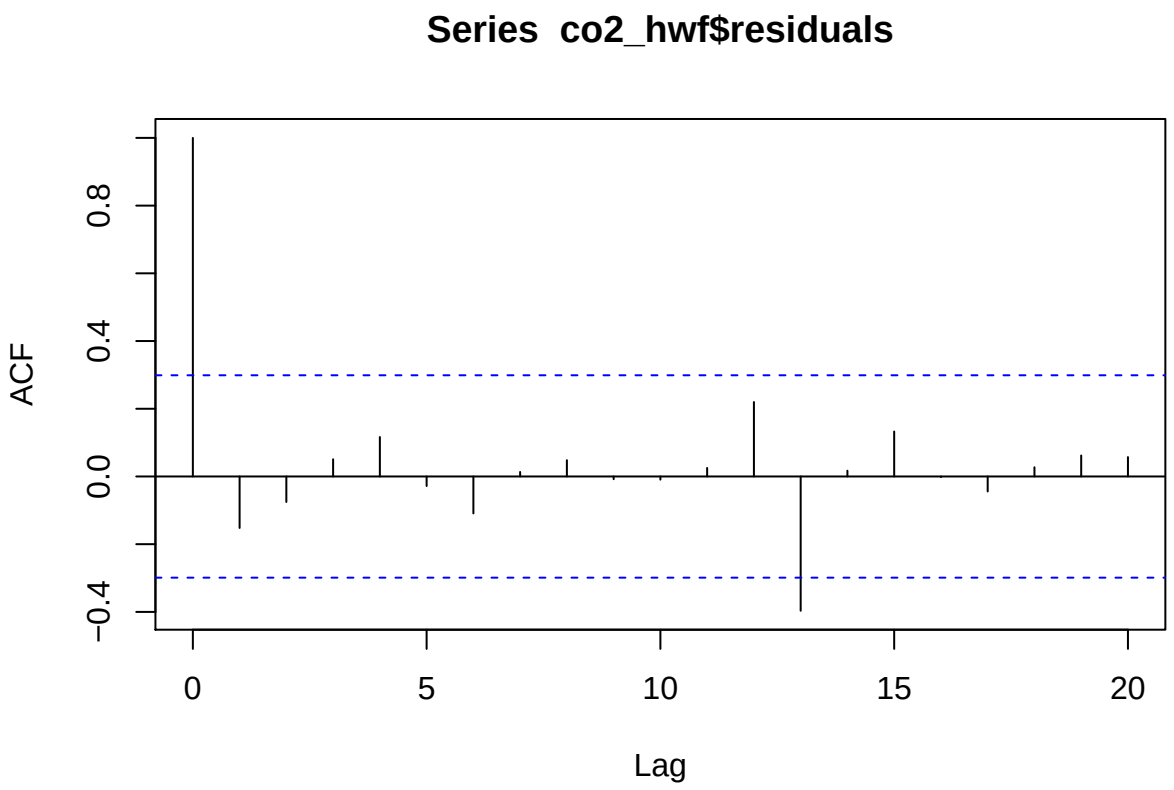
```
## [1] 0.019 0.027 0.037 0.007 0.011 0.048 0.139
```

If the model cannot be improved upon, there should be no correlations between forecast errors for successive predictions. This is very similar to the regression models you've learned about earlier in the Machine Learning Specialization.

If there are in fact correlations between forecast errors for successive predictions, it is likely that the simple exponential smoothing forecasts could be improved upon by another forecasting technique; Conceptually, it means there are some systematic components that could be mined or "extracted" from the series.

To see if this is the case, we can obtain a correlogram of the in-sample forecast errors for lags 1-20 using the `acf()` function and specifying the maximum lag that we want to look at using `acf.max` parameter:

```
# use na.action to tell the function that we should just omit the one NA value in our hwf
acf(co2_hwf$residuals, lag.max=20, na.action=na.pass)
```



`acf` is a built-in function that compute and (by default plots) the autocorrelation (default) or covariance of our univariate time series. We specified 20 to be the maximum lag at which to calculate the acf. If this is the first time you've across the word "lag" and autocorrelation, allow me to explain in greater details what they mean.

You've learned that we use correlation or covariance to compare two series of values and see how similar two time series are in their variation. Autocorrelation aims to describe the how similar the values of a time series is with other values within it.

Lag can be understood as the delay, or the gap, between values. For lag 0, the autocorrelation is comparing the entire time series with itself (and hence it is 1 in the ACF plot). For lag 1, the computation works by shifting the time series by 1 before comparing that time series with the shifted-by-1 series and it continues doing so for the whole length of the time series.

For a a timeseries that is "white noise", where it comprises of completely random values throughout the length of its series, we expect to see no correlation anywhere else apart from at lag 0. It is a useful technique that reveals at what "lag" in the time series can we expect correlation.

Imagine stock prices of a public company over the last many years, sampled daily. You will find that the time series will exhibit a relatively high correlation in lag=1 (because the values of one day to a great extent correlates with the price tomorrow), but decrease quickly at lag=2. Supposed the company file its annual returns every 12 months and we use a monthly time unit, then we should expect some correlation at lag=12 as well.

Going back to the ACF plot above, by default it plots the 95% confidence interval as dotted blue lines: Observe that all sample autocorrelations except the one at 13 fall well inside the 95% confidence bounds indicating the residuals appear to be random. By pure random chance (because of the 95% CI), you'd expect to see one lag exceeding the 95% threshold for every 20 values.

Understanding ACF and PACF

Unlike regular sampling data, time-series are ordered which means there may exist additional information (any temporal patterns) in our data that we can take advantage of. The autocorrelation function gives us the correlation between point separated by various time lags. The notation is ACF(n) where n is the number of time periods between points, such that ACF(n) gives us the correlation between points separated by n time periods.

```
acf(co2_hwf$residuals, lag.max=5, na.action=na.pass, plot=F)
```

```
##
## Autocorrelations of series 'co2_hwf$residuals', by lag
##
##      0      1      2      3      4      5
##  1.000 -0.152 -0.075  0.051  0.117 -0.028
```

Observe from the correlogram that ACF(0) = 1, because all data are perfectly correlated with themselves. Our ACF(1) = -0.152, indicating that the correlation between a point and its next point is -0.152. ACF(3) is 0.051, indicating the correlation between a point and the point two lag away (or two time steps ahead) is 0.051 and so on.

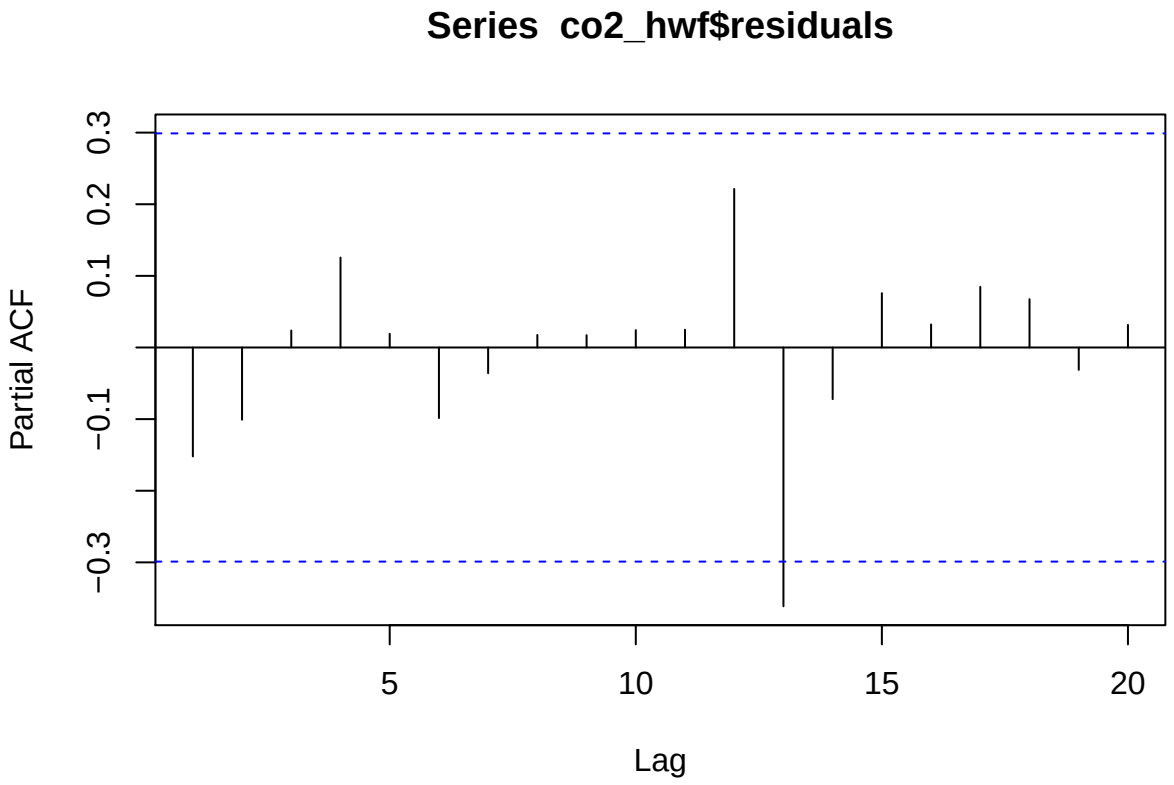
So we've learned earlier that the ACF tells you how correlated the points are with each other in a given time series, based on how many time steps they are separated by. Typically, we would expect the autocorrelation function to fall towards 0 as points become more separated because it is as a rule of thumb harder to forecast further into the future from a given point of time.

The ACF and its close relative, the *partial* autocorrelation function (PACF) are used in the Box-Jenkins/ARIMA modeling approach to determine how past and future data points are related in a time series. The PACF can be thought of as the correlation between two points *but* with the effect of the intervening correlations removed. This is important because in reality, if each data point is only directly correlated with the *next* data point and none other, that correlation may "dribble down" leading us to believe that there is a correlation between the current point and a point down the future. For example, T1 is directly correlated with T2, which is directly correlated with T3. This causes T1 to look like it is directly correlated with T3 - so the PACF is a measure we use so any intervening correlation with T2 is eliminated leading us to better discern true patterns of correlations.

In general, the "partial" correlation between two variables is the amount of correlation between them that

is not explained by their mutual correlations with a specified set of other variables. For example, if we are regressing variable "Y" on variables X1, X2 and X3, the partial correlation of X3 and Y can be computed as the square root of the reduction in variance that is achieved by adding X3 to the regression of Y on X1 and X2.

```
pacf(co2_hwf$residuals, lag.max=20, na.action=na.pass)
```



Ljung-box Test

Now going back to the earlier correlogram we plotted: notice that the autocorrelation at lag 13 is just touching the significance bounds - to test whether there is significant evidence for non-zero correlations at lag 1-20, we can perform a Ljung-Box test using the `Box.test()` function. The maximum lag that we want to look at is specified using the `lag` parameter. We will use the Ljung-Box test to determine whether our residuals are random:

```
Box.test(co2_hwf$residuals, lag=20, type="Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  co2_hwf$residuals
## X-squared = 17.942, df = 20, p-value = 0.5912
```

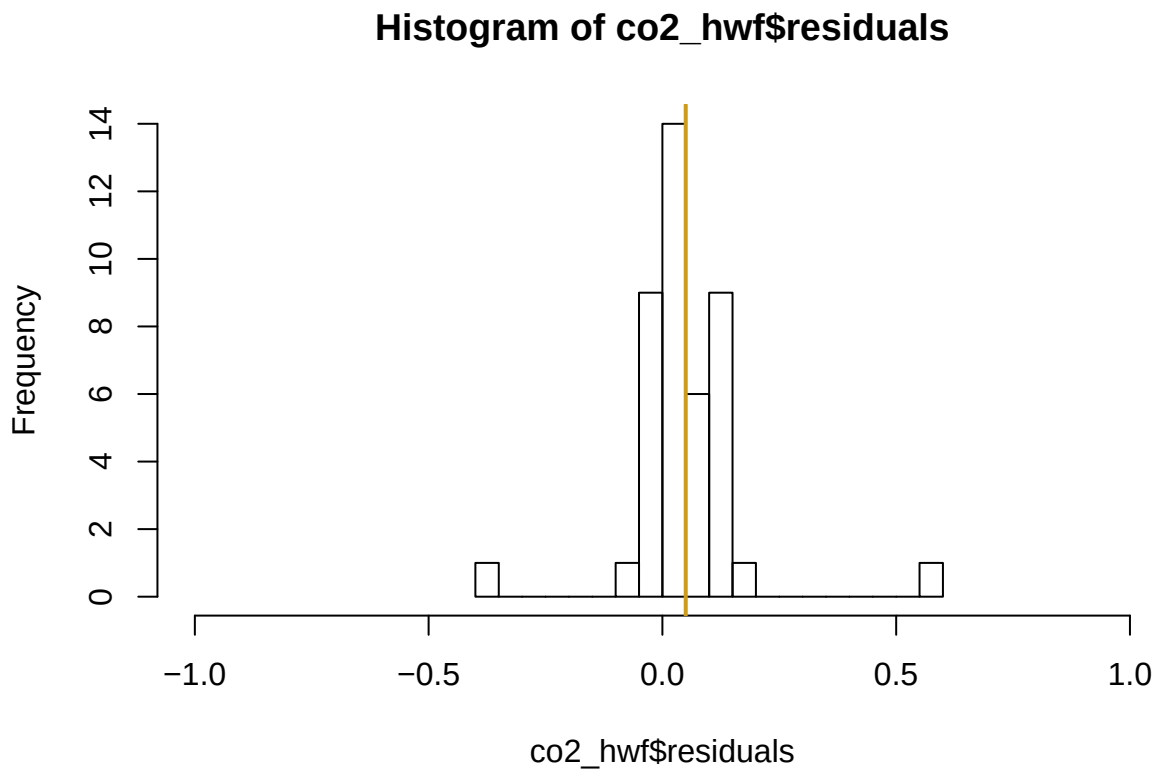
Here the Ljung-Box test statistic is 17.94, and the p-value is 0.6, so there is little evidence of non-zero autocorrelations in the in-sample forecast errors at lag 1-20. The Ljung-Box test statistic (X-squared) gets larger as the sample auto-correlations of the residuals get larger - without going too far beyond of this scope of the coursebook, here's the general idea:

For a *p-value* < 0.05: You can reject the null hypothesis (no auto-correlation) and know that you have less than 5% chance of making an error. This also mean we can assume that our values are showing dependence on each other. So really we want to see large values.

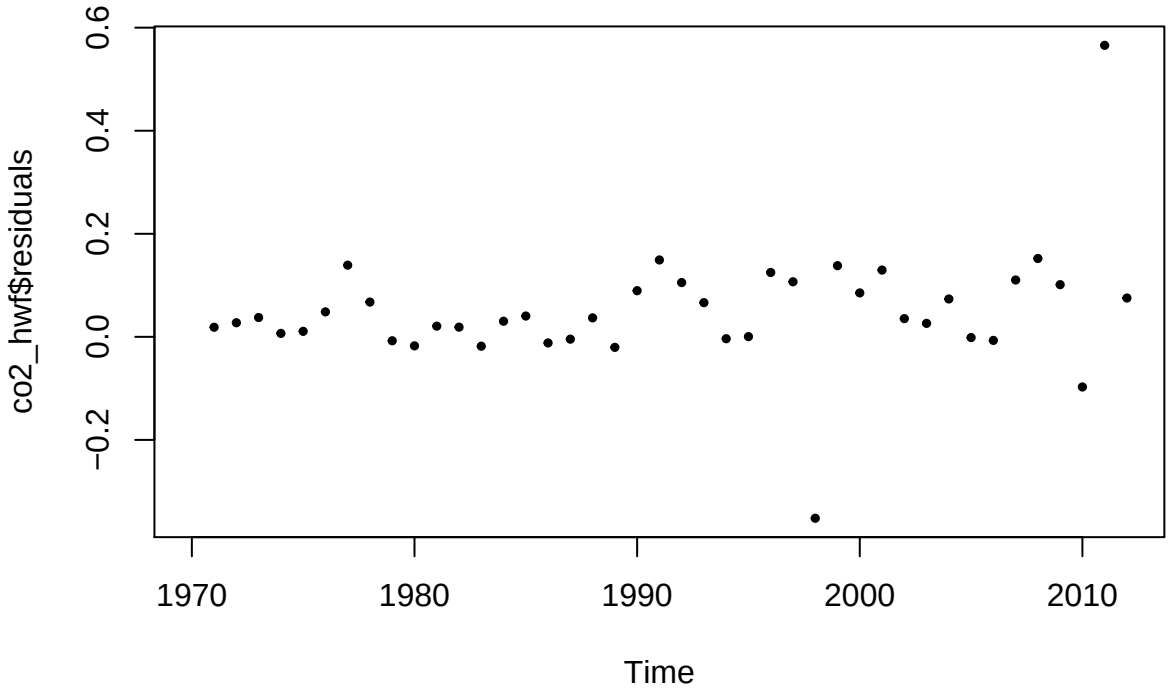
For a *p-value* > 0.05: Don't have enough evidence to reject the null hypothesis (no correlation), so we assume our values are random (no structure in the data) - or that there is no dependence on each other.

Lastly, to be sure that the predictive model cannot be furthed improved upon, let's check that the errors are approximately normally distributed with mean zero and constant variance:

```
hist(co2_hwf$residuals, breaks=20, xlim=c(-1,1))
abline(v=mean(co2_hwf$residuals, na.rm=T), col="goldenrod3", lwd=2)
```




```
plot(co2_hwf$residuals, type="p", pch=19, cex=0.5)
```



Observe that the in-sample forecast errors seem to have roughly constant variance over time, although the size of the fluctuations at around 1998 and 2010 may be slightly larger than that at later dates.

Before we conclude, recall again from our Ljung-Box test that there is little evidence of non-zero correlations in the in-sample forecast errors, and the distribution of our errors do also seem to approximate normal with mean ~0: both of which suggests that the simple exponential smoothing do provide an adequate predictive model for the Indonesia's CO2 emission forecast. We'd reason that the assumptions that the 95% prediction intervals were based upon are valid given: - No autocorrelations in the forecast errors
- Forecast error (residuals) are normally distributed

Holt's Exponential Smoothing

If we have a time series that can be described using an additive model with a trend component but no seasonality, we can use Holt's exponential smoothing to make short-term forecasts.

Holt's exponential smoothing (known also as Double Exponential Smoothing) estimates the level and slope at the current time point. Smoothing is controlled by:
- α: the estimate of the level at the current time point
- β: the estimate of the slope b of the trend component at the current time point t

Both alpha and beta take a value between 0 and 1 and values closer to 0 puts less weight on recent observations - just as it works in simple exponential smoothing. By introducing a second equation (beta) Holt's exponential smoothing uses the formulas:
- $s_1 = x_1$
- $b_1 = x_1 - x_0$

And for t $>$ 2, it is represented by:

- $s_t = \alpha x_t + (1-a)(s_{t-1} + b_{t-1})$
- $b_t = \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1}$

Recalling that α is our overall smoothing factor and β is our trend smoothing factor.

To forecast beyond x_t:
$F_{t+m} = s_t + mb_t$

Some suggestions to initialize b_0 is to use:
- $b_1 = x_1 - x_0$ - $b_1 = \frac{1}{3}[(x_1 - x_0) + (x_2 - x_1) + (x_3 - x_2)]$ - $b_1 = \frac{x_n - x_1}{n-1}$

Notice that in the first smoothing equation, we adjust s_t directly for the trend of the previous period (hence adding $s_{t-1} + b_{t-1}$). This helps to eliminate the lag and brings s_t to the appropriate base of the current value.

The second smoothing equation then updates the trend, which is expressed as the difference between the last two values $(s_t - s_{t-1})$. The equation is similar to the basic form of single smoothing, but here applied to the updating of the trend.

If you have followed along the earlier exercise, you've re-written the following code (removing `beta=FALSE` from the function call):

```
co2_holt <- HoltWinters(co2_ts, gamma=F)
co2_holt
```

```
## Holt-Winters exponential smoothing with trend and without seasonal component.
##
## Call:
## HoltWinters(x = co2_ts, gamma = F)
##
## Smoothing parameters:
##  alpha: 0.7468631
##  beta : 0.1106535
##  gamma: FALSE
##
## Coefficients:
##         [,1]
## a 2.38054022
## b 0.09548633
```

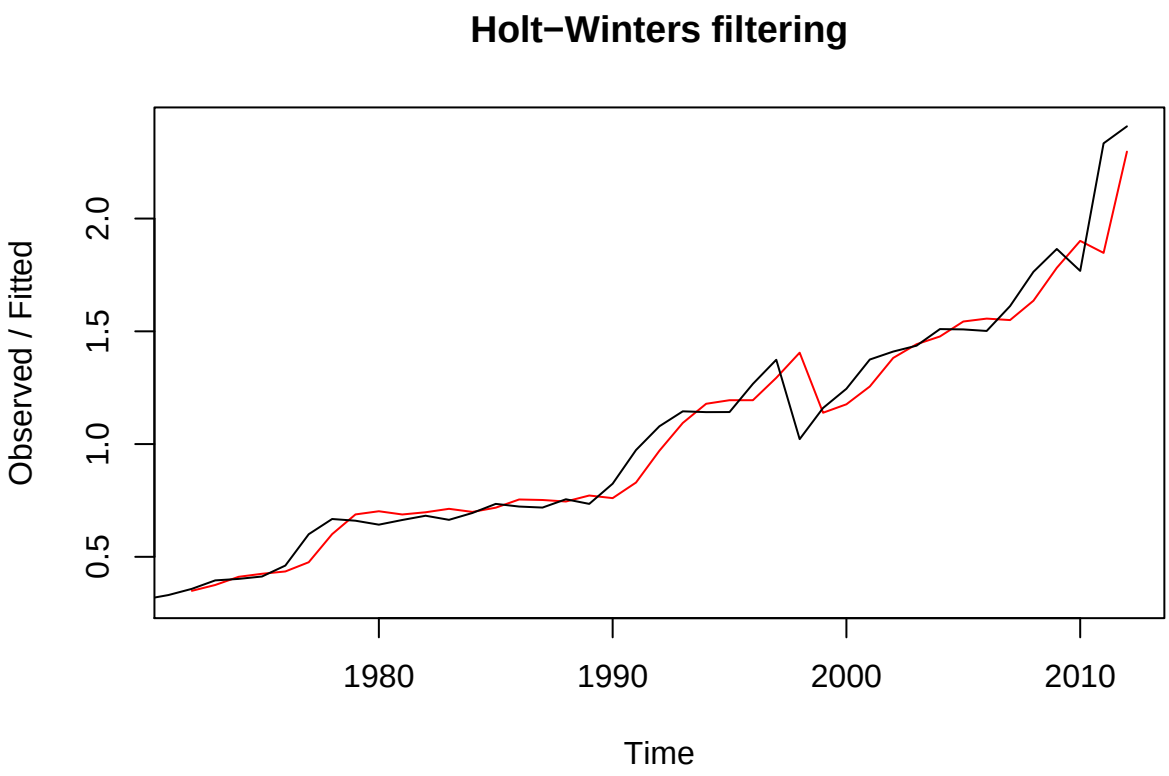
What you have done in fact, is to perform Holt's Exponential Smoothing on the time series by including a trend component - your estimates are also better, a result you can confirm visually by plotting the improved forecast or by comparing the sum of squared errors between the two model. Do one of the above:

```
# confirmation that the Holt's model is an improvement over co2_hw
```

Notice that the Holt's Exponential Smoothing model estimates the parameters for the trend component (beta) in our time series in addition to `alpha`, which we're familiar with.

The estimated value of alpha is 0.75 and of beta is 0.11. Alpha is high - indicating that the estimate of the overall smoothing are based more strongly on very recent observations in the time series; The trend smoothing (slope) is estimated from observations that are recent as well as more distant and that makes pretty good intuitive sense if we visually inspect the time series again:

```
plot(co2_holt)
```



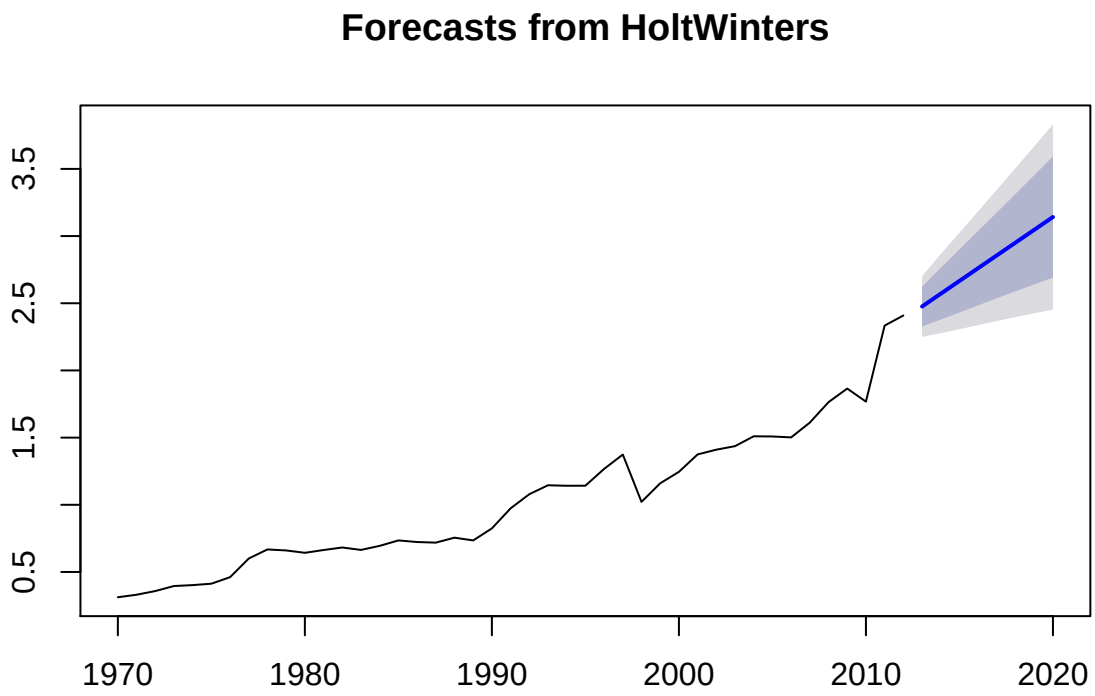
Observe that the in-sample forecasts agree pretty well with the observed values, although they tend to lag behind the observed values a bit. If we want, we can specify the initial values of the level and slope b of the trend component using the `l.start` and `b.start` arguments. It is common to set the initial value of the level to the first value in the time series (recall $s_1 = x_1$) and the initial value of the slope to the second value minus the first value (recall $b_1 = x_1 - x_0$), so fitting a predictive model with initial values would look like this:

```
co2_holt2 <- HoltWinters(co2_ts, gamma=F, l.start=co2_ts[1], b.start=co2_ts[2]-co2_ts[1])
head(co2_holt2$x)
```

```
## Time Series:
## Start = 1970
## End = 1975
## Frequency = 1
## [1] 0.3119519 0.3306215 0.3580080 0.3954703 0.4021567 0.4128050
```

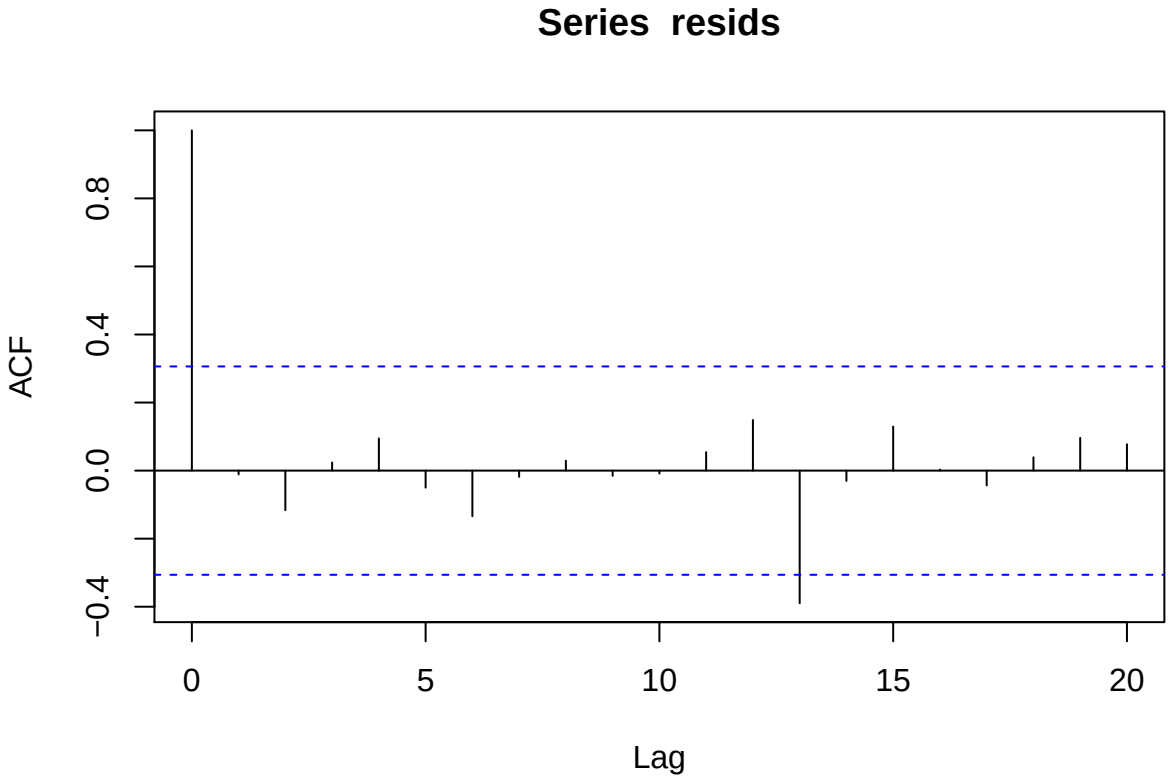
We can also make forecasts for future times not covered by the original time series using the `forecast.HoltWinters()` function. Let's make predictions for 2013 to 2020 (8 more data points), and plot them:

```
co2_hwf2 <- forecast(co2_holt2, h=8)
plot(co2_hwf2)
```



The plot gives us a forecast (blue line), a 80% prediction interval and a 95% prediction interval. Let's apply what we've learned earlier to see if the model could be improved upon by identifying whether the in-sample forecast errors show any non-zero autocorrelations at lags 1-20 using a correlogram plot:

```
# remove the first two residuals values as they are NA (no residuals)
resids <- co2_hwf2$residuals[3:43]
acf(resids, lag.max=20)
```



Observe from the above correlogram that the in-sample forecast errors at lag 13 exceeds the significance bounds. Again, we would expect one in 20 of the autocorrelations to exceed the 95% significance bounds by chance alone.

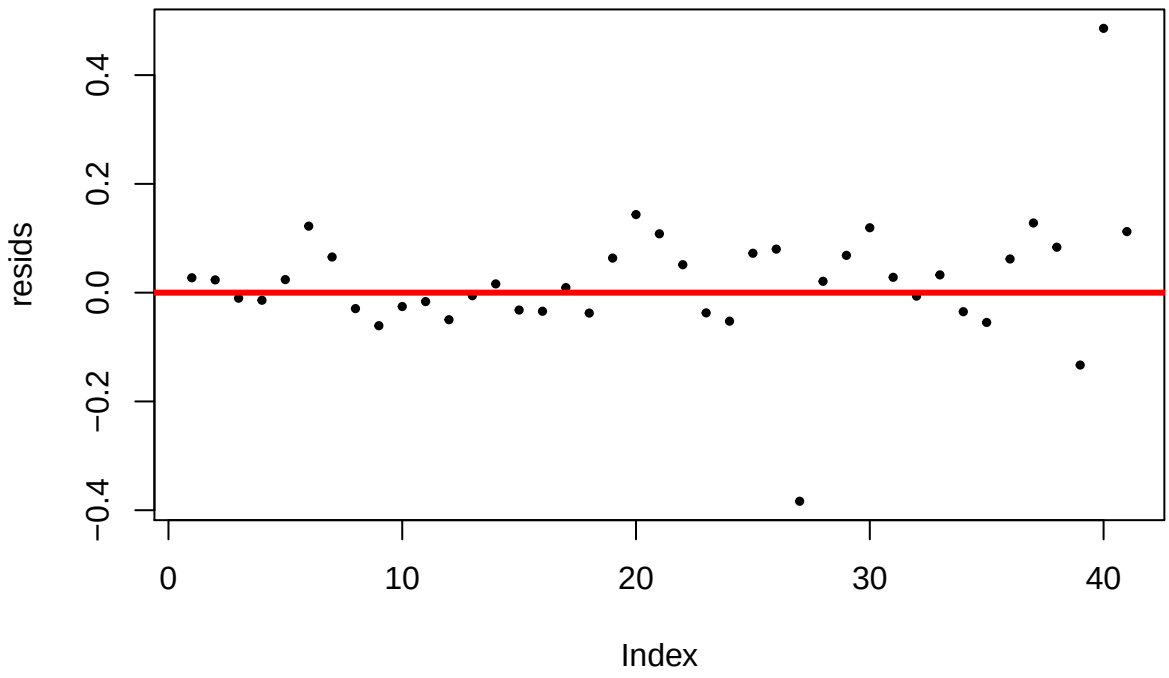
```
Box.test(resids, lag=20, type="Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  resids
## X-squared = 15.973, df = 20, p-value = 0.7183
```

Indeed, with the Ljung-Box test we obtain a p-value of 0.72, indicating that there is little evidence of a non-zero autocorrelations in the in-sample forecast errors at lag 1-20.

We could further check that the forecast errors have constant variance over time and are normally distributed with mean zero. We can use a time series plot and a histogram to inspect the distribution of the errors:

```
plot(resids, cex=0.5, pch=19)
abline(h=0, col="red", lwd=3)
```



Observe that the errors do seem to have roughly constant variance over time and that that our errors are normally distributed with mean zero and almost constant variance.

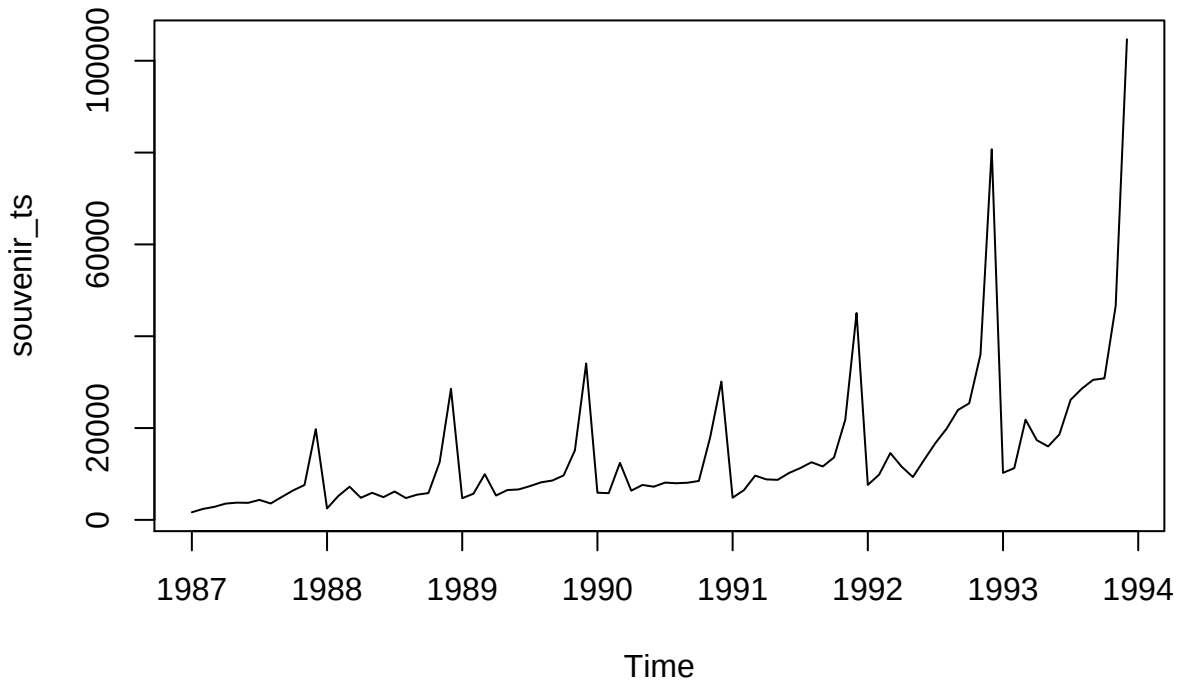
To summarize, our Ljung-Box test shows that there is little evidence of autocorrelations in the forecast errors, while the time plot and histogram of forecast errors show that it is plausible that the forecast errors are normally distributed with mean zero and constant variance. We can therefore conclude that our double exponential smoothing (Holt's exponential smoothing) provides an adequate predictive model for skirt diameters.

Holt-Winters Exponential Smoothing

If we have a time series that can be described using an additive model with increasing or decreasing trend and seasonality, we can use Holt-Winters exponential smoothing to make short-term forecasts.

Holt-Winters exponential smoothing **estimates the level, slope and seasonal component** at the current time point, and is controlled by α, β and γ for the estimates of the level, slope b of the trend component, and the seasonal component respectively. They all take values between 0 and 1, with 0 giving little weight to the most recent observations when making forecasts of future values. Let's read the time series of the log of monthly sales for the souvenir shop at a beach resort town in Queensland:

```
souvenir <- scan("data_input/fancy.dat", quiet = T)
souvenir_ts <- ts(souvenir, frequency = 12, start = c(1987,1))
plot(souvenir_ts)
```



It looks like it's a multiplicative time series - we'll apply a log-transformation on the series and apply Holt Winters on the resulting series:

```
souv_hw <- HoltWinters(log(souvenir_ts))
souv_hw
```

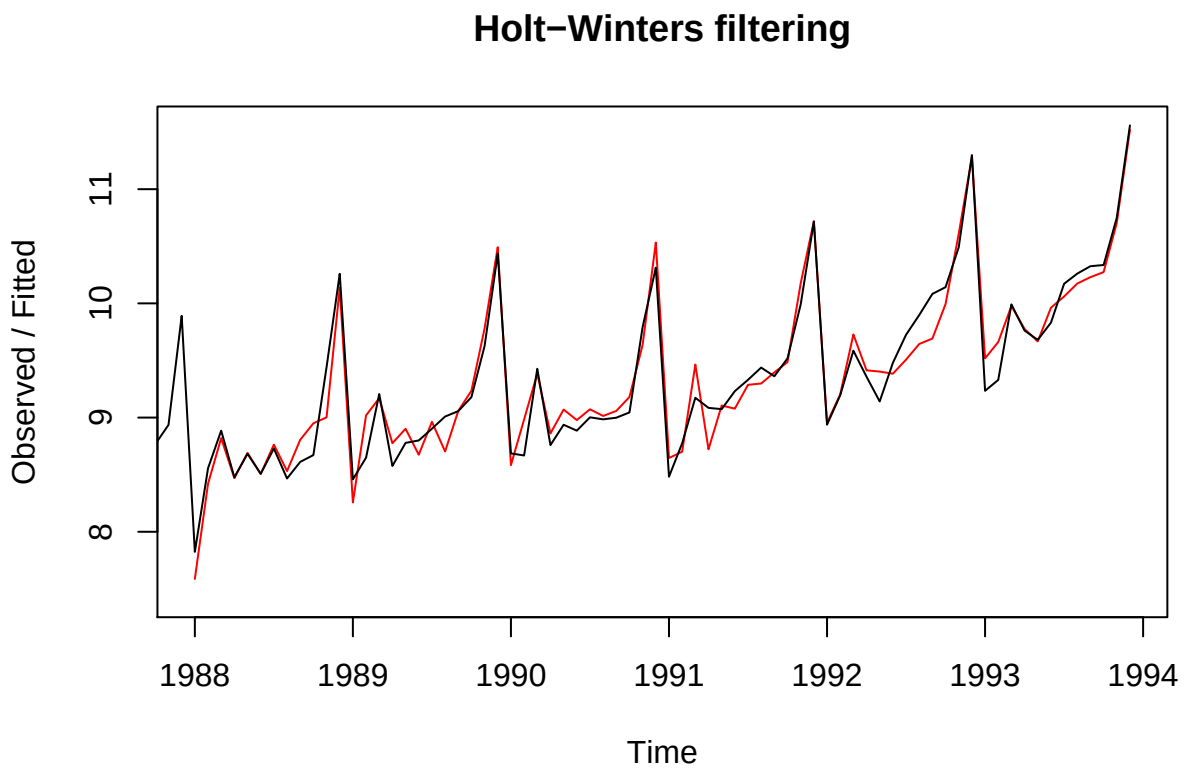
```
## Holt-Winters exponential smoothing with trend and additive seasonal component.
##
## Call:
## HoltWinters(x = log(souvenir_ts))
##
## Smoothing parameters:
##  alpha: 0.413418
##  beta : 0
##  gamma: 0.9561275
##
## Coefficients:
##           [,1]
## a   10.37661961
## b    0.02996319
## s1  -0.80952063
## s2  -0.60576477
## s3   0.01103238
## s4  -0.24160551
## s5  -0.35933517
## s6  -0.18076683
## s7   0.07788605
```

```
## s8   0.10147055
## s9   0.09649353
## s10  0.05197826
## s11  0.41793637
## s12  1.18088423
```

The estimated values of alpha (0.41) is relatively low, indicating that the estimate of the level at the current time point is based upon both recent observations and some observations in the more distant past.

The value of beta is 0.00 indicating that the estimate of the slope b of the trend component is not updated over the time series and instead is set equal to its initial value. This makes good intuitive sense as the level changes quite a bit over time but the slope b of the trend component remains roughly the same:

```
plot(souv_hw)
```



In contrast, the value of gamma (0.96) is high, indicating that the estimate of the seasonal component at the current time point is just based upon very recent observations.

From the forecast plot above we see that the Holt-Winters exponential method is very successful in predicting the seasonal peaks, which occur roughly in November every year.

Understanding Holt-Winters

Holt-Winters method, also known as the triple exponential smoothing was first suggested by Holt's student Peter Winters and introduces a third equation in addition to the two in Holt's method to account for seasonality. Generally, to produce forecasts (assuming additive seasonality), the formula would be:
Forecast = Most recent estimated level + trend + seasonality

And for multiplicative seasonality, we would instead use:
Forecast = Most recent estimated (level + trend) * seasonality

The equations for multiplicative seasonality are: - Overall smoothing: $s_t = \alpha \frac{x_t}{I_{t-L}} + (1-\alpha)(s_{t-1} + b_{t-1})$
- Trend smoothing: $b_t = \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1}$
- Seasonal smoothing: $I_t = \gamma \frac{x_t}{s_t} + (1-\gamma)I_{t-L}$
-Forecast: $F_{t+m} = (s_t + mb_t)I_{t-L+m}$

Where alpha, beta and gamma are estimated in a way that minimizes the MSE of the error, and: - x is the observation
- s is the smoothed observation
- b is the trend factor
- I is the seasonal index (sequence)
- F is the forecast at m periods ahead
- t is an index denoting a time period
- L is the number of periods in a season

Notice the I_{t-L} in our formula: this makes sense because I_t is the sequence of seasonal correction factors so a t value of 6 in a 4-period season (Q1 to Q4) would refer to the seasonal correction factor of the second period(Q2).

A complete season's data consists of L periods, and we need to estimate the trend factor from one period to the next. To accomplish this, it is advisable to have at least two complete seasons.

The general formula to estimate the initial trend b:
$b = \frac{1}{L}(\frac{x_{L+1}-x_1}{L} + \frac{x_{L+2}-x_2}{L} ... + \frac{x_{L+L}-x_L}{L})$

Now onto finding the initial values for the seasonal indices. To make things less abstract, assume we're working with data that consist of 5 years and 4 periods (4 Quarters per year):

- Step 1: Compute the averages of each of the 5 years ($A_1 ... A_5$)

- Step 2: Divide the observations by the appropriate yearly mean

```
library(htmlTable)
A <- rbind.data.frame(
  c("y<sub>1</sub>/A<sub>1</sub>","y<sub>5</sub>/A<sub>2</sub>","y<sub>9</sub>/A<sub>3</sub>","y<sub>13
  c("y<sub>2</sub>/A<sub>1</sub>","y<sub>6</sub>/A<sub>2</sub>","y<sub>10</sub>/A<sub>3</sub>","y<sub>14
  c("y<sub>3</sub>/A<sub>1</sub>","y<sub>7</sub>/A<sub>2</sub>","y<sub>11</sub>/A<sub>3</sub>","y<sub>15
  c("y<sub>4</sub>/A<sub>1</sub>","y<sub>8</sub>/A<sub>2</sub>","y<sub>12</sub>/A<sub>3</sub>","y<sub>16
)

htmlTable(A, css.cell="padding:0 2em;font-size:0.8em", align="c", header=c("Year 1","Year 2","Year 3","Y
```

Year 1

Year 2

Year 3

Year 4

Year 5

1

y1/A1

y5/A2

y9/A3

y13/A4

y17/A5

2

y2/A1

y6/A2

y10/A3

y14/A4

y18/A5

3

y3/A1

y7/A2

y11/A3

y15/A4

y19/A5

4

y4/A1

y8/A2

y12/A3

y16/A4

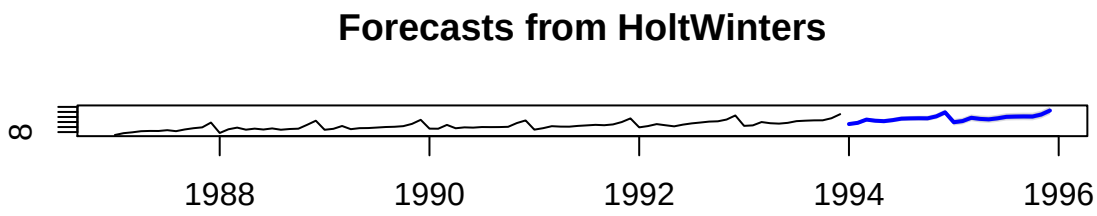
y20/A5

- Step 3: Now the seasonal indices are formed by computing the average of each row. Thus the seasonal indices are computed:
  $I_1 = (y_1/A_1 + y_5/A_2 + y_9/A_3 + y_{13}/A_4 + y_{17}/A_5)/5$
  (substitute I_1 for I_t)

Forecast using Holt-Winters

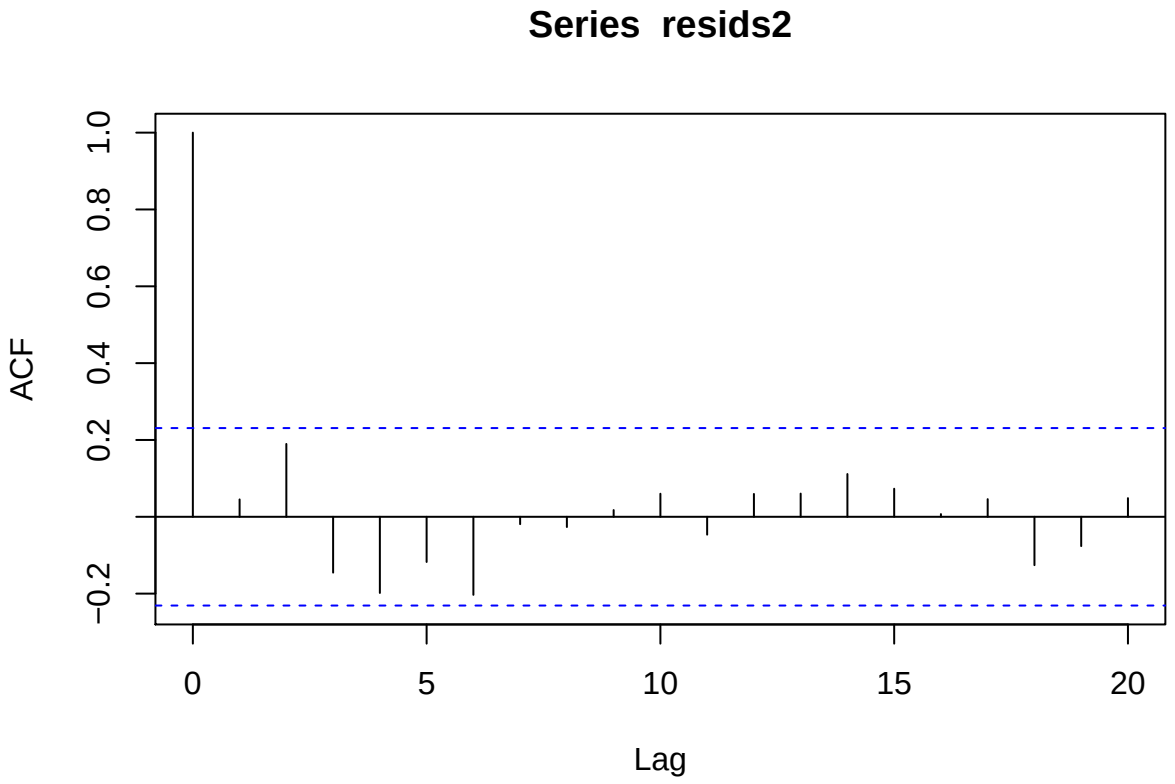
Making a forecast is simple enough in R using the `forecast` function. Let's try forecast with 24 more months (`h=24`):

```
souv_hwf <- forecast(souv_hw, h=24)
plot(souv_hwf)
```



We obtain the forecast (blue line) with a 80% (dark grey) and 95% (light grey) prediction intervals, respectively. We'll also check that the predictive model cannot be improved upon by observing if the in-sample forecast errors show non-zero autocorrelations at lag 1-20 and performing the Ljung-Box test:

```
resids2 <- souv_hwf$residuals[!is.na(souv_hwf$residuals)]
acf(resids2, lag.max=20)
```

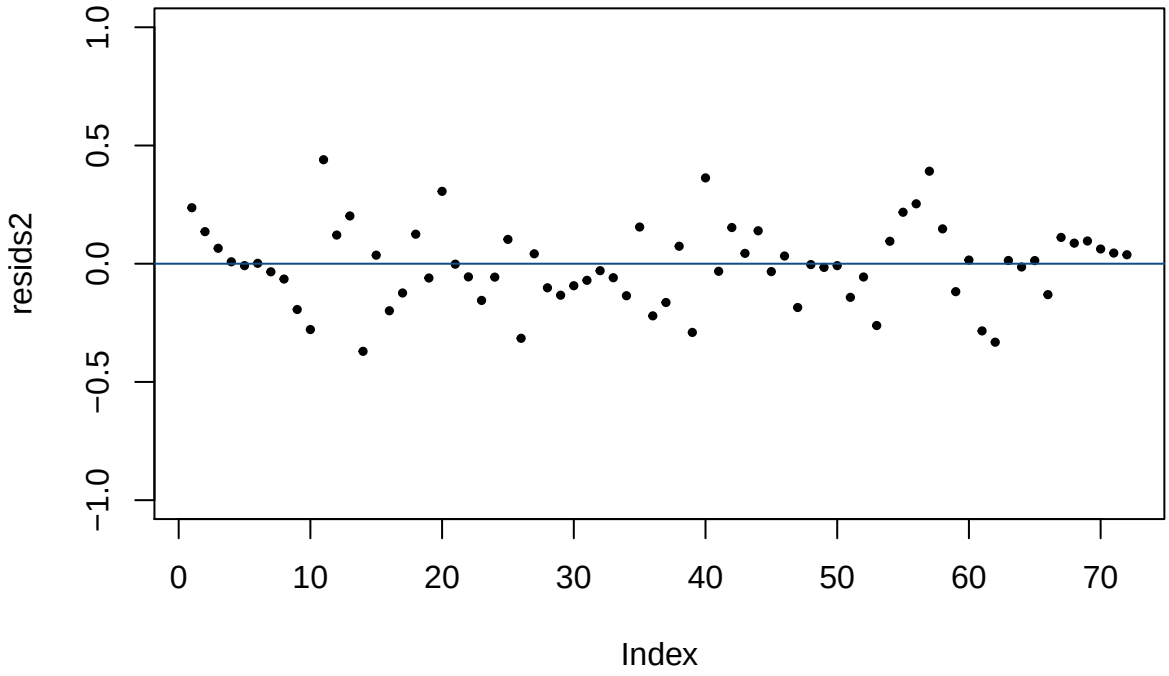



```
Box.test(resids2, lag=20, type="Ljung-Box")
```

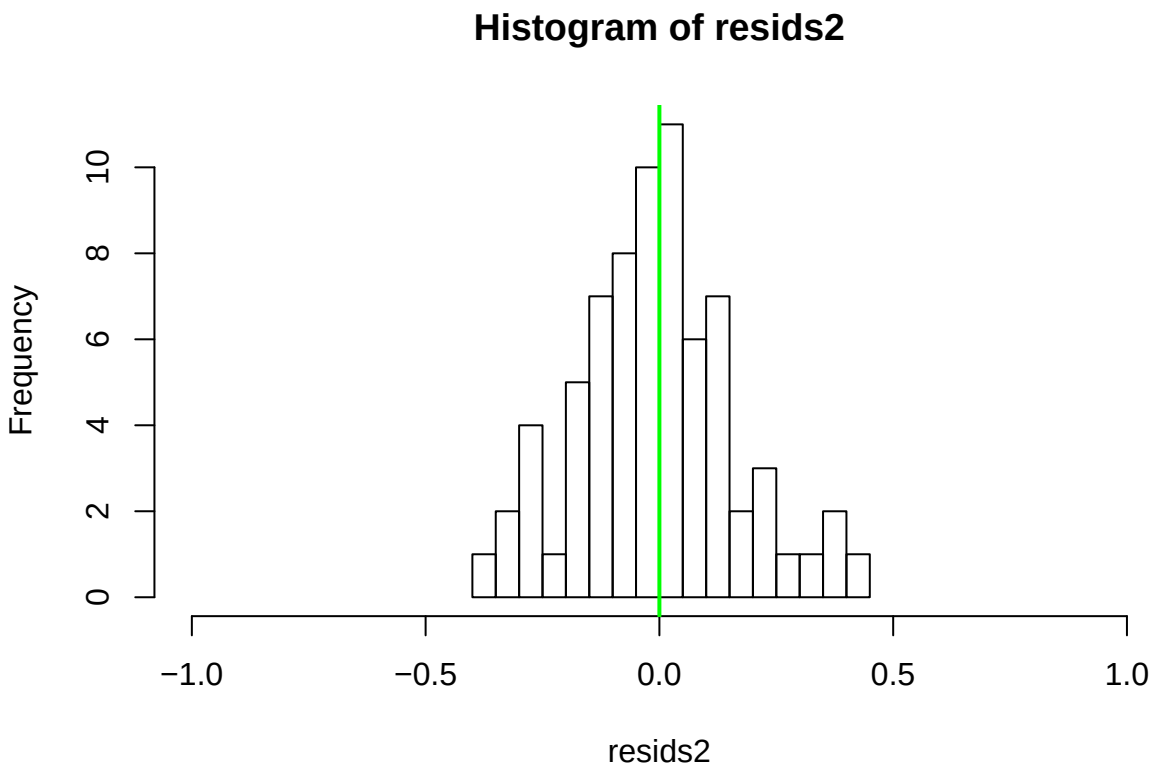
```
##
##  Box-Ljung test
##
## data:  resids2
## X-squared = 17.53, df = 20, p-value = 0.6183
```

The correlogram shows that the autocorrelations for the in-sample forecast errors do not exceed the significance bounds for lags 1-20. Furthermore, the p-value from our Ljung-Box test is 0.62, indicating that there is little evidence of a non-zero autocorrelations at lags 1-20. We'll also use a time plot to check that the forecast errors follow a normal distribution with mean zero and roughly constant variance:

```
plot(resids2, pch=19, cex=0.5, ylim=c(-1, 1))
abline(h=0, col="dodgerblue4")
```




```
hist(resids2, breaks=15, xlim=c(-1, 1))
abline(v=0, lwd=2, col="green")
```



It appears plausible from both our plots that the forecast errors do follow these assumptions – combined with the Ljung-Box test and acf plot, we may say that the Holt-Winters exponential smoothing appear to provide an adequate predictive model for the log of sales at the souvenir shop and cannot be improved upon.

ARIMA Models

Stationarity and Differencing

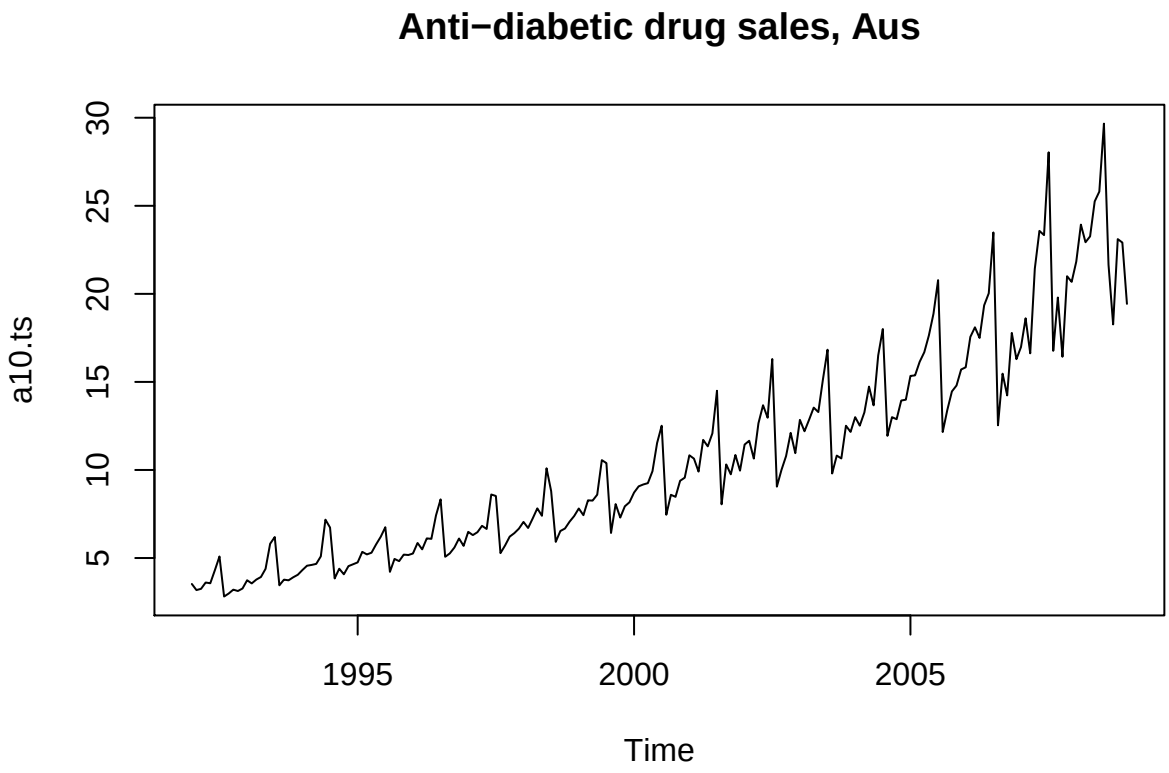
An important concept when dealing with time series is the idea of stationarity. A stationary time series is one whose **properties** do not depend on the time at which the series is observed, so a time series with trends or seasonality are not stationary - the trend and seasonality will affect the value of the time series at different times. A white noise series on the other hand is stationary.

Some cases can be less clear-cut: a time series with cyclic behavior (but not trend or seasonality) is stationary. This is because the cycles are not of fixed length so before we observe the series we cannot be sure where the peaks and troughs of the cycles will be. In general, a stationary time series will have no predictable patterns in the long term and their plots will show roughly constant variance.

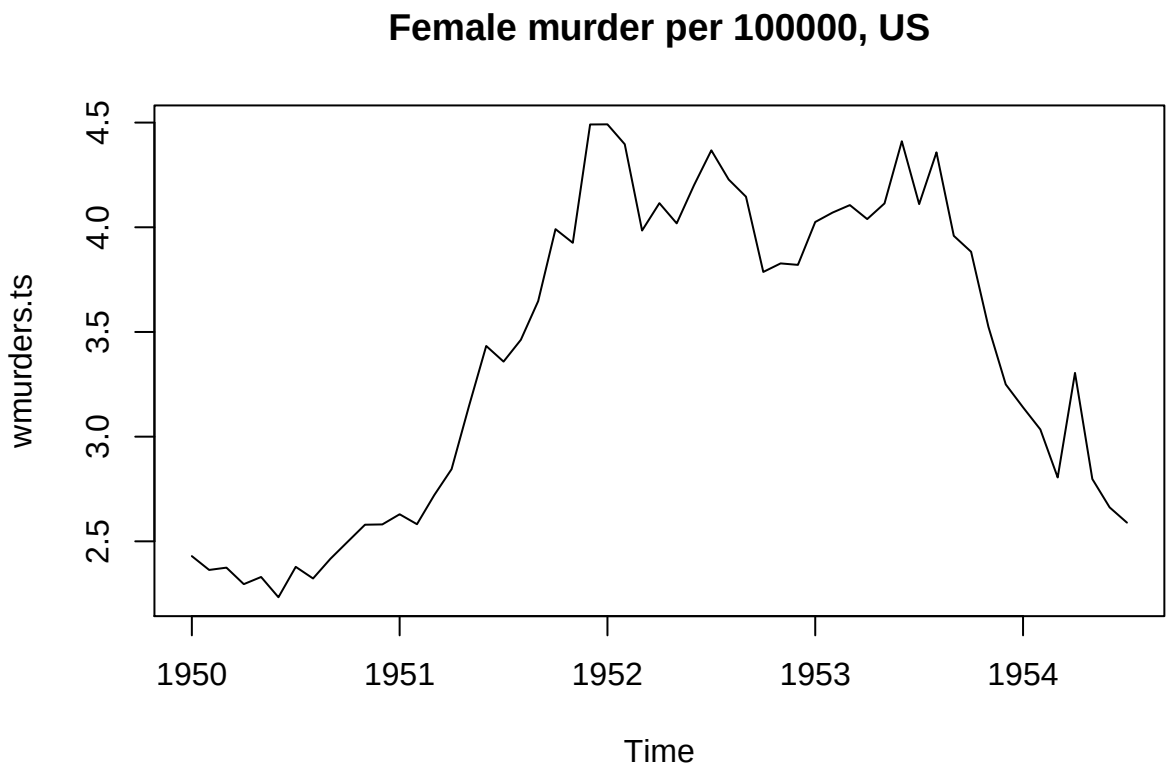
Quiz: Which of the three time series are stationary?

```
library(fpp)
#op <- par(mfrow=c(2,2), pty="s", mar=c(0,0,0,0))

a10.ts <- ts(a10, frequency = 12, start=c(1992,1))
debit.ts <- ts(debitcards, frequency = 12, start=c(2000,1))
wmurders.ts <- ts(wmurders, frequency = 12, start=c(1950,1))
```

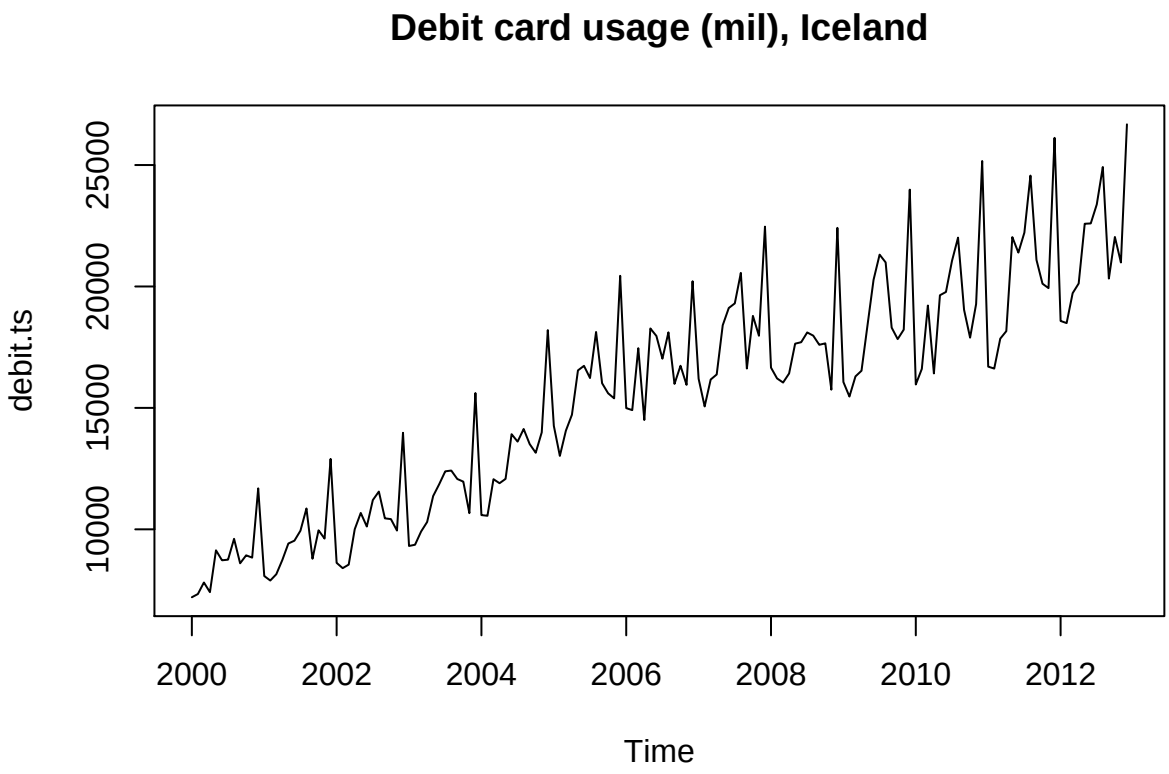
```r
#par(mfrow=c(4,1))
plot(a10.ts)
title("Anti-diabetic drug sales, Aus")
```




```r
plot(wmurders.ts)
title("Female murder per 100000, US")
```




```
plot(debit.ts)
title("Debit card usage (mil), Iceland")
```




```
# end of plotting, reset to previous settings
#reset_par()
```

Obvious seasonality rules out the debit card usage time series and anti-diabetic drug sales, while trend rules out female murder rate series. In addition to seasonality, increasing variance in the anti-diabetic drug sales series also violated the assumptions required for stationarity.

Let's take the anti-diabetic drug sales series and see if we can make it stationary: a common technique that springs to mind is to deal with non-constant variance by taking the logarithm or square root of the series, so we'll try that.

Apart from taking the log, another common technique to deal with non-stationary data is to difference the data. That is, given the series Z_t, create the new series:
$Y_i = Z_i - Z_{i-1}$

The differenced data will contain one less point than the original data. Although we can difference the data more than once, one difference is usually sufficient.

See `diff()` in action:

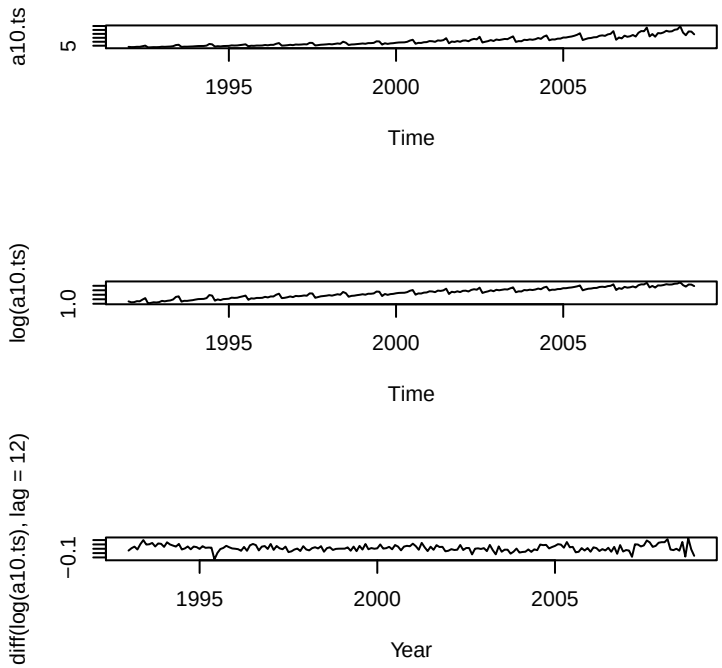
```
prices <- c(10, 4, 17, 3, 1)
diff(prices)
```

```
## [1]  -6  13 -14  -2
```

Let's plot the original diabetic drug sales series, the log-transformed series, and the differenced series:

```
par(mfrow=c(3,1))
plot(a10.ts)
plot(log(a10.ts))
```

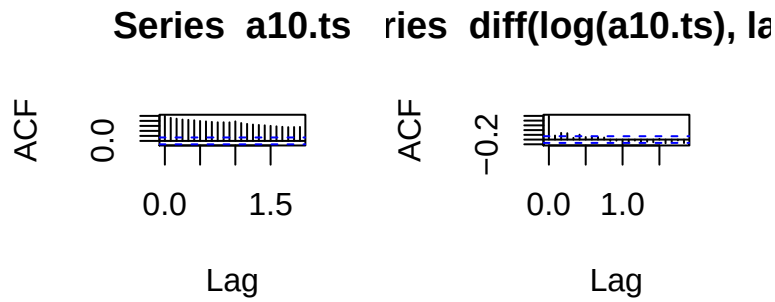
```
plot(diff(log(a10.ts), lag=12), xlab="Year")
```



Transformations such as logarithms help to stabilize the variance of a time series and techniques such as differencing (compute differences between consecutive observations) help stabilize the mean of a time series by removing changes in the level of a time series and so eliminating trend and seasonality.

Another technique that is helpful is the ACF plot we've learned about earlier: **The ACF of a stationary time series will drop to zero relatively quickly** while the ACF of a non-stationary data decreases slowly:

```
par(mfrow=c(1,2))
acf(a10.ts)
acf(diff(log(a10.ts), lag=12))
```



Second-order differencing and Seasonal differencing:
Occasionally the differenced data will not appear stationary and it may be necessary to difference the data a second time to obtain a stationary series. In this case we are modeling the "change in the changes" of the original data - in practice, it is almost never necessary to go beyond second-order differences.

Second order differencing:

```
prices <- c(10, 4, 17, 3, 1)
prices
```

```
## [1] 10  4 17  3  1
```

```
diff(prices)
```

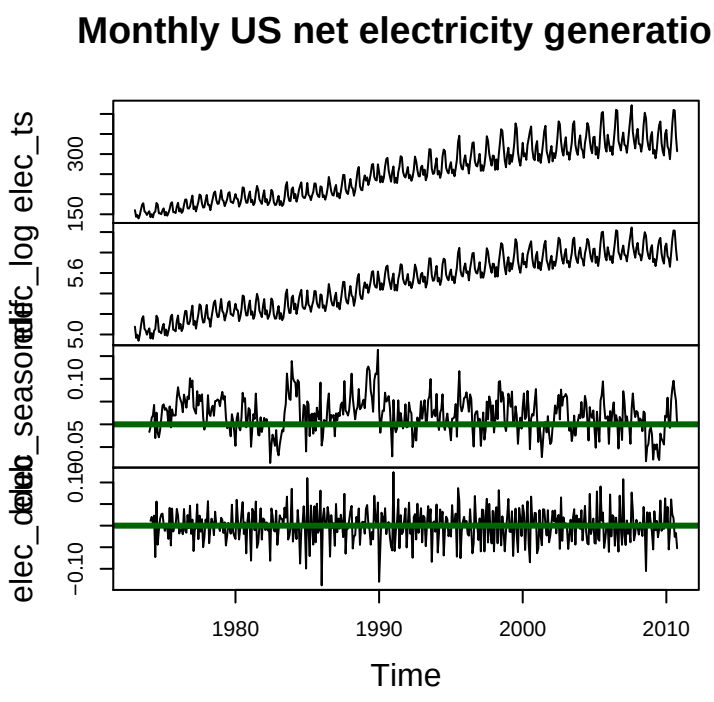
```
## [1]  -6  13 -14  -2
```

```
diff(diff(prices))
```

```
## [1]  19 -27  12
```

Seasonal difference is the difference between an observation and the corresponding observation from the previous season, and are also called "lag-m differences" as we subtract the observation after a lag of m periods.

```
elec_ts <- ts(usmelec, frequency=12, start=c(1973,1))
elec_log <- log(elec_ts)
elec_seasondif <- diff(log(elec_ts), lag=12)
elec_doub <- diff(elec_seasondif, lag=1)

elec.df <- ts(data.frame(
  cbind(elec_ts, elec_log, elec_seasondif, elec_doub)
), frequency=12, start=c(1973,1))
plot.ts(elec.df, plot.type="m", main="Monthly US net electricity generation", panel = function(x, col, l
  lines(x=x, col=col)
  abline(h=0, col="darkgreen", lwd=3)})
```



Sometimes it is necessary to do both a seasonal difference and a first difference to obtain stationary data as shown in the panel above. When both seasonal and first differences are applied it makes no difference which is done first as the result will be the same. However, if the data have a strong seasonal pattern, it is

recommended that seasonal differencing be done first because sometimes the resulting series will be stationary and there will be no need for a further first difference.

Augmented Dickey-Fuller (ADF) test:
Another way to determine more objectively if differencing is required is to use a unit root test and one of the most popular test is the *Augmented Dickey-Fuller (ADF)* test.

The null-hypothesis for an ADF test is that the data are non-stationary, so large p-values are indicative of non-stationarity and **small p-values suggest stationarity**. Using the usual 95% threshold, differencing is required if the p-value is greater than 0.05.

```
adf.test(elec_ts, alternative = "stationary")
```

```
## Warning in adf.test(elec_ts, alternative = "stationary"): p-value smaller
## than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  elec_ts
## Dickey-Fuller = -9.9254, Lag order = 7, p-value = 0.01
## alternative hypothesis: stationary
```

```
adf.test(souvenir_ts, alternative = "stationary")
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  souvenir_ts
## Dickey-Fuller = -2.0809, Lag order = 4, p-value = 0.5427
## alternative hypothesis: stationary
```

Discuss: Between the two time series, which one is not stationary? How would you transform the time series into one that is stationary? Use the Augmented Dickey-Fuller Test to confirm that your transformed time series is in fact now stationary:

```
### Your answer below:
```

Autoregressive models

Recall from our regression models class that a multiple regression model "forecast" the variable of interest using a linear combination of predictors. In an autoregressive model, we forecast the variable of interest using a linear combination of *past values of the variable*. The term autoregression indicates that it is a regression of the variable against itself, just as the term autocorrelation refers to the practice of performing correlation on a time series against itself. Personally, I find it easier to think of the term "auto" in both cases as in "autonomous" rather than "automatic".

Given that definition, an autoregression model of order p can be described as:
$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + e_t$

Where c is a constant and e is white noise. This is like a multiple regression but with lagged values of y_t as predictors. We refer to this as an AR(p) model. Note that changing the parameters $\phi_1, ...\phi_p$ will result in different time series patterns, and also that the variance of the error term e_t will only change the scale of the series but not the patterns.

For an AR(1) model:
- When $\phi_1 = 0$, y_t is equivalent to white noise $(c + e_t)$ - When $\phi_1 = 1$ and $c = 0$, y_t is equivalent to a random walk

We normally restrict autoregressive models to stationary data and then some constraints on the values of the parameters are required:
- For an AR(1) model: $-1 \leq \phi_1 < 1$
- For an AR(2) model: $-1 \leq \phi_2 < 1$, $\phi_1 + \phi_2 < 1$, $\phi_1 - \phi_2 < 1$

To see an example of AR in action, we'll use the time series for monthly total generation of electricity. The length of our time series is 454, and `ar()` will select the optimal value of order if we don't specify a maximum order by using the optional `order.max` parameter:

```
elec_ar <- ar(elec_ts)
elec_ar
```

```
##
## Call:
## ar(x = elec_ts)
##
## Coefficients:
##       1        2        3        4        5        6        7        8
##  0.9307  -0.0796   0.0169  -0.1177   0.1437   0.0353   0.0497  -0.0769
##       9       10       11       12       13       14       15       16
## -0.0105   0.0651  -0.0349   0.4881  -0.4270   0.0349  -0.1109   0.1631
##      17       18       19       20       21       22       23       24
## -0.0370  -0.1711   0.0755   0.0337  -0.0407  -0.0034   0.0604   0.2555
##      25
## -0.2533
##
## Order selected 25  sigma^2 estimated as  143.1
```

When p >= 3, the restrictions are much more complicated - fortunately R takes care of these restrictions when estimating a model.

Moving average models

Rather than using past values of the forecast variable in a regression, a moving average model uses past forecast errors in a regression-like model:
$y_t = c + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q}$
Where e_t is white noise. We refer to this as an MA(q) model. Notice that each value of y_t can be thought of as a weighted moving average of the past few forecast errors.

Non-seasonal ARIMA models

If we combine differencing, autoregression and a moving average model, we obtain a non-seasonal ARIMA model. The AR part of ARIMA indicates that the evolving variable of interest is regressed on its own lagged values while the MA part indicates that the regression error is actually a linear combination of error terms whose values occured contemporaneously and at various times in the past. The I (for "integrated") indicates that the data values have been replaced with the difference between their values and the previous values (and this differencing process may have been performed more than once, even though in practice it's rarely necessary to have more than 2 levels of differencing before a white noise series is obtained). The purpose of each of these features is to make the model fit the data as well as possible.

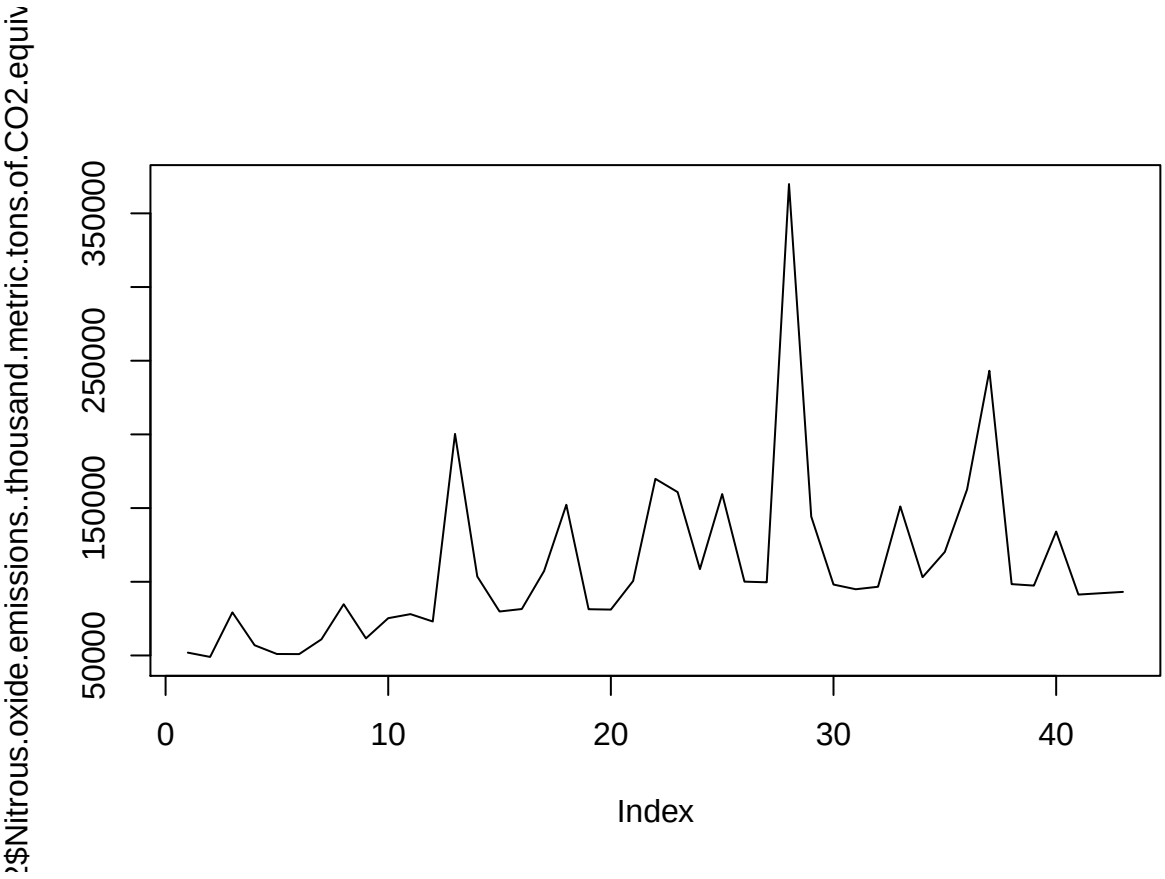
Non-seasonal ARIMA models are generally denoted ARIMA(p,d,q) where parameters p,d, and q are non-negative integers. - p is the order (number of time lags) of the AR model
- d is the degree of differencing (number of times the data have had past values subtracted)
- q is the order of the moving-average model

ARIMA(1,0,0) is AR(1); ARIMA(0,1,0) is I(1), and ARIMA(0,0,1) is MA(1).

Selecting appropriate values for p, d and q can be difficult but fortunately for us the `auto.arima()` function will do it for us.

The Nitrous Oxide emissions into Indonesia's atmosphere is a time series with no seasonality and no trend; Nitrous oxide emissions are emissions from agricultural biomass burning, industrial activities, and livestock management. Take a look at the observations over the years:

```
plot(co2$Nitrous.oxide.emissions..thousand.metric.tons.of.CO2.equivalent., type="l")
```



We'll first create a time series object using observations from that data frame, specifying the `start` and `frequency`:

```
agri_ts <- ts(co2$Nitrous.oxide.emissions..thousand.metric.tons.of.CO2.equivalent., start=1970, frequen
```

And use `auto.arima` on the time series, and since we've observed earlier that there seems to be no seasonal pattern, we'll restrict the search to non-seasonal models using `seasonal=F`:

```
agri_arima <- auto.arima(agri_ts, seasonal=F)
agri_arima
```

```
## Series: agri_ts
## ARIMA(0,1,1)
##
## Coefficients:
##           ma1
##       -0.8244
## s.e.   0.0842
```

```
##
## sigma^2 estimated as 3210286376:  log likelihood=-519.34
## AIC=1042.68   AICc=1042.99   BIC=1046.16
```

The output of `auto.arima` suggests an ARIMA(0,1,1), so that's equivalent to a one-order differencing with a MA(1) model.

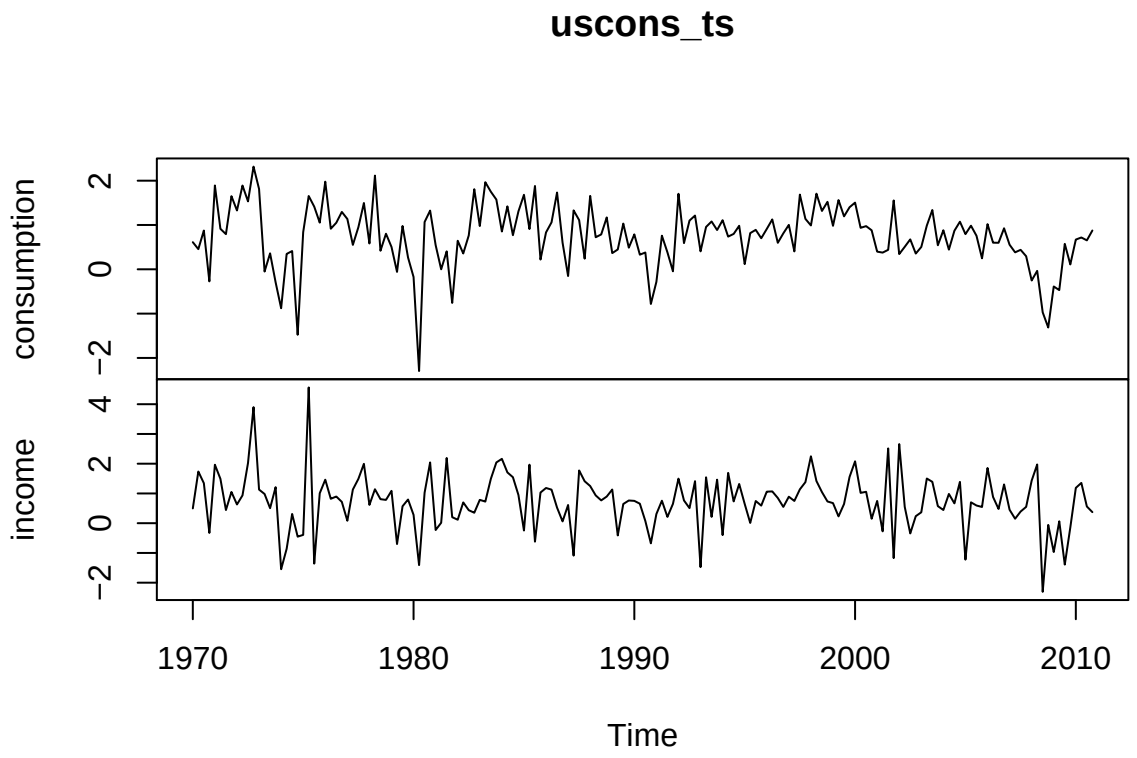
Discussion: `usconsumption` is a multivariate time series measuring percentage changes in quarterly personal consumption expenditure and personal disposable income for the US from 1970 to 2010:

```
head(usconsumption)
```

```
##         consumption    income
## 1970 Q1   0.6122769  0.496540
## 1970 Q2   0.4549298  1.736460
## 1970 Q3   0.8746730  1.344881
## 1970 Q4  -0.2725144 -0.328146
## 1971 Q1   1.8921870  1.965432
## 1971 Q2   0.9133782  1.490757
```

When we plot the multivariate time series, we'll see the following pattern:

```
uscons_ts <- ts(usconsumption, start = c(1970,1), frequency = 4)
plot(uscons_ts)
```



1. Do you think the time series is stationary? (Hint: I mentioned earlier that it is the percentage changes in quarterly PCE and PDI, strongly suggesting that it is a differenced series)
2. Is the time series seasonal?

3. If the time series is stationary (Question 1), what do you think the value of I would be in ARIMA?

Observe from the plot (quarterly percentage changes in US consumption expenditure and income) above - there appears to be no seasonal pattern so we will fit a non-seasonal ARIMA model:

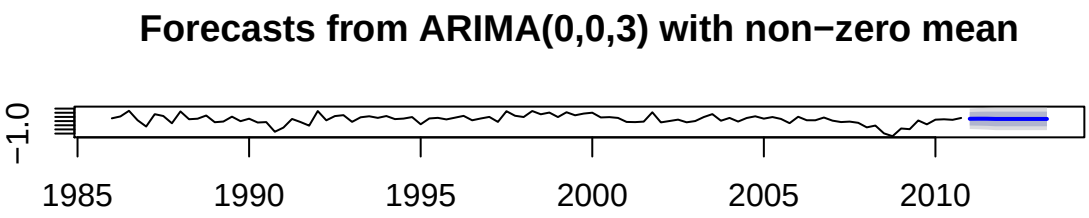
```
uscons.arima <- auto.arima(uscons_ts[,1], seasonal=F)
uscons.arima
```

```
## Series: uscons_ts[, 1]
## ARIMA(0,0,3) with non-zero mean
##
## Coefficients:
##          ma1     ma2     ma3    mean
##       0.2542  0.2260  0.2695  0.7562
## s.e.  0.0767  0.0779  0.0692  0.0844
##
## sigma^2 estimated as 0.3953:  log likelihood=-154.73
## AIC=319.46   AICc=319.84   BIC=334.96
```

The result suggested an ARIMA(0,0,3) model, recall this is equivalent to a MA(3) model:
$y_t = 0.7562 + e_t + 0.2542e_{t-1} + 0.2260e_{t-2} + 0.2695e_{t-3}$
where 0.7562 is the constant term and e_t is white noise with standard deviation 0.62 (or: sqrt(0.3953)).

We can now plot our forecast, specifying `h=10` to forecast for 10 future values and including past 95 observations in the plot.

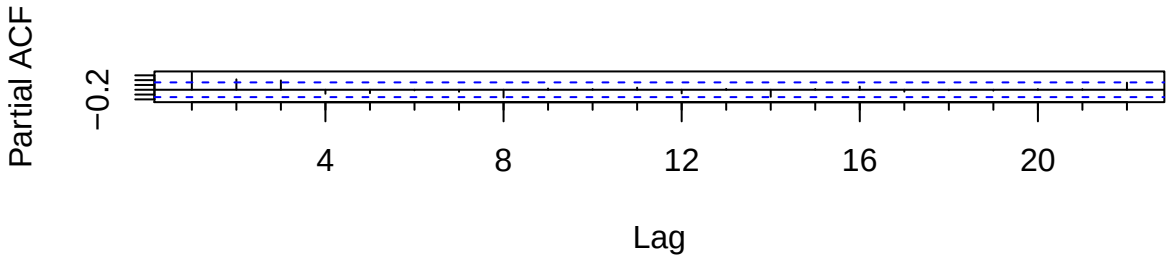
```
plot(forecast(uscons.arima, h=10), include=100)
```



To tell whether values of p and q chosen are appropriate for the time series we can try applying the ACF and PACF plot. Recall that an ACF plot shows the autocorrelations which measures the relationship between y_i and y_{t-k}, now if y_t and y_{t-1} are correlated then y_{t-1} and y_{t-2} must also be correlated. But then y_t and y_{t-2} may be correlated simply because they are both connected to $yt-1$ rather than because of any new information that can be useful to our forecast model.

To overcome this problem, we use **partial autocorrelations** to help us measure the relationship between y_t and y_{t-k} after removing the effects of other lags between t and k.

```
Pacf(uscons_ts[,1], main="")
```



The partial autocorrelations have the same critical values of $\pm 1.96/\sqrt{T}$ where T is the number of points in the time series. In here, the critical values are 0.15 and -0.15 respectively.

```
1.96/sqrt(length(uscons_ts[,1]))
```

```
## [1] 0.1530503
```

```
-1.96/sqrt(length(uscons_ts[,1]))
```

```
## [1] -0.1530503
```

Observe the pattern in the first three spikes from the PACF plot - this is what we would expect from an ARIMA(0,0,3) as the PACF tends to decay exponentially. In this case, the PACF lead us to the same model as the automatic procedure we perform using `auto.arima` above (both suggested ARIMA(0,0,3)).

How `auto.arima` works behind the scene is a combination of KPSS tests for choosing the parameter d and choosing p and q to minimize the AIC - the details of which is beyond the scope of this coursebook. However, if you want to fit a model using your own parameters, you can do so using the `Arima()` function. To fit a `ARIMA(0,0,3)` model to the US consumption data:

```
uscons_arima2 <- Arima(uscons_ts[,1], order=c(0,0,3))
uscons_arima2
```

```
## Series: uscons_ts[, 1]
## ARIMA(0,0,3) with non-zero mean
##
## Coefficients:
##          ma1     ma2     ma3    mean
##       0.2542  0.2260  0.2695  0.7562
## s.e.  0.0767  0.0779  0.0692  0.0844
##
## sigma^2 estimated as 0.3953:  log likelihood=-154.73
## AIC=319.46   AICc=319.84   BIC=334.96
```

This gives us exactly the same result as the `auto.arima()` function call we made above.

Let's summarize the general approach to fitting an ARIMA model:
1. Plot the data; Identify any unusual observations
2. If necessary, transform the data (Box-Cox Transformation) to stabilize the variance
3. If data are non-stationary, take first differences of the data until data are stationary
4. Examine the ACF / PACF: Is an AR(p) or MA(q) model appropriate?
5. Try your chosen model(s)
6. Check the residuals from our models by plotting the ACF of the residuals - they have to look like white noise
7. If residuals look like white noise, calculate forecasts

The automated algorithm only takes care of steps 3-5, so we'll have to do the rest manually!

Seasonal ARIMA models

ARIMA models are also capable of modelling seasonal data. A seasonal ARIMA model is formed by including additional seasonal terms in the ARIMA model we have seen and discussed so far.

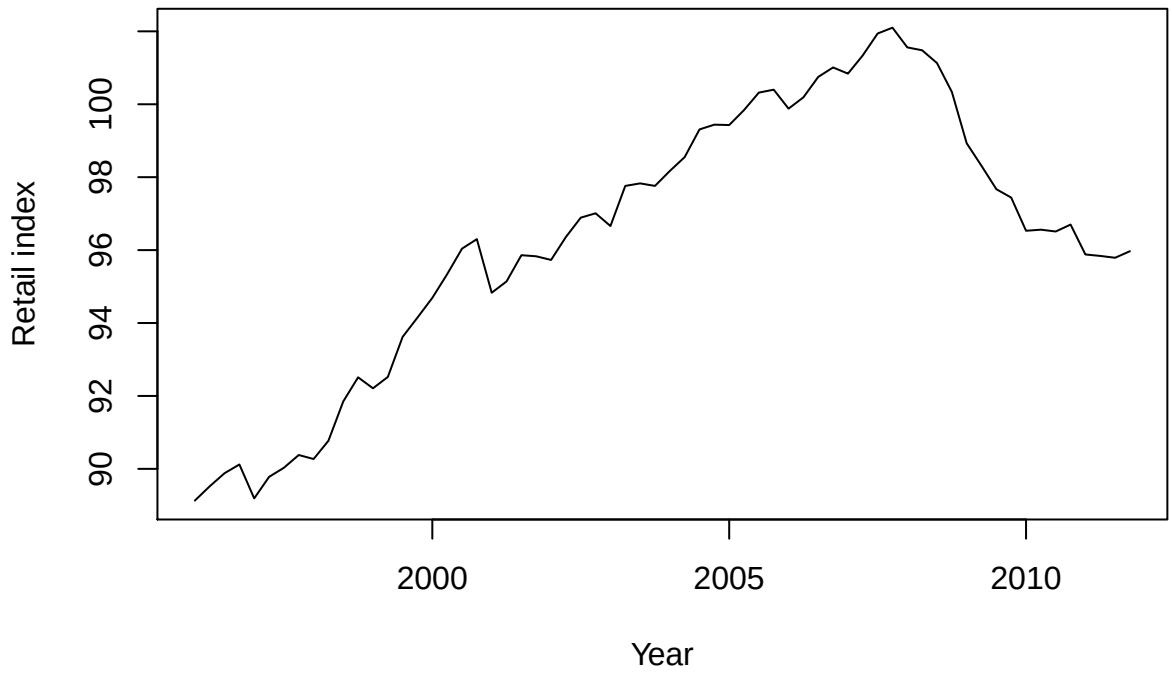
Seasonal ARIMA models are usually denoted $ARIMA(p,d,q)(P,D,Q)_m$ where m refers to the number of periods in each season, and the uppercase P,D,Q refers to the autoregressive, differencing and moving average terms for the seasonal part of the ARIMA model. Notice that as a convention, we use uppercase notation for the seasonal parts of the model and lowercase for the non-seasonal parts.

The seasonal part of an AR or MA model will be seen in the seasonal lags of the PACF and ACF. For example, an $ARIMA(0,0,0)(0,0,1)_{12}$ model will show:
- a spike at lag 12 in the ACF but no other significant spikes
- exponential decay in the seasonal lags of the PACF; that is, at lags 12, 24, 36...

Similarly, an $ARIMA(0,0,0)(1,0,0)_{12}$ model will show:
- exponential decay in the seasonal lags of the ACF
- a significant spike at lag 12 in the PACF

In considering the appropriate seasonal orders for an ARIMA model, restrict attention to the seasonal lags. The modelling procedure is almost the same as for non-seasonal data, except that we need to select seasonal AR and MA terms as well as the non-seasonal components of the model. Let's see some examples:

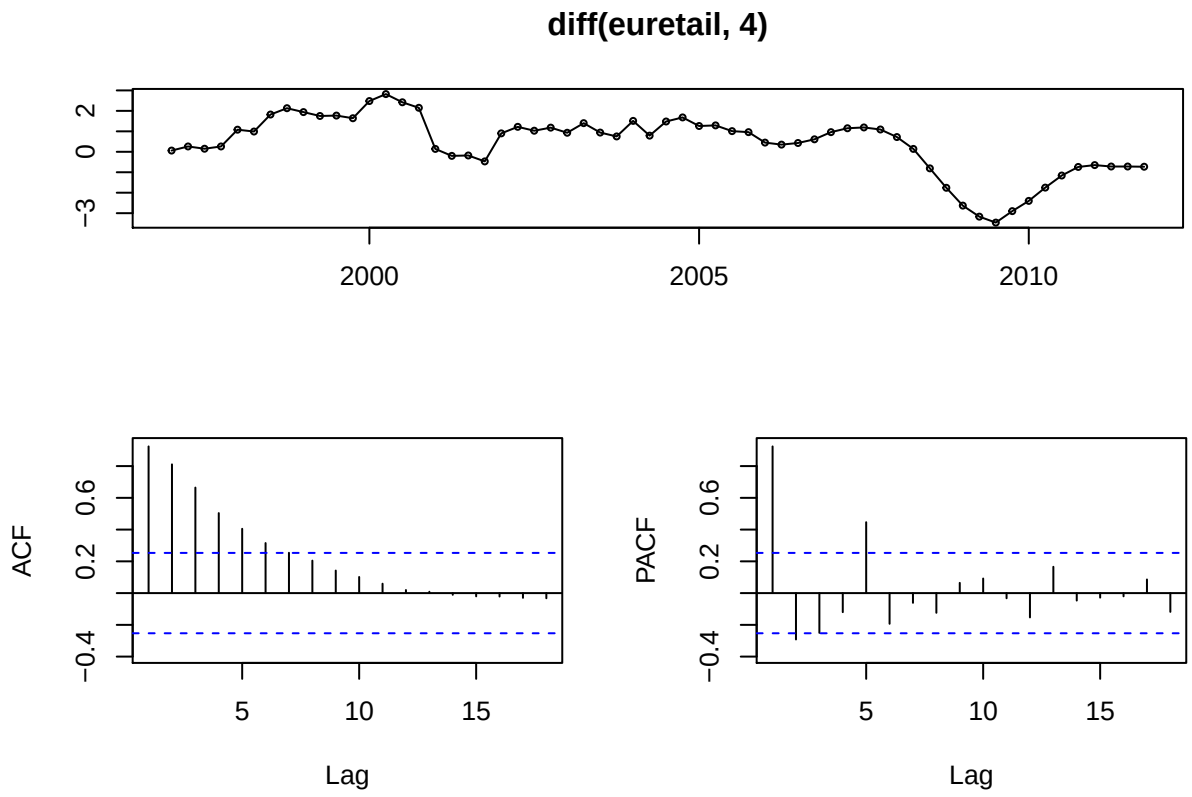
```
plot(euretail, ylab="Retail index", xlab="Year")
```



This is a plot of the quarterly retail trade index in the Euro area covering wholesale and retail trade and

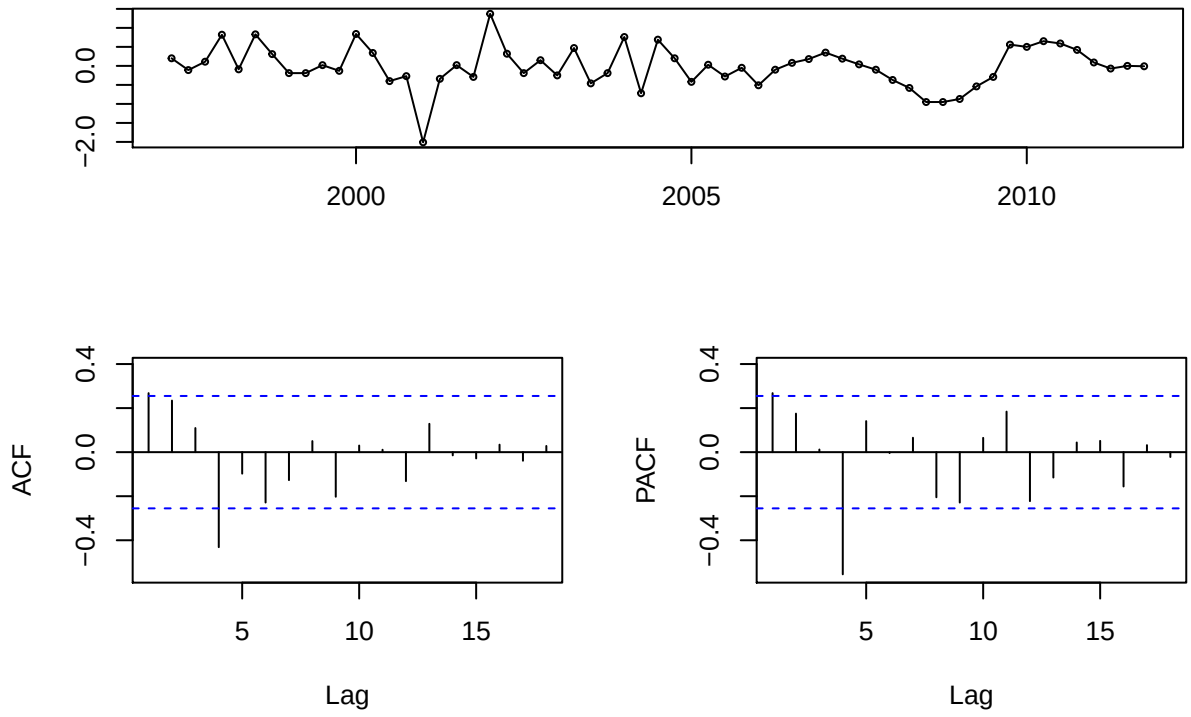
repair of motor vehicles. The data is clearly non-stationary with some seasonality, so let's take a seasonal difference and plot the seasonally differenced data:

```
tsdisplay(diff(euretail, 4)) # quarterly differenced
```



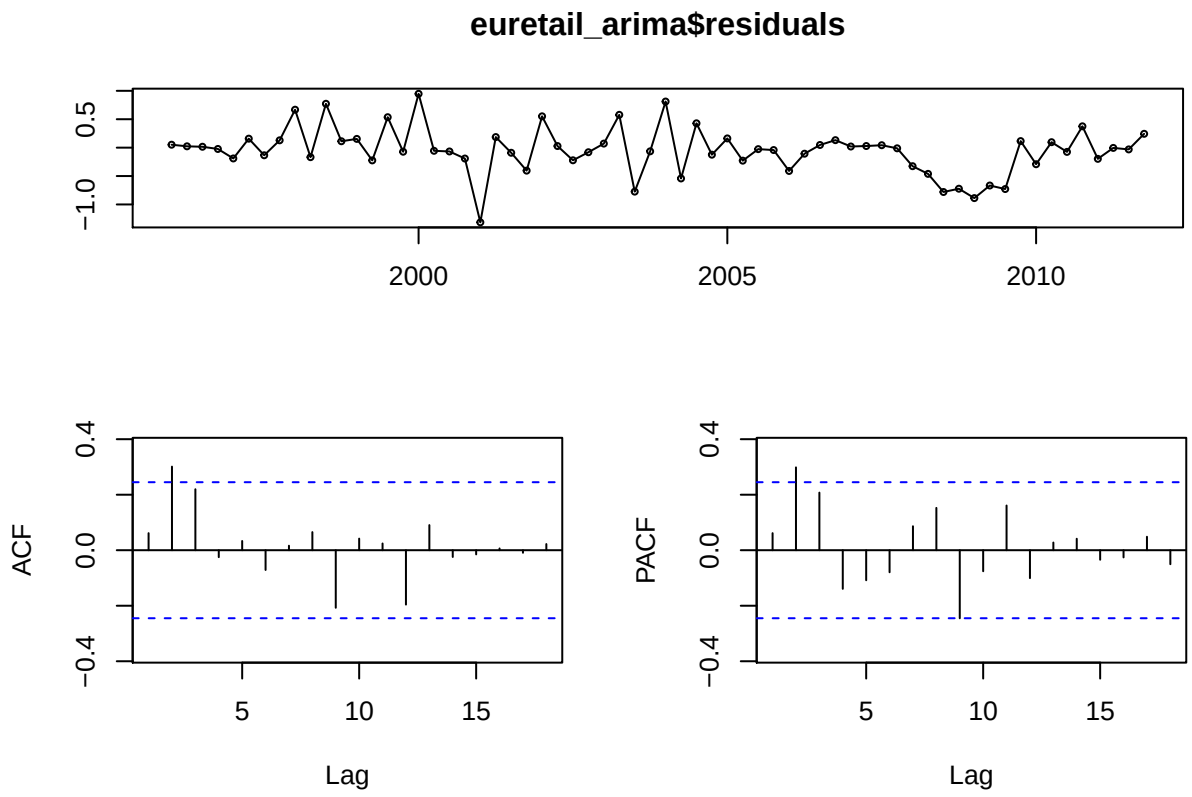
The data still appear to be non-stationary so we take an additional first difference:

```
tsdisplay(diff(diff(euretail,4),1), main="")
```



Our aim now is to find an appropriate ARIMA model based on the ACF and PACF shown. The significant spike at lag 1 in the ACF and PACF suggests a non-seasonal MA(1) component, and the significant spike at lag 4 (and to a lesser extent at lag 8, lag 12, lag 16) in the PACF suggest a seasonal MA(1) component. Consequently, we begin with an $ARIMA(0,1,1)(0,1,1)_4$ model, indicating a first and seasonal difference, and non-seasonal and seasonal MA(1) components. The residuals for the fitted model are:

```
euretail_arima <- Arima(euretail, order=c(0,1,1), seasonal=c(0,1,1))
tsdisplay(euretail_arima$residuals)
```



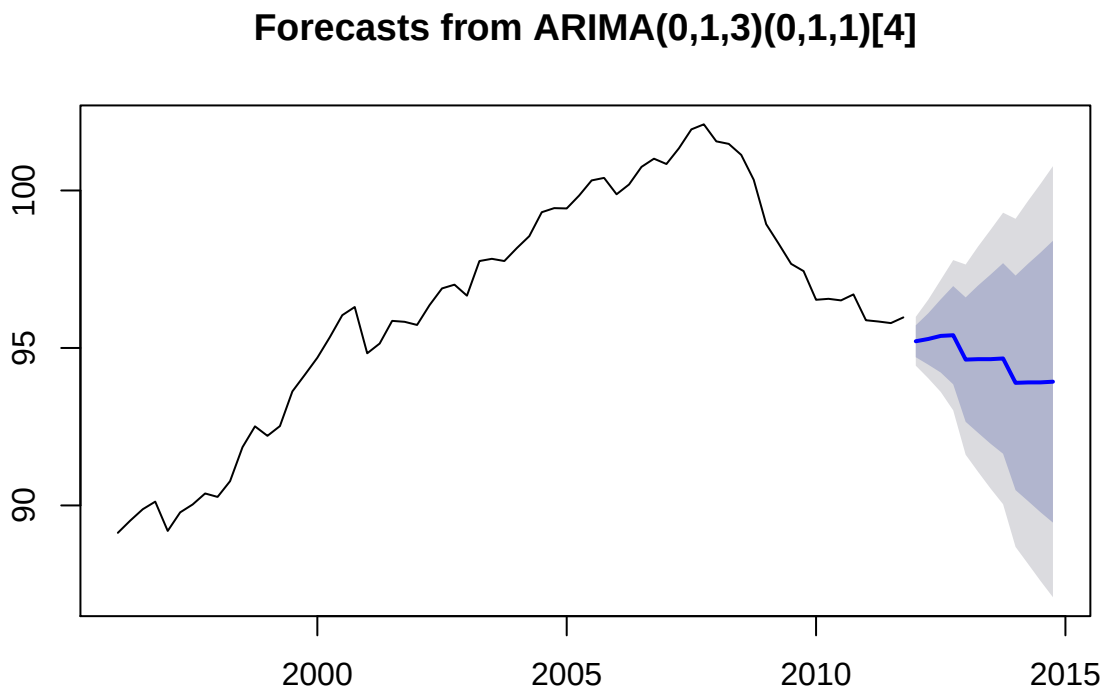
Both the ACF and PACF show significant spikes at lag 2, and almost significant spikes at lag 3, indicating some additional non-seasonal terms need to be included in the model. Using the Ljung-Box test we try a few other models and arrive at a lowest AIC value with the $ARIMA(0,1,3)(0,1,1)_4$ model:

```
euretail_final <- Arima(euretail, order=c(0,1,3), seasonal=c(0,1,1))

Box.test(euretail_final$residuals, lag=16, type="Ljung")
```

```
##
##  Box-Ljung test
##
## data:  euretail_final$residuals
## X-squared = 7.0105, df = 16, p-value = 0.9731
```

Now that we have a seasonal ARIMA model that passes the required checks, we'll plot the forecasts from the model for the next 3 years. Notice that the forecasts follow the recent trend in the data:

```
plot(forecast(euretail_final, h=12))
```



Notice if we have used `auto.arima`, we would obtain a different ARIMA model:

```
auto.arima(euretail)
```

```
## Series: euretail
## ARIMA(1,1,2)(0,1,1)[4]
##
## Coefficients:
##          ar1      ma1     ma2     sma1
##       0.7345  -0.4655  0.2162  -0.8413
## s.e.  0.2239   0.1995  0.2096   0.1869
##
## sigma^2 estimated as 0.1592:  log likelihood=-29.69
## AIC=69.37   AICc=70.51   BIC=79.76
```

This model have a larger AIC value than the one we manually fitted (69.37 compared to 67.4), as well as a slightly-higher residual sum of squares (8.754 vs 8.6) - this is because `auto.arima` takes some short-cuts in order to speed up the computation and is not guaranteed to return the best model. We could turn the short-cuts off when calling the auto arima function:

```
auto.arima(euretail, stepwise=F, approximation=F)
```

```
## Series: euretail
## ARIMA(0,1,3)(0,1,1)[4]
##
## Coefficients:
##          ma1     ma2     ma3     sma1
##       0.2625  0.3697  0.4194  -0.6615
```

```
## s.e.  0.1239  0.1260  0.1296   0.1555
##
## sigma^2 estimated as 0.1564:  log likelihood=-28.7
## AIC=67.4   AICc=68.53   BIC=77.78
```

This time it returned the same model we had identified by hand:

```
euretail_final
```

```
## Series: euretail
## ARIMA(0,1,3)(0,1,1)[4]
##
## Coefficients:
##          ma1     ma2     ma3     sma1
##       0.2625  0.3697  0.4194  -0.6615
## s.e.  0.1239  0.1260  0.1296   0.1555
##
## sigma^2 estimated as 0.1564:  log likelihood=-28.7
## AIC=67.4   AICc=68.53   BIC=77.78
```

Additional Tips to Working with Time Series

Sometimes you'll need to convert a series of daily observations into a weekly one through aggregation. If you want to avoid using a for-loop, then R's `xts` (stands for extensible time-series) has functions that allow you to convert the daily observations into weekly, monthly, quarterly or even yearly observations.

Pay attention to the final 2 lines of code in the following chunk:

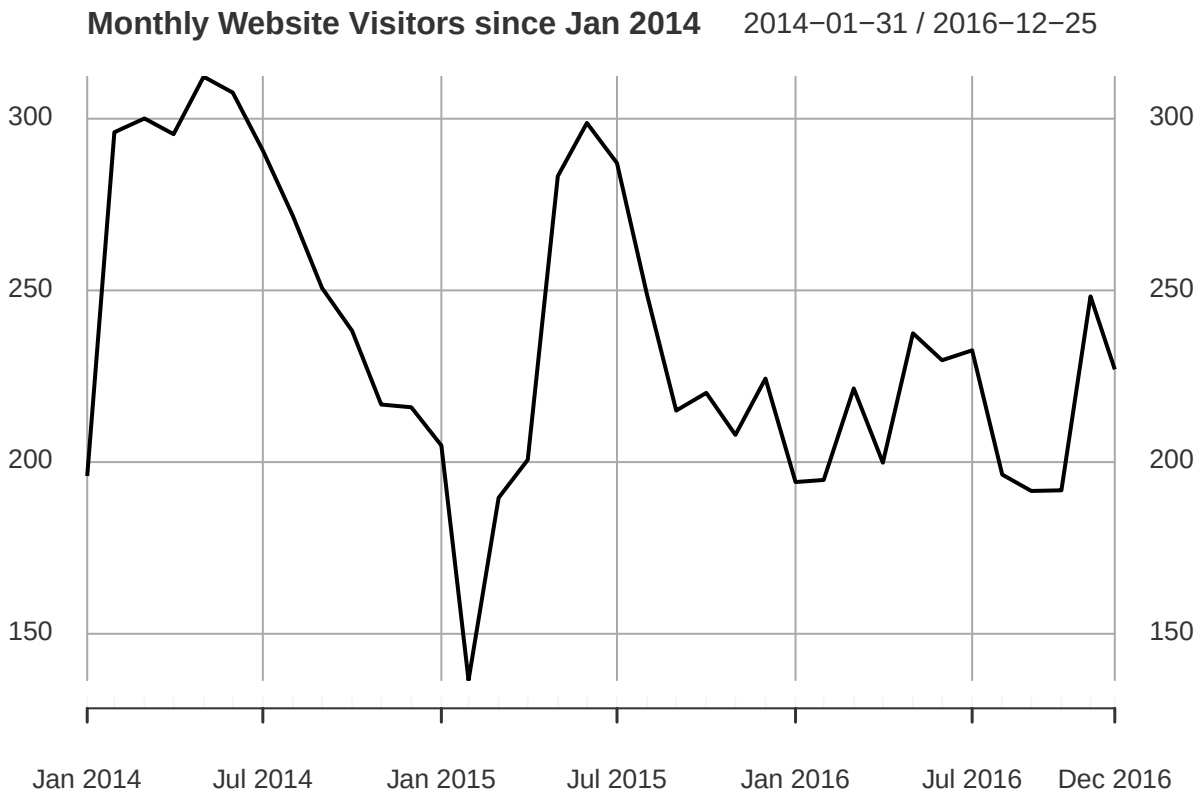
```
library(xts)
webtraffic <- read.csv("data_input/workshop.csv")
webtraffic$weekdays <- weekdays(as.Date(webtraffic$Day.Index))
webtraffic <- webtraffic[7:1084, ]
webtf <- as.xts(webtraffic$Sessions, order.by=as.Date(webtraffic$Day.Index))
webtfw <- apply.weekly(webtf,sum)
```

As an exercise, try and change the final line of code to `apply.quarterly()`, name it `webtfq` and print the first 6 values of `webtfq`

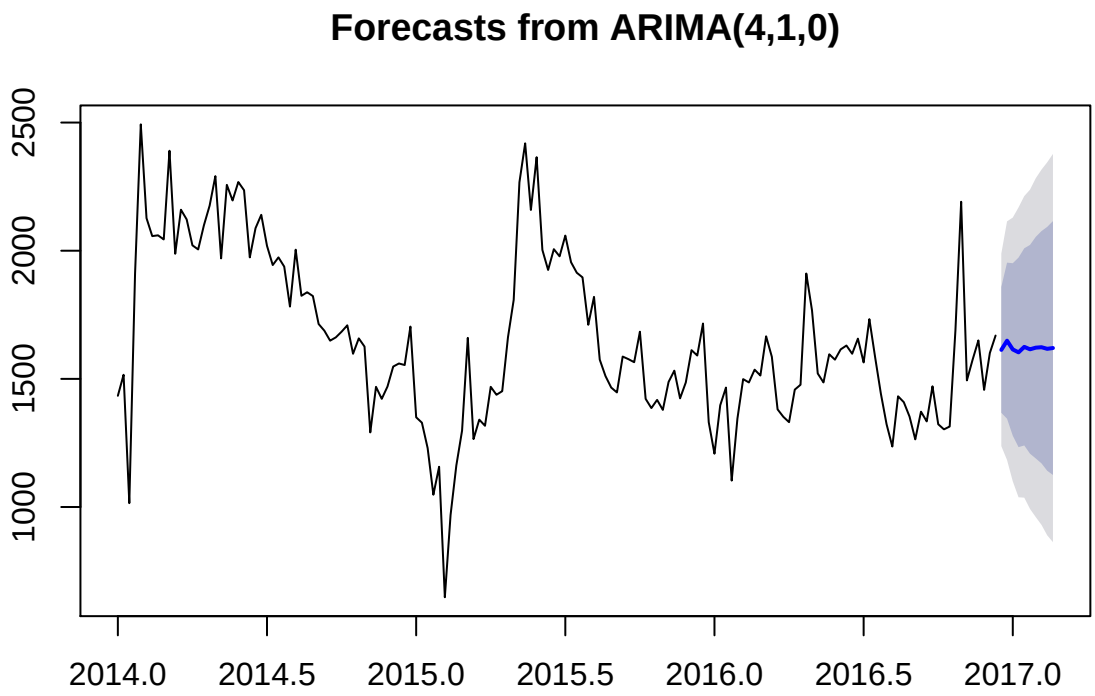
```
# Your answer below:
```

You can also use `plot()` on an `xts` object:

```
webtfm <- apply.monthly(webtf, mean)
plot(webtfm, main="Monthly Website Visitors since Jan 2014")
```




```
webts <- ts(webtfw, start=c(2014,01,19), frequency=52)
plot(forecast(auto.arima(webts), h=10))
```



Another tip I want to give you is on the use of `quantmod` to extract stock data. R's `quantmod` package made this a breeze. When I execute the following code chunk, an object named `AAPL` will be created containing the stock we specified through the symbol:

```
options("getSymbols.warning4.0"=FALSE)
library(quantmod)
start <- as.Date("2014-09-10")
end <- as.Date("2018-09-09")

# Let's get Apple stock data; Apple's ticker symbol is AAPL. We use the
# quantmod function getSymbols, and pass a string as a first argument to
# identify the desired ticker symbol, pass 'yahoo' to src for Yahoo!
# Finance, and from and to specify date ranges

# The default behavior for getSymbols is to load data silently (and directly) into the
# global environment, with the object being named after the loaded ticker
# symbol.

getSymbols("AAPL", src = "yahoo", from = start, to = end)
```

Taking a peek at the data:

```
head(AAPL, 8)
```

```
apl <- as.ts(AAPL[,1])
plot(apl)
```

Now that we've learned about moving averages, let's do a fun practice of adding a SMA to our bar plot.

Recall that the bigger `n` is for moving averages, the smoother the line:

```
barChart(AAPL, theme='white.mono')
addSMA(n=15, col = "goldenrod4")
```

One last tip: When you plot a multivariate time series earlier, it plots each time series separately (`plot.type="multiple"`) but if you prefer to have time plotting as a single plot, you can specify that:

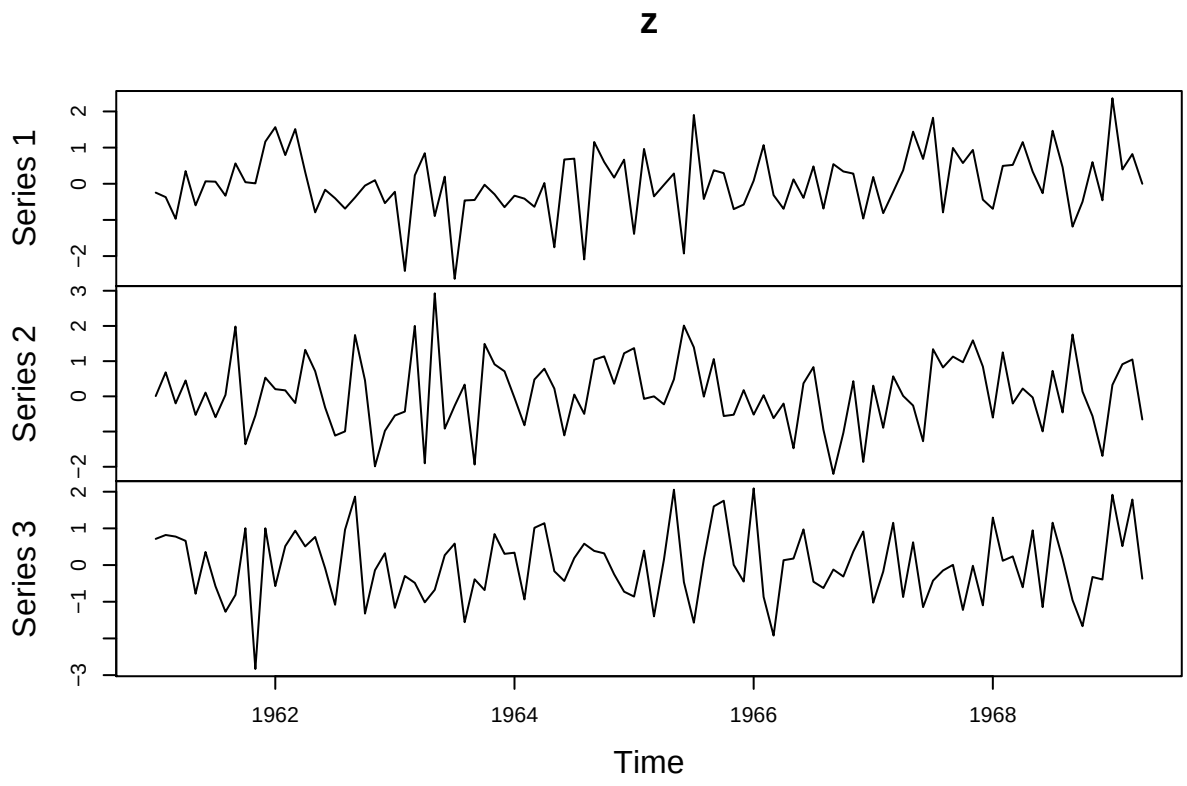
```
## Multivariate
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1), frequency = 12)
class(z)
```

```
## [1] "mts"    "ts"     "matrix"
```

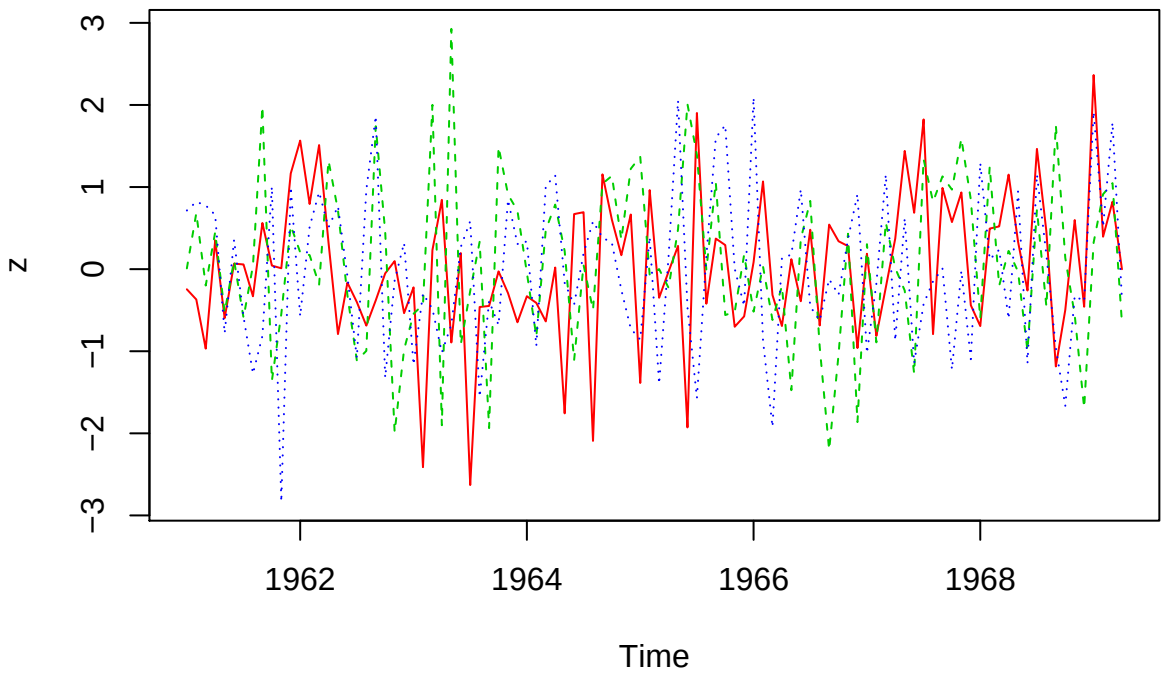
```
head(z) # as "matrix"
```

```
##              Series 1     Series 2   Series 3
## Jan 1961 -0.24536589  0.008902794  0.7140891
## Feb 1961 -0.36949968  0.682049851  0.8200063
## Mar 1961 -0.96848964 -0.199593570  0.7765008
## Apr 1961  0.35023022  0.450809164  0.6598183
## May 1961 -0.59721961 -0.526713092 -0.7813457
## Jun 1961  0.06727673  0.106733270  0.3569942
```

```
plot(z)
```



```
plot(z, plot.type = "single", lty = 1:3, col=2:4)
```



Graded Quizzes

This section of the workshop needs to be completed in the classroom in order to obtain a score that count towards your final grade.

Learn-by-Building

This week's learn-by-building may require a non-trivial amount of pre-processing so I recommend you schedule extra sessions with your mentor or work in a group if necessary. Download the dataset from Chicago Crime Portal, and use a sample of these data to build a forecasting project where you inspect the seasonality and trend of crime in Chicago. Submit you project in the form of a RMD format, and address the following question:

- Is crime generally rising in Chicago in the past decade (last 10 years)?
- Is there a seasonal component to the crime rate?
- Which time series method seems to capture the variation in your time series better? Explain your choice of algorithm and its key assumptions

Student should be awarded the full (3) points if they address at least 2 of the above questions. The questions are by no means definitive, but can be used as a "guide" in the preparation of your project. The data contains a variety of offenses, but you can sample only a type of crime you're interested in (eg. theft, narcotic, battery etc). Use visualization if it helps support your narrative.

Alternatively, read on in the Extra Content and peek at the `wiki.R` script that I gave. Adjust the script to scrap some data relating to Wikipedia views on any subject that you are interested in (choose between the English or Bahasa Indonesia version of Wikipedia). Produce a well-formatted RMD report that outline your findings and forecasts. Use any forecasting or time series method and argue your case. Show at least 1 visualization to complement your writing.

Student should be awarded the full (3) points if the report displays: - Appropriate use of any time series method
- Visualization
- Analysis on the trend (are more people interested in that subject over time? Are people reading about data science algorithms on Monday the most? Are Indonesians considerably more interested in the history and definition of Bhinneka Tunggal Ika as we near Independence Day each year?) Show originality and your own analysis / opinion on the matter supported by a simple visualization of the trend.

Extra Content:

Using the `wikipediatrend` library, I've downloaded the daily wikipedia views on the English[9] and Bahasa Indonesia[10] version of Joko Widodo's article. The full R code is in `wiki.R`. For convenience, I've saved a copy of that dataset into your `data_input` directory:

```
jokowi_wiki <- read.csv("data_input/wikiviews.csv")
str(jokowi_wiki)
```

```
## 'data.frame':    1158 obs. of  8 variables:
##  $ project    : Factor w/ 1 level "wikipedia": 1 1 1 1 1 1 1 1 1 1 ...
##  $ article    : Factor w/ 1 level "Joko_Widodo": 1 1 1 1 1 1 1 1 1 1 ...
##  $ access     : Factor w/ 1 level "all-access": 1 1 1 1 1 1 1 1 1 1 ...
##  $ agent      : Factor w/ 1 level "all-agents": 1 1 1 1 1 1 1 1 1 1 ...
##  $ granularity: Factor w/ 1 level "daily": 1 1 1 1 1 1 1 1 1 1 ...
##  $ date       : Factor w/ 1158 levels "2015-07-01","2015-07-02",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ en_views   : int  939 974 616 569 651 899 793 786 745 745 ...
##  $ id_views   : int  711 640 604 576 581 652 620 688 741 611 ...
```

We'll take only the `date` and `en_views` variables (the `id_views` variable correspond to the daily Wikipedia views on the Bahasa Indonesia version of the entry), and do the necessary pre-processing:

```
library(dplyr)
jokowi_en <- jokowi_wiki %>%
  mutate(ds = as.Date(date), y = en_views) %>%
  select(ds, y)
```

Using Prophet

Prophet is a forecasting tool released and made open source by Facebook's Core Data Science team. The team introduced it as "a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series

[9] Joko Widodo, Wikipedia (English)
[10] Joko Widodo, Wikipedia (Bahasa Indonesia)

that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well".

Let's load the library into our environment and run the prophet forecaster of the dataframe. A requirement of this package is the necessary preparation to obtain a dataframe with a `ds` column and a `y` column that correspond to a valid date type and the time series, respectively.

```
library(prophet)
```

```
## Loading required package: Rcpp
```

```
## Loading required package: rlang
```

```
jokowi_model <- prophet(jokowi_en)
```

```
## Disabling daily seasonality. Run prophet with daily.seasonality=TRUE to override this.
```

Prophet tells us that the optimization terminated normally and we'll now use `make_future_dataframe` to make a dataframe object containing rows from our history data plus 365 days ahead (1158+365 = 1523). This new dataframe is going to have 1,523 rows, with only one variable: the date (`ds`). At this point we are still not doing any prediction but merely constructing the "unseen" data for our prediction to take place:

```
# include_history = FALSE if we wish to exclude the historical data points
jokowi_fd <- make_future_dataframe(jokowi_model, periods = 365)
tail(jokowi_fd)
```

```
##             ds
## 1518 2019-08-26
## 1519 2019-08-27
## 1520 2019-08-28
## 1521 2019-08-29
## 1522 2019-08-30
## 1523 2019-08-31
```

Let's now use predict, passing in our model and our dataframe. This is a process that you should be very familiar by now. From the resulting dataframe (we name it `jokowi_fc`), we have the predicted

```
jokowi_fc <- predict(jokowi_model, jokowi_fd)
str(jokowi_fc)
```

```
## 'data.frame':    1523 obs. of  19 variables:
##  $ ds                        : POSIXct, format: "2015-07-01" "2015-07-02" ...
##  $ trend                     : num  874 875 875 876 877 ...
##  $ additive_terms            : num  19.5 -38.5 -125.7 -234.1 -185.5 ...
##  $ additive_terms_lower      : num  19.5 -38.5 -125.7 -234.1 -185.5 ...
##  $ additive_terms_upper      : num  19.5 -38.5 -125.7 -234.1 -185.5 ...
##  $ weekly                    : num  106.8 51.7 -31 -133.6 -77.7 ...
##  $ weekly_lower              : num  106.8 51.7 -31 -133.6 -77.7 ...
##  $ weekly_upper              : num  106.8 51.7 -31 -133.6 -77.7 ...
##  $ yearly                    : num  -87.3 -90.1 -94.6 -100.6 -107.7 ...
##  $ yearly_lower              : num  -87.3 -90.1 -94.6 -100.6 -107.7 ...
##  $ yearly_upper              : num  -87.3 -90.1 -94.6 -100.6 -107.7 ...
##  $ multiplicative_terms      : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ multiplicative_terms_lower: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ multiplicative_terms_upper: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ yhat_lower                : num  338.9 286 139.8 43.3 113.2 ...
##  $ yhat_upper                : num  1470 1403 1295 1207 1271 ...
##  $ trend_lower               : num  874 875 875 876 877 ...
##  $ trend_upper               : num  874 875 875 876 877 ...
```

```
##  $ yhat                        : num  894 836 750 642 691 ...
```

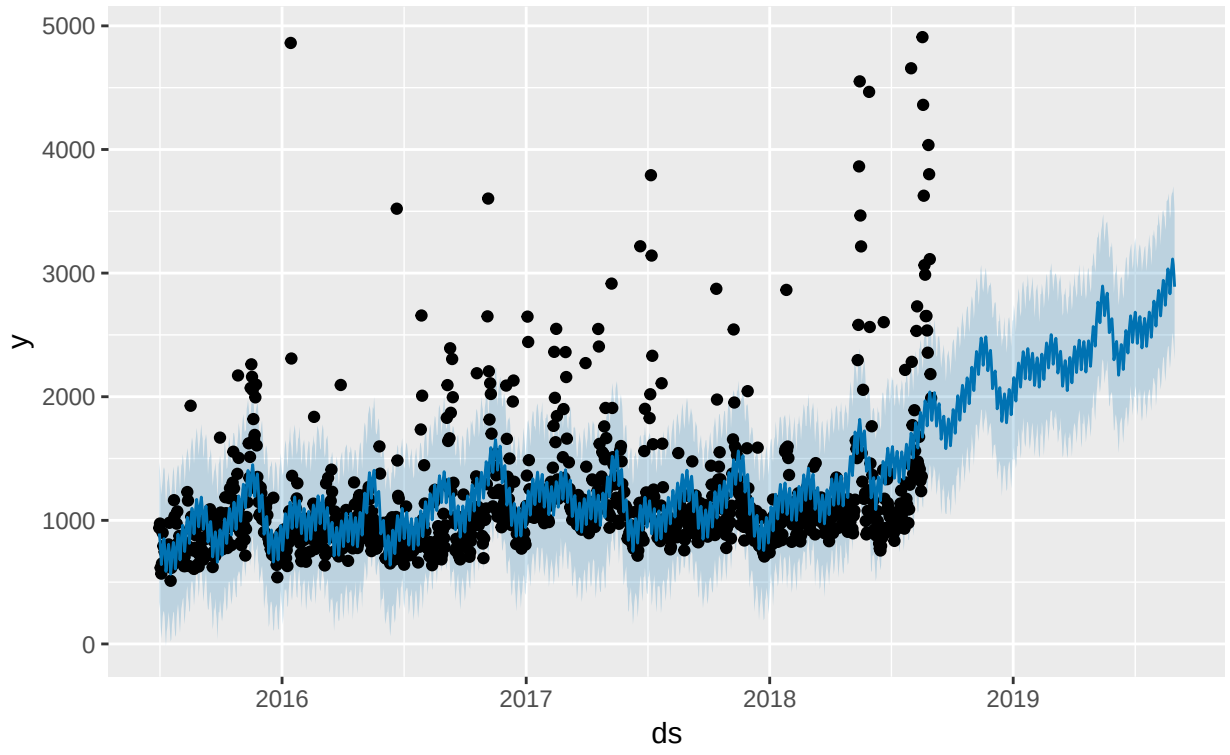
I'm particularly interested in the upper and lower bound (prediction intervals) so we can add some code to combine that into a data frame:

```
actual_pred <- cbind(jokowi_en, jokowi_fc[1:nrow(jokowi_en), c("yhat_lower", "yhat", "yhat_upper")])
head(actual_pred)
```

```
##           ds   y yhat_lower     yhat yhat_upper
## 1 2015-07-01 939  338.92334 893.8886   1469.940
## 2 2015-07-02 974  286.03067 836.4773   1402.882
## 3 2015-07-03 616  139.80084 749.8351   1295.117
## 4 2015-07-04 569   43.32205 641.8995   1207.106
## 5 2015-07-05 651  113.19251 691.1475   1270.615
## 6 2015-07-06 899  215.79828 782.2759   1321.016
```

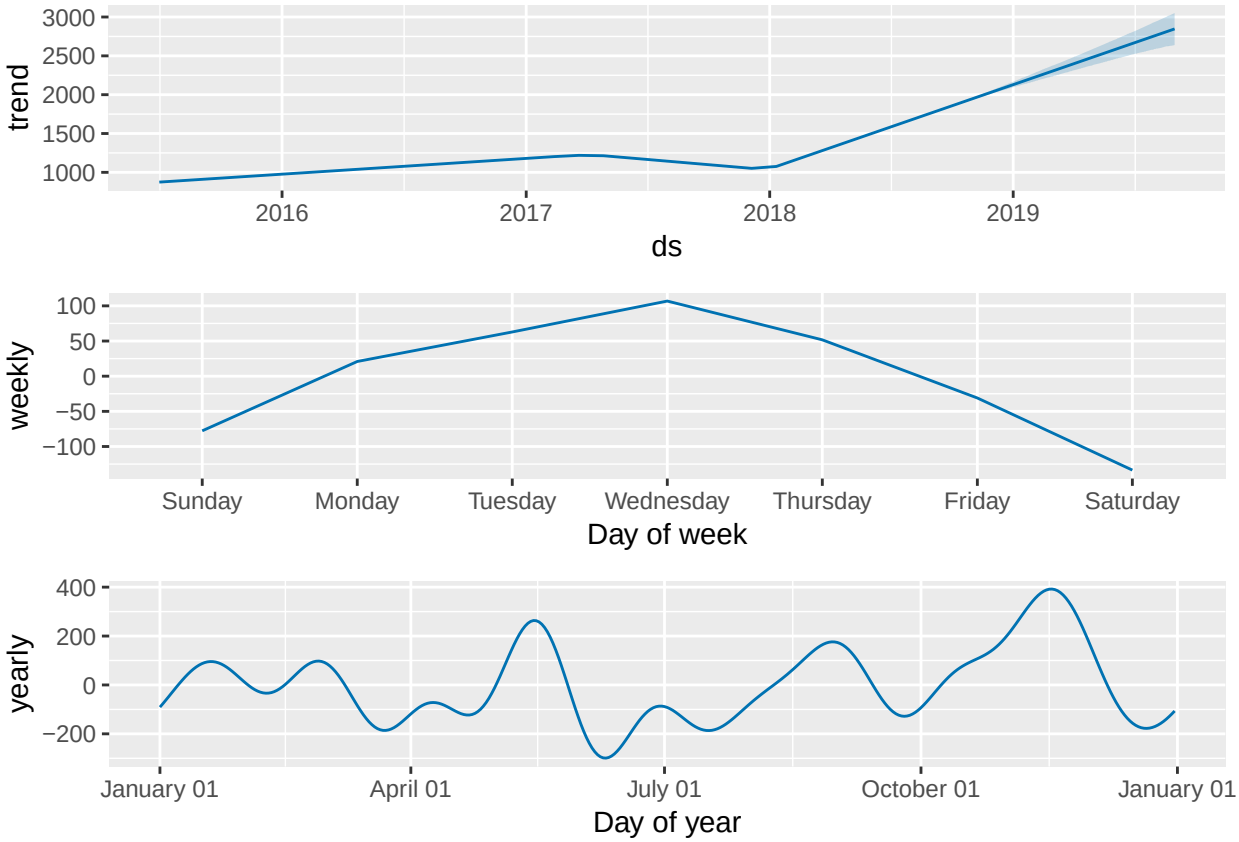
We can also use `prophet`'s plotting capabilities to plot our forecast against the historical observations:

```
plot(jokowi_model, jokowi_fc)
```



As with the case of most time series techniques we've learned so far, the package also supports decomposition that ease our task of breaking down trends in various degree of resolution:

```
prophet_plot_components(jokowi_model, jokowi_fc)
```



What did we learn from the decomposition above? Which day of week are people more likely to read facts and information about our President and what time of year happen to be more popular. As a homework, do you suspect the same trend for other politicians in Indonesia? Do you see a similar consumption behavior when we perform the analysis on the "Bahasa Indonesia" version of this article?

Annotations