Time Series Exploration and Exponential Smoothing with R

Nikolaos Kourentzes (nikolaos@kourentes.com)

Contents

1 Basic R computations

In this workshop we will learn some basic forecasting in R. To do this we will also get used to some basic R syntax and computations.

To execute a line in R we place the cursor at that line and press **Ctrl+Enter** (or **Command+Enter** in Mac OS). We can also select part of the code (including multiple lines) and execute it in the same way. Anything after `#` is a comment, so it is not executed. Any code that is executed and its result will be reported in the console window. The lab notes are written so that you can copy and paste code from this page to R and execute them. You can compare the results with the outputs reported here.

```
# This line is a comment! The following line is not
1+1
1+1 # I can also have comments, together with commands. Anything after # is ignored.
```

To assign a value to a variable in memory we use the symbol `<-`. For example to assign to variable `a` the value 5 we write:

```
a <- 5
```

If you write `A = 5`, you will get the same result, but it is good practice to avoid using the symbol `=`, as this is used for a different purpose in R (to assign values to functions).

Let us assign the value 2 to variable `b`:

```
b <- 2
```

Now we can perform some basic computations, like addition, subtraction, multiplication, division:

```
a + b  # addition
```

```
## [1] 7
```

```
a - b  # subtraction
```

```
## [1] 3
```

```
a * b  # multiplication
```

```
## [1] 10
```

```
a / b  # division
```

```
## [1] 2.5
```

We can also store the result in a new variable `total`, like this:

```
total <- a + b
```

Note that the result is not printed on the console, as it is stored to a variable instead. We can get the result by simply typing `total` or `print(total)`:

```
total
```

```
## [1] 7
```

```
print(total)
```

```
## [1] 7
```

The command `print()` is a function, which is some code to do a specific task. You will know that I am referring to a function because it will always be followed by (). All functions have a common syntax, that is function(...), i.e. the function name, followed by brackets that contain the input arguments. If there are several arguments, these are separated by commas. For example:

```
sum(a,b)
```

```
## [1] 7
```

To compare two results I can use `==`, for example:

```
total == sum(a,b)
```

```
## [1] TRUE
```

```
total == a - b
```

```
## [1] FALSE
```

A final very useful command that we will see before we jump into some forecasting is `c()`. This is useful to collect several values/variables into a single variable. For example:

```
c(5,8)
```

```
## [1] 5 8
```

```
all <- c(a,b,total)
print(all)
```

```
## [1] 5 2 7
```

For any function, you can type it, select it and press F1 to bring up its help page, or type it with a `?` in front. For example:

```
?sum
```

2 Loading packages into R

R is a great analytics tool because of the vast number of packages that implement rather complicated code in an easy to use way. There are quite a few useful packages for time series forecasting, the most popular is **forecast**. A list of

the various packages useful for forecasting can be found here. We will also be using the package **tsutils** to help us with time series exploration tasks.

We load a pre-installed package in the memory using the `library()`. This will most probably give an error, but it is fine, read on to understand why!

```
library(forecast)
```

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

There is a good chance that the first time you want to use a package it is not installed on your computer. In this case, you need to install the package manually. This is a one-off process and you do not need to re-install the package again. However, from time to time packages are updated to introduce new functionalities and fix bugs. Installing and updating packages in Rstudio is very easy and can be done with the command:

```
install.packages('forecast')
# We still need to load the package
library(forecast)
# We repeat the same for the tsutils package
install.packages('tsutils'); library(tsutils)
# Note that I used ; to have several commands in the same line.
```

We can get documentation about a package using:

```
?'forecast-package'
```

Packages typically come with several functions and sometimes datasets.

3 Loading data into R

There are many alternatives on how to load data in R. A very easy way is to save your data in a comma-delimited data file (.csv) in Excel and then load it to R. You can import Excel files directly or use .Rdata file, which is the native file format for R. Lookup online for ready code to load data from whatever source you need. There are way far too many tutorials online!

We will load some data from a prepared .csv file and store the data into a variable called `Y`. (Use the Rstudio menu to find the right path for the dataset. Go to *Session -> Set Working Directory -> Choose Directory...* and select the folder where you have saved the data.)

```
Y <- read.csv("./workshop1R.csv")
print(Y) # The result is not shown here!
```

The file contains 8 columns, and each column is a time series of 60 months long. To get the names of the columns/time series:

```
colnames(Y)
```

```
## [1] "Level_A"     "Level_B"     "LevelShift"  "Trend_A"     "Trend_B"
## [6] "Season_A"    "Season_B"    "TrendSeason"
```

As you can see, we have to model some series with a known structure. Nonetheless, we will treat them as series with unknown structure. As an example, we will model one of them, Level_A. Using the same approach you can model the rest.

First, let us take that series from the matrix of data. We will store in variable `y` (in contrast to capital `Y` that has all series).

```
y <- Y[,1]
print(y)
```

```
##  [1] 309.5927 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226
##  [9] 302.5102 319.5777 332.5051 342.4510 339.0753 334.1232 361.4546 336.1173
```

```
## [17] 319.6460 317.3951 328.0461 304.5760 355.3091 330.1500 342.8376 327.5952
## [25] 328.0343 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126
## [33] 345.3493 313.4708 304.8354 311.6021 307.4285 347.2009 326.6501 308.4443
## [41] 299.1128 341.9722 302.1470 295.1502 351.9557 332.8738 324.6155 306.7664
## [49] 320.3624 350.8271 317.0764 300.9870 336.8473 316.1264 321.9198 336.2565
## [57] 335.7531 332.0578 324.6613 352.9658
```

The syntax `[,1]` means take all rows and the first column of `Y`. We could for example say we want rows 1-10 and columns 3-4 by typing `Y[1:10,3:4]`. Or we could take rows 1 and 5 from all columns by typing `Y[c(1,5),]`. Give these a try to understand the logic.

If want to try another series you simply choose a different column in `Y`, for example, `Y[,2]`. For the sake of this example, let us proceed with the first column for now.

Currently `y` is a vector of values. Let us tell R that this is a time series by using the function `ts()`

```
y <- ts(y,frequency=12)
print(y)
```

```
##        Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 1 309.5927 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226
## 2 339.0753 334.1232 361.4546 336.1173 319.6460 317.3951 328.0461 304.5760
## 3 328.0343 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126
## 4 307.4285 347.2009 326.6501 308.4443 299.1128 341.9722 302.1470 295.1502
## 5 320.3624 350.8271 317.0764 300.9870 336.8473 316.1264 321.9198 336.2565
##        Sep      Oct      Nov      Dec
## 1 302.5102 319.5777 332.5051 342.4510
## 2 355.3091 330.1500 342.8376 327.5952
## 3 345.3493 313.4708 304.8354 311.6021
## 4 351.9557 332.8738 324.6155 306.7664
## 5 335.7531 332.0578 324.6613 352.9658
```

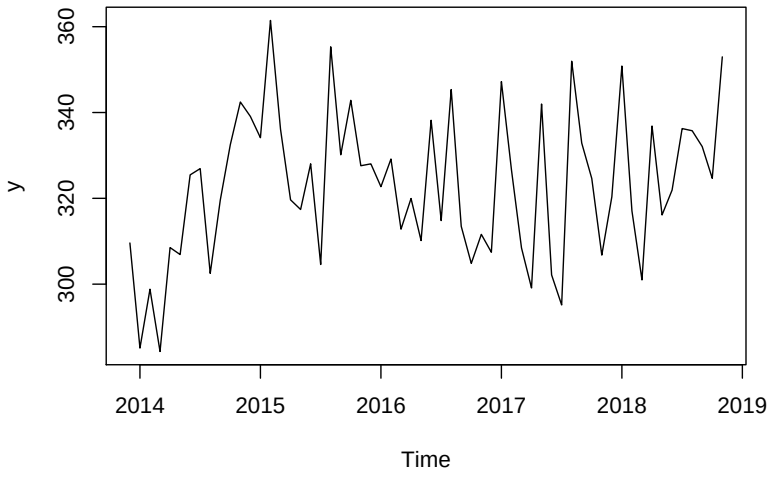
The argument `frequency=12` tells `ts()` that this time series has 12 observations per season, i.e. it is monthly data. We could also specify the start or end date if it was known. For our purpose let us assume that the data end in November.

```
y <- ts(y,frequency=12,end=c(2018,11))
# The syntax of end is c(Year,Month), or more generally
# c(season,seasonal period).
print(y)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2013
## 2014 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226 302.5102
## 2015 334.1232 361.4546 336.1173 319.6460 317.3951 328.0461 304.5760 355.3091
## 2016 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126 345.3493
## 2017 347.2009 326.6501 308.4443 299.1128 341.9722 302.1470 295.1502 351.9557
## 2018 350.8271 317.0764 300.9870 336.8473 316.1264 321.9198 336.2565 335.7531
##           Sep      Oct      Nov      Dec
## 2013                            309.5927
## 2014 319.5777 332.5051 342.4510 339.0753
## 2015 330.1500 342.8376 327.5952 328.0343
## 2016 313.4708 304.8354 311.6021 307.4285
## 2017 332.8738 324.6155 306.7664 320.3624
## 2018 332.0578 324.6613 352.9658
```

We can easily plot the time series with the function `plot()`, if we did not convert `y` to a time series plot would behave very differently

```
plot(y)
```



4 Constructing estimation and hold-out sets

For any model building exercise, it is useful to set aside some holdout data, to test the outcome of your modelling. This step is very important in the initial setup of the forecasting process, i.e. specifying a model for a time series, but naturally, when we want to produce a true forecast for the upcoming months we use all data, using the model/approach we evaluated as best in the hold-out sample.

We will keep as hold-out (or test set) the last year. For this the function `tail()` can help us, which keeps the last n observations. This function behaves differently if the **forecast** package is not loaded, so make sure that you have followed that step above.

```
y.tst <- tail(y,12)
print(y.tst)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2017
## 2018 350.8271 317.0764 300.9870 336.8473 316.1264 321.9198 336.2565 335.7531
##           Sep      Oct      Nov      Dec
## 2017                           320.3624
## 2018 332.0578 324.6613 352.9658
```

The rest is kept as fitting (or estimation or training) set. The function `head()` is the opposite of `tail()`.

```
y.trn <- head(y,48)
print(y.trn)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2013
## 2014 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226 302.5102
## 2015 334.1232 361.4546 336.1173 319.6460 317.3951 328.0461 304.5760 355.3091
## 2016 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126 345.3493
## 2017 347.2009 326.6501 308.4443 299.1128 341.9722 302.1470 295.1502 351.9557
##           Sep      Oct      Nov      Dec
## 2013                           309.5927
## 2014 319.5777 332.5051 342.4510 339.0753
## 2015 330.1500 342.8376 327.5952 328.0343
## 2016 313.4708 304.8354 311.6021 307.4285
## 2017 332.8738 324.6155 306.7664
```

When splitting ts objects (time series objects that hold date and frequency information), then the functions `head()`, `tail()`, `window()` retain this information. Other ways to split the series, such as `y[1:48]` will not carry forward this information and the data must be specified as a ts object again. For example:

```
yy <- y[1:48]
print(yy)
```

```
##  [1] 309.5927 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226
##  [9] 302.5102 319.5777 332.5051 342.4510 339.0753 334.1232 361.4546 336.1173
## [17] 319.6460 317.3951 328.0461 304.5760 355.3091 330.1500 342.8376 327.5952
## [25] 328.0343 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126
## [33] 345.3493 313.4708 304.8354 311.6021 307.4285 347.2009 326.6501 308.4443
## [41] 299.1128 341.9722 302.1470 295.1502 351.9557 332.8738 324.6155 306.7664
```

Variable `yy` is no longer a ts object. We can use the function `class()` to see this.

```
class(y) # Our time series object
```

```
## [1] "ts"
```

```
class(yy) # A simple vector of numeric values
```

```
## [1] "numeric"
```

But we can always carry the properties to yy:

```
yy <- ts(yy, frequency=frequency(y), start=start(y))
print(yy)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2013
## 2014 285.0966 298.8200 284.3028 308.5171 306.8993 325.4628 326.9226 302.5102
## 2015 334.1232 361.4546 336.1173 319.6460 317.3951 328.0461 304.5760 355.3091
## 2016 322.7103 329.1623 312.8083 319.9644 310.1535 338.1874 314.8126 345.3493
## 2017 347.2009 326.6501 308.4443 299.1128 341.9722 302.1470 295.1502 351.9557
##           Sep      Oct      Nov      Dec
## 2013                            309.5927
## 2014 319.5777 332.5051 342.4510 339.0753
## 2015 330.1500 342.8376 327.5952 328.0343
## 2016 313.4708 304.8354 311.6021 307.4285
## 2017 332.8738 324.6155 306.7664
```

```
class(yy)
```

```
## [1] "ts"
```

This is now the same as `y.trn`. We can use `==` to compare

```
yy == y.trn
```

```
##       Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
## 2013                                                        TRUE
## 2014 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 2015 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 2016 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 2017 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
all(yy==y.trn) # all() provides the overall logical comparison result
```

```
## [1] TRUE
```

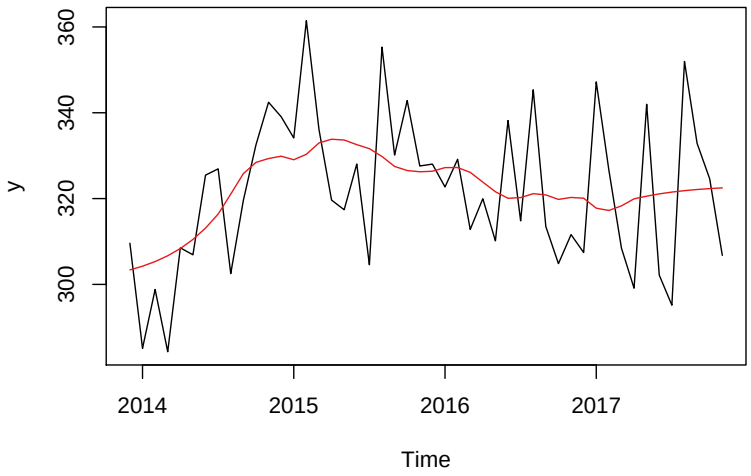
5 Exploring a time series

There are a few alternative ways to do that, using different packages. For this we will be using the **tsutils** package. We loaded this before, but just in case, we can call it again.

```
library(tsutils)
```

First, we calculate a Centred Moving Average and plot it. Note that we will be using `y.trn` and not `y.tst` or the complete `y`.

```
cma <- cmav(y.trn,outplot=1) # The argument outplot produces a plot
```



This function extrapolates the CMA values for the first few and last observations that we cannot calculate from the data. This is done by fitting an appropriate exponential smoothing model. The order of the average is taken automatically from the frequency argument of the time series.

Question: Is this time series trended?

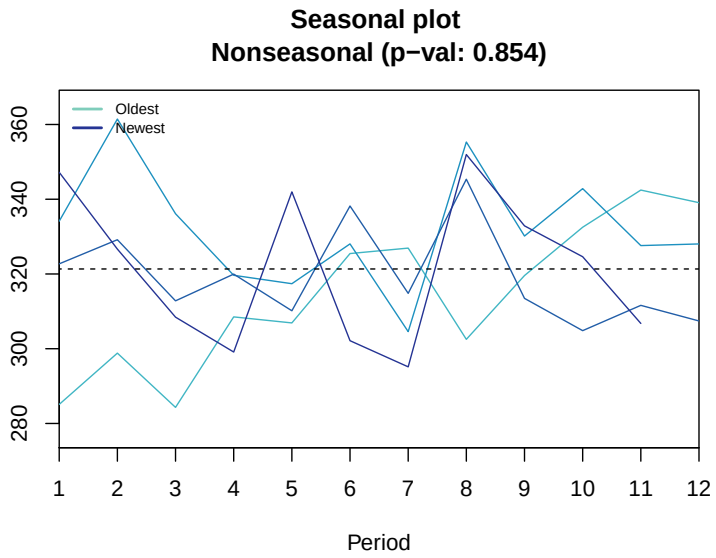
If needed, the variable 'cma' contains the CMA calculated values.

```
print(cma)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2013
## 2014 304.2395 305.3201 306.6708 308.3591 310.4696 313.1166 316.3878 321.0404
## 2015 329.0542 330.3231 332.9636 333.8346 333.6461 332.5671 331.6315 329.8105
## 2016 327.2172 327.2288 326.1188 323.8404 321.5906 320.0656 320.2275 321.1433
## 2017 317.7662 317.2222 318.3059 319.9386 320.5613 321.0955 321.5123 321.8458
##           Sep      Oct      Nov      Dec
## 2013                          303.3751
## 2014 325.8091 328.4317 329.3328 329.8777
## 2015 327.4937 326.5358 326.2473 326.3682
## 2016 320.8568 319.8061 320.2631 320.0872
## 2017 322.1126 322.3260 322.4968
```

Next, we proceed to explore the seasonality of the time series. To produce a seasonal diagram we can use the function `seasplot()`. This will detrend the series, if needed, and produce the seasonal plot. It will also provide a p-value for the presence of seasonality. Do not trust the p-value more than you trust your eyes! It is there only for automation, but not as reliable as your critical thinking!

```
seasplot(y.trn)
```

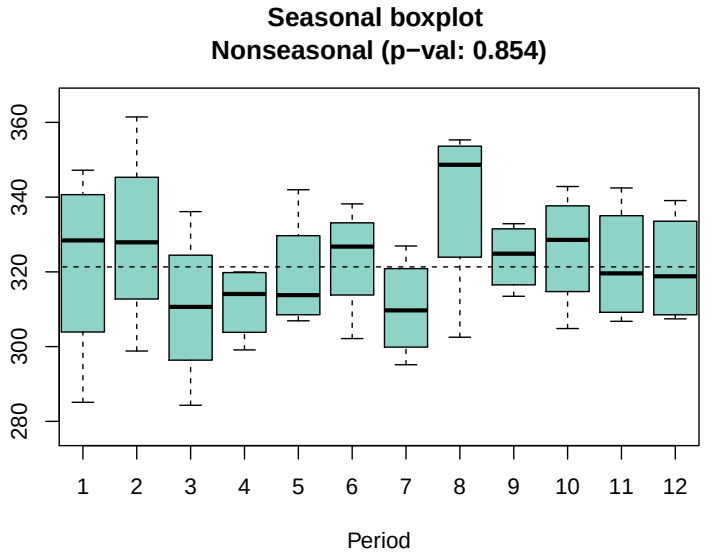



```
## Results of statistical testing
## Evidence of trend: FALSE  (pval: 0.154)
## Evidence of seasonality: FALSE  (pval: 0.854)
```

Question: Is this time series seasonal?

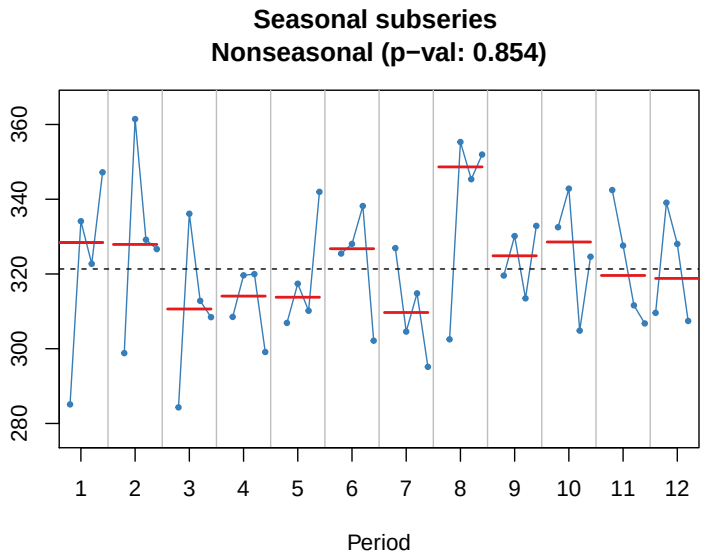
The function `seasplot()` provides a few alternative visualisations (see Lecture 1).

```
seasplot(y.trn,outplot=2) # Boxplots of the values of each month
```



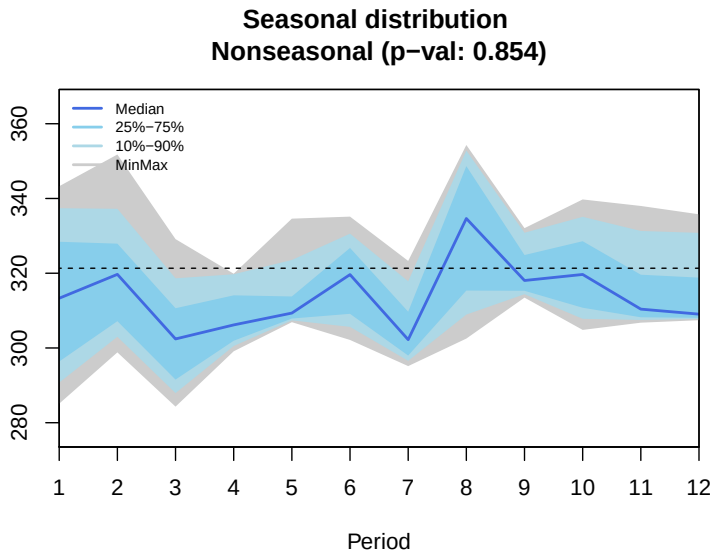

```
## Results of statistical testing
## Evidence of trend: FALSE  (pval: 0.154)
## Evidence of seasonality: FALSE  (pval: 0.854)
```

```
# The average (red) value for each month with a series of each month across years (blue)
seasplot(y.trn,outplot=3)
```




```
## Results of statistical testing
## Evidence of trend: FALSE  (pval: 0.154)
## Evidence of seasonality: FALSE  (pval: 0.854)
```

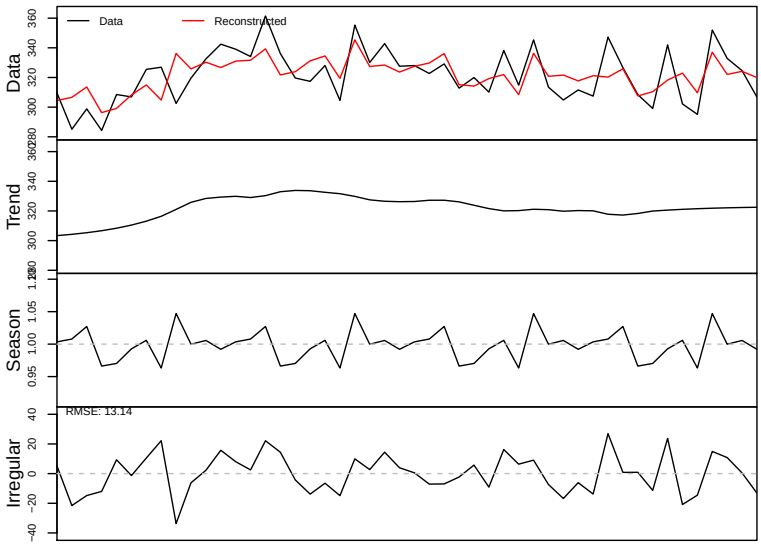
```
seasplot(y.trn,outplot=4) # A `connected' boxplot across months.
```




```
## Results of statistical testing
## Evidence of trend: FALSE  (pval: 0.154)
## Evidence of seasonality: FALSE  (pval: 0.854)
```

We can also decompose the time series, using the function `decomp()`. This function is not smart enough to know when there is a trend or seasonality, so it will always assume both components are there. Remember, in decomposition we only remove the trend or the season if it is present! Otherwise, we would be modelling noise.

```
dc <- decomp(y.trn,outplot=1)
```



6 Forecasting

6.1 Model fitting

At this point we have established the nature of the time series at hand. We will use the function `ets()` to build exponential smoothing models. `ets()` allows building all standard (30) exponential smoothing models. Remember that the models are codified with 3 letters **ETS(Error,Trend,Season)**, where each Error, Trend, and Season can be (N,A,M), for *None*, *Additive*, and *Multiplicative*. (In fact Error can be only A or M, as all models include an error term). To set a damped trend we need to use the argument `damped=TRUE` (or `damped=FALSE` for linear trend). If no option is specified for damped, then `ets()` decides on its own using the AICc information criterion. The AICc is similar to the Akaike Information Criterion (AIC), adjusted for small sample sizes.

```
# Model ANN is (A)dditive errors, (N)o trend and (N)o season, i.e. the single exponential
# smoothing (local level model).
ets(y.trn,model="ANN")
```

```
## ETS(A,N,N)
##
## Call:
##  ets(y = y.trn, model = "ANN")
##
##   Smoothing parameters:
##     alpha = 0.2315
##
##   Initial states:
##     l = 304.1632
##
##   sigma:  17.7462
##
##      AIC     AICc      BIC
## 465.8872 466.4326 471.5008
```

Observe that the output gives us information about the smoothing parameter (alpha), the initial level (l), and the standard deviation or the errors (uncertainty; sigma). It also presents results on various information criteria.

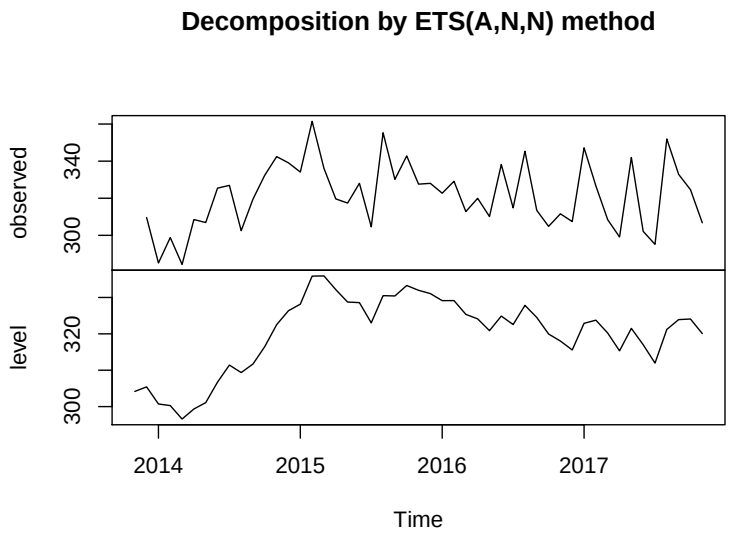
We store the model in a variable in the memory:

```
fit1 <- ets(y.trn,model="ANN")
print(fit1)
```

```
## ETS(A,N,N)
##
## Call:
##  ets(y = y.trn, model = "ANN")
##
##   Smoothing parameters:
##     alpha = 0.2315
##
##   Initial states:
##     l = 304.1632
##
##   sigma:  17.7462
##
##      AIC     AICc      BIC
## 465.8872 466.4326 471.5008
```

We can also produce a plot of the model using `plot()`:

```
plot(fit1)
```



This plot shows the modelled time series components. Personally, I do not find this graph to be particularly helpful. We will produce an alternative one, but before that, we need to understand what is stored in `fit`, which is a list of variables that stores all the model information. We will use the function `names()` to extract the names of the variables stored in `fit1`. This function is appropriate for lists.

```
class(fit1)
```

```
## [1] "ets"
```

```
# names gives you the variables in the list fit1, an object ets (the result of class) is a list.
names(fit1)
```

```
##  [1] "loglik"     "aic"        "bic"        "aicc"       "mse"
##  [6] "amse"       "fit"        "residuals"  "fitted"     "states"
## [11] "par"        "m"          "method"     "series"     "components"
## [16] "call"       "initstate"  "sigma2"     "x"
```

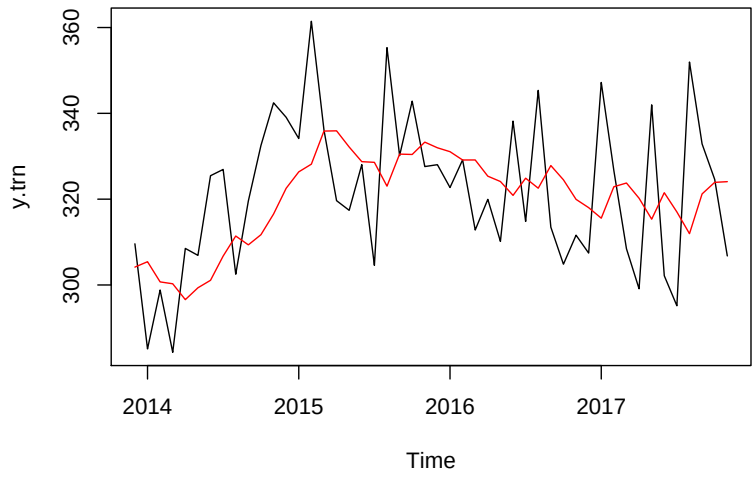
Observe that one of the variables is called `fitted`, this contains the fitted values in the historical data. To access this variable we use the dollar sign, `$`, for example:

```
fit1$fitted
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 2013
## 2014 305.4201 300.7152 300.2764 296.5785 299.3423 301.0918 306.7338 311.4076
## 2015 326.3602 328.1574 335.8658 335.9240 332.1556 328.7385 328.5782 323.0216
## 2016 331.0612 329.1279 329.1359 325.3560 324.1078 320.8773 324.8847 322.5529
## 2017 315.5674 322.8906 323.7610 320.2151 315.3298 321.4977 317.0179 311.9554
##           Sep      Oct      Nov      Dec
## 2013                        304.1632
## 2014 309.3478 311.7161 316.5288 322.5299
## 2015 330.4963 330.4161 333.2917 331.9730
## 2016 327.8304 324.5061 319.9523 318.0192
## 2017 321.2157 323.9146 324.0768
```

Now, let us return to plotting the result of the model fitting. We will use functions `plot()' and`lines()‘. The latter allows us to add lines to an existing plot.

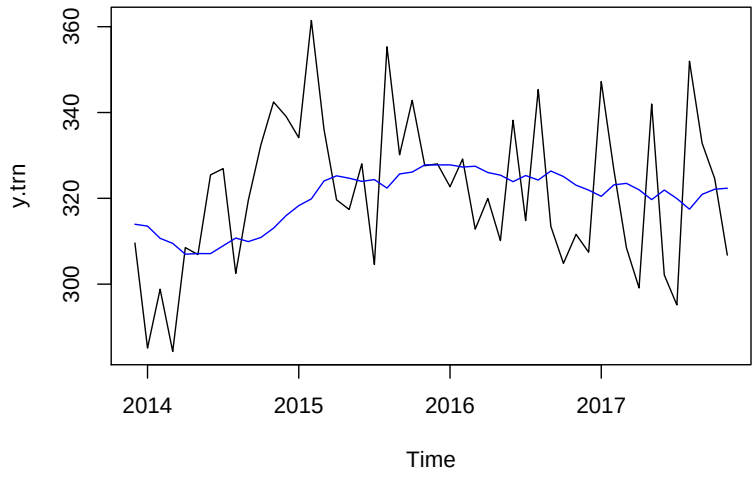
```
plot(y.trn)
lines(fit1$fitted, col="red") # col=... specifies the colour of the line
```



Question: Given your understanding of single exponential smoothing, is this a good model fit? Does the forecast look “smooth”? What should we do with the alpha parameter?

Let us try a different alpha.

```
fit2 <- ets(y.trn, model = "ANN", alpha = 0.1)
plot(y.trn)
lines(fit2$fitted,col="blue")
```



We can try different values for alpha until we are happy. If we do no specify an alpha, the value we get is the one that minimises the in-sample Mean Squared Error (MSE), which was the case for `fit1`. In fact, we can extract the in-sample MSE and compare:

```
fit1$mse
```

```
## [1] 301.8053
```

```
fit2$mse
```

```
## [1] 313.4249
```

So the optimiser correctly suggests the alpha to be 0.2315, as it gives the minimum in-sample MSE. That may not be true for the out-of-sample MSE, as we will see below. But before we do that, we need to produce some out-of-sample forecasts!

6.2 Forecasting

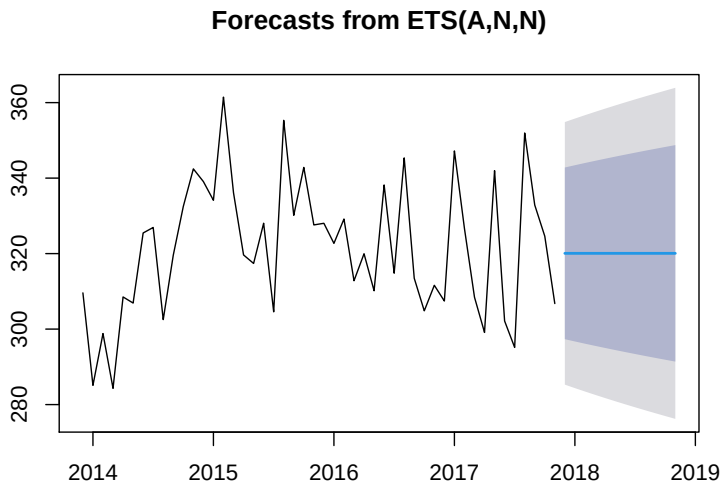
We can generate forecasts using the function `forecast()`, which has two necessary arguments, the fitted model and the forecast horizon (`h`)

```
frc1 <- forecast(fit1, h=12)
print(frc1)
```

```
##          Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
## Dec 2017       320.0694 297.3267 342.8121 285.2875 354.8513
## Jan 2018       320.0694 296.7253 343.4135 284.3676 355.7712
## Feb 2018       320.0694 296.1389 343.9999 283.4708 356.6679
## Mar 2018       320.0694 295.5666 344.5722 282.5955 357.5433
## Apr 2018       320.0694 295.0073 345.1315 281.7402 358.3986
## May 2018       320.0694 294.4602 345.6786 280.9035 359.2353
## Jun 2018       320.0694 293.9246 346.2142 280.0844 360.0544
## Jul 2018       320.0694 293.3997 346.7391 279.2817 360.8571
## Aug 2018       320.0694 292.8850 347.2538 278.4945 361.6443
## Sep 2018       320.0694 292.3798 347.7590 277.7219 362.4169
## Oct 2018       320.0694 291.8837 348.2551 276.9631 363.1757
## Nov 2018       320.0694 291.3962 348.7426 276.2175 363.9213
```

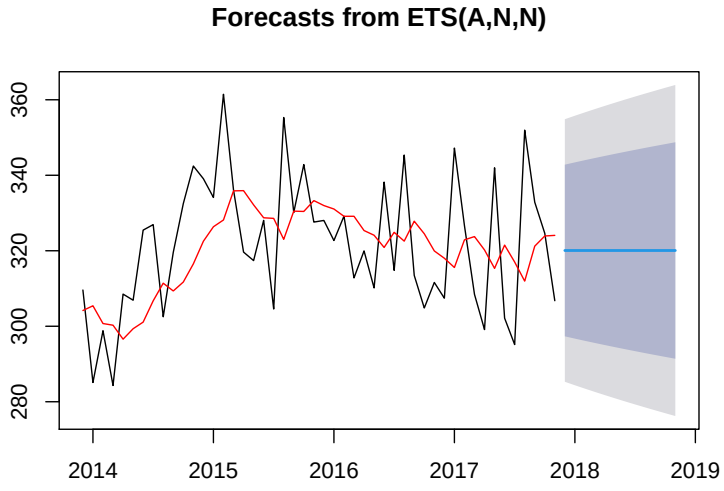
Observe that it gives us a point forecast, and lower (Lo) and upper (Hi) prediction intervals. By default, it gives us the 80% and 95% intervals. We can control that with the argument `level=...` that by default is `level=c(80,95)`. We can get a plot of everything using `plot()`:

```
plot(frc1)
```



I find it helpful to add the in-sample fit as well:

```
plot(frc1)
lines(fit1$fitted,col="red")
```



The `frc1` variable, where we stored the forecasts, is a list. Therefore, we can use `names()` to see what it contains.

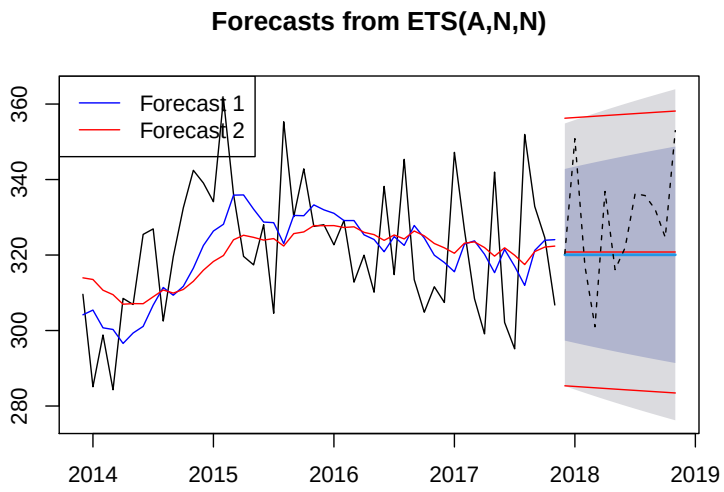
```
names(frc1)
```

```
##  [1] "model"     "mean"      "level"     "x"         "upper"     "lower"
##  [7] "fitted"    "method"    "series"    "residuals"
```

The variable `mean` contains the point forecasts, and the `upper`/`lower` the prediction intervals. Let us superimpose the forecasts from 'fit2' on the same plot, as well as the out-of-sample data.

```
frc2 <- forecast(fit2,h=12) # Store the  forecasts
plot(frc1)
lines(fit1$fitted,col="blue")
```

```
lines(frc2$mean,col="red")
lines(fit2$fitted,col="red")
lines(frc2$lower[,2],col="red")  # 95% lower
lines(frc2$upper[,2],col="red")  # 95% upper
lines(y.tst,lty=2)  # lty=2 gives us a dashed line.
# Add legend to the plot
legend("topleft",c("Forecast 1","Forecast 2"),col=c("blue","red"),lty=1)
```



So although the point forecasts are very similar the model that produces a smoother model fit, also provides narrower (overall) intervals.

Question: Which is the best forecast?

We should **never** select between forecasts using the out-of-sample errors. However, we can use these to evaluate whether our selection was correct after all. Let us use Mean Absolute Error (MAE).

```
MAE1 <- mean(abs(y.tst - frc1$mean))
MAE2 <- mean(abs(y.tst - frc2$mean))
MAE <- c(MAE1, MAE2) # Collect them in a single vector
names(MAE) <- paste0("Forecast ",1:2) # Name the errors
round(MAE,3) # round to 3rd decimal point
```

```
## Forecast 1 Forecast 2
##     13.087     12.794
```

So going for the smoother in-sample forecast, with the somewhat lower alpha value, gave us a lower forecast error. This is not true in general and here I only got suspicious because I found that the fitted values of `fit1` looked somewhat too wiggly for my liking. *Yes, this is not scientific at all!* But what is scientific is that the series is level, with no peculiar events, so ideally I want a long average and therefore a small alpha value: it is all down to exploring your data!

7 Exercises

Practice with the remaining time series in `Y`, for which we know the model forms: Level_A, Level_B, LevelShift, Trend_A, Trend_B, Season_A, Season_B, TrendSeason. DO not attempt fitting other models than ETS(ANN) yet! We will cover this in detail.

1. Retain some data for out-of-sample evaluation.
2. Perform data exploration. The functions you need to use are: `cma()`, `seasplot()` and `decomp()`.

- Does your understanding of the plots agree with the underlying model?

3. Forecast using exponential smoothing
   - Fit exponential smoothing models with automatic and manual parameters.
   - Which one is best using your judgement?
   - Which one is best using errors?
4. Calculate out-of-sample accuracy.
   - Does the selected model perform best in the out-of-sample data?
5. Finally use some real data to repeat the exercise. You can use the `AirPassengers` time series, by setting `y <- AirPassengers`.

Remember: R has a fantastic online community. Online you will find answers to almost anything R related!

Happy forecasting!