Introduction to Time Series

Gracie Yao
November 26, 2017

install.packages("chron")
install.packages("lubridate")
install.packages("forecast")
install.packages("TTR")

```
library(chron) # version2.3-51
library(lubridate)  # version 1.6.0
library(forecast) # version 8.2
library(TTR) # version 0.23-2
```

Datasets:
Lynx: Annual Canadian Lynx trapping for 1821-1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).
Nottem: A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

Version of R: 3.3.2
Version of R studio: 1.0.136

Introduction

Time series refers to a series of data points in time order. Time series data are prevalent in our lives, especially in financial, economic, and medical applications. Examples include the stock returns, daily temperature changes, and the values of the consumer price index. Time series analysis has become one of the crucial fields in statistical programming because it helps us to understand the present--the data we already collected--and to predict the future. Therefore, the motivation for this post is to familiarize the readers with some background knowledge necessary for doing time series analysis using R.
This post consists of three main sections. The first section shows ways to handle date and time data. After that, it introduces some basics regarding time series. And the last part talks about time series analysis. The following is the structure of the post:

Structure

I. Dates and Time

1. Classes

```
a.  POSIXt class
b.  Date class
c.  Chron class
```

2. Lubridate

```
a.  Input and time zone
b.  Weekdays and intervals
c.  Calculations with time
  1)    Basic calculations
  2)    Leap years
```

II. Time Series Basics

1. Time series vector and time lag

```
a.  Generate a time series
b.  Time lag
c.  Primitive forecasting methods
```

2. Model comparison

III. Exploratory Time Series Analysis

1. Time series class: ts

2. Decomposing a seasonal time series

3. Smoothing the data

4. Time series visualization

I. Dates and Time

1. Classes

a. POSIXt class:

- One of the key classes used to handle dates and time
- Divided into POSIXct and POSIXlt, both of which can handle time zones, dates, and time

```
#store data in POSIXct format
a = as.POSIXct("2017-11-26 18:30:25")
#store data in POSIXlt format
b = as.POSIXlt("2017-11-26 18:30:25")
a
```

```
## [1] "2017-11-26 18:30:25 PST"
```

```
b
```

```
## [1] "2017-11-26 18:30:25 PST"
```

```
# they give the same result, but behave differently
unclass(a) # POSIXct used 01.01.1970 as a reference point. It calculates the amount of seconds passed since
```

```
## [1] 1511749825
## attr(,"tzone")
## [1] ""
```

```
unclass(b) # POSIXlt returns a list of the elements of a date and also the time zone
```

```
## $sec
## [1] 25
##
## $min
## [1] 30
##
## $hour
## [1] 18
##
## $mday
## [1] 26
##
## $mon
## [1] 10
##
## $year
## [1] 117
##
## $wday
## [1] 0
##
## $yday
## [1] 329
##
## $isdst
## [1] 0
##
## $zone
## [1] "PST"
##
## $gmtoff
## [1] NA
```

b. Date class:

- Also has a reference date of 01.01.1970
- Calculates in days instead of seconds

```
x = as.Date("2017-11-26 18:30:25")
x
```

```
## [1] "2017-11-26"
```

```
class(x)
```

```
## [1] "Date"
```

```
unclass(x) # returns the number of days after the reference point
```

```
## [1] 17496
```

c. Chron class:

- calculates the days passed since 1970, but transfers time as decimals
- chron format:
  - x = chron("dd/mm/yyyy", "hh:mm:ss")
- Chron is suitable when we have constant time zones since it does not require a time zone in its arguments

```
x = chron("11/26/2017", "18:30:25")
x
```

```
## [1] (11/26/17 18:30:25)
```

```
class(x)
```

```
## [1] "chron" "dates" "times"
```

```
unclass(x)
```

```
## [1] 17496.77
## attr(,"format")
##   dates   times
## "m/d/y" "h:m:s"
## attr(,"origin")
## month   day  year
##     1     1  1970
```

2. Lubridate

- Can handle time and dates with different time zones

a. Input and time zone

```
# ways to input dates
ymd(20171126)
```

```
## [1] "2017-11-26"
```

```
dmy(26112017)
```

```
## [1] "2017-11-26"
```

```
mdy(11262017)
```

```
## [1] "2017-11-26"
```

```
# all the time zones
tzones<-olson_time_zones()
```

```
## Warning: 'olson_time_zones' is deprecated; use 'OlsonNames' instead.
## Deprecated in version '1.5.8'.
```

```
head(tzones)
```

```
## [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
## [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Bamako"
```

```
# use time and date together
mytime <- ymd_hm("2017-11-26 11:23", tz = "Europe/Prague")
mytime
```

```
## [1] "2017-11-26 11:23:00 CET"
```

```
# extract information from mytime
minute(mytime)
```

```
## [1] 23
```

```
hour(mytime)
```

```
## [1] 11
```

```
# change time value
hour(mytime) <- 05
mytime
```

```
## [1] "2017-11-26 05:23:00 CET"
```

```
#change time zones
with_tz(mytime, tz = "Europe/London")
```

```
## [1] "2017-11-26 04:23:00 GMT"
```

b. Weekdays and intervals

```
# check the weekday of x
wday(mytime)
```

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

```
wday(mytime, label = T, abbr = F) # label to display the name of the day, with abbreviation
```

```
## [1] Sunday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

```
wday(mytime, label = T, abbr = T) # label to display the name of the day, no abbreviation
```

```
## [1] Sun
## Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
```

```
# get the time interval between two dates, which is useful for further calculations
time1 <- ymd_hm("2016-11-26 11:23", tz = "Europe/Prague")
time2 <- mytime
interval <- interval(time1, time2)
interval
```

```
## [1] 2016-11-26 11:23:00 CET--2017-11-26 05:23:00 CET
```

c. Calculations with time

1. Basic calculation

```
#minutes(x) returns an integer x as minutes
minutes(7)
```

```
## [1] "7M 0S"
```

```
# dminutes(x) returns the duration of x minutes in seconds
dminutes(3)
```

```
## [1] "180s (~3 minutes)"
```

```
# add seconds and minutes
minutes(2) + seconds(5)
```

```
## [1] "2M 5S"
```

```
minutes(2) + seconds(76) # no automatic translation into minutes
```

```
## [1] "2M 76S"
```

```
# as.duration(x) operates with automatic translation
as.duration(minutes(2) + seconds(76))
```

```
## [1] "196s (~3.27 minutes)"
```

2. Leap years

```
# which year is a leap year: returns TRUE if it is a leap year
leap_year(2009:2016)
```

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

```
# add year to a timepoint
ymd(20160101) + years(1)
```

```
## [1] "2017-01-01"
```

```
ymd(20160101) + dyears(1)
```

```
## [1] "2016-12-31"
```

Difference between the standard one and the duration has something to do with the leap years:

- the standard one (the period) makes the year a whole new unit (+1)
- the duration always adds 365 yeas

II. Time Series Basics

What defines a time series?

- Time series data have a particular order, which is often specified by the time stamp (data collected on a yearly, monthly etc. basis)
- This order can also be specified by a vector which has a unique ID attached to the variable

1. Time Series Vector and Time Lag

a. Generate a time series

- function ts(): attaches a time stamp to a vector; useful if we set up a time series from scratch
- function xts(): for importing time series data

b. Time lag: the gap between two observations

Ex. lynx dataset

```
length(lynx)
```

```
## [1] 114
```

```
tail(lynx)
```

```
## Time Series:
## Start = 1929
## End = 1934
## Frequency = 1
## [1]  485  662 1000 1590 2657 3396
```

*lag*1 = 3396 – 2657
*lag*2 = 3396 – 1590

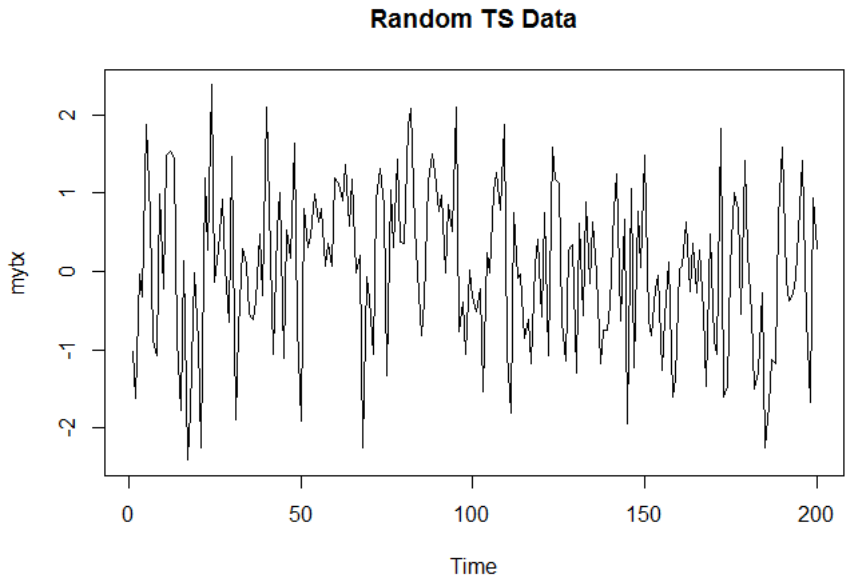
c. Primitive forecasting methods

Primitive forecasting methods can be quite useful in certain circumstances:

- for mostly or completely random data
- they do well with stock data or financial data Three simple models:

1. Naive method:
   - projects the last observation into the future
   - use the `naive()` function (`forecast` package)
2. Average method:
   - calculates the average of the data and projects that into the future
   - use the `meanf()` function (`forecast` package)
3. Drift method:
   - calculates the difference between first and last observation and projects that difference into the future
   - use the `rwf()` function (`forecast` package) Example:

```
# random normally distributed time series data, starting at 1818
set.seed(95)
mytx <- ts(rnorm(200), start(1818))
plot(mytx, main="Random TS Data")
```



The graph above plots a random time series dataset.

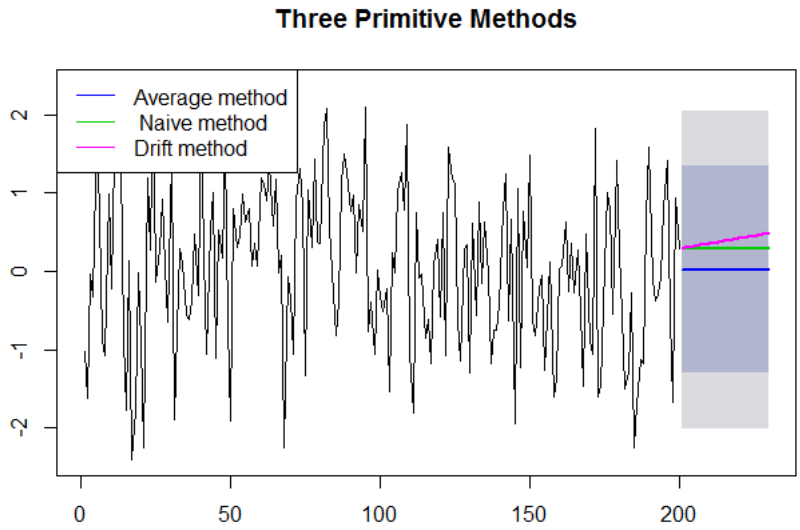
```
#use mean model to forecast 30 years
mean_model <- meanf(mytx, h=30)
#use naive model to forecast 30 years
naive_model <- naive(mytx, h=30)
#drift set to TRUE for the drift method
drift_model <- rwf(mytx, h=30, drift = T)
```

We can extract particular objects using `$`):

- fitted values: the forecasted values applied to the original dataset
- residuals: x minus the fitted values
- mean values: the forecasted values of the same mean

```
# plot the mean_model object
plot(mean_model,
     main = "Three Primitive Methods")

#naive_model$mean and drift_model$mean provide the 30 forecast values of each object
lines(naive_model$mean, col = 123, lwd = 2)
lines(drift_model$mean, col = 22, lwd = 2)
legend("topleft", lty = 1,
       col=c(4,123,22),
       legend = c("Average method", " Naive method", "Drift method"))
```



The graph above shows the predictions we get using three primitive methods.

2. Model Comparison

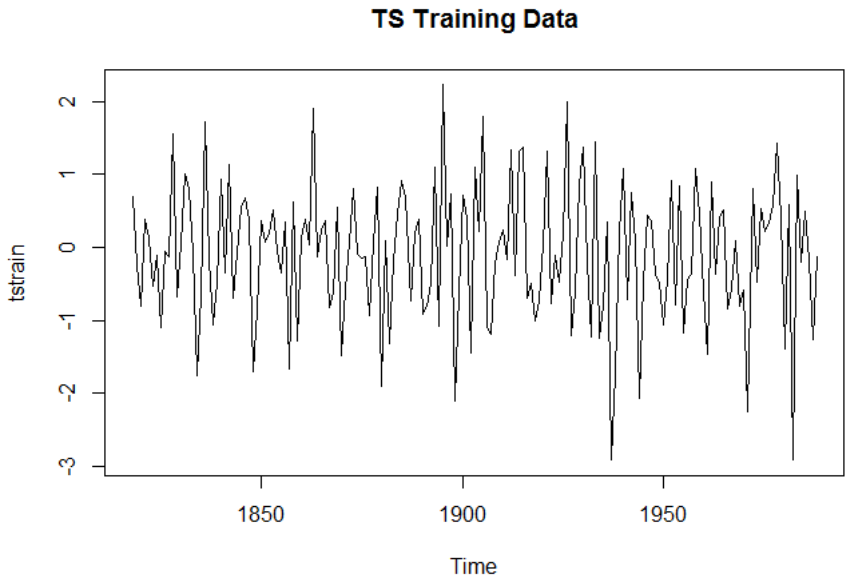
- It is important to know which model performs best with the given dataset
- In order to determine the forecast accuracy, we need to find the difference between the actual value and the corresponding forecasted value
- The simplest way is through a scale dependent error, meaning that all the models we want to compare need to be on the same scale
- Such errors include MAE(mean absolute error), RMSE(root mean squared error), MASE(mean absolute scaled error), MAPE(mean absolute percentage error), and AIC(Akaike information criterion).
- `accuracy()` function in the `forecast` package gives all these accuracy statistics except the AIC
- We divide the time series data into the training data and a test set using `window()` function
  - the training data is used to fit the model
  - the test set is used to see how well the model performs
  - `window()` function allows us to extract a time frame

STEP1: generate 200 random numbers -> `ts`

```
ts <- ts(rnorm(200), start = 1818)
```

STEP2: take the first 170 observations of `ts` as the training data -> `tstrain`

```
tstrain <- window(ts, start = 1818, end = 1988)
plot(tstrain, main="TS Training Data")
```



The graph above shows the first 170 observations of the 200 randomly-generated data.

STEP3: set up three forecasting models on `tstrain` to get 30 observations into the future -> meanm, naivem, driftm

```
meanm <- meanf(tstrain, h = 30)
naivem <- naive(tstrain, h = 30)
driftm <- rwf(tstrain, h = 30, drift = T)
```

STEP4: take the last 30 observations of `ts` as the test set -> `tstest`

```
tstest <- window(ts, start = 1988)
```

STEP5: use `accuracy()` function to see the error statistics of the three models compared to the error statistics of `tstest`

```
accuracy(meanm, tstest)
```

```
##                       ME      RMSE       MAE       MPE      MAPE
## Training set -1.009651e-17 0.9278796 0.7336360 104.85216 107.15623
## Test set      3.477060e-01 0.9473922 0.7427798  97.01645  98.69904
##                   MASE       ACF1 Theil's U
## Training set 0.6472821 -0.1267550        NA
## Test set     0.6553497 -0.2609843  1.061401
```

```
accuracy(naivem, tstest)
```

```
##                      ME      RMSE       MAE      MPE     MAPE      MASE
## Training set -0.004899599 1.3956559 1.1334099 86.19750 342.5778 1.0000000
## Test set      0.384044151 0.9613232 0.7538557 95.85658 103.1460 0.6651219
##                   ACF1 Theil's U
## Training set -0.5257653        NA
## Test set     -0.2609843  1.079209
```

```
accuracy(driftm, tstest)
```

```
##                      ME     RMSE       MAE      MPE     MAPE      MASE
## Training set 8.732292e-18 1.395647 1.1339287 86.44923 342.7066 1.0004577
## Test set     4.575381e-01 0.986836 0.7808886 93.79767 115.7111 0.6889728
##                   ACF1 Theil's U
## Training set -0.5257653        NA
## Test set     -0.2796690  1.114853
```

```
# we can see that the mean method does the best. We have the smallest value for RMSE, MAE, MASE, and MAPE
```

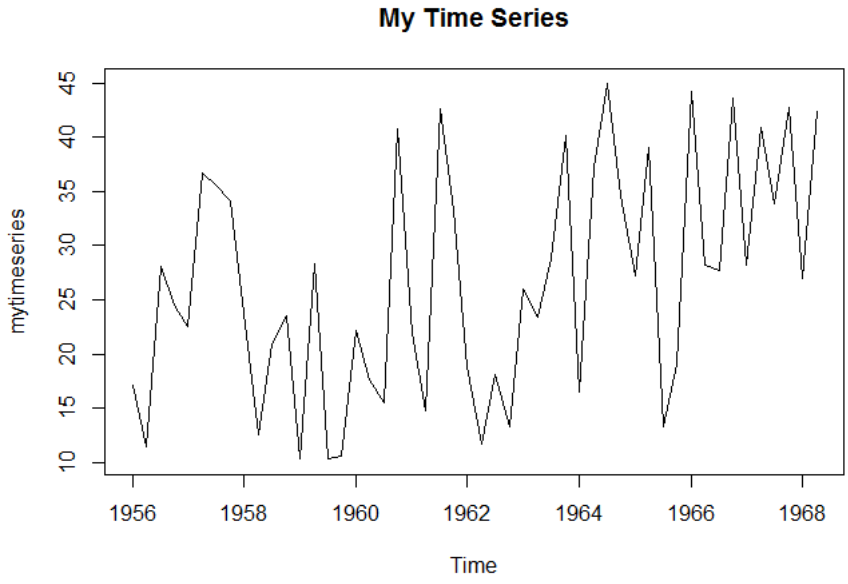
III. Exploratory Time Series Analysis

General idea: examine the data of the past and the present, look for patterns, and use them to do future forecasting

1. Time Series Class: ts

- In order to use all the analytical tools R offers, we need to have dataset in the time series class
- A time series object is like a vector with labels attached
- Labels = time marks

```
# get data (y axis)
mydata <- runif(n = 50, min = 10, max = 45)
# use ts() function and attach a time stamp (x axis) to the created dataset
mytimeseries <- ts(mydata, start = 1956, frequency = 4)
# plot the data
plot(mytimeseries, main="My Time Series")
```



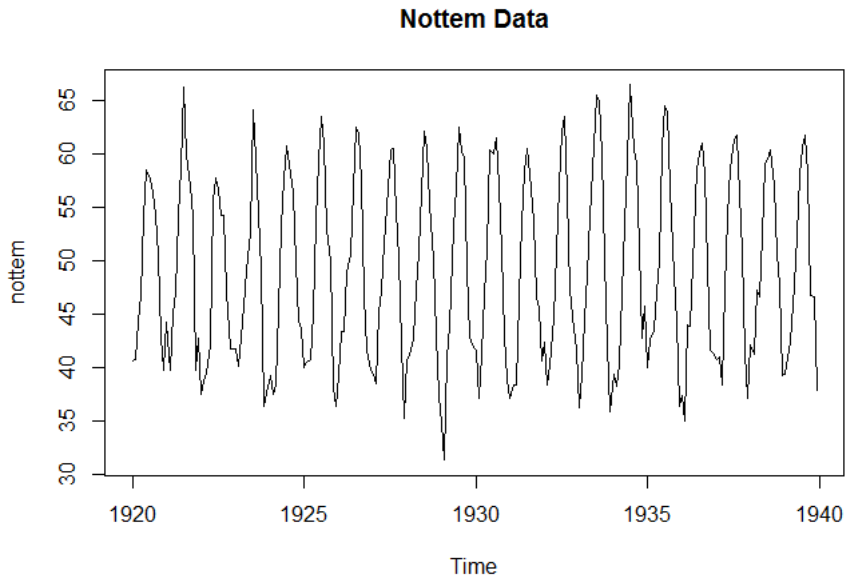
The graph above shows the random time series data we generated.

2. Decomposing a Seasonal Time Series

Three main components of time series dataset: trend, seasonality, and white noise
These three components can be either added (additive model) or multiplied (multiplicative model)

- A multiplicative model suits datasets with increasing seasonality
- Dataset with stable seasonality can be described using additive model

```
plot(nottem, main="Nottem Data")
```



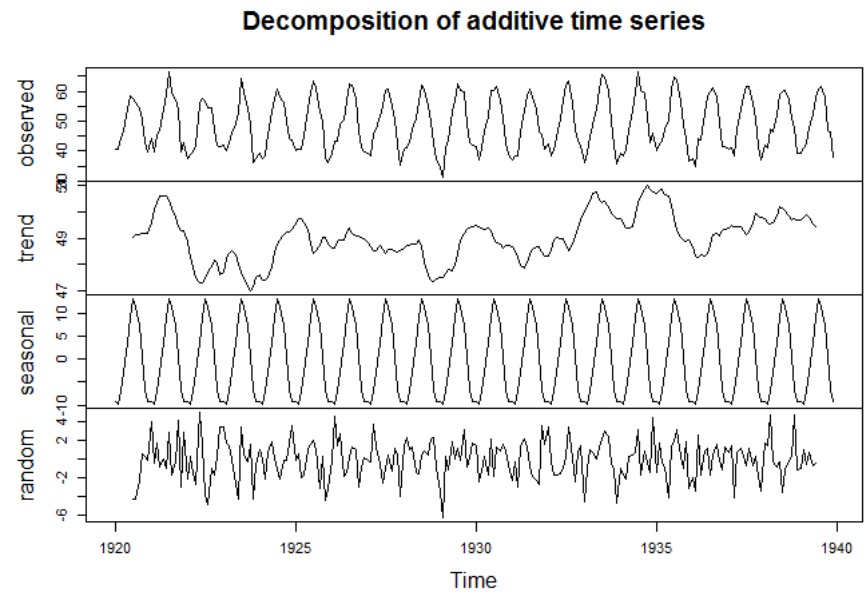
As we can see through the plot, the dataset has:

- stable seasonality
- no trend
- can be described with an additive model

```
# make sure the dataset has a preset frequency, which can be seen by checking the length of the dataset
length(nottem) # in this case the frequency is 12 (12 months) -> 240/20=12
```

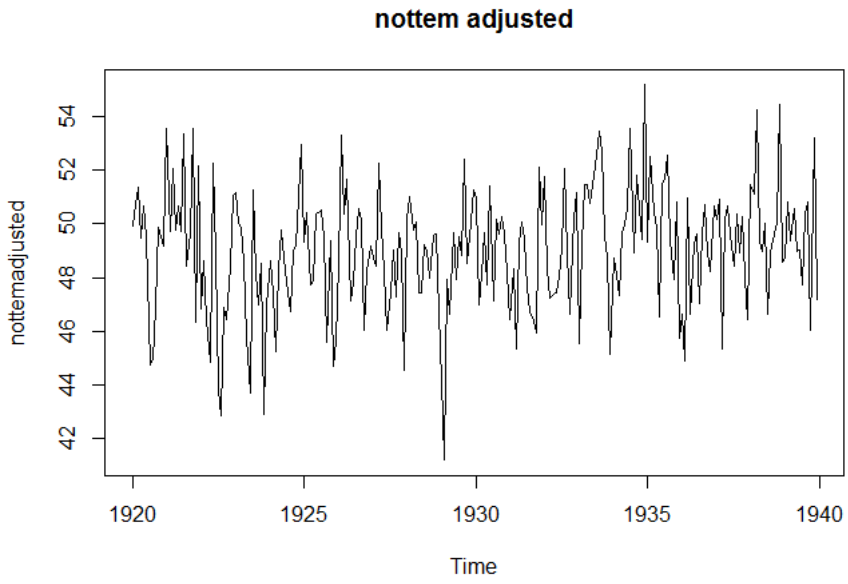
```
## [1] 240
```

```
#plot the decomposition
nottemdata <- decompose(nottem, "additive")
plot(decompose(nottem, "additive"))
```



The graph above shows the decomposition of the nottem time series data.

```
#get seasonal adjusted dataset by substracting seasonal element
nottemadjusted <- nottem - nottemdata$seasonal
# get a plot
plot(nottemadjusted, main = "nottem adjusted")
```



The graph above shows the adjusted nottem time series data.

3. Smoothing the Data

- Why?
  - to get the general trend
  - to remove the white noise
- How? by decreasing the impact of the extreme values
- Classic smoother: simple moving average (`SMA(x, n)`)
  - Define the number of period `n` we want to use. For example, if we have an `SMA` of 3, it means we take the average of three successive values. Then we move the procedure one step future and so on.
- Works best with non-seasonal data

```
#little experiment
x = c(1:7)
SMA(x, n = 3)
```

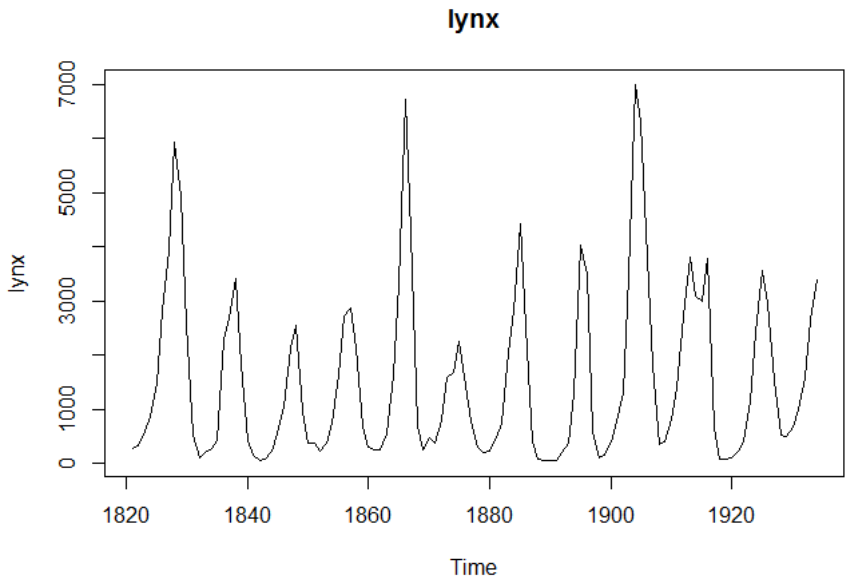
```
## [1] NA NA  2  3  4  5  6
```

We have some NAs at the beginning because at those stages we don't have enough data for the calculation. Only when we have three data available can SMA be calculated

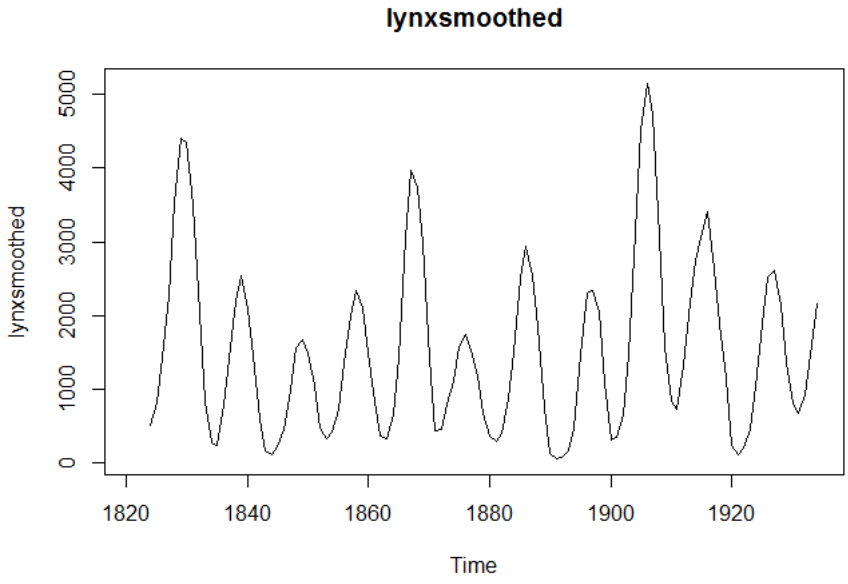
```
lynxsmoothed <- SMA(lynx, n = 4)
lynxsmoothed
```

```
## Time Series:
## Start = 1821
## End = 1934
## Frequency = 1
##   [1]      NA      NA      NA  511.50  813.00 1438.00 2273.75 3541.75
##   [9] 4410.50 4349.50 3498.25 2037.00  845.50  271.00  242.50  789.25
##  [17] 1414.50 2197.00 2550.75 2081.75 1448.25  607.25  168.25  119.25
##  [25]  218.00  465.00  980.25 1561.00 1663.75 1495.75 1057.75  480.00
##  [33]  330.75  423.25  738.50 1363.50 1991.25 2338.25 2099.75 1493.25
##  [41]  834.50  366.00  333.00  664.00 1432.75 3051.75 3977.25 3743.25
##  [49] 2979.25 1417.25  443.25  467.50  802.25 1103.00 1576.25 1736.75
##  [57] 1527.25 1183.00  670.50  371.25  299.50  408.75  869.00 1514.50
##  [65] 2505.00 2948.75 2535.50 1851.00  753.00  137.50   55.00   83.75
##  [73]  168.25  479.00 1472.00 2298.75 2351.25 2054.50 1085.00  308.00
##  [81]  350.75  651.25 1479.25 3130.25 4519.00 5140.75 4733.50 3072.00
##  [89] 1589.25  842.75  730.75 1322.75 2177.25 2748.00 3147.25 3416.50
##  [97] 2635.00 1882.50 1156.25  235.75  124.50  204.00  467.00 1048.00
## [105] 1884.25 2518.25 2619.50 2143.75 1371.50  803.25  669.00  934.25
## [113] 1477.25 2160.75
```

```r
# compare the smoothed vs the original data
plot(lynx, main="lynx")
```




```r
plot(lynxsmoothed, main="lynxsmoothed")
```

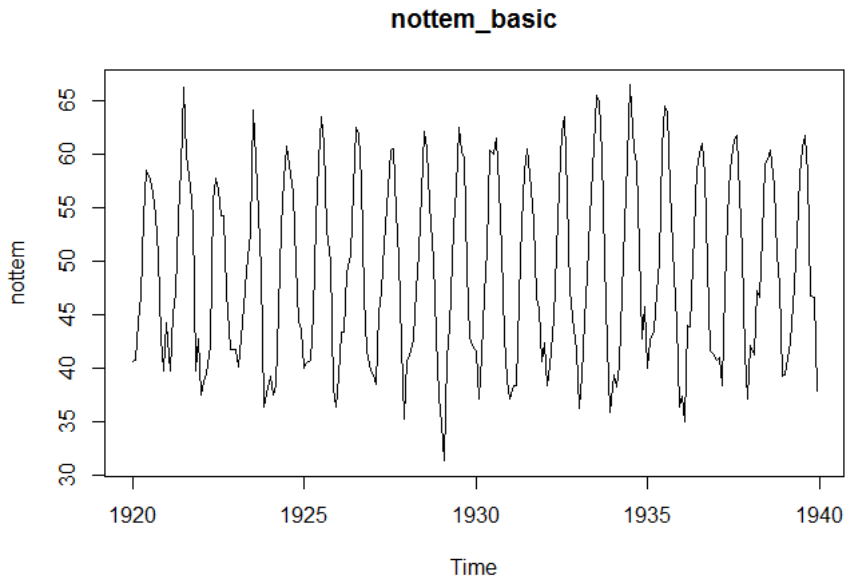


Comparing these two plots, we can see that the second plot is smoother.

4. Time Series Visualization

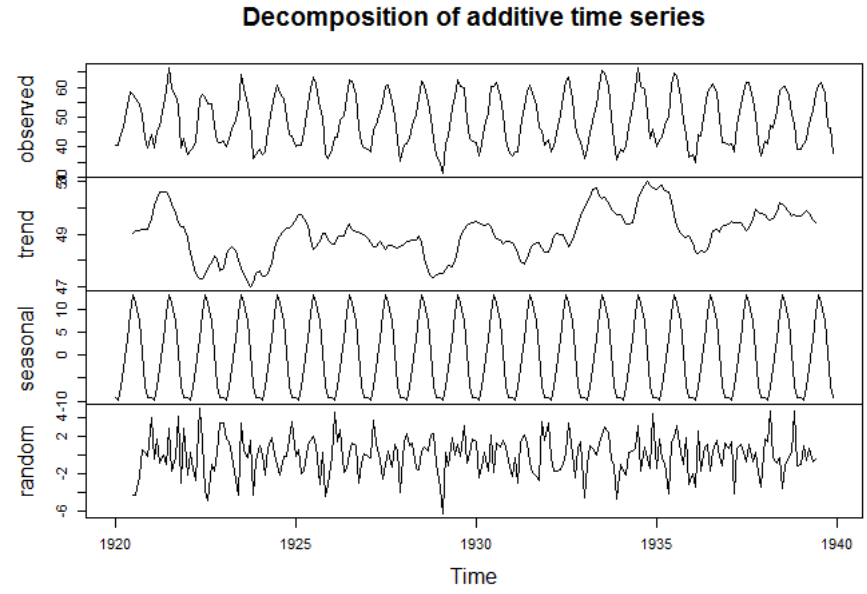
- Purposes of time series visualization
  - Data is presentable to the audience
  - Patterns can be more easily understood
- Features for visualization
  - `plot()` function
  - Other add-on packages such as `ggplot2` and `lattice`
- The line chart is key tool for plotting time series data
- Function `plot.ts()` plots non-time-series data like it was time series data
- Function `seasonplot()` plots each seasonal cycle of the dataset in a separate line

```r
#standard R base plots
plot(nottem, main = "nottem_basic")
```



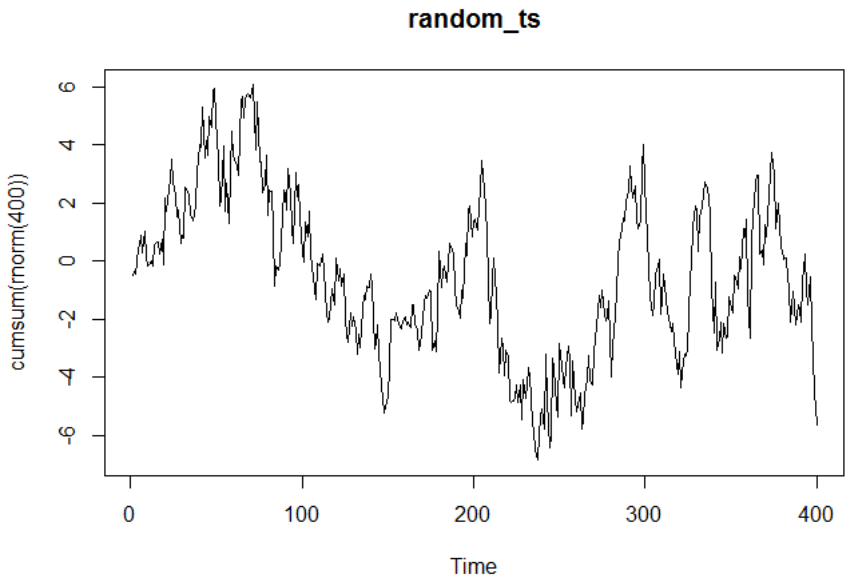
The graph above shows the nottem time series data using standard plot.

```r
# plot of components
plot(decompose(nottem))
```



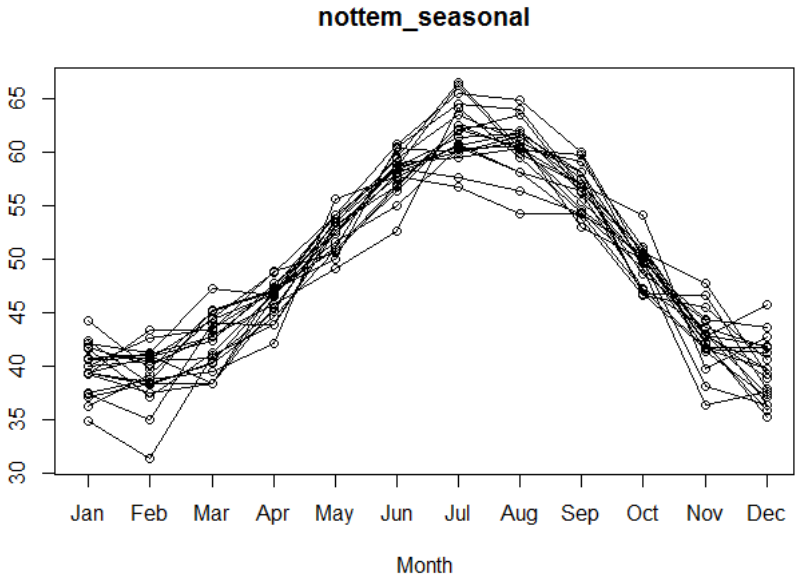
The graph above shows the decomposition plot of nottem time series data.

```r
# random data
set.seed(100)
plot.ts(cumsum(rnorm(400)), main = "random_ts")
```



The graph above shows how we plot non-time-series data like it was time series data.

```
# seasonal data plot
seasonplot(nottem, main = "nottem_seasonal")
```



The graph above shows the seasonal plot of the nottem time series data.

Conclusion & Take-Home Message

The underlying idea of time series analysis is to use our understanding of the present to predict the future, which is an extremely powerful tool applied in all the fields. Rather than diving into complicated time series models, this post focuses on some basic knowledge and techniques needed for the more advanced time series analysis. The major take-home message of this post is the fundamental idea of time series analysis: examine the events of the past, look for patterns, create models, and apply them to the future. I also expect you to know the following points:

- How to handle dates and time
  - Three classes
  - `lubridate` package
- Basic concepts related to time series
- Ts model comparison
- Some ways to plot time series

Referene: https://en.wikipedia.org/wiki/Time_series
https://cran.r-project.org/web/views/TimeSeries.html
https://cran.r-project.org/web/packages/lubridate/index.html
https://cran.r-project.org/doc/Rnews/Rnews_2004-1.pdf
http://people.duke.edu/~rnau/411home.htm
https://www.datacamp.com/home
https://onlinecourses.science.psu.edu/stat510/node/33
https://www.statmethods.net/advstats/timeseries.html