Monthly Inflation Rates of Germany

Gian Carlo Sanfuego

2024-06-25

## Introduction

In this project, we will deal with a topic that affects us all on a daily basis. It is the inflation rate. By definition, the inflation rate is a measure of change in purchasing power. Macro economists and finance professionals love to work with inflation rates. Hence the sort of data is very often used in time series analysis. But on top of that, it is also important to know how the inflation moves, since it directly affects investment opportunities. Prices of stocks, property, precious metals or oil are at least indirectly affected by the rate of inflation.

This dataset deals with the monthly inflation rates of Germany from 2008 to October 2017. So it is almost ten years of monthly data. The source of the data is <https://www.statbureau.org/en/germany/inflation-tables>, which has a rich database of inflation rates available for many countries. We are using a month on month inflation rate here. Now, this is somewhat unusual because when you read the news generally the year on year inflation rate is used. To further understand the data, let’s now start setting our directory.

# Set working directory  
#getwd()  
#setwd('D:\Desktop\DATA ANALYSIS\R\TIME SERIES ANALYSIS AND FORECASTING IN R\PROJECT 2 - SEASONAL DATA')

# install and load readxl package to import the data in xlsx format  
#install.packages("readxl")  
library(readxl)

## Warning: package 'readxl' was built under R version 4.3.3

After installing and loading the necessary package, let us now read the excel file for this analysis in the directory that we set.

# Read the excel file  
ger\_inflation <- read\_excel("ger\_inflation.xlsx", col\_names = FALSE)

## New names:  
## • `` -> `...1`  
## • `` -> `...2`  
## • `` -> `...3`  
## • `` -> `...4`  
## • `` -> `...5`  
## • `` -> `...6`  
## • `` -> `...7`  
## • `` -> `...8`  
## • `` -> `...9`  
## • `` -> `...10`  
## • `` -> `...11`  
## • `` -> `...12`  
## • `` -> `...13`

To further understand the data, lets see the what’s in it.

# Display the data to understand its structure  
print(ger\_inflation)

## # A tibble: 10 × 13  
## ...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11 ...12 ...13  
## <dbl> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>  
## 1 2008 -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41   
## 2 2009 -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81   
## 3 2010 -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60   
## 4 2011 -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19   
## 5 2012 -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29   
## 6 2013 -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38   
## 7 2014 -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00   
## 8 2015 -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## 9 2016 -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74   
## 10 2017 -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 <NA> <NA>

str(ger\_inflation)

## tibble [10 × 13] (S3: tbl\_df/tbl/data.frame)  
## $ ...1 : num [1:10] 2008 2009 2010 2011 2012 ...  
## $ ...2 : chr [1:10] "-0.31" "-0.51" "-0.60" "-0.20" ...  
## $ ...3 : chr [1:10] "0.41" "0.61" "0.40" "0.60" ...  
## $ ...4 : chr [1:10] "0.51" "-0.20" "0.50" "0.59" ...  
## $ ...5 : chr [1:10] "-0.20" "0.10" "0.10" "0.00" ...  
## $ ...6 : chr [1:10] "0.61" "-0.10" "-0.10" "0.00" ...  
## $ ...7 : chr [1:10] "0.20" "0.30" "0.00" "0.10" ...  
## $ ...8 : chr [1:10] "0.61" "0.00" "0.20" "0.20" ...  
## $ ...9 : chr [1:10] "-0.30" "0.20" "0.10" "0.10" ...  
## $ ...10: chr [1:10] "-0.10" "-0.30" "-0.10" "0.20" ...  
## $ ...11: chr [1:10] "-0.20" "0.00" "0.10" "0.00" ...  
## $ ...12: chr [1:10] "-0.51" "-0.10" "0.10" "0.20" ...  
## $ ...13: chr [1:10] "0.41" "0.81" "0.60" "0.19" ...

As you can see, the first column of this dataset is a year variable. And the dataset is in tibble/dataframe format. Also, we can observe that all the variables are in character. Another thing that i observed here, is that the dataset is in wide format, meaning to say that rows represent the values in a single year, while columns represents all the months starting from january to december, that’s the reason why we have 12 variables, excluded the year. To convert it into a time series class, we have remove the year variable, change the format to numeric and then make it a vector first.

## Pre-processing

years <- ger\_inflation[[1]]  
inflation\_data <- ger\_inflation[, -1]  
inflation\_data

## # A tibble: 10 × 12  
## ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11 ...12 ...13  
## <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>  
## 1 -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41   
## 2 -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81   
## 3 -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60   
## 4 -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19   
## 5 -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29   
## 6 -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38   
## 7 -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00   
## 8 -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## 9 -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74   
## 10 -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 <NA> <NA>

Let’s now convert all variables at once to numeric using apply function.

# Convert the character data to numeric  
inflation\_data <- apply(inflation\_data, 2, as.numeric)  
inflation\_data

## ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11 ...12 ...13  
## [1,] -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41  
## [2,] -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81  
## [3,] -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60  
## [4,] -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19  
## [5,] -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29  
## [6,] -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38  
## [7,] -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00  
## [8,] -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## [9,] -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74  
## [10,] -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 NA NA

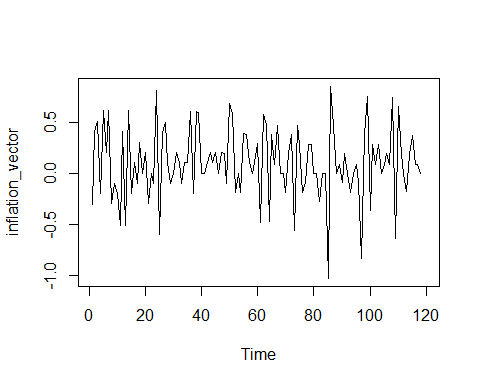
In R, time series functions typically operate on columns rather than rows. This is because time series data conventionally has time indices (dates, months, etc.) as rows, and observations (variables, measurements) as columns. Since our data is structured with time indices in rows, we need to transpose our data to align with this convention before applying time series analysis functions effectively.

# Reshape the data to a vector for time series conversion  
inflation\_vector <- as.vector(t(inflation\_data))  
inflation\_vector

## [1] -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41  
## [13] -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81  
## [25] -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60  
## [37] -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19  
## [49] -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29  
## [61] -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38  
## [73] -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00  
## [85] -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## [97] -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74  
## [109] -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 NA NA

We are done with our pre-processing steps. Now let’s try to plot the data to have an overview of what the time series dataset looks like, and to see what model is applicable to it.

plot.ts(inflation\_vector)



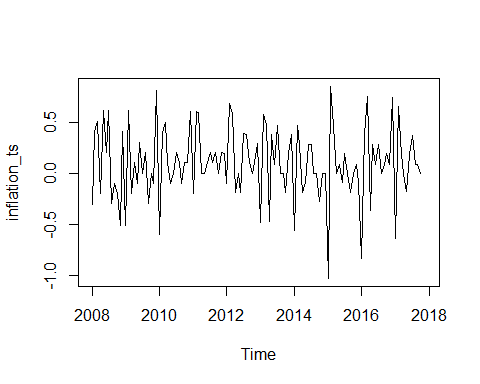
This initial plot tells us a lot about the data. With this knowledge, we can select a suitable model. We know what the problems could arise and know how to interpret results. First and foremost, this dataset has no trend. The mean stays constant over the whole series. That will surely affect the output of Arima or exponential smoothing methods. Second, this dataset is seasonal. After all, it is a monthly rate and this is also reflected on the plot where there are several observational time points over one year. Thirdly, this data set contains negative values. As you can see, the line goes towards both sides of this zero y mark. Negative values are critical in exponential smoothing, since they prohibit multiplicative exponential smoothing models. That means we are left with additive models only. Lastly, the up and down movement, amplitude seems stable over the series. Thus the variance is stable, which might affect the D parameter in an Arima model. Stable variance means that there is less differencing required, hence the low D, a low middle parameter in Arima.

So now that we have the data in R, we need to convert it to a time series object because functions and models we are using in time series analysis and forecasting requrie this format. the start argument specifies the start time for the time series. While the frequency argument specifies the frequency of the observations per year. Here, 12 indicates monthly data, meaning our time series has observations for each month (January to December).

# Create a time series object starting from January 2008  
inflation\_ts <- ts(inflation\_vector, start = 2008, frequency = 12)  
  
# Display the time series object  
print(inflation\_ts)

## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
## 2008 -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41  
## 2009 -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81  
## 2010 -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60  
## 2011 -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19  
## 2012 -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29  
## 2013 -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38  
## 2014 -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00  
## 2015 -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## 2016 -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74  
## 2017 -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 NA NA

# Plot the time series  
plot(inflation\_ts)



And indeed, this whole thing seems to work. The x axis looks good and we have a line chart which demonstrates that we got the Time series running.

## Modeling

# SEASONAL DECOMPOSITION

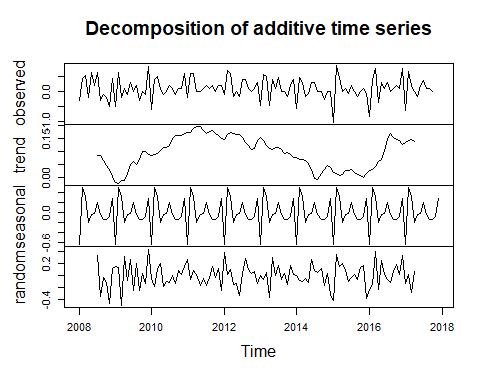
Let’start this with seasonal decomposition. But what does seasonal decomposition do? With seasonal decomposition, what this do is basically to divide the data into trend seasonality and remainder, which should be random.You either can choose a method which adds these three components up, or you can opt for a multiplicative model which multiplies them. In general, if this seasonal component stays constant over several cycles, it is best to opt for an additive decomposition. The strength of the method, however, is its ease of use and its simplicity.

decompose(inflation\_ts)

## $x  
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
## 2008 -0.31 0.41 0.51 -0.20 0.61 0.20 0.61 -0.30 -0.10 -0.20 -0.51 0.41  
## 2009 -0.51 0.61 -0.20 0.10 -0.10 0.30 0.00 0.20 -0.30 0.00 -0.10 0.81  
## 2010 -0.60 0.40 0.50 0.10 -0.10 0.00 0.20 0.10 -0.10 0.10 0.10 0.60  
## 2011 -0.20 0.60 0.59 0.00 0.00 0.10 0.20 0.10 0.20 0.00 0.20 0.19  
## 2012 -0.10 0.68 0.58 -0.19 0.00 -0.19 0.39 0.38 0.10 0.00 0.10 0.29  
## 2013 -0.48 0.57 0.48 -0.47 0.38 0.09 0.47 0.00 0.00 -0.19 0.19 0.38  
## 2014 -0.56 0.47 0.28 -0.19 -0.09 0.28 0.28 0.00 0.00 -0.28 0.00 0.00  
## 2015 -1.03 0.85 0.47 0.00 0.09 -0.09 0.19 0.00 -0.19 0.00 0.09 -0.09  
## 2016 -0.84 0.38 0.75 -0.37 0.28 0.09 0.28 0.00 0.09 0.19 0.09 0.74  
## 2017 -0.64 0.65 0.18 0.00 -0.18 0.18 0.37 0.09 0.09 0.00 NA NA  
##   
## $seasonal  
## Jan Feb Mar Apr May Jun  
## 2008 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2009 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2010 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2011 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2012 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2013 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2014 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2015 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2016 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## 2017 -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## Jul Aug Sep Oct Nov Dec  
## 2008 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2009 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2010 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2011 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2012 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2013 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2014 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2015 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2016 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
## 2017 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
##   
## $trend  
## Jan Feb Mar Apr May  
## 2008 NA NA NA NA NA  
## 2009 -0.016250000 -0.020833333 -0.008333333 -0.008333333 0.017083333  
## 2010 0.084166667 0.088333333 0.092500000 0.105000000 0.117500000  
## 2011 0.174166667 0.174166667 0.186666667 0.195000000 0.195000000  
## 2012 0.147083333 0.166666667 0.174166667 0.170000000 0.165833333  
## 2013 0.155833333 0.143333333 0.123333333 0.111250000 0.107083333  
## 2014 0.078750000 0.070833333 0.070833333 0.067083333 0.055416667  
## 2015 0.020416667 0.016666667 0.008750000 0.012500000 0.027916667  
## 2016 0.027916667 0.031666667 0.043333333 0.062916667 0.070833333  
## 2017 0.135416667 0.142916667 0.146666667 0.138750000 NA  
## Jun Jul Aug Sep Oct  
## 2008 NA 0.085833333 0.085833333 0.064583333 0.047500000  
## 2009 0.050833333 0.063750000 0.051250000 0.071666667 0.100833333  
## 2010 0.117083333 0.125000000 0.150000000 0.162083333 0.161666667  
## 2011 0.182083333 0.169166667 0.176666667 0.179583333 0.171250000  
## 2012 0.165833333 0.154166667 0.133750000 0.125000000 0.109166667  
## 2013 0.114583333 0.115000000 0.107500000 0.095000000 0.098333333  
## 2014 0.031666667 -0.003750000 -0.007500000 0.016250000 0.032083333  
## 2015 0.027916667 0.032083333 0.020416667 0.012500000 0.008750000  
## 2016 0.105416667 0.148333333 0.167916667 0.155416667 0.147083333  
## 2017 NA NA NA NA NA  
## Nov Dec  
## 2008 0.030416667 0.005000000  
## 2009 0.100833333 0.088333333  
## 2010 0.161666667 0.170000000  
## 2011 0.163333333 0.151250000  
## 2012 0.113333333 0.140833333  
## 2013 0.090416667 0.078750000  
## 2014 0.047500000 0.039583333  
## 2015 0.001250000 0.016666667  
## 2016 0.143333333 0.127916667  
## 2017 NA NA  
##   
## $random  
## Jan Feb Mar Apr May  
## 2008 NA NA NA NA NA  
## 2009 0.1440832369 0.1393610147 -0.5048982446 0.3135739776 -0.0830000965  
## 2010 -0.0463334298 -0.1798056520 0.0942684221 0.2002406443 -0.1834167631  
## 2011 0.2636665702 -0.0656389853 0.0901017554 0.0102406443 -0.1609167631  
## 2012 0.3907499035 0.0218610147 0.0926017554 -0.1547593557 -0.1317500965  
## 2013 0.0019999035 -0.0648056520 0.0434350887 -0.3760093557 0.3069999035  
## 2014 -0.0009167631 -0.0923056520 -0.1040649113 -0.0518426890 -0.1113334298  
## 2015 -0.4125834298 0.3418610147 0.1480184221 0.1927406443 0.0961665702  
## 2016 -0.2300834298 -0.1431389853 0.3934350887 -0.2276760224 0.2432499035  
## 2017 -0.1375834298 0.0156110147 -0.2798982446 0.0664906443 NA  
## Jun Jul Aug Sep Oct  
## 2008 NA 0.3288980517 -0.3437408372 -0.0362408372 -0.1108704668  
## 2009 0.2730936535 -0.2590186150 0.1908424961 -0.2433241705 0.0357961998  
## 2010 -0.0931563465 -0.1202686150 -0.0079075039 -0.1337408372 0.0749628665  
## 2011 -0.0581563465 -0.1644352816 -0.0345741705 0.1487591628 -0.0346204668  
## 2012 -0.3319063465 0.0405647184 0.2883424961 0.1033424961 0.0274628665  
## 2013 -0.0006563465 0.1597313850 -0.0654075039 0.0333424961 -0.1517038002  
## 2014 0.2722603202 0.0884813850 0.0495924961 0.1120924961 -0.1754538002  
## 2015 -0.0939896798 -0.0373519483 0.0216758295 -0.0741575039 0.1278795332  
## 2016 0.0085103202 -0.0636019483 -0.1258241705 0.0629258295 0.1795461998  
## 2017 NA NA NA NA NA  
## Nov Dec  
## 2008 -0.4665186150 0.1229258295  
## 2009 -0.1269352816 0.4395924961  
## 2010 0.0122313850 0.1479258295  
## 2011 0.1105647184 -0.2433241705  
## 2012 0.0605647184 -0.1329075039  
## 2013 0.1734813850 0.0191758295  
## 2014 0.0263980517 -0.3216575039  
## 2015 0.1626480517 -0.3887408372  
## 2016 0.0205647184 0.3300091628  
## 2017 NA NA  
##   
## $figure  
## [1] -0.63783324 0.49147232 0.31323158 -0.20524064 -0.03408324 -0.02392699  
## [7] 0.19526861 -0.04209250 -0.12834250 -0.13662953 -0.07389805 0.28207417  
##   
## $type  
## [1] "additive"  
##   
## attr(,"class")  
## [1] "decomposed.ts"

If we run decompose on our German inflation dataset, we get the whole numbers for each of the three components. The data was divided in two. Note the NA’s at the beginning and the end of the time series due to the nature of the calculations. These observations are not available, which might cause problems when doing further analysis with this data. Now we can simply plot the whole thing to make it more visual. We just need to wrap the decompose function in a plot command and we are good to go.

plot(decompose(inflation\_ts))



This is the data we previously got with the decompose function. The first one on top shows the original dataset. Then we get the trend component. Next we see how the seasonal part looks and the rest is white noise. This is interesting info since it tells me that there is no trend in the series. The whole trend component makes a half circular move from 2009 to 2015 and starts to rise again in late 2016. But overall there is no clear trend direction visible.

The interesting part here is the seasonal component. One seasonal cycle is calculated and used for all years. The series starts with a low January inflation, which is instantly compensated for in February. Overall, this pattern makes perfect sense since the summer holidays as well as the Christmas period are times of great consumption. Hence, it makes sense for the retailers to adjust the prices.

You can also use some new versions of decomposition methods such as X11, Seats and STL method. The strong point with these methods is that we can now get model values for all observations, meaning to say, no more noise. Furthermore, the models allow the seasonal part to be adjusted over time, unlike the standard decompose function where the seasonal part stays constant throughout the model. So let’s take a look at such a method.

# Remove NA values  
inflation\_ts.nona <- na.omit(inflation\_ts)

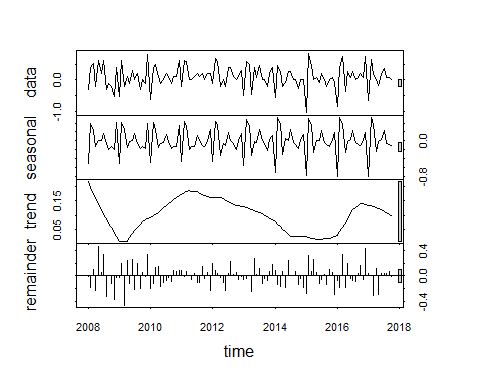
Removing first the NA will make the STL function work.

# Load the forecast package to use the stl method  
library(forecast)

## Warning: package 'forecast' was built under R version 4.3.3

## Registered S3 method overwritten by 'quantmod':  
## method from  
## as.zoo.data.frame zoo

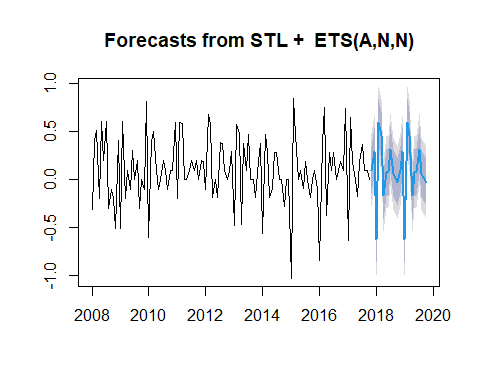
plot(stl(inflation\_ts.nona, s.window = 7))



So if we run this line, we can see that the seasonal patterns do change over time. The inflation amplitude gets wider towards the 2017 end of the series. With the change of the seasonal cycles, the patterns do not necessarily stay constant with this method. And this of course affects the trend part as well. The trend curve gets rounder overall, and the last part of the curve has a smaller amplitude than with a decompose function. This is due to the fact that the seasonal part was already increased, so this needs to be subtracted from the trend part.

Now let’s use STL decomposition for forecasting.

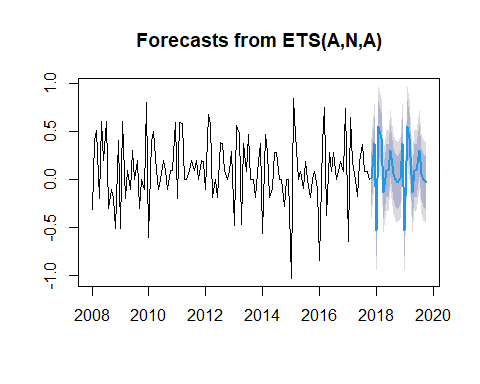
# stl forecasting  
plot(stlf(inflation\_ts.nona, method = 'ets'))



As you can see, the forecasted intervals seem to be identical. The peaks and lows are at the same height in both years. We can compare this with a standard ETS forecast of the same length. Therefore, we simply get the plot of the forecast of ETS with an age of 24.

# Comparison with a standard ets forecast  
plot(forecast(ets(inflation\_ts), h = 24))

## Warning in ets(inflation\_ts): Missing values encountered. Using longest  
## contiguous portion of time series



We can see that here the two intervals are the same again, but the amplitudes are slightly different than before. The December peaks are higher than the July ones, which is the opposite with the STL method. Although we used an ETS system for both forecasts with the STLF function, the forecast is a combined one.

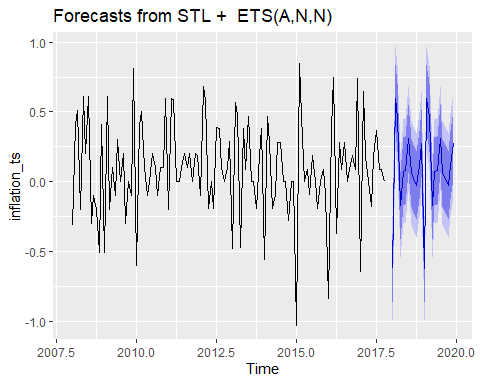
Whereas if we use ETS exponential smoothing on itself, the model is different. With the STLF function we get an ETS(A,N,N), which means additive season, no trend and no error or remainder. The remainder is handled by the STL part of the model with ETS on its own we get an A which is additive seasonality, no trend but an additive error. The underlying model for ETS, you can see in the headers of the plot, that way we can explain that the models do have some differences on the forecasting part.

# Using autoplot  
library(ggplot2)

## Warning: package 'ggplot2' was built under R version 4.3.3

autoplot(stlf(inflation\_ts, method = 'ets'))

## Warning: Removed 2 rows containing missing values or values outside the scale range  
## (`geom\_line()`).



Like with most of the plots we are using here, we can as well use the auto plot function available in packages ggplot2 and forecast, which gives probably a slightly better visualization. Especially the background lines allow a more accurate estimation of the y axis values.

# SEASONAL ARIMA

Now there are basically two methods how to set up an Arima model in R, a manual method where you find out the parameters step by step by using ACF and Pacf plots and differencing the dataset if required. But as you might imagine, the process complexity is a lot higher. But luckily for us there is also the automated method with the auto.arima() function which we owe to Rob Hindman and his forecast package.

# Seasonal Arima (package forecast)  
auto.arima(inflation\_ts, stepwise = T,  
 approximation = F, trace = T)

##   
## ARIMA(2,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,0)[12] with drift : 31.97975  
## ARIMA(1,0,0)(1,1,0)[12] with drift : 22.54641  
## ARIMA(0,0,1)(0,1,1)[12] with drift : 6.801821  
## ARIMA(0,0,0)(0,1,0)[12] : 29.92568  
## ARIMA(0,0,1)(0,1,0)[12] with drift : 29.30375  
## ARIMA(0,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,1)(1,1,0)[12] with drift : 23.5733  
## ARIMA(0,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,1)[12] with drift : 5.572693  
## ARIMA(0,0,0)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(1,1,0)[12] with drift : 23.12354  
## ARIMA(0,0,0)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,0)(0,1,1)[12] with drift : 6.390183  
## ARIMA(1,0,1)(0,1,1)[12] with drift : 1.682711  
## ARIMA(1,0,1)(0,1,0)[12] with drift : 22.07834  
## ARIMA(1,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,1)(1,1,0)[12] with drift : 17.05319  
## ARIMA(1,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,1)[12] with drift : 0.6937448  
## ARIMA(2,0,1)(0,1,0)[12] with drift : 22.8497  
## ARIMA(2,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(1,1,0)[12] with drift : 15.85279  
## ARIMA(2,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,0)(0,1,1)[12] with drift : 2.297761  
## ARIMA(3,0,1)(0,1,1)[12] with drift : 2.580427  
## ARIMA(2,0,2)(0,1,1)[12] with drift : 0.2340872  
## ARIMA(2,0,2)(0,1,0)[12] with drift : Inf  
## ARIMA(2,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,2)(1,1,0)[12] with drift : 16.68961  
## ARIMA(2,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,1)[12] with drift : 0.0488431  
## ARIMA(1,0,2)(0,1,0)[12] with drift : 22.52145  
## ARIMA(1,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(1,1,0)[12] with drift : 15.08675  
## ARIMA(1,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,2)(0,1,1)[12] with drift : 4.437503  
## ARIMA(1,0,3)(0,1,1)[12] with drift : 1.945777  
## ARIMA(0,0,3)(0,1,1)[12] with drift : 4.681465  
## ARIMA(2,0,3)(0,1,1)[12] with drift : 2.305476  
## ARIMA(1,0,2)(0,1,1)[12] : -2.165707  
## ARIMA(1,0,2)(0,1,0)[12] : 20.35976  
## ARIMA(1,0,2)(1,1,1)[12] : Inf  
## ARIMA(1,0,2)(0,1,2)[12] : Inf  
## ARIMA(1,0,2)(1,1,0)[12] : 12.87818  
## ARIMA(1,0,2)(1,1,2)[12] : Inf  
## ARIMA(0,0,2)(0,1,1)[12] : 2.269405  
## ARIMA(1,0,1)(0,1,1)[12] : -0.4490879  
## ARIMA(2,0,2)(0,1,1)[12] : Inf  
## ARIMA(1,0,3)(0,1,1)[12] : -0.3068146  
## ARIMA(0,0,1)(0,1,1)[12] : 4.749055  
## ARIMA(0,0,3)(0,1,1)[12] : 2.485182  
## ARIMA(2,0,1)(0,1,1)[12] : -1.521243  
## ARIMA(2,0,3)(0,1,1)[12] : -0.01360729  
##   
## Best model: ARIMA(1,0,2)(0,1,1)[12]

## Series: inflation\_ts   
## ARIMA(1,0,2)(0,1,1)[12]   
##   
## Coefficients:  
## ar1 ma1 ma2 sma1  
## -0.7944 0.7783 0.2114 -0.7618  
## s.e. 0.1201 0.1470 0.1013 0.1291  
##   
## sigma^2 = 0.04883: log likelihood = 6.38  
## AIC=-2.77 AICc=-2.17 BIC=10.55

# Assign it to an object  
inflation\_ts.arima <- auto.arima(inflation\_ts,  
 stepwise = T,  
 approximation = F,  
 trace = T)

##   
## ARIMA(2,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,0)[12] with drift : 31.97975  
## ARIMA(1,0,0)(1,1,0)[12] with drift : 22.54641  
## ARIMA(0,0,1)(0,1,1)[12] with drift : 6.801821  
## ARIMA(0,0,0)(0,1,0)[12] : 29.92568  
## ARIMA(0,0,1)(0,1,0)[12] with drift : 29.30375  
## ARIMA(0,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,1)(1,1,0)[12] with drift : 23.5733  
## ARIMA(0,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,1)[12] with drift : 5.572693  
## ARIMA(0,0,0)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(1,1,0)[12] with drift : 23.12354  
## ARIMA(0,0,0)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,0)(0,1,1)[12] with drift : 6.390183  
## ARIMA(1,0,1)(0,1,1)[12] with drift : 1.682711  
## ARIMA(1,0,1)(0,1,0)[12] with drift : 22.07834  
## ARIMA(1,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,1)(1,1,0)[12] with drift : 17.05319  
## ARIMA(1,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,1)[12] with drift : 0.6937448  
## ARIMA(2,0,1)(0,1,0)[12] with drift : 22.8497  
## ARIMA(2,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(1,1,0)[12] with drift : 15.85279  
## ARIMA(2,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,0)(0,1,1)[12] with drift : 2.297761  
## ARIMA(3,0,1)(0,1,1)[12] with drift : 2.580427  
## ARIMA(2,0,2)(0,1,1)[12] with drift : 0.2340872  
## ARIMA(2,0,2)(0,1,0)[12] with drift : Inf  
## ARIMA(2,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,2)(1,1,0)[12] with drift : 16.68961  
## ARIMA(2,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,1)[12] with drift : 0.0488431  
## ARIMA(1,0,2)(0,1,0)[12] with drift : 22.52145  
## ARIMA(1,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(1,1,0)[12] with drift : 15.08675  
## ARIMA(1,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,2)(0,1,1)[12] with drift : 4.437503  
## ARIMA(1,0,3)(0,1,1)[12] with drift : 1.945777  
## ARIMA(0,0,3)(0,1,1)[12] with drift : 4.681465  
## ARIMA(2,0,3)(0,1,1)[12] with drift : 2.305476  
## ARIMA(1,0,2)(0,1,1)[12] : -2.165707  
## ARIMA(1,0,2)(0,1,0)[12] : 20.35976  
## ARIMA(1,0,2)(1,1,1)[12] : Inf  
## ARIMA(1,0,2)(0,1,2)[12] : Inf  
## ARIMA(1,0,2)(1,1,0)[12] : 12.87818  
## ARIMA(1,0,2)(1,1,2)[12] : Inf  
## ARIMA(0,0,2)(0,1,1)[12] : 2.269405  
## ARIMA(1,0,1)(0,1,1)[12] : -0.4490879  
## ARIMA(2,0,2)(0,1,1)[12] : Inf  
## ARIMA(1,0,3)(0,1,1)[12] : -0.3068146  
## ARIMA(0,0,1)(0,1,1)[12] : 4.749055  
## ARIMA(0,0,3)(0,1,1)[12] : 2.485182  
## ARIMA(2,0,1)(0,1,1)[12] : -1.521243  
## ARIMA(2,0,3)(0,1,1)[12] : -0.01360729  
##   
## Best model: ARIMA(1,0,2)(0,1,1)[12]

According to the results, the best model is Arima(1,0,2)(0,1,1) of 12 months. That means for the non seasonal part, we have one order of autocorrelation, no differencing and two orders of moving averages. The seasonal part has one step of differencing and one order of moving average. Our M is 12. That means 12 observations per interval, which is a year. In this case, the corresponding coefficients for the model are available down here. Autoregressive coefficient one is -0.79. The moving average one is 0.778 and moving average two is 0.211. So these are the coefficients for the non seasonal part and the one coefficient for the seasonal moving average is -0.761. So the seasonal coefficients always have the prefix s for seasonal. The model indicators like info criteria are available at the bottom of the output, much like in a standard Arima model.

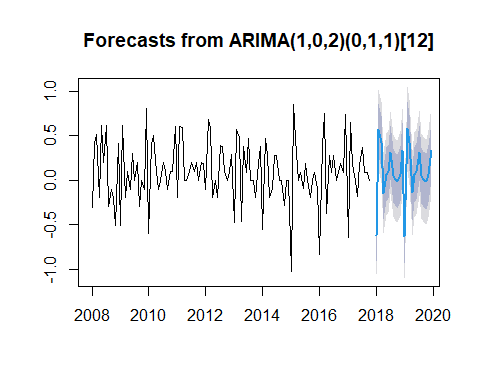
The trace argument in the function list all alternative models that R tested for us, however, the best model with lowest AIC is the best model itself, so we will now proceed to our analysis. Just remember that the lower info criteria, the better.

Now we can simply use the standard forecasting procedure to use this model for a forecast.

# Assign it to an object  
inflation\_ts.arima <- auto.arima(inflation\_ts,  
 stepwise = T,  
 approximation = F,  
 trace = T)

##   
## ARIMA(2,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,0)[12] with drift : 31.97975  
## ARIMA(1,0,0)(1,1,0)[12] with drift : 22.54641  
## ARIMA(0,0,1)(0,1,1)[12] with drift : 6.801821  
## ARIMA(0,0,0)(0,1,0)[12] : 29.92568  
## ARIMA(0,0,1)(0,1,0)[12] with drift : 29.30375  
## ARIMA(0,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,1)(1,1,0)[12] with drift : 23.5733  
## ARIMA(0,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,1)[12] with drift : 5.572693  
## ARIMA(0,0,0)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(1,1,0)[12] with drift : 23.12354  
## ARIMA(0,0,0)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,0)(0,1,1)[12] with drift : 6.390183  
## ARIMA(1,0,1)(0,1,1)[12] with drift : 1.682711  
## ARIMA(1,0,1)(0,1,0)[12] with drift : 22.07834  
## ARIMA(1,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,1)(1,1,0)[12] with drift : 17.05319  
## ARIMA(1,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,1)[12] with drift : 0.6937448  
## ARIMA(2,0,1)(0,1,0)[12] with drift : 22.8497  
## ARIMA(2,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(1,1,0)[12] with drift : 15.85279  
## ARIMA(2,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,0)(0,1,1)[12] with drift : 2.297761  
## ARIMA(3,0,1)(0,1,1)[12] with drift : 2.580427  
## ARIMA(2,0,2)(0,1,1)[12] with drift : 0.2340872  
## ARIMA(2,0,2)(0,1,0)[12] with drift : Inf  
## ARIMA(2,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,2)(1,1,0)[12] with drift : 16.68961  
## ARIMA(2,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,1)[12] with drift : 0.0488431  
## ARIMA(1,0,2)(0,1,0)[12] with drift : 22.52145  
## ARIMA(1,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(1,1,0)[12] with drift : 15.08675  
## ARIMA(1,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,2)(0,1,1)[12] with drift : 4.437503  
## ARIMA(1,0,3)(0,1,1)[12] with drift : 1.945777  
## ARIMA(0,0,3)(0,1,1)[12] with drift : 4.681465  
## ARIMA(2,0,3)(0,1,1)[12] with drift : 2.305476  
## ARIMA(1,0,2)(0,1,1)[12] : -2.165707  
## ARIMA(1,0,2)(0,1,0)[12] : 20.35976  
## ARIMA(1,0,2)(1,1,1)[12] : Inf  
## ARIMA(1,0,2)(0,1,2)[12] : Inf  
## ARIMA(1,0,2)(1,1,0)[12] : 12.87818  
## ARIMA(1,0,2)(1,1,2)[12] : Inf  
## ARIMA(0,0,2)(0,1,1)[12] : 2.269405  
## ARIMA(1,0,1)(0,1,1)[12] : -0.4490879  
## ARIMA(2,0,2)(0,1,1)[12] : Inf  
## ARIMA(1,0,3)(0,1,1)[12] : -0.3068146  
## ARIMA(0,0,1)(0,1,1)[12] : 4.749055  
## ARIMA(0,0,3)(0,1,1)[12] : 2.485182  
## ARIMA(2,0,1)(0,1,1)[12] : -1.521243  
## ARIMA(2,0,3)(0,1,1)[12] : -0.01360729  
##   
## Best model: ARIMA(1,0,2)(0,1,1)[12]

forec <- forecast(inflation\_ts.arima)  
plot(forec)



As you can see, we get two seasonal cycles as the forecast period. The cycles are identical and they would stay like that even if you have 5 or 10 cycles. This is fundamentally a linear model type and there is no trend or something similar in the model. Hence there is no variation which could be used to alter the cycles over time. So as you can see, Arima is a viable option for a seasonal data set due to the auto.arima() function, it is fairly easy to calculate a suitable model. However, when you have a dataset like this, it is also advisable to compare against an exponential smoothing model like Holt-winters or an ETS model. Exponential smoothing models are generally an excellent alternative when it comes to seasonal datasets.

Let’s try it now with Exponential smoothing.

## EXPONENTIAL SMOOTHING WITH ETS

If we want to model this data set with the exponential smoothing system, there are two main ways in how to do this. The classic exponential smoothing method used for a seasonal data set is called Holt-winters seasonal method with a corresponding hw() function. The standard alternative to this method would be the ETS method with the ETS function. No matter which of these two systems you go for, the underlying principle is always the same. We will use the ETS() function because it is more efficient since it is auto generated.

# Auto generated  
ets(inflation\_ts)

## Warning in ets(inflation\_ts): Missing values encountered. Using longest  
## contiguous portion of time series

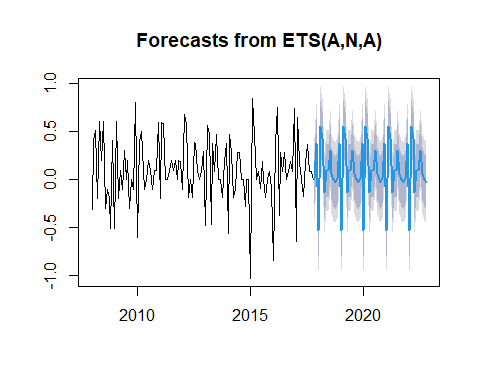
## ETS(A,N,A)   
##   
## Call:  
## ets(y = inflation\_ts)   
##   
## Smoothing parameters:  
## alpha = 1e-04   
## gamma = 1e-04   
##   
## Initial states:  
## l = 0.1021   
## s = 0.2741 -0.0868 -0.1336 -0.1038 -0.0377 0.2009  
## -0.0085 -0.0111 -0.2365 0.323 0.4503 -0.6303  
##   
## sigma: 0.2145  
##   
## AIC AICc BIC   
## 214.7276 219.4335 256.2878

The model is preset to automate it so there is nothing we need to adjust here. It is really easy to use it with the default settings, but nonetheless the results are generally quite good. The suggested model is essentially an additive, one without the trend. And that is pretty much what we already learned from seasonal decomposition. There is no trend in the data. Therefore, we have an N at the T component. The variations in the dataset are mainly due to additive seasonal patterns, hence the A at the end. Furthermore, due to the nature of this time series, it is not even possible to get a multiplicative model. Multiplicative does not work when you have negative or null values in your data. Therefore, this is even the only solution possible given the dataset. We get the corresponding smoothing parameters. Alpha for the error and gamma for the seasonality, which are both close to zero, indicating a model which takes even older data into account. We get standard additional info down here like the info criteria or the initial levels which are used to calculate the forecasts.

# Forecast plot  
inflation\_ts.ets <- ets(inflation\_ts)

## Warning in ets(inflation\_ts): Missing values encountered. Using longest  
## contiguous portion of time series

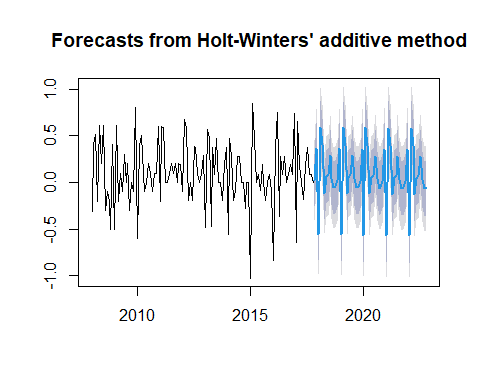
plot(forecast(inflation\_ts.ets, h = 60))



The seasonal patterns stay constant over the five years. By the way, the holt-winters forecast looks a bit more smooth as we can see here. As we initially discussed in this video, Holt-winters would be the alternative to this model since hw() allows a seasonal data set. The plot shows holt-winters of the same length. This automatically generates a forecast. Both models are based on an additive seasonality.

# Comparison with seasonal Holt Winters model  
plot(hw(inflation\_ts, h = 60))

## Warning in ets(x, "AAA", alpha = alpha, beta = beta, gamma = gamma, phi = phi,  
## : Missing values encountered. Using longest contiguous portion of time series



## Model Evaluation

Now since we have several models calculated, it is, of course, paramount to know which of the models is the best one. Of course, it is not a good idea to just use info criteria to compare different systems of models. Arima And exponential smoothing cannot be compared just with information criteria. The underlying statistics used to calculate the info criteria are different for both models systems. Therefore, we have to concentrate on prediction or forecast accuracy. Mean squared errors from forecast residuals have proven to be a measure of choice to compare Arima and exponential smoothing methods. This time, we will use the function tscv() function for time series cross validation.

## Cross Validation of 2 models  
inflation\_ts.ets = ets(inflation\_ts)

## Warning in ets(inflation\_ts): Missing values encountered. Using longest  
## contiguous portion of time series

inflation\_ts.arima = auto.arima(inflation\_ts,   
 stepwise = T,   
 approximation = F,   
 trace = T)

##   
## ARIMA(2,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,0)[12] with drift : 31.97975  
## ARIMA(1,0,0)(1,1,0)[12] with drift : 22.54641  
## ARIMA(0,0,1)(0,1,1)[12] with drift : 6.801821  
## ARIMA(0,0,0)(0,1,0)[12] : 29.92568  
## ARIMA(0,0,1)(0,1,0)[12] with drift : 29.30375  
## ARIMA(0,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,1)(1,1,0)[12] with drift : 23.5733  
## ARIMA(0,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,1)[12] with drift : 5.572693  
## ARIMA(0,0,0)(1,1,1)[12] with drift : Inf  
## ARIMA(0,0,0)(0,1,2)[12] with drift : Inf  
## ARIMA(0,0,0)(1,1,0)[12] with drift : 23.12354  
## ARIMA(0,0,0)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,0)(0,1,1)[12] with drift : 6.390183  
## ARIMA(1,0,1)(0,1,1)[12] with drift : 1.682711  
## ARIMA(1,0,1)(0,1,0)[12] with drift : 22.07834  
## ARIMA(1,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,1)(1,1,0)[12] with drift : 17.05319  
## ARIMA(1,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,1)[12] with drift : 0.6937448  
## ARIMA(2,0,1)(0,1,0)[12] with drift : 22.8497  
## ARIMA(2,0,1)(1,1,1)[12] with drift : Inf  
## ARIMA(2,0,1)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,1)(1,1,0)[12] with drift : 15.85279  
## ARIMA(2,0,1)(1,1,2)[12] with drift : Inf  
## ARIMA(2,0,0)(0,1,1)[12] with drift : 2.297761  
## ARIMA(3,0,1)(0,1,1)[12] with drift : 2.580427  
## ARIMA(2,0,2)(0,1,1)[12] with drift : 0.2340872  
## ARIMA(2,0,2)(0,1,0)[12] with drift : Inf  
## ARIMA(2,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(2,0,2)(1,1,0)[12] with drift : 16.68961  
## ARIMA(2,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,1)[12] with drift : 0.0488431  
## ARIMA(1,0,2)(0,1,0)[12] with drift : 22.52145  
## ARIMA(1,0,2)(1,1,1)[12] with drift : Inf  
## ARIMA(1,0,2)(0,1,2)[12] with drift : Inf  
## ARIMA(1,0,2)(1,1,0)[12] with drift : 15.08675  
## ARIMA(1,0,2)(1,1,2)[12] with drift : Inf  
## ARIMA(0,0,2)(0,1,1)[12] with drift : 4.437503  
## ARIMA(1,0,3)(0,1,1)[12] with drift : 1.945777  
## ARIMA(0,0,3)(0,1,1)[12] with drift : 4.681465  
## ARIMA(2,0,3)(0,1,1)[12] with drift : 2.305476  
## ARIMA(1,0,2)(0,1,1)[12] : -2.165707  
## ARIMA(1,0,2)(0,1,0)[12] : 20.35976  
## ARIMA(1,0,2)(1,1,1)[12] : Inf  
## ARIMA(1,0,2)(0,1,2)[12] : Inf  
## ARIMA(1,0,2)(1,1,0)[12] : 12.87818  
## ARIMA(1,0,2)(1,1,2)[12] : Inf  
## ARIMA(0,0,2)(0,1,1)[12] : 2.269405  
## ARIMA(1,0,1)(0,1,1)[12] : -0.4490879  
## ARIMA(2,0,2)(0,1,1)[12] : Inf  
## ARIMA(1,0,3)(0,1,1)[12] : -0.3068146  
## ARIMA(0,0,1)(0,1,1)[12] : 4.749055  
## ARIMA(0,0,3)(0,1,1)[12] : 2.485182  
## ARIMA(2,0,1)(0,1,1)[12] : -1.521243  
## ARIMA(2,0,3)(0,1,1)[12] : -0.01360729  
##   
## Best model: ARIMA(1,0,2)(0,1,1)[12]

Now, with our seasonal inflation data set, we used seasonal decomposition with stlf(), we used seasonal arima and we used ETS for exponential smoothing. The forecast with stlf() contained, also the ETS method. Therefore, if you compare ETS alone and stlf() with ETS, you will find that the errors are nearly the same. Therefore, we will focus on the comparison of a seasonal arima model and an ETS model for our Germany inflation dataset.

For the tscv() function. You need the model and the dataset. I would recommend to set up a function for the model which is then easily fed into tscv() and you can generate the mean of the error rates to compare the models. Therefore it will be a three step process. Writing the function with the underlying model. Running the cross-validation and computing the error rate. So let’s start with creating the two functions for the two models.

forecastets = function(x, h) {  
 forecast(ets(x), h = h)  
}  
  
forecastarima = function(x, h) {  
 forecast(auto.arima(x), stepwise = T, approximation = F, h=h)  
}

Next we feed these models into the tscv() function from the package forecast.

etserror = tsCV(inflation\_ts, forecastets, h=1)  
arimaerror = tsCV(inflation\_ts, forecastarima, h=1)

So these two lines will now generate the forecast errors for each of the observations step by step, and therefore it will take a while to compute all these measures depending on the length of your dataset or the capacity of your machine.

mean(etserror^2, na.rm=TRUE)

## [1] 0.09073782

mean(arimaerror^2, na.rm=TRUE)

## [1] 0.07700064

We have these two numbers down here, 0.0912 for the ETS model and 0.0795 for the Arima model. The better model is the one with the lower error rate, which is the Arima model in this case. Logically, with all of these error and accuracy indicators, you want to pick the lowest one.

In this case we compared ETS and seasonal Arima on our German inflation data set and we found out that the Arima model is the better fit.