UC business Analytics R programming Guide 3

Koji Mizumura

2019-06-28

Table of Contents

# Time series analysis

## Exploring & visualizing time series

Time series forecasting is performed in nearly every organization that works with quantifiable data. Retail stores forecast sales. Energy companies forecast reserves, production, demand, and prices. Educational institutions forecast enrollment. Goverments forecast tax receipts and spending. International financial organizations forecast inflation and economic activity. The list is long but the point is short - forecasting is a fundamental analytic process in every organization. The purpose of this tutorial is to get you started doing some fundamental time series exploration and visualization.

### tl;dr

This tutotorial serves as an introduction to exploring and visualizing time series data and covers:

1. [Replication requirements](#TS_RR): What you will need to reproduce the analysis
2. [Creating time series objects](#TS_TS): Convert your data to a ts object for time series analysis.
3. [Time series plots](#TS_Plot): Basic visualization of ts objects and differentiating trends, seasonality, and cycle variation.
4. [Seasonal plots](#TS_Seasonal): Plotting seasonality trends in time series data.
5. [Autocorrelation of time series](#RS_ACF): Computing and visualizing autocorrelation.
6. [White noise](#TS_WN): Differentiating signal from the noise.
7. [Exercises](#TS_Exercise): Practice what you’ve learned.

### Replication Requirements

This tutorial leverages a variety of data sets to illustrate unique time series features. The data sets are all provided by the forecast and fpp2 packages. Furthermore, these packages provide various functions for computing and visualizing basic time series components.

library(tidyverse)  
library(tidymodels)  
library(magrittr)  
library(forecast)  
library(fpp2)

### Creating time series objects

A time series can be thought of as a vector or matrix numbers, along with some information about what times those numbers were recorded. This information is stored in a ts object in R.

In most examples and exercies throughout the forecasting tutorial ou will use data that are already in the time series format. However, if you want to work with your own data, you need to know how to create a ts object in R.

Here, I illustrate how to convert a data frame to a ts object. First, let’s assume we have a data frame named pass.df that looks like the following where we have the total number of airline passengers for each month for the years 1949-1960.

airpass %>% head()  
## Jan Feb Mar Apr May Jun  
## 1949 112 118 132 129 121 135

We can convert this data frame to a time series object by us the ts() function. Here, the…

* **first argument** supplies it the pass.df data frame and we index for just the columns with the data (we store the date-time data separately).
* **second argument** supplies the start date for the first observation (first period in 1949).
* **third argument** identifies the frequency, which in this case is monthly (hence 12 months in a year).

airpass %>% as.tibble() %>%   
 ts(., start=c(1949,1), frequency=12)  
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
## 1949 112 118 132 129 121 135 148 148 136 119 104 118  
## 1950 115 126 141 135 125 149 170 170 158 133 114 140  
## 1951 145 150 178 163 172 178 199 199 184 162 146 166  
## 1952 171 180 193 181 183 218 230 242 209 191 172 194  
## 1953 196 196 236 235 229 243 264 272 237 211 180 201  
## 1954 204 188 235 227 234 264 302 293 259 229 203 229  
## 1955 242 233 267 269 270 315 364 347 312 274 237 278  
## 1956 284 277 317 313 318 374 413 405 355 306 271 306  
## 1957 315 301 356 348 355 422 465 467 404 347 305 336  
## 1958 340 318 362 348 363 435 491 505 404 359 310 337  
## 1959 360 342 406 396 420 472 548 559 463 407 362 405  
## 1960 417 391 419 461 472 535 622 606 508 461 390 432  
  
# pass.ts <- ts(pass.df["AirPassengers"], start = c(1949, 1), frequency = 12)

We now have convered our data fraime into a time series object

str(airpass)  
## Time-Series [1:144] from 1949 to 1961: 112 118 132 129 121 135 148 148 136 119 ...  
airpass  
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
## 1949 112 118 132 129 121 135 148 148 136 119 104 118  
## 1950 115 126 141 135 125 149 170 170 158 133 114 140  
## 1951 145 150 178 163 172 178 199 199 184 162 146 166  
## 1952 171 180 193 181 183 218 230 242 209 191 172 194  
## 1953 196 196 236 235 229 243 264 272 237 211 180 201  
## 1954 204 188 235 227 234 264 302 293 259 229 203 229  
## 1955 242 233 267 269 270 315 364 347 312 274 237 278  
## 1956 284 277 317 313 318 374 413 405 355 306 271 306  
## 1957 315 301 356 348 355 422 465 467 404 347 305 336  
## 1958 340 318 362 348 363 435 491 505 404 359 310 337  
## 1959 360 342 406 396 420 472 548 559 463 407 362 405  
## 1960 417 391 419 461 472 535 622 606 508 461 390 432

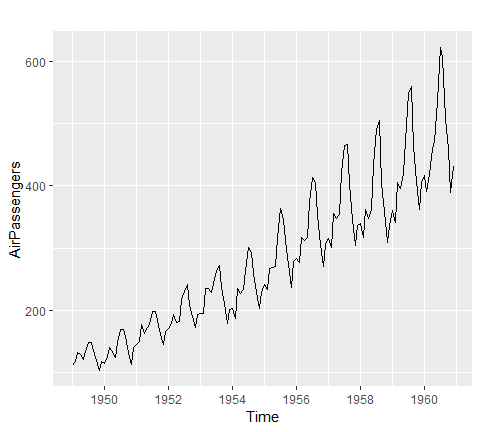
Go ahead and compute this pass.ts time series to the built-in AirPassengers data set.

### Time series plot

The first step in any data analysis is to plot the data. Graphs enable you to visualize many features of the data, including patterns, unusual observations, changes over time, and relationships between variables. Just as the type of data determines which forecasting method to use, it also determines which graphs are appropriate.

Here, we use the autoplot() function to produce time plots of ts data. In time series plots, we should always look for outliers, seasonal patterns, overall trends, and other interesting features. This plot starts to illustrate the obvious trends that emerge over time.

autoplot(AirPassengers)



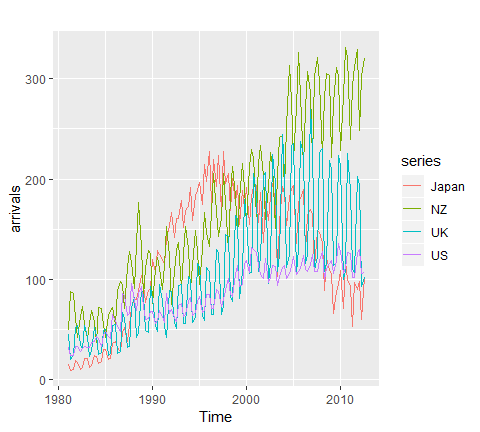
Often, we will have have time series data that has multiple variables. For example, the fpp2::arrivals data set has time series data for “quarterly international arrivals (in thousands) Australia from Japan, New Zealand, UK and the US 1919Q1 - 2012Q3”.

So this time series data has two variables (over and above the time stamp data) - (1) arrivals in thousands and (2) country.

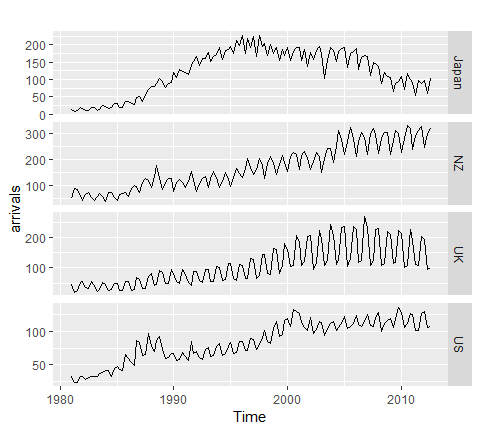
head(arrivals)  
## Japan NZ UK US  
## 1981 Q1 14.763 49.140 45.266 32.316  
## 1981 Q2 9.321 87.467 19.886 23.721  
## 1981 Q3 10.166 85.841 24.839 24.533  
## 1981 Q4 19.509 61.882 52.264 33.438  
## 1982 Q1 17.117 42.045 53.636 33.527  
## 1982 Q2 10.617 63.081 34.802 28.366

We can compare the trends across the different variables (countries) either in one plot or use the facetting option to separate the plots:

autoplot(arrivals)

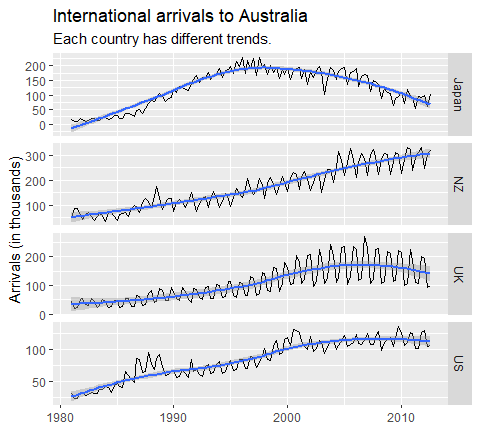


autoplot(arrivals, facets = T)



You may have noticed that autoplot() looks a lot like ggplot2 outputs. That’s because many of the visualizations in the forecast package are built on top of ggplot2. This allows us to easily add on to these plots with ggplot2 syntax. For example, we can add a smooth trend line and adjust titles:

autoplot(arrivals, facets = TRUE) +  
 geom\_smooth() +  
 labs(title = "International arrivals to Australia",  
 y = "Arrivals (in thousands)",  
 x = NULL,  
 subtitle = "Each country has different trends.")



These initial visualizations may spor additional questions such as what was the min, max, or average arrival amount for Japan. We can index and use many normail functions to assess these types of questions.

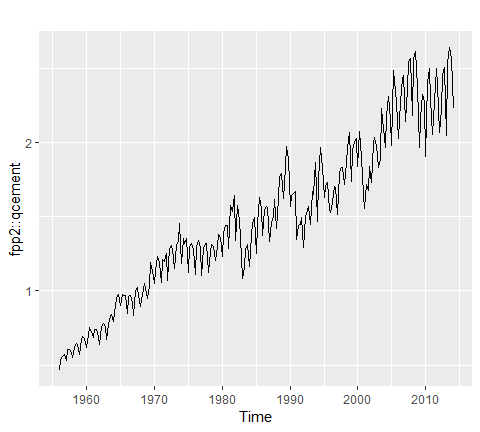
# index for Japan  
japan <- arrivals[, "Japan"]  
  
# Identify max arrival amount  
summary(japan)  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 9.321 74.135 135.461 122.080 176.752 227.641  
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 9.321 74.135 135.461 122.080 176.752 227.641

You can also use the frequency() function to get the number of observations per unit time. This example returns 4 which means the data are recorded on a quarterly interval.

frequency(japan)  
## [1] 4

In viewing time series plots we can describe different components. We can describe the common components using this quarterly cement production data.

autoplot(fpp2::qcement)



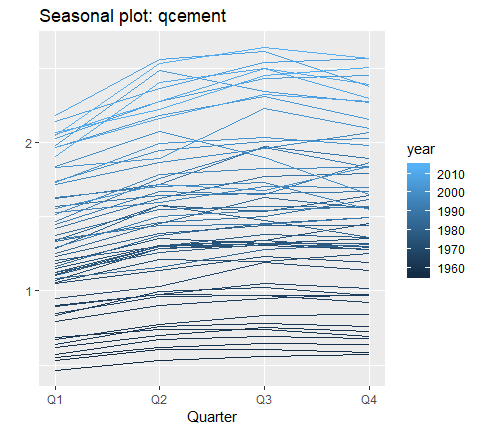
* the **trend** is the long-term increase or decrease in the data. There is an increasing trend in the cement data.
* the **seasonal** pattern occurs when a time series is affected by seasonal factors such as the time of the year or the day of the week. The quarterly cement data above shows seasonality likely induced by the change in weather and its impact on being able to pour cement.
* the **cycle** occurs when the data exhibit rises and falls that are not of a fixed period. These fluctuations are usually due to economic conditions and are often related to the “business cycle”. We can see a few cycles in our cement data in the early ’80s, ’90s, ’00s, and around 2008 - all these date ranges are around economic depressions that occurred.

Later tutorials will illustrate how to decompose each of these components; however, next we will look at how to do some initial investigating regarding seasonal patterns.

### Seasonal plots

There are a few useful ways of plotting data to emphasize seasonal patterns and show changes in these patterns over time. First, a seasonal plot is similar to a time plot except that the data are plotted against the individual “seasons” in which the data were observed. We can produce a seasonal plot with ggseasonplot():

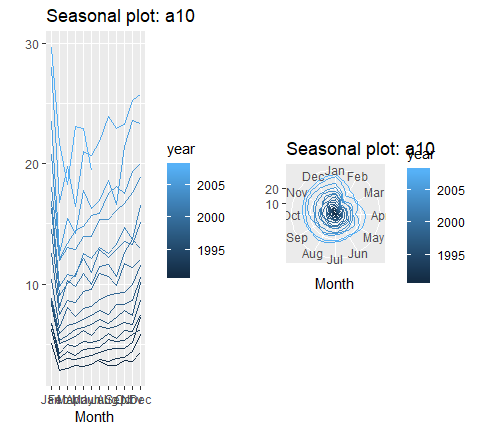
ggseasonplot(qcement, year.labels = FALSE, continuous = T)



This is the same qcement data shown above, but now the data from each season are overlapped. A seasonal plot allows the underlying seasonal pattern to be seen more clearly, and can be useful in identifying years in which the pattern changes. Here, we see that cement production has consistently increased over the years as the lower (darker) lines represent earlier years and the higher (lighter) lines represent recent years. Also, we see that cement production tends to be the lowest in Q1 and typically peaks in Q3 before leveling off or decreasing slightly in Q4.

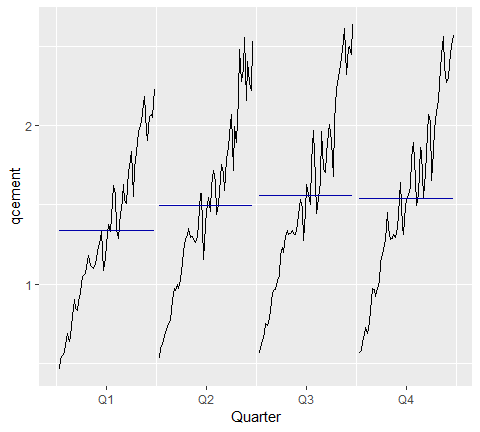
A particular useful variant of a season plot uses polar coordinates, where the time axis is circular rather than horizontal. Here, we plot the a10 data with the conventional seasonal plot versus a polar coordinate option to illustrate this variant. Both plots illustrate a sharp decrease in values in Feb and then a slow increase from Apr-Jan.

# left  
p1 <- ggseasonplot(a10, year.labels=FALSE, continuous=TRUE)  
  
#right  
p2 <- ggseasonplot(a10, year.labels=FALSE, continuous=TRUE, polar = TRUE)  
  
gridExtra::grid.arrange(p1,p2, ncol=2)



An alternative plot that emphasizes the seasonal patterns is where the data for each season (quarter in our example) are collected together in separate mini time plots. A subseries plot produced by ggsubseriesplot() creates mini time plots for each season. Here, the mean for each season is shown as a blue horizontal line.

ggsubseriesplot(qcement)



This form of plot enables the underlying seasonal pattern to be seen clearly, and also shows the changes in seasonality over time. It is especially useful in identifying changes within particular seasons. In this example, the plot is not particularly revealing; but in some cases, this is the most useful way of viewing seasonal changes over time.

### Autocorrelation of Time Series

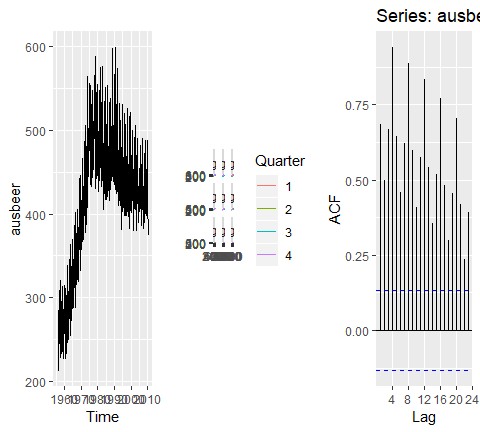
Another way to look at time series data is to plot each observation against another observation that occured some time previously. For example, you could plot against . This is called a lag plot because you are plotting the time series against lags of itself. The gglagplot() function produces various types of lag plots.

The correlations associated with the lag plots form what is called the “autocorrelation function”. Autocorrelation is nearly the same as correlation, which you can learn about in the Assessing Correlations tutorial. However, autocorrelation is the correlation of a time series with a delayed copy of itself. Autocorrelation between and for different values of k can be written as:

where T is the length of the time series. And similar to correlation, autocorrelation will always between +1 and -1.

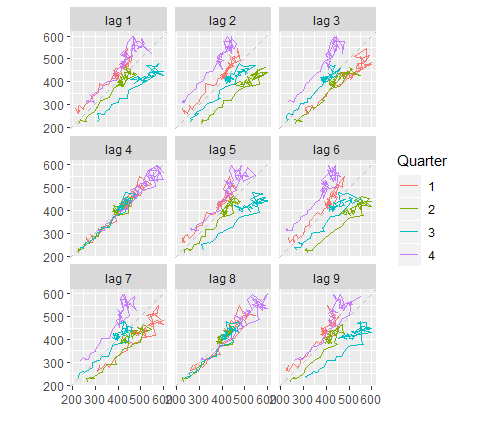
When these autocorrelations are plotted, we get an ACF plot. The ggAcf() function produces ACF plots. Here we look at the total quarterly beer production in Australia (in megalitres) from 1956:Q1 to 2010:Q2. The data are available in the fpp2::ausbeer time series data.

# left: autoplot of the beer data  
p1 <- autoplot(ausbeer)  
  
# middle: lag plot of the beer data  
p2 <- gglagplot(ausbeer)  
  
# right: ACF plot of the beer data  
p3 <- ggAcf(ausbeer)  
  
gridExtra::grid.arrange(p1,p2,p3, ncol=3)



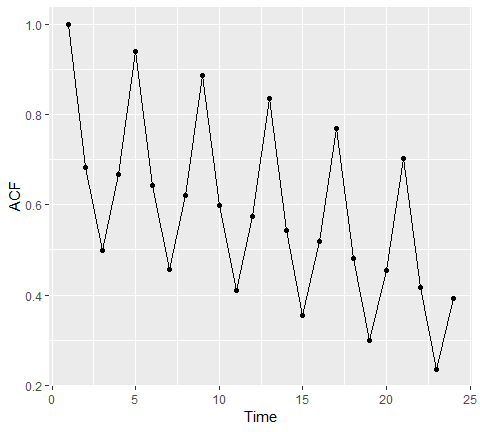
The middle plot provides the bivariate scatter plot for each level of lag (1-9 lags). The right plot provides a condensed plot of the autocorrelation values for the first 23 lags. The right plot shows that the greatest autocorrelation values occur at lags 4, 8, 12, 16, and 20. We can adjust the gglagplot to help illustrate this relationship. Here, we create a scatter plot for the first 16 lags. If you look at the right-most column (lags 4, 8, 12, 16) you can see that the relationship appears strongest for these lags, thus supporting our far right plot above.

p2



We can also access these autocorrelation values with Acf. Here, we can see that the autocorrelation for the two strongest lags (4 and 8) is 0.94 and 0.887.

acf <- acf(ausbeer, plot = F)  
  
acf$acf %>% as.tibble() %>%   
 rename(ACF =V1) %>%   
 mutate(Time = 1:24) %>%   
 select(Time, ACF) %>%   
 ggplot(aes(Time, ACF))+  
 geom\_point()+  
 geom\_line()

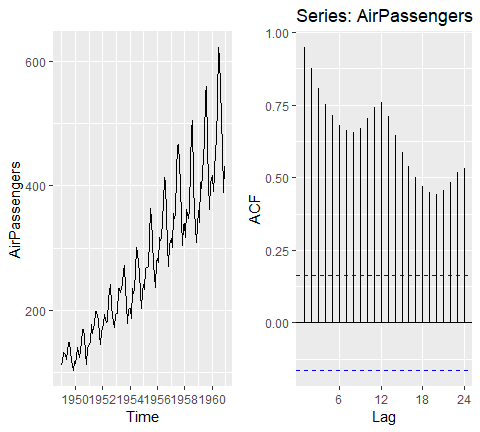


When data are either seasonal or cyclical, the ACF will peak around the seasonal lags or at the average cycle length. Thus, we see that the maximal autocorrelation for the ausbeer data occurs at a lag of 4 (right plot above). This makes sense since this is quarterly production data so the highest correlated value for a particular quarter will be the same quarter in the previous year.

A simplified approach to thinking about time series features and autocorrelation is as follows:

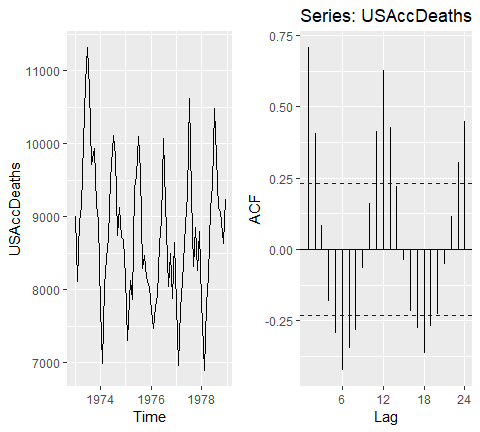
1. **Trends** induce passive correlations in the early lags. Strong trend will result in the more recent observations being of closer value to one another.

# left plot  
p1 <- autoplot(AirPassengers)  
  
# right plot  
p2 <- ggAcf(AirPassengers)  
  
gridExtra::grid.arrange(p1, p2, ncol=2)



1. **Seasonality** will induce peaks at the seasonal lags. Think about the holidays, each holiday will have certain products that peak at the time each year and so the strongest correlation will be the values at thatsame time each year.

# left plot  
p1 <- autoplot(USAccDeaths)  
  
# right plot  
p2 <- ggAcf(USAccDeaths)  
  
gridExtra::grid.arrange(p1, p2, ncol=2)



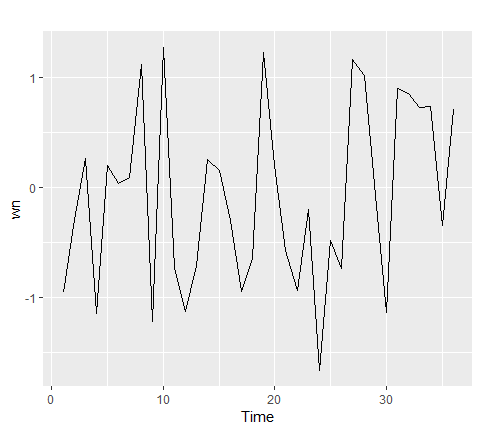
1. **Cyclicity** induces peaks at the average cycle length. Here we see that there tends to be cyclic impact to the mink population every 10 years. We also see this cause a peak in the ACF plot.

### White Noise

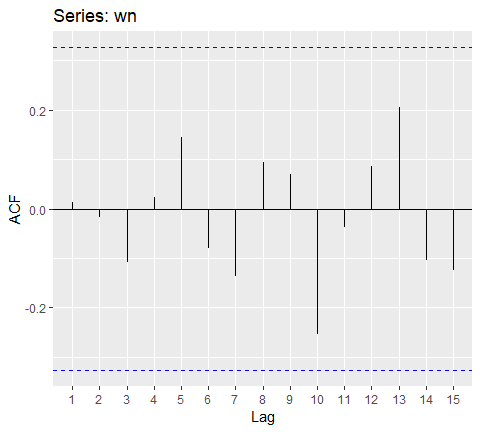
Time series that show no autocorrelation are called “white noise”. For example, the following plots 36 random numbers and illustrates a white noise series.

This data is considered independent and identically distributed (**iid**) because there is no trend, no seasonality, no autocorrelation … just randomness.

set.seed(3)  
wn <- ts(rnorm(36))  
autoplot(wn)



ggAcf(wn)

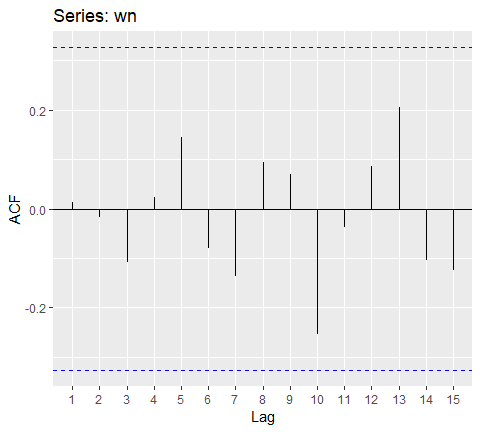


For white noise series, we expect each autocorrelation to be close to zero. Of course, they are not exactly equal to zero as there is some random variation. For a white noise series, we expect 95% of the spikes in the ACF to lie within where T is the length of the time series.

It is common to plot these bounds on a graph of the ACF. If there are one or more large spikes outside these bounds, or if more than 5% of spikes are outside these bounds, then the series is probably not white noise.

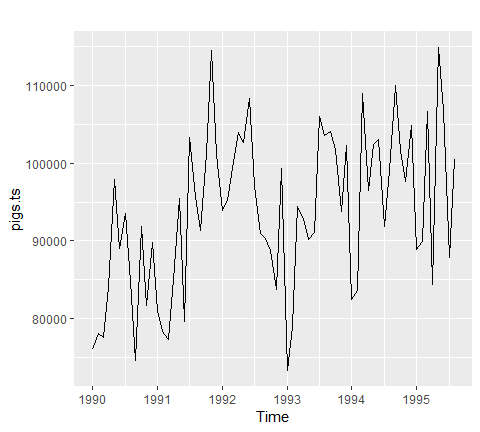
When using ggAcf, the dotted blue lines represent the 95% threshold. Here, we see that none of the autocorrelations exceed the blue line os we can be confident that there is no time series component to this data.

ggAcf(wn)

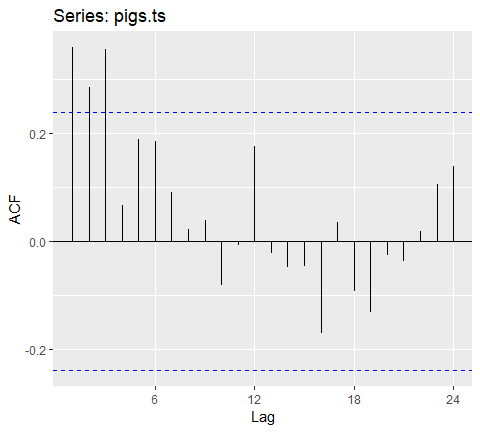


Assessing autocorrelation can be quite useful for data sets where trends and seasonalities are hard to see. For example, the following displays the monthly number of pigs slaughtered in Victoria, Australia from 1990-1995. There may be a slight trend over time but it is unclear.

pigs.ts <- ts(pigs[121:188], start = c(1990, 1), frequency = 12)  
autoplot(pigs.ts)

 However, looking at the ACF plot makes the feature more clear. There is more information in this data then the plain time series plot provided. We see that the first three lags clearly exceed the blue line suggesting there is possible some signal in this time series component that can be used in a forecasting approach.

ggAcf(pigs.ts)



The ACF plots test if an individual lag autocorrelation is different than zero. An alternative approach is to use the **Ljung-Box test**, which tests whether any of a group of autocorrelations of a time series are different from zero. In essence it tests the “overall randomness” based on a number of lags. If the result is a small p-value than it indicates the data are probably not white noise.

Here, we perform a Ljung-Box test on the first 24 lag autocorrelations. The resulting p-value is significant at , so this supports our ACF plot consideration above where we stated its likely this is not purely whitenoise and that some time series information exists in this data.

Box.test(pigs, lag = 24, fitdf = 0, type = "Lj")  
##   
## Box-Ljung test  
##   
## data: pigs  
## X-squared = 634.15, df = 24, p-value < 2.2e-16  
##   
## Box-Ljung test  
##   
## data: pigs  
## X-squared = 634.15, df = 24, p-value < 2.2e-16

## Benchmark Methods and Forecast Accuracy

In this tutorial, you will learn general tools that are useful for many different forecasting situations. t will describe some methods for benchmark forecasting, methods for checking whether a forecasting model has adequately utilized the available information, and methods for measuring forecast accuracy. Each of the tools discussed in this tutorial will be used repeatedly in subsequent tutorials as you develop and explore a range of forecasting methods.

### tl;dr

1. [Replication requirements](#TS_BM_RR): What you’ll need to reproduce the analysis.
2. [Naive Forecasting Methods](#TS_BM_Naive): A simple but useful benchmark approach.
3. [Fitted Values and Residuals](#TS_BM_Pred): Always check the residuals.
4. [Training and Test Sets](#TS_BM_Split): How to partition time series data.
5. [Evaluating Forecast Accuracy](#TS_BM_Eval): How to evaluate accuracy of non-seasonal and non-season forecast methods.
6. [Time Series Cross-validation](#TS_BM_CV): A more sophisticated approach for evaluating predictive accuracy.
7. Exercises: Practice what you’ve learned.

### Replication Requirements

This tutorial leverages a variety of data sets to illustrate unique time series features. The data sets are all provided by the forecast and fpp2 packages. Furthermore, these packages provide various functions for computing and visualizing basic time series components.

library(forecast)  
library(fpp2)

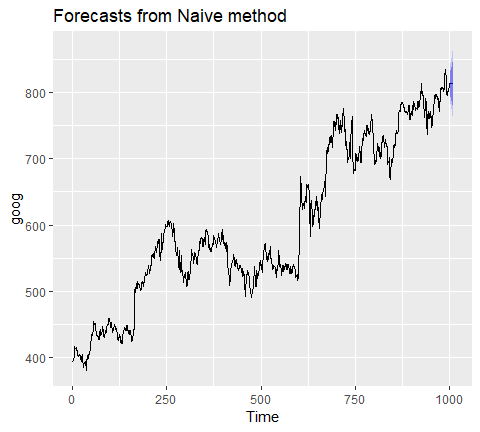
### Naive Forecasting Methods

Although it is tempting to apply “sophisticated” forecasting methods, one must remeber to consider *naive forecasting*. A naive forecast is simply the most recently observed value. In other words, at the time *t*, the *k*-step-ahead naive forecast () equals the observed value at time t().

Sometimes, this is the best that can be done for many time series including most stock price data (for reasons illustrated in the [previous tutorial’s exercise](http://uc-r.github.io/ts_exploration#exercises)). Even if it is not a good forecasting method, it provides a useful benchmark for other forecasting methods. We can perform a naive forecast with the naive function.

Here, we use naive to forecast the next 10 values. the resulting output is an object of class forecast. This is the core class of objects in the forecast package, and there are many functions for dealing with them. We can print off the model summary with summary(), which provides us with the residual standard deviation, some error measures and actual forecasted values.

fc\_goog <- naive(goog, 10)  
summary(fc\_goog)  
##   
## Forecast method: Naive method  
##   
## Model Information:  
## Call: naive(y = goog, h = 10)   
##   
## Residual sd: 8.7285   
##   
## Error measures:  
## ME RMSE MAE MPE MAPE MASE  
## Training set 0.4212612 8.734286 5.829407 0.06253998 0.9741428 1  
## ACF1  
## Training set 0.03871446  
##   
## Forecasts:  
## Point Forecast Lo 80 Hi 80 Lo 95 Hi 95  
## 1001 813.67 802.4765 824.8634 796.5511 830.7889  
## 1002 813.67 797.8401 829.4999 789.4602 837.8797  
## 1003 813.67 794.2824 833.0576 784.0192 843.3208  
## 1004 813.67 791.2831 836.0569 779.4322 847.9078  
## 1005 813.67 788.6407 838.6993 775.3910 851.9490  
## 1006 813.67 786.2518 841.0882 771.7374 855.6025  
## 1007 813.67 784.0549 843.2850 768.3777 858.9623  
## 1008 813.67 782.0102 845.3298 765.2505 862.0895  
## 1009 813.67 780.0897 847.2503 762.3133 865.0266  
## 1010 813.67 778.2732 849.0667 759.5353 867.8047  
autoplot(fc\_goog)



You will notice the forecast output provides a point forecast(the last observed value in the goog data set) and prediction confidence levels at the 80% and 95% level. A prediction interval gives an inbterval within which we expect to lie with a specified probability. For example, assuming the forecast errors are uncorrelated and normally distributed, then a simple 95% prediction interval for the next observation in a time series is

where an estimate of the standard deviation of the forecast distribution. In forecasting, it is common to calculate 80% intervals and 95% intervals, although any percentage may be used.

When forecasting one-step ahead, the standard deviation of the forecast distribution is almost the same as the standard deviation of the residuals. (In fact, the two standard deviations are identical if there are no parameters to be estimated such as with the naive method. For forecasting methods involving parameters to be estimated, the standard deviation of the forecast distribution is slightly larger than the residual standard deviation, although this difference is often ignored.)

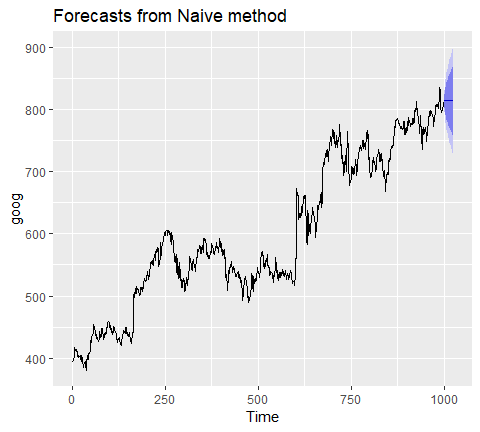
For example, consider our naive forecast for the goog data. The last value of the observed series is 838.96, so the forecast of the next value is 838.96 and the standard deviation of the residuals from the naive method is 8.91. Hence, a 95% prediction interval for the next value of goog is

Similarl, an 80% prediction interval is given by

The vsalue of prediction intervals is that the express the uncertainty in the forecasts. If we only produce point forecasts, there is no way of telling how accurate the forecats are. But if we also produce prediction intervals, then it is clear how much uncertainty is associated with each forecast. Thus, with the naive forecast on the next goog value, we can be 80% confident that the next value will be in the range of 828-850 and 95% confident that the the value will be between 821-856.

We can illustrate this prediction interval by plotting the naive model (fc\_goog). Here, we see the black point estimate line flat-line (equal to the last observed value) and the colored bands represent our 80% and 95% prediction confidence interval. A common feature of prediction intervals is that they increase in length as the forecast horizon increases. The further ahead we forecast, the more uncertainty is associated with the forecast, and so the prediction intervals grow wider.

# forecast next 25 values  
fc\_goog <- naive(goog, 25)  
autoplot(fc\_goog)



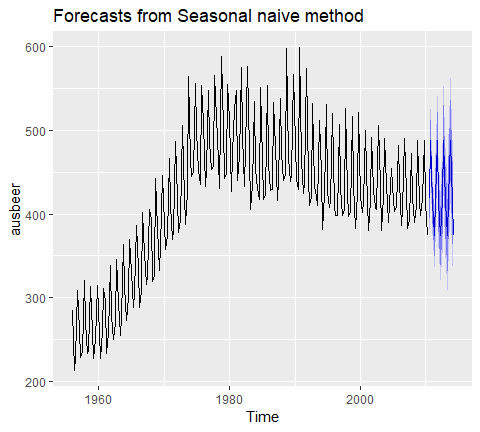
For seasonal data, a related idea is to use the corresponding season from the last year of data. For example, if you want to forecast the sales volume for next March, you would use the sales volume from the previous March. For a series with M seasons, we can write this as

This is implemented in the snaive function meaning, *seasonal naive*. Here, I use snaive to forecast the next 16 values for the ausbeer series. Here, we see that the 4th quarter for each future year is 488 which is the last observed 4th quarter value in 2009.

fc\_beer <- snaive(ausbeer, 16)  
summary(fc\_beer)  
##   
## Forecast method: Seasonal naive method  
##   
## Model Information:  
## Call: snaive(y = ausbeer, h = 16)   
##   
## Residual sd: 19.1207   
##   
## Error measures:  
## ME RMSE MAE MPE MAPE MASE ACF1  
## Training set 3.098131 19.32591 15.50935 0.838741 3.69567 1 0.01093868  
##   
## Forecasts:  
## Point Forecast Lo 80 Hi 80 Lo 95 Hi 95  
## 2010 Q3 419 394.2329 443.7671 381.1219 456.8781  
## 2010 Q4 488 463.2329 512.7671 450.1219 525.8781  
## 2011 Q1 414 389.2329 438.7671 376.1219 451.8781  
## 2011 Q2 374 349.2329 398.7671 336.1219 411.8781  
## 2011 Q3 419 383.9740 454.0260 365.4323 472.5677  
## 2011 Q4 488 452.9740 523.0260 434.4323 541.5677  
## 2012 Q1 414 378.9740 449.0260 360.4323 467.5677  
## 2012 Q2 374 338.9740 409.0260 320.4323 427.5677  
## 2012 Q3 419 376.1020 461.8980 353.3932 484.6068  
## 2012 Q4 488 445.1020 530.8980 422.3932 553.6068  
## 2013 Q1 414 371.1020 456.8980 348.3932 479.6068  
## 2013 Q2 374 331.1020 416.8980 308.3932 439.6068  
## 2013 Q3 419 369.4657 468.5343 343.2438 494.7562  
## 2013 Q4 488 438.4657 537.5343 412.2438 563.7562  
## 2014 Q1 414 364.4657 463.5343 338.2438 489.7562  
## 2014 Q2 374 324.4657 423.5343 298.2438 449.7562

Similar to naive, we can plot the snaive model with autoplot.

autoplot(fc\_beer)



### Fitted Values and Residuals

When applying a forecasting method, it is important to always check that the residuals are well-behaved (i.e., no outliers or patterns) and resemble white noise. Essential assumptions for an appropriate forecasting model include residuals being:

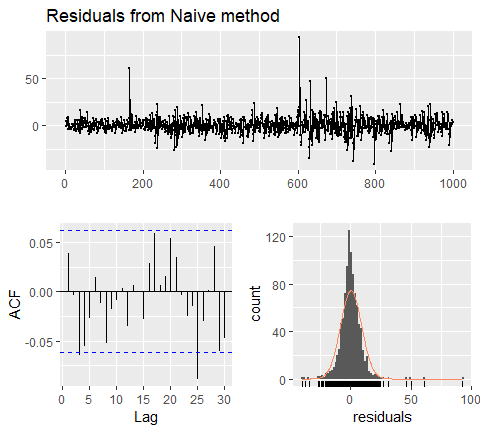
* uncorrelated
* centered at mean zero

Furthermore, the prediction intervals are computed assuming that the residuals:

* have constant variance
* are normally distributed

A convenient function to use to check these assumptions is the checkresiduals function. This function produces a time plot, ACF plot, histogram, and a **Ljung-Box test** on the residuals. Here, I use checkresiduals for the fc\_goog naive model. We see that the top plot shows residuals that appear to be white noise (no discernable pattern), the bottom left plot shows only a couple lags that exceed the 95% confidence interval, bottom right plot shows the residuals to be approximately normally distributed, and the Ljung-Box test results give a p-value of 0.22 suggesting the residuals are white noise. This is a good thing as it suggests that are model captures all (or most) of the available signal in the data.

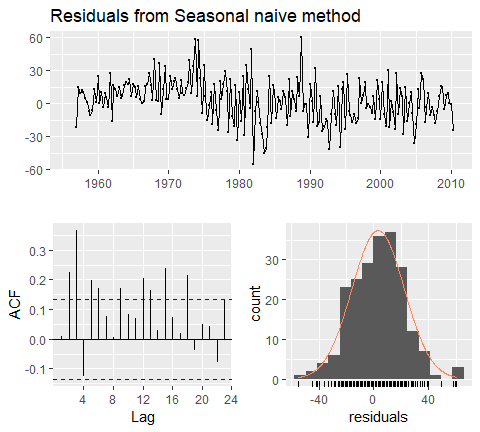
checkresiduals(fc\_goog)



##   
## Ljung-Box test  
##   
## data: Residuals from Naive method  
## Q\* = 13.123, df = 10, p-value = 0.2169  
##   
## Model df: 0. Total lags used: 10

If we compare that to the fc\_beer seasonal naive model we see that there is an apparent pattern in the residual time series plot, the ACF plot shows several lags exceeding the 95% confidence interval, and the Ljung-Box test has a statistically significant p-value suggesting the residuals are not purely white noise. This suggests that there may be another model or additional variables that will better capture the remaining signal in the data.

checkresiduals(fc\_beer)



##   
## Ljung-Box test  
##   
## data: Residuals from Seasonal naive method  
## Q\* = 60.535, df = 8, p-value = 3.661e-10  
##   
## Model df: 0. Total lags used: 8

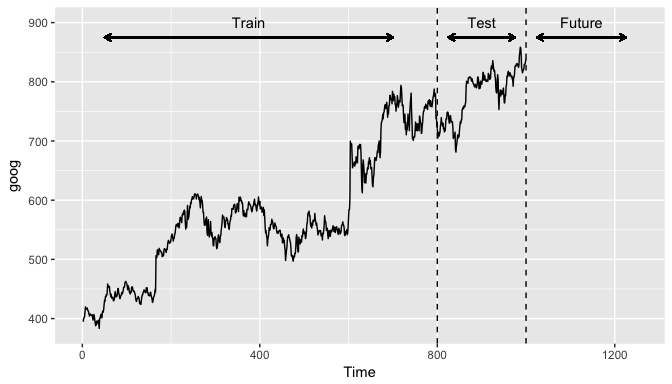
### Training & Test sets

A training set is a data set that is used to discover possible relationships. A test set is a data set that is used to verify the strength of these potential relationships. When you separate a data set into these parts, you generally allocate more of the data for training, and less for testing. There is one important difference between data partitioning in cross-sectional and time series data. In cross-sectional data the partitioning is usually done randomly, with a random set of observations designated as training data and the remainder as test data. However, in time series, a random partition creates two problems:

1. It does not mimic the temporal uncertainty where we use the past and present to forecast the future.
2. It creates two time series with “holes”, whereas many standard forecasting methods cannnot handle time series with missing value.

Therefore, time series partitioning into training and test sets is done by taking a training partition from earlier observations and then using a later partition for the test set.

knitr::include\_graphics("image/partitioning-1.png")



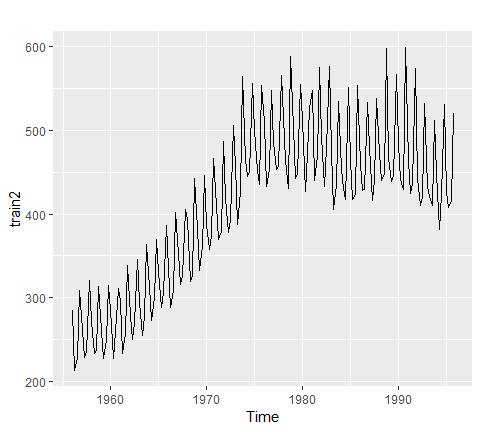
One function that can be used to create training and test sets is subset.ts(), which returns a subset of a time series where the start and end of the subset are specified using *index* values. The gold time series comprises daily gold prices over 1108 traiding days. Let’s use the first 1000 days as a training set. We can also create a test data set of the remaining data.

train <- subset.ts(gold, end = 1000)  
test <- subset.ts(gold, start = 1001, end = length(gold))

A similar approach can be used for data where we want to maintain season features. For example, if we use subset.ts to take the first 1000 observations this may include half of a year in the training data and the other half in the test data.

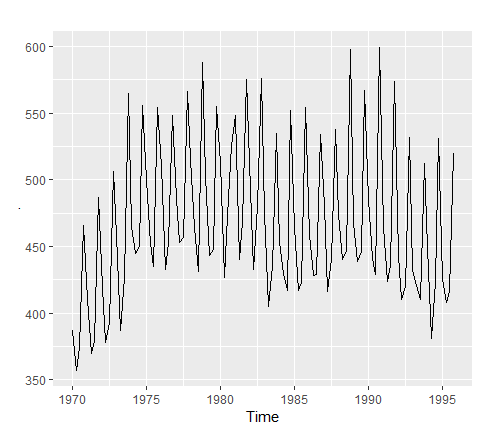
Rather, for forecasting methods that include seasonal features (i.e. snaive), we prefer to not split a full cycle of seasons (i.e. year, month, week). We can use the window function to do this. For example, the ausbeer data set has quarterly data from 1956-2010. Here, we take data from 1956 through the 4th quarter of 1995 to be a training data set.

train2 <- window(ausbeer, end = c(1995, 4))  
autoplot(train2)



The window function is also particularly useful when we want to take data from a defined period. For example, if we decide that data from 1956-1969 is not appropriate due to regulatory changes then we can create a training data set from the 1st qtr of 1970 through the 4th qtr of 1995.

train3 <- window(ausbeer, start = c(1970, 1), end = c(1995, 4))  
train3 %>% autoplot()



### Evaluating forecast accuracy

For evaluating predictive performance, several measures are commonly used to assess the predictive accuracy of a forecasting method. In all cases, the *measures are based on the test data set*, which services as a more objective basis than the training period to assess predictive accuracy. Given a forecast and its given errors (), the commonly used accuracy measures are listed below.

Definition

* Observation:
* Forecast:
* Forecast error:

Accuracy Measure: - Mean absolute error: MAE = - Mean squared error: MSE = - Mean absolute percentage error: - Mean absolute scaled error:

Note that each measure has its strengths and weaknesses. For example, if you want to compare forecast accuracy between two series on very different scales you can’t compare the MAE or MSE for these forecast as these measures depend on the scale of the time series data. MAPE is often better for comparisons but only if our data are all positive and have no zeros or small values. It also assumes a natural zero so it can’t be used for temperature forecasts as these are based on arbitrary zero scales. MASE is similar to MAE but is scaled so that it can be compared across different data series. You can read more about each of these measures [here](https://www.otexts.org/fpp/2/5); however, for now just keep in mind that for all these measures a smaller value signifies a better forecast.

#### Evaluating forecast accuracy of no-seasonal methods

We can compute all of these measures by using the accuracy function. The accuracy measures provided also include root mean squared error (RMSE) which is the square root of the mean squared error (MSE). Minimizing RMSE, which corresponds with increasing accuracy, is the same as minimizing MSE. In addition, other accuracy measures not illustrated above are also provided (i.e. ACF1 - autocorrelation at lag 1; Theil’s U). The output of accuracy allows us to compare these accuracy measures for the residuals of the training data set against the forecast errors of the test data. However, our main concern is how well different forecasting methods improve the predictive accuracy on the test data.

Using the training data we created from the gold data set (train), let’s create two different forecasts:

1. A naive forecast and
2. a forecast equal to the mean of all observations

Here, I use h = 108 to predict for the next 108 days (note that the gold data set has 1108 observations, the training data set has 1000, so we want to predict the next 108 observations and compare that to the test data)

# create training data  
train <- subset.ts(gold, end = 1000)  
  
# compute naive forecasts and save to naive\_fc  
naive\_fc <- naive(train, h = 108)  
  
# compute mean forecasts and save to mean\_fc  
accuracy(naive\_fc, gold)  
## ME RMSE MAE MPE MAPE MASE  
## Training set 0.09161392 6.33977 3.158386 0.01662141 0.794523 1  
## Test set -6.53834951 15.84236 13.638350 -1.74622688 3.428789 NA  
## ACF1 Theil's U  
## Training set -0.3098928 NA  
## Test set 0.9793153 5.335899  
accuracy(mean\_fc, gold)  
## ME RMSE MAE MPE MAPE MASE  
## Training set -4.239671e-15 59.17809 53.63397 -2.390227 14.230224 16.98145  
## Test set 1.319363e+01 19.55255 15.66875 3.138577 3.783133 NA  
## ACF1 Theil's U  
## Training set 0.9907254 NA  
## Test set 0.9793153 6.123788

What accuracy is doning is taking the model output from the training data (naive\_fc & means\_fc) and computing the accuracy measures of the 108 forecasted values to those values in gold that are not included in the model (aka test data). This means we do not need to directly feed it a test data set, although we do have the option as illustrated as below:

train <- subset.ts(gold, end = 1000)  
test <- subset.ts(gold, start = 1001, end = length(gold))  
naive\_fc <- naive(train, h = 108)  
accuracy(naive\_fc, test)  
## ME RMSE MAE MPE MAPE MASE  
## Training set 0.09161392 6.33977 3.158386 0.01662141 0.794523 1  
## Test set -6.53834951 15.84236 13.638350 -1.74622688 3.428789 NA  
## ACF1 Theil's U  
## Training set -0.3098928 NA  
## Test set 0.9793153 5.335899

If we compare the test set accuracy measures for both models we see that the naive approach has lower scores across all measures indicating better forecasting accuracy.

#### Evaluating forecast accuracy of seasonal methods

We can use a similar approach to evaluate the accuracy of seasonal forecasting models. The primary difference is we want to use the window function for creating our training data so that we appropriately capture seasonal cycles.

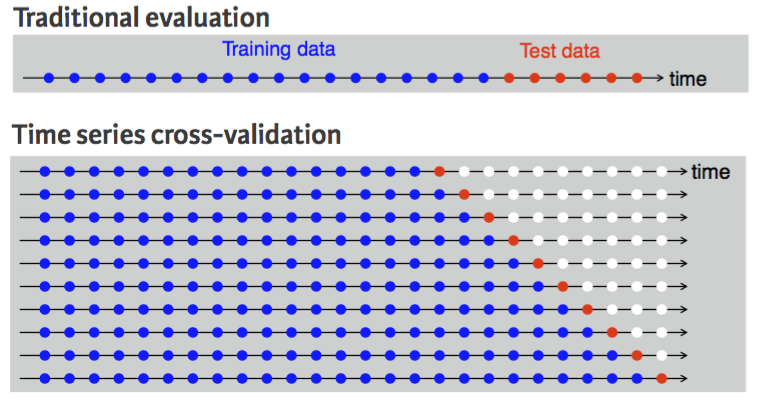
Here, I illustrate with the ausbeer data set. We see the snaive model produces lower scores across all measures indicating better forecasting accuracy.

# create training data  
train2 <- window(ausbeer, end = c(1995, 4))  
  
# create specific test data of interest  
test <- window(ausbeer, start = c(1996, 1), end = c(2004, 4))  
  
# Compute snaive forecasts and save to snaive\_fc  
snaive\_fc <- snaive(train2, h = length(test))  
  
# Compute mean forecasts and save to mean\_fc  
mean\_fc <- meanf(train2, h = length(test))  
  
# Use accuracy() to compute forecast accuracy  
accuracy(snaive\_fc, test)  
## ME RMSE MAE MPE MAPE MASE  
## Training set 4.730769 20.60589 16.51282 1.284697 3.957622 1.0000000  
## Test set -8.527778 18.06854 13.08333 -2.011727 3.038430 0.7923137  
## ACF1 Theil's U  
## Training set 0.01783674 NA  
## Test set -0.39101873 0.2890478  
## ME RMSE MAE MPE MAPE MASE  
## Training set 4.730769 20.60589 16.51282 1.284697 3.957622 1.0000000  
## Test set -8.527778 18.06854 13.08333 -2.011727 3.038430 0.7923137  
## ACF1 Theil's U  
## Training set 0.01783674 NA  
## Test set -0.39101873 0.2890478  
accuracy(mean\_fc, test)  
## ME RMSE MAE MPE MAPE MASE  
## Training set -2.279427e-14 96.37835 78.64437 -6.646984 21.996740 4.762625  
## Test set 2.393472e+01 49.38745 34.07500 4.640789 7.243654 2.063548  
## ACF1 Theil's U  
## Training set 0.72972747 NA  
## Test set -0.09479634 0.8164006  
## ME RMSE MAE MPE MAPE MASE  
## Training set -2.279427e-14 96.37835 78.64437 -6.646984 21.996740 4.762625  
## Test set 2.393472e+01 49.38745 34.07500 4.640789 7.243654 2.063548  
## ACF1 Theil's U  
## Training set 0.72972747 NA  
## Test set -0.09479634 0.8164006

### Time series cross-validation

A more sophisticted version of training/test sets is cross validation. You can see how CV works for cross-sectional data [here](http://uc-r.github.io/resampling_methods). For time series data, the procedure is similar but the training set consists only of observations that occurred prior to the observation that forms the test set. So in traditional time series partitioning, we select a certain point in time where everything before that point (blue) is the training data and everything after that point (red) is the test data.

knitr::include\_graphics("image/ts\_validation.png")

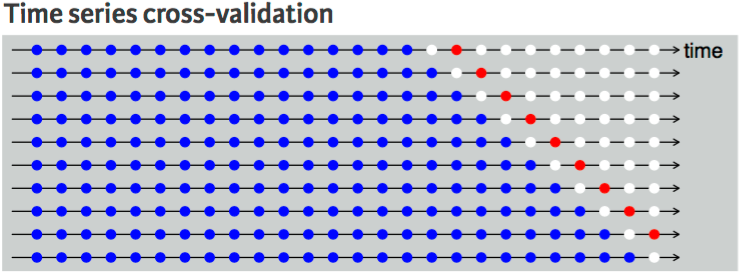
 However, assuming we want to perform 1-step forecast (predicting the next value in the series), time series cross-validation will:

1. Select the observation at, and prior to, time k (blue dots).
2. Select the observation at for the test data (red dots).
3. Discard the observation (white dots)
4. Compute the error on the forecast for time .
5. Repeat steps 1-4 above for where is the total number of observations.
6. Compute the forecast accuracy measures based on the errors obtained.

This procedure is sometimes known as a “rolling forecast origin” because the “origin” at which the forecast is based rolls foward in times as displayed by each row in the above illustration.

With time series forecasting, one-step forecasts may not be as relevant as multi-step forecasts. In this case, the cross-validation procedure based on a rolling forecasting origin can be modified to allow multi-step errors to be used. Suppose we are interested in models that produce good h-step-ahead forecasts. Here, we simply adjust the above algorithm so that we select the observation at time for the test set, use the observations at time to estimate the forecasting model, compute the step error on the forecast for time , rinse & repeat until we can compute the forecasting accuracy for all errors caluclated. For a 2-step-ahead forecast this looks like

knitr::include\_graphics("image/two\_step\_cv.png")



This seems tedious, however, there is a simple function that implements this procedure. The tsCV function applies a forecasting model to a sequence of training sets from a give time series and provides the errors as the output. However, we need to compute our own accuracy measures with these errors.

As an example, let’s perform a cross-validation approach for a 1-step ahead (h=1) naive model with the goog data. We can then compute the MSS which is 79.59.

errors <- tsCV(goog, forecastfunction = naive, h = 1)  
mean(errors^2, na.rm = TRUE)  
## [1] 76.28775  
## [1] 79.58582

We can compute and compare the MSE for different forecast horizons (1-10) to see if certain forecasting horizons perform better than others. Here, we see that as the forecasting horizon extends the predictive accuracy becomes poorer.

# create empty vector to hold MSE values  
MSE <- vector("numeric", 10)  
for(h in 1:10) {  
 errors <- tsCV(goog, forecastfunction = naive, h = h)  
 MSE[h] <- mean(errors^2, na.rm = TRUE)  
}  
  
MSE  
## [1] 76.28775 117.60458 158.87772 197.78048 234.10338 268.50152 302.10622  
## [8] 335.04660 366.69461 397.21987  
## [1] 79.58582 167.14427 253.92996 330.20539 398.47090 464.83012 531.09256  
## [8] 596.40191 654.25322 712.70988

## Moving Average

Smoothing methods are a family of forecasting methods that average values over multiple periods in order to reduce the noise and uncoiver patterns in the data.

Moving averages are one such smoothing method. Moving averages is a smoothing approach that averages values from a window of consecutive time periods, thereby generating a series of averages. The moving average approaches primarily differ based on the number of values averaged, how the average is computed, and how many times averaging is performed. This tutorial will walk you through the basics of performing moving averages.

### tl;dr

1. [Replication Requirements](#TS_MA_RR): What you’ll need to replicate the analysis in this tutorial
2. [Centered Moving Averages](#TS_CMA): Technique for calculating and displaying a simple moving average
3. [Moving Averages of Moving Averages](#TS_MAMA): Using the concept of simple moving averages to perform multi-step smoothing
4. [Weighted Moving Averages](#TS_WMA): Smoothing by use of weights specifically chosen for their mathematical properties

### Replication requirements

There are four packages outside of the base set of functions that will be used in the tutorial:

library(tidyverse) # data manipulation and visualization  
library(lubridate) # easily work with dates and times  
library(fpp2) # working with time series data  
library(zoo) # working with time series data

### Centered Moving Averages

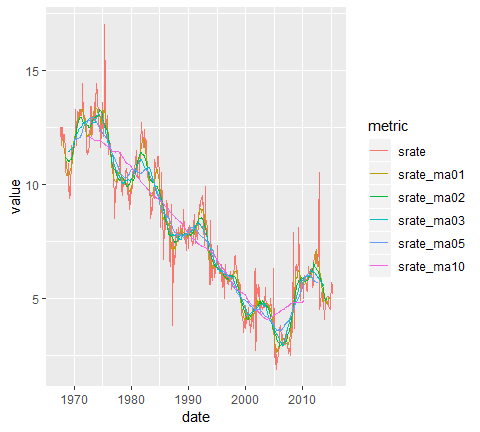
The most straightforward method is called a simple moving average. For this method, we choose a number of nearby points and average them to estimate the trend. When calculating a simple moving average, it is beneficial to use an odd number of points so that the calculation is symmetric. For example, to calculate a 5 point moving average, the formula is:

where t is the time step that you are smoothing at and 5 is the number of points being used to calculate the average (which moving forward will be denoted as k). To compute moving averages on our data we can leverage the rollmean function from the zoo package. Here, we focus on the personal savings rate (psavert) variable in the economics data frame. Using mutate and rollmean, I compute the 13, 25, …, 121 month moving average values and add this data back to the data frame. Note that we need to explicitly state to fill any years that cannot be computed (due to lack of data) with NA.

savings <- economics %>%   
 select(date, srate = psavert) %>%   
 mutate(srate\_ma01 = rollmean(srate, k = 13, fill = NA),  
 srate\_ma02 = rollmean(srate, k = 25, fill = NA),  
 srate\_ma03 = rollmean(srate, k = 37, fill = NA),  
 srate\_ma05 = rollmean(srate, k = 61, fill = NA),  
 srate\_ma10 = rollmean(srate, k = 121, fill = NA))

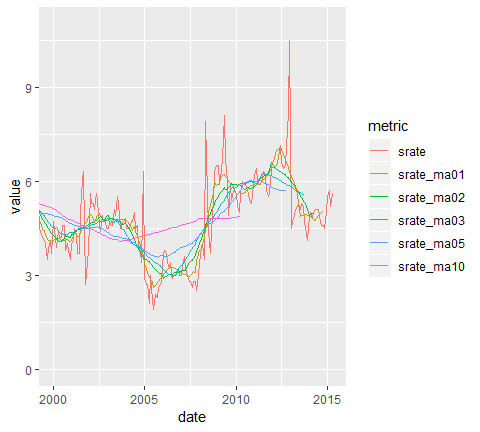
Now we can go ahead and plot these values and compare the actual data to the different moving average smoothers.

savings %>%   
 gather(metric, value, srate:srate\_ma10) %>%   
 ggplot(aes(date, value, color = metric))+  
 geom\_line()



You may notice that as the number of points used for the average increases, the curve becomes smoother. Choosing a value for is a balance between eliminating noise while still capturing the data’s true structure. For this set, the 10 year moving average () eliminates most of the pattern and is probabily little more than just looking at the data itself. We can see this by zooming into the 2000-2015 time range:

savings %>%  
 gather(metric, value, srate:srate\_ma10) %>%  
 ggplot(aes(date, value, color = metric)) +  
 geom\_line() +  
 coord\_cartesian(xlim = c(date("2000-01-01"), date("2015-04-01")), ylim = c(0, 11))



To understand how thse different moving averages compare we can compute the [MSE and MAPE](http://uc-r.github.io/ts_benchmarking#accuracy). Both of these error rates will increase as you choose a larger to average over; however, if you or your leadership are indifferent between a 6-9% error rate then you may want to illustrate trends with a 3 year moving average rather than a 1 year moving average.

savings %>%   
 gather(metric, value, srate\_ma01:srate\_ma10) %>%   
 group\_by(metric) %>%   
 summarise(  
 MSE = mean((srate - value)^2, na.rm=TRUE),  
 MAPE = mean(abs((srate - value)/srate), na.rm=TRUE))  
## # A tibble: 5 x 3  
## metric MSE MAPE  
## <chr> <dbl> <dbl>  
## 1 srate\_ma01 0.391 0.0642  
## 2 srate\_ma02 0.564 0.0822  
## 3 srate\_ma03 0.724 0.0956  
## 4 srate\_ma05 0.832 0.109   
## 5 srate\_ma10 1.32 0.152

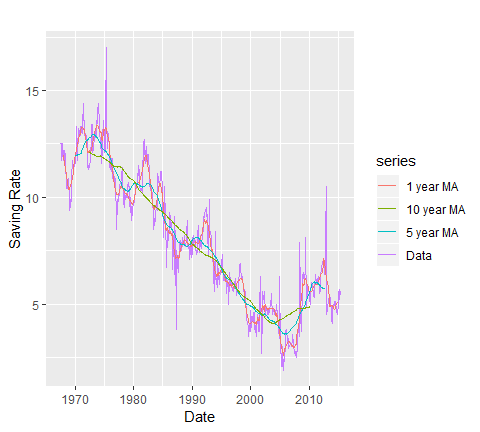
#### Using the fpp2 package

A simple moving average can also be plotted by using autoplot() contained in the fpp2 package. This is helpful if your data is already in time series data object. For example, if our savings rate data were already converted to a time series object as here…

savings.ts <- economics %>%   
 select(srate = psavert) %>%   
 ts(start = c(1967, 7), frequency = 12)  
  
head(savings.ts, 30)  
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
## 1967 12.5 12.5 11.7 12.5 12.5 12.1  
## 1968 11.7 12.2 11.6 12.2 12.0 11.6 10.6 10.4 10.4 10.6 10.4 10.9  
## 1969 10.0 9.4 9.9 9.5 10.0 10.9 11.7 11.5 11.5 11.3 11.5 11.7

we can plot this data with autoplot. Here, the data is plotted in line 1 of the following code, while the moving average (calculated using the ma() function) is plotted in the second layer.

autoplot(savings.ts, series = "Data")+  
 autolayer(ma(savings.ts, 13), series = "1 year MA")+  
 autolayer(ma(savings.ts, 61), series = "5 year MA")+  
 autolayer(ma(savings.ts, 121), series = "10 year MA")+  
 xlab("Date")+  
 ylab("Saving Rate")



#### Trailing Moving Averages for Forecasting

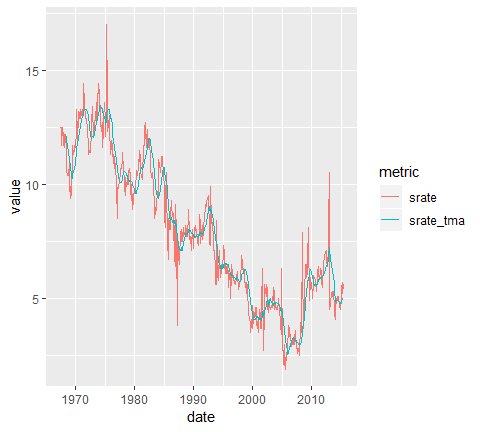
Centered moving averages are computed by averaging across data both in the past and future of a given time point. In that sense they cannot be used for forecasting because at the time of forecasting, the future is typically unknown. Hence, for purposes of forecasting, we use trailing moving averages, where the window of k periods is placed over the most recent available k values of the series. For example, if we have data up to time period t, we can predict the value for t+1 by averaging over k periods prior to t+1. If we want to use the 5 most recent time periods to predict for t+1 then our function looks like:

So, if we wanted to predict the next month’s savings rate based on the previous year’s average, we can use rollmean with the align = "right" argument to compute the training moving average. We can see that if we wanted to predict what the saving rates would be for 2015-05-01 based on the last 12 months, our prediction would be 5.06% (the 12-month average for 2015-04-01). This is now similar to using a naive forecast but with an averaged value rather than the last actual value.

savings\_tma <- economics %>%   
 select(date, srate = psavert) %>%   
 mutate(  
 srate\_tma = rollmean(srate, k = 12, fill = NA, align = "right")  
 )  
  
tail(savings\_tma,12)  
## # A tibble: 12 x 3  
## date srate srate\_tma  
## <date> <dbl> <dbl>  
## 1 2014-05-01 5.1 4.90  
## 2 2014-06-01 5.1 4.88  
## 3 2014-07-01 5.1 4.88  
## 4 2014-08-01 4.7 4.83  
## 5 2014-09-01 4.6 4.78  
## 6 2014-10-01 4.6 4.77  
## 7 2014-11-01 4.5 4.79  
## 8 2014-12-01 5 4.87  
## 9 2015-01-01 5.5 4.92  
## 10 2015-02-01 5.7 4.97  
## 11 2015-03-01 5.2 5.01  
## 12 2015-04-01 5.6 5.06

We can visualize how the 12-month trailing moving average predicts future savings rates with the following plot. It’s easy to see that trailing moving averages have a delayed reaction to changes in patterns and trends.

savings\_tma %>%   
 gather(metric, value, -date) %>%   
 ggplot(aes(date, value, color = metric))+  
 geom\_line()



### Moving averages of moving averages

The concept of simple moving averages can be extended to taking moving averages of moving averages. This technique is often employed with an even number of data points so that the final product is symmetric around each point. For example, let’s look at the built-in data set elecsales provided by the fpp2 package. For our first example we convert to a data frame. This data frame is even numbered with 20 rows.

# convert to data frame   
elecsales.df <- data.frame(year = time(elecsales),  
 sales = elecsales)  
  
elecsales.df  
## year sales  
## 1 1989 2354.34  
## 2 1990 2379.71  
## 3 1991 2318.52  
## 4 1992 2468.99  
## 5 1993 2386.09  
## 6 1994 2569.47  
## 7 1995 2575.72  
## 8 1996 2762.72  
## 9 1997 2844.50  
## 10 1998 3000.70  
## 11 1999 3108.10  
## 12 2000 3357.50  
## 13 2001 3075.70  
## 14 2002 3180.60  
## 15 2003 3221.60  
## 16 2004 3176.20  
## 17 2005 3430.60  
## 18 2006 3527.48  
## 19 2007 3637.89  
## 20 2008 3655.00

An even-numbered moving average is unbalanced, and for our purposes, the unbalancing will be in factor of more recent observations. For example, to calcuate a 4-MA, the equation is as follows:

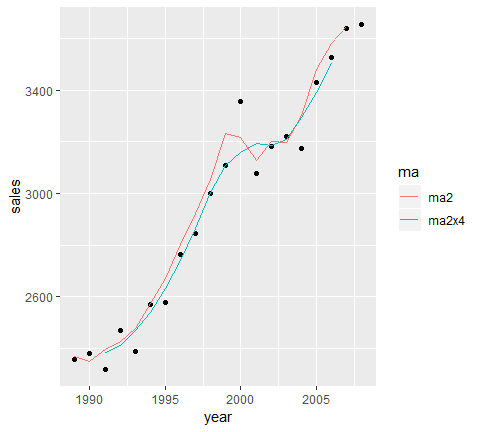
To make the moving averages symmetric (and therefore more accurate), we then take a 2-M of the 4-MA to create **2 x 4-MA**. For the 2-MA step, we average the current and previous moving averages, thus resulting in an overall estimate of:

This two-step process can be performed easily with the ma function by setting order = 4 and center = TRUE.

elecsales.df %>%  
 mutate(ma4 = ma(sales, order = 4, centre = TRUE)) %>%  
 head()  
## year sales ma4  
## 1 1989 2354.34 NA  
## 2 1990 2379.71 NA  
## 3 1991 2318.52 2384.359  
## 4 1992 2468.99 2412.047  
## 5 1993 2386.09 2467.918  
## 6 1994 2569.47 2536.784  
## year sales ma4  
## 1 1989 2354.34 NA  
## 2 1990 2379.71 NA  
## 3 1991 2318.52 2384.359  
## 4 1992 2468.99 2412.047  
## 5 1993 2386.09 2467.918  
## 6 1994 2569.47 2536.784

To compuare this moving average to a regular moving average we can plot the two outputs:

# Compute 2 and 2\*4 moving averages  
  
elecsales.df %>%   
 mutate(ma2 = rollmean(sales, k=2, fill = NA),  
 ma2x4 = ma(sales, order = 4, centre = TRUE)) %>%   
 gather(ma, value, ma2:ma2x4) %>%   
 ggplot(aes(x = year))+  
 geom\_point(aes(y = sales))+  
 geom\_line(aes(y = value, color = ma))



This 2 x 4-MA process produces the best fit yet. It massages out some of the noise while maintaining the overall trend of the data. Other combinations of moving averages are possible, such as 3 x 3-MA. To maintain symmetry, if your first moving average is an even number of points, the follow-up MA should also contain an even number. Likewise, if your first MA uses an odd number of points, the follow-up should use an odd number of points. Just keep in mind that moving averages of moving averages will lose information as you do not retain as many data points.

#### Using fpp2 package

If your data is already in time series data object, then you can apply the ma function directly to that object with order = 4 and centre = TRUE. For example, the built-in elecsales data set is a time series object:

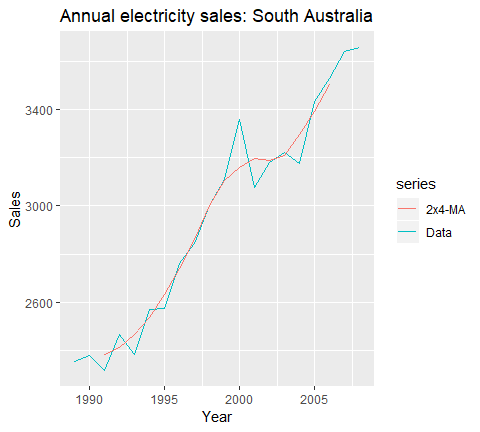
class(elecsales)  
## [1] "ts"

Wecan compute the ma2x4 moving average directly:

ma(elecsales, order = 4, centre = TRUE)  
## Time Series:  
## Start = 1989   
## End = 2008   
## Frequency = 1   
## [1] NA NA 2384.359 2412.047 2467.918 2536.784 2630.801  
## [8] 2742.006 2862.457 3003.352 3106.600 3157.988 3194.662 3186.188  
## [15] 3207.888 3295.610 3391.006 3502.892 NA NA

And we can use autoplot to plot the 2x4 moving average against the raw data:

autoplot(elecsales, series = "Data")+  
 autolayer(ma(elecsales, order = 4, centre = TRUE), series = "2x4-MA")+  
 labs(x = "Year", y = "Sales")+  
 ggtitle("Annual electricity sales: South Australia")



### Weighted moving averages {##TS\_WMA}

A moving average of a movingaverage can be thought of as a symmetric MA that has different weights on each nearby observation. For example, the 2x4-MA discussed above is equivalent to a weighted 5-MA with weights given by . In general, a weighted m-MA can be written as:

where k=(m-1)/2 and the weights are given by . It is important that the weights all sum to one and that they are symmetric so that . This simple m-MA is a special case where all the weights are equal to . This simple m-MA is a special case where all the weights are equal to A major advantage of weighted moving averages is that they yield a smoother estimate of the trend-cycle. Instead of observations entering and leaving the calculation at full weight, their weights are slowly increased and then slowly decreased resulting in a smoother curve. Some specific sets of weights are widely used such as the following:

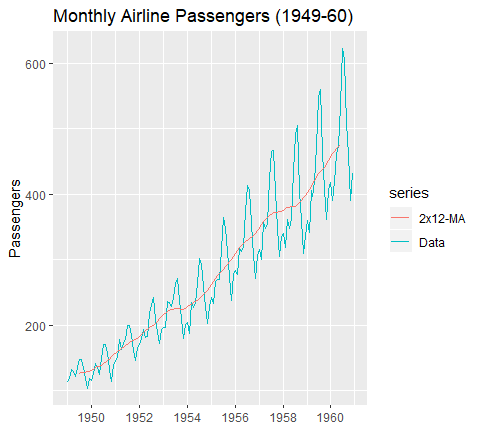
For example, the AirPassengers data contains an entry for every month in a 12 year span, so a time period would consist of 12 time units. A 2 x 12-MA set-up is the preferred method for such data. The observation itself, as well as the 5 observations immediately before and after it, receives weight , while the data point for that month last year and that month the following year both receive weight .

We can produce this weighted moving average using the ma function as we did in the last section.

ma(AirPassengers, order = 12, centre = TRUE)  
## Jan Feb Mar Apr May Jun Jul  
## 1949 NA NA NA NA NA NA 126.7917  
## 1950 131.2500 133.0833 134.9167 136.4167 137.4167 138.7500 140.9167  
## 1951 157.1250 159.5417 161.8333 164.1250 166.6667 169.0833 171.2500  
## 1952 183.1250 186.2083 189.0417 191.2917 193.5833 195.8333 198.0417  
## 1953 215.8333 218.5000 220.9167 222.9167 224.0833 224.7083 225.3333  
## 1954 228.0000 230.4583 232.2500 233.9167 235.6250 237.7500 240.5000  
## 1955 261.8333 266.6667 271.1250 275.2083 278.5000 281.9583 285.7500  
## 1956 309.9583 314.4167 318.6250 321.7500 324.5000 327.0833 329.5417  
## 1957 348.2500 353.0000 357.6250 361.3750 364.5000 367.1667 369.4583  
## 1958 375.2500 377.9167 379.5000 380.0000 380.7083 380.9583 381.8333  
## 1959 402.5417 407.1667 411.8750 416.3333 420.5000 425.5000 430.7083  
## 1960 456.3333 461.3750 465.2083 469.3333 472.7500 475.0417 NA  
## Aug Sep Oct Nov Dec  
## 1949 127.2500 127.9583 128.5833 129.0000 129.7500  
## 1950 143.1667 145.7083 148.4167 151.5417 154.7083  
## 1951 173.5833 175.4583 176.8333 178.0417 180.1667  
## 1952 199.7500 202.2083 206.2500 210.4167 213.3750  
## 1953 225.3333 224.9583 224.5833 224.4583 225.5417  
## 1954 243.9583 247.1667 250.2500 253.5000 257.1250  
## 1955 289.3333 293.2500 297.1667 301.0000 305.4583  
## 1956 331.8333 334.4583 337.5417 340.5417 344.0833  
## 1957 371.2083 372.1667 372.4167 372.7500 373.6250  
## 1958 383.6667 386.5000 390.3333 394.7083 398.6250  
## 1959 435.1250 437.7083 440.9583 445.8333 450.6250  
## 1960 NA NA NA NA NA

And to compare this moving average to the actual time series:

autoplot(AirPassengers, series = "Data") +   
 autolayer(ma(AirPassengers, order = 12, centre = T), series = "2x12-MA") +  
 ggtitle("Monthly Airline Passengers (1949-60)") +  
 labs(x = NULL, y = "Passengers")



You can see we’ve smoothed out the seasonality but have captured the overall trend.

Exercises Using the economics data set provided by the ggplot2 package:

1. Compute and plot the 1, 3, and 5 year moving average for the personal consumption expenditures.
2. Compute the mean square error of these moving averages.
3. Forecast the personal consumption expenditure for 2015-05-01 using a 1, 3, and 5 year trailing moving average.
4. Compute and plot a 2x12 weighted smoothing average.

## Exponential smoothing

Expontential smoothing another smoothing method and has been around since the 1950s. Where niave forecasting places 100% weight on the most recent observation and moving averages place equal weight on k values, exponential smoothing allows for weighted averages where greater weight can be placed on recent observations and lesser weight on older observations. Exponential smoothing methods are intuitive, computationally efficient, and generally applicable to a wide range of time series. Consequently, exponentially smoothing is a great forecasting tool to have and this tutorial will walk you through the basics.

### tl;dr

1. [Replication requirements](#ES_RR): What you’ll need to reproduce the analysis in this tutorial.
2. [Simple Exponential Smoothing](#ES_Simple): Technique for data with no trend or seasonality.
3. [Holt’s Method](#ES_Holt): Technique for data with trend but no seasonality.
4. [Holt-Winters Seasonal Method](#ES_Holt-Winters): Technique for data with trend and seasonality.
5. [Damped Trend Methods](#ES_Dampled_Trend): Technique for trends that are believed to become more conservative or “flat-line” over time.
6. Exercises: Practice what you’ve learned

### Replication requirements

This tutorial primarily uses the fpp2 package. fpp2 will automatically load the forecast package (among others), which provides many of the key forecasting functions used throughout.

library(tidyverse)  
library(fpp2)

Furthermore, we’ll use a couple data sets to illustrate. The goog and qcement data are provided by the fpp2 package. Let’s go ahead and set up training and validation sets:

# create training and validation of the Google stock data  
goog.train <- window(goog, end = 900)  
goog.test <- window(goog, start = 901)  
  
# create training and validation of the AirPassengers data  
qcement.train <- window(qcement, end = c(2012, 4))  
qcement.test <- window(qcement, start = c(2013, 1))

### Simple exponential smoothing

The simplest of the exponentially smoothing methods is called “simple exponential smooothing” (SES). The key point to remmember is that SES is suitable for data with no trend or seasonal pattern. This section will illustrate why.

For exponential smoothing, we weight the recent observation more heavily than older observations. The weight of each observation is determined through the use of *smoothing parameter*, which we will denote . For a data set with observations, we calculate our predicted value , which will be based on through as follows.

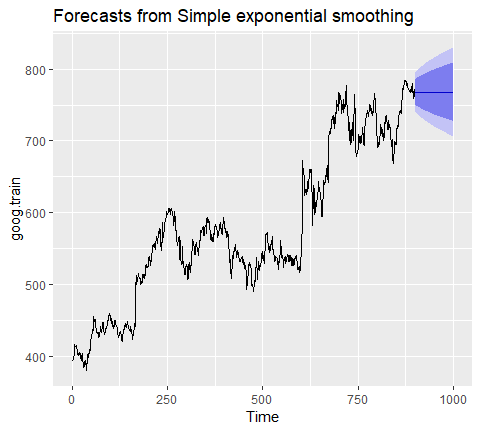
where . It is also common to come to use the *component form* of this model, which uses the following set of equations.

$$
\hat{y}\_{t+1} = l\_t \\
l\_t = \alpha y\_t + (1-\alpha)l\_{t-1}
$$

In both equations, we can see that the most weights is placed on the most recent observation. In practice, equals to 0.1-0.2 tends to perform quite well but we will demonstrate shortly how to tune this parameter. When is close to 1 we consider this fast learning because the algorithm gives more historical data more weight. When is closer to 1, we consider the fast learning because the algorithms give more weight to the most recent observation; therefore, recent changes in the data will have a bigger impact on forecasted values. The following table illustrates how weighting changes based on the parameters.

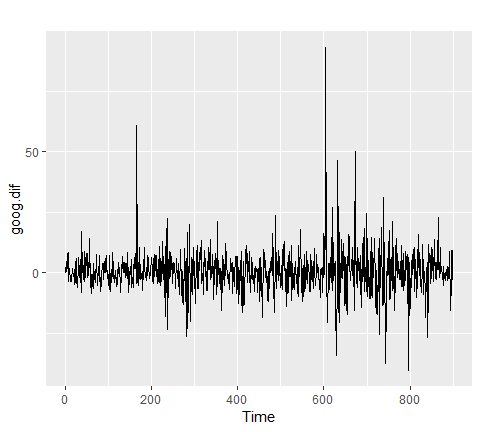
Let’s go ahead and apply SES to the Google data using the ses function. We manually set the for our initial model and forecast foward 100 steps with . We see that our forecast proejcts a flatlined estimate into the future, which does not capture the positive trend in the data. This is why SES should not be used on data with a trend or seasonal component.

ses.goog <- ses(goog.train, alpha=.2, h=100)  
autoplot(ses.goog)



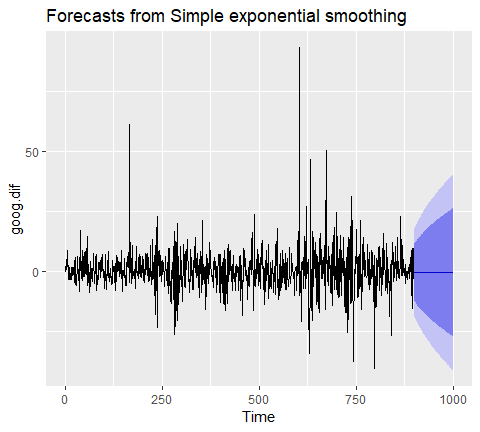
One approach to correct for this is to difference our data to remove the trend. Now, goog.dif represents the change in stock price from the previous day.

goog.dif <- diff(goog.train)  
autoplot(goog.dif)



Once we have differenced we have effectively removed the trend from our data and can reapply the SES model.

ses.goog.dif <- ses(goog.dif, alpha = .2, h = 100)  
autoplot(ses.goog.dif)

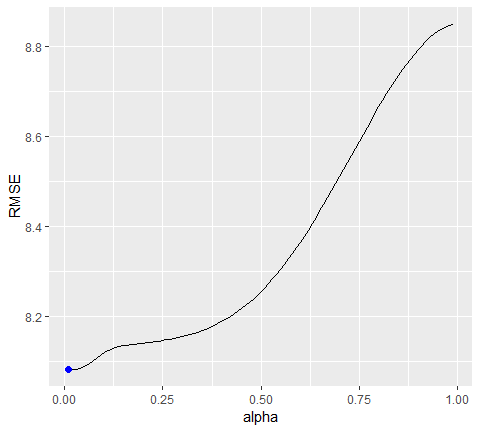


To understand how well the model predicts we can compare our forecasts to our validation data set. But first we need to create a differenced validation set since our training data was built on differenced data. We see that performance measures are smaller on the test set than the training so we are not overfitting our model.

goog.dif.test <- diff(goog.test)  
  
# accuracy calculation  
forecast::accuracy(ses.goog.dif, goog.dif.test) %>%   
 broom::tidy()  
## # A tibble: 2 x 9  
## .rownames ME RMSE MAE MPE MAPE MASE ACF1 Theil.s.U  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 Training set -0.0137 9.32 6.40 100.0 254. 0.757 -0.0544 NA   
## 2 Test set 0.972 8.14 6.12 110. 178. 0.724 0.123 0.990

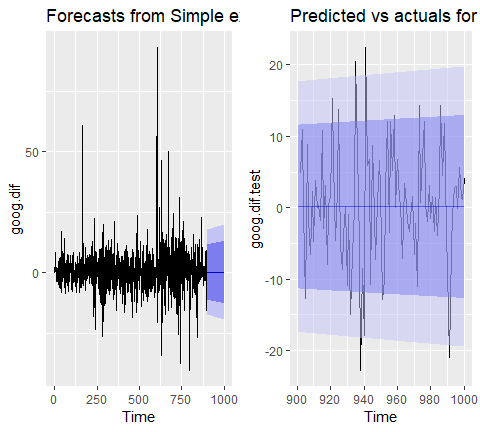
In our model, we used the standard ; however, we can tune our alpha parameter to identify the value that reduces our forecasting error. Here, we loop through alpha values from 0.01-0.99 and identify the level that minimizes our test RMSE. Turns our that minimize our prediction error.

# identify optimal alpha parameter  
alpha <- seq(.01, .99, by=.01)  
RMSE <- NA  
  
for (i in seq\_along(alpha)){  
 fit <- ses(goog.dif, alpha = alpha[i], h=100)  
 RMSE[i] <- accuracy(fit, goog.dif.test)[2,2]  
}  
  
# convert to a data frame and idenitify min alpha value  
alpha.fit <- data\_frame(alpha, RMSE)  
alpha.min <- filter(alpha.fit, RMSE == min(RMSE))  
  
# plot RMSE vs. alpha  
ggplot(alpha.fit, aes(alpha, RMSE)) +  
 geom\_line() +  
 geom\_point(data = alpha.min, aes(alpha, RMSE), size = 2, color = "blue")



Now we can re-fit our SES with . Our performance metrics are not significantly different from our model, where ; however, you will notice that the predicted confidence intervals are narrower (left chart). And when we zoom into the predicted versus actuals (right chart) you see that for most observations, our predicted confidence intervals did well.

# refit model with alpha = .05  
ses.goog.opt <- ses(goog.dif,   
 alpha = .05,  
 h = 100)  
  
# performance eval  
accuracy(ses.goog.opt,  
 goog.dif.test)  
## ME RMSE MAE MPE MAPE MASE  
## Training set -0.01188991 8.939340 6.030873 109.97354 155.7700 0.7136602  
## Test set 0.30483955 8.088941 6.028383 97.77722 112.2178 0.7133655  
## ACF1 Theil's U  
## Training set 0.01387261 NA  
## Test set 0.12278141 0.9998811  
  
# plotting results  
p1 <- autoplot(ses.goog.opt)+  
 theme(legend.position = "bottom")  
p2 <- autoplot(goog.dif.test)+  
 autolayer(ses.goog.opt, alpha=.5)+  
 ggtitle("Predicted vs actuals for the test data set")  
  
gridExtra::grid.arrange(p1, p2, nrow=1)



### Holt’s Method

As mentionedd and observed in the previous section, SES does not perform well with data that has a long-term trend. In the last section, we will illustrate how you can remove the rend with differencing and then perform SES. An alternative method to apply exponential smoothing while capturing trend in the data is to use *Holt’s Method*.

Holt’s Method makes predictions for data with a trend using two smoothing parameters, and , which correspond to the level and trend components, respectively. For Holt’s method, the prediction will be a line of some non-zero slope that extends from the time step after the last collected data point onwards.

The methodology for predictions using data with a trend (Holt’s Method) uses the following equation with observations. The k-step-ahead forecast is given by combining the level estimate at time t() and the trend estimate (which in this example is assumed additive) at time t ().

The level () and trend () are updated through a pair of updating equations, which is where you see the presence of the two smoothing parameters:

$$
L\_t = \alpha y\_t + \alpha(1-\alpha)(L\_t + T\_{t-1}) \\
T\_t = \beta(L\_t - L\_{t-1}) + (1-\beta)T\_{t-1}
$$

In these equations, the first means that the level at time t is a weighted average of the actual value at time t and the level in the previous period, adjusted for trend. The second equation means that the trend at time t is a weighted average of the trend in the previous period and the more recent information on the change in the level. Similar to SES, and are constrained to 0-1 with higher values giving faster learning and lower values providing slower learning.

To capture a **multiplicative** (exponential) trend we make a minor adjustment in the above equations.

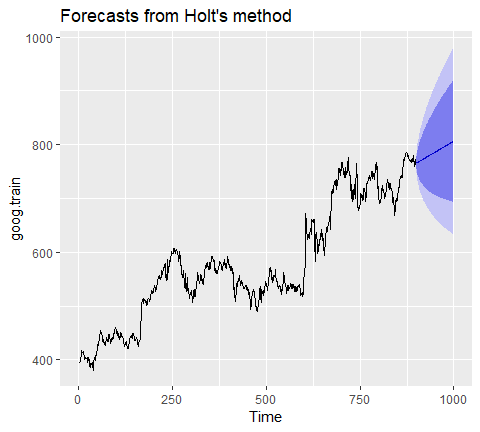
$$
\hat{y}\_{T+1} = L\_t \* kT\_k \\
L\_t = \alpha y\_t + \alpha(1-\alpha)(L\_t \* T\_{t-1}) \\
T\_t = \beta(L\_t / L\_{t-1}) + (1-\beta)T\_{t-1}
$$

Holt’s method also has the alternative Component Form operations. In this case these represent the additive trend component form:

$$
\hat{y}\_{t+h} = l\_t \* hb\_t \\
l\_t = \alpha y\_t + \alpha(1-\alpha)(l\_t + b\_{t-1}) =
\beta (l\_t - l\_{t-1}) + (1-\beta)b\_{t-1}
$$

If we go back to our Google stock data, we can apply Holt’s method in the following manner. Her, we will not manually set the and for our initial model and forecast foward 100 step with h=100. We see that our forecast now does a better job capturing the positive trend in the data.

holt.goog <- holt(goog.train, h = 100)  
autoplot(holt.goog)



Within holt you cna manually set the and parameters; however, if you leave those parameters at NULL, the holt function will actually identify the optimal model parameters. It does by ,minimizing AIC and BIC values. We can see the model selected by holt. In this case, meaning fast learning in the daty-today movements and which means slow learning for the trend.

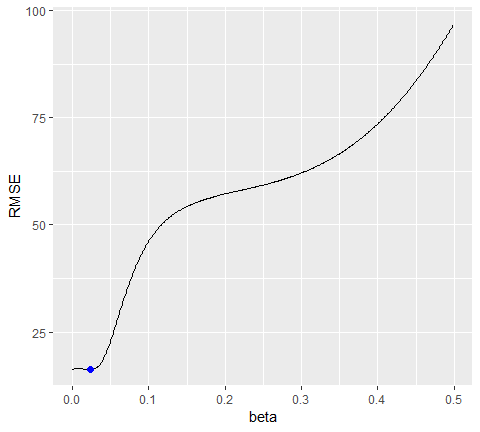
holt.goog$model  
## Holt's method   
##   
## Call:  
## holt(y = goog.train, h = 100)   
##   
## Smoothing parameters:  
## alpha = 0.9999   
## beta = 1e-04   
##   
## Initial states:  
## l = 401.1276   
## b = 0.4091   
##   
## sigma: 8.8149  
##   
## AIC AICc BIC   
## 10045.74 10045.81 10069.75

Let’s check the predictive accuracy of our model. According to our MAPE we have about a 2% error rate.

accuracy(holt.goog, goog.test)  
## ME RMSE MAE MPE MAPE  
## Training set -0.003332796 8.795267 5.821057 -0.01211821 1.000720  
## Test set 0.545744415 16.328680 12.876836 0.03013427 1.646261  
## MASE ACF1 Theil's U  
## Training set 1.002452 0.03100836 NA  
## Test set 2.217538 0.87733298 2.024518

Similr to SES, we can tune the parameter to see if we can improve our predictive accuracy. The holt function identified an optimal ; however, this optimal value is based on minimizing errors on the training set, not minimizing predicton errors on the test set. Let’s assess a tradespace of values and see if we gain some predictive accuracy. Here, we loop through a series of values starting at 0.0001 all the way up to 0.5. We see that there is a dip in our RMSE at 0.0601.

# identify optimal alpha parameter  
beta <- seq(.0001, .5, by=0.001)  
RMSE <- NA  
  
for (i in seq\_along(beta)){  
 fit <- holt(goog.train, beta = beta[i], h=100)  
 RMSE[i] <- accuracy(fit, goog.test)[2,2]  
}  
  
# convert to a data frame and identify min alpha value  
beta.fit <- data\_frame(beta, RMSE)  
beta.min <- filter(beta.fit, RMSE == min(RMSE))  
  
# plot RMSE vs alpha  
ggplot(beta.fit, aes(beta, RMSE))+  
 geom\_line()+  
 geom\_point(data=beta.min, aes(beta,RMSE), size=2, color="blue")



Now let’s refit our model with this optimal value and compare our predictive accuracy to our original model. We see that our new model reduces our error rate (MAPE) down to 1.78%.

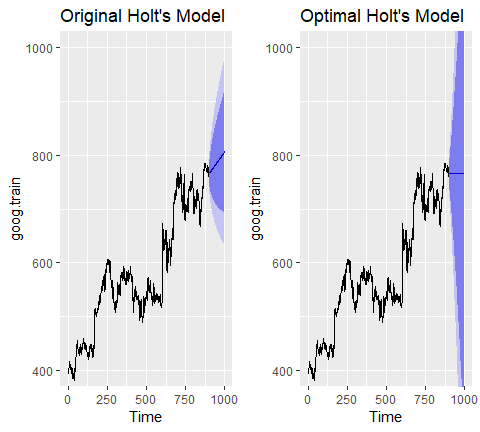
# new model with optimal beta  
holt.goog.opt <- holt(goog.train,   
 h = 100,  
 beta = 0.0601)  
  
# accuracy of first model  
accuracy(holt.goog, goog.test)  
## ME RMSE MAE MPE MAPE  
## Training set -0.003332796 8.795267 5.821057 -0.01211821 1.000720  
## Test set 0.545744415 16.328680 12.876836 0.03013427 1.646261  
## MASE ACF1 Theil's U  
## Training set 1.002452 0.03100836 NA  
## Test set 2.217538 0.87733298 2.024518  
## ME RMSE MAE MPE MAPE  
## Training set -0.003922069 8.938319 5.974921 -0.01188414 1.004290  
## Test set -8.945342029 21.099897 16.268344 -1.15371296 2.039945  
## MASE ACF1 Theil's U  
## Training set 0.9997514 0.03746482 NA  
## Test set 2.7220944 0.89540070 2.481272  
  
# accuracy of new optimal model  
accuracy(holt.goog.opt, goog.test)  
## ME RMSE MAE MPE MAPE  
## Training set -0.01493114 8.960214 6.058869 -0.005524151 1.039572  
## Test set 21.41138275 28.549029 23.841097 2.665066997 2.988712  
## MASE ACF1 Theil's U  
## Training set 1.043406 0.009696325 NA  
## Test set 4.105709 0.895371665 3.435763  
## ME RMSE MAE MPE MAPE MASE  
## Training set -0.01098347 9.109332 6.218278 -0.00502392 1.043517 1.040471  
## Test set -0.33180592 18.536622 14.287508 -0.09452438 1.780052 2.390652  
## ACF1 Theil's U  
## Training set 0.01293146 NA  
## Test set 0.88797970 2.156113

If we plot our original versus more recent optimal model we will notice a couple things.

* First, our predicted values for the optimal model are more conservative; in other words, they are assuming a more gradual slope.
* Second, the confidence intervals are much more extreme. So although our predictions were more accuracte, our uncertainty increases.

The reasons for this is that by increasing our value we are assuming faster learning from more recent observations. And since there some quite a bit of turbulence in the recent time period, this is causing greater variance to be incorporated into our prediction intervals. This requires a more indepth discussion than this tutorial will go into, but the important thing to keep in mind is that although we increase our prediction accuracy with parameter tuning, there are additional side effects that can occur, which may be harder to explain to decision-makers.

p1 <- autoplot(holt.goog)+  
 ggtitle("Original Holt's Model")+  
 coord\_cartesian(ylim = c(400, 1000))  
  
p2 <- autoplot(holt.goog.opt)+  
 ggtitle("Optimal Holt's Model")+  
 coord\_cartesian(ylim = c(400, 1000))  
  
gridExtra::grid.arrange(p1, p2, nrow=1)

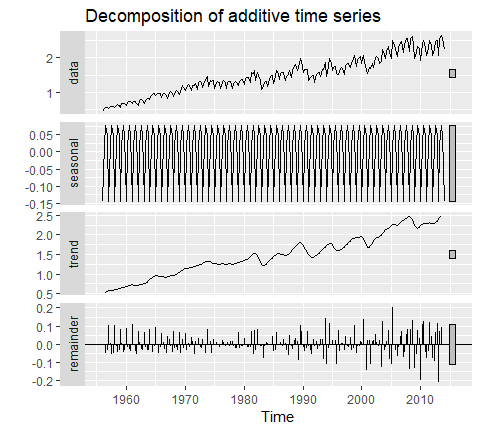


### Holt-Winters Seasonal Method

To make predictions using data with a trend and seasonality, we turn to the Holt Winters Seasonal Method. This method can be implemented with an “Additive” structure or “Multiplicative” structure, where the choice of method depends on the data set. The Additive model is best used when the seasonal trend is of the same magnitude throughout the data set, while the multiplicative model is preferred when the magnitude of seasonality changes as time increase.

Since the Google data does not have seasonality, we’ll use the qcement data that we set up in the Replication section to demonstrate. This data has seasonality and trend; however, it is unclear if seasonality is additive or multiplicative. We’ll use the Holt-Winters method to identify the best fit model.

autoplot(decompose(qcement))



#### Additive

For the additive model, the regular equation form is:

The level (), trend () and season () are updated through a pair of updating equations, which is where you see the presence of the three smoothing parameters:

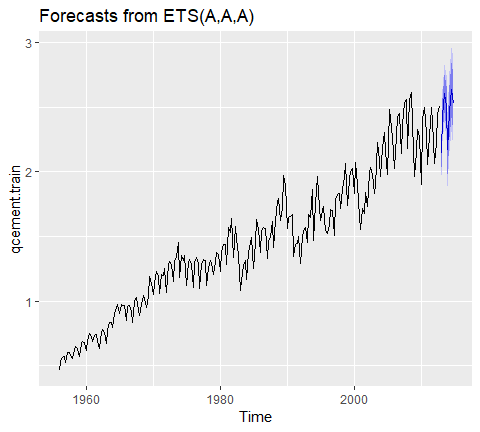
$$
L\_t = \alpha(y\_t - S\_{t-m} + (1-\alpha)(L\_{t-1}+T\_{t-1})) \\
T\_t = \beta(L\_t - L\_{t-1}) + (1-\beta)T\_{t-1} \\
S\_t = \gamma(y\_t - L\_t) + (1-\gamma)S\_{t-m}
\\
$$

where are the three smoothing parameters to deal with the level pattern, the trend, and the seasonality, respectively. Similar to SES and Holt’s method, all three parameters are constrained to 0-1. The component equations are as follows.

$$
\hat{y}\_{t+1} = l\_t + ht\_k + s\_{t+k-m} \\
l\_t = \alpha(y\_t - s\_{t-m} + (1-\alpha)(l\_{t-1}+b\_{t-1})) \\
b\_t = \beta(l\_t - l\_{t-1}) + (1-\beta)b\_{t-1} \\
s\_t = \gamma(y\_t - l\_t) + (1-\gamma)s\_{t-m}
\\
$$

To apply the Holt-Winters method we’ll introduce a new function, ets which stands for error, trend, and seasonality. The important thing to understand about the ets model is how to select the model = parameter. In total you have 36 model options to choose from. The parameter settings in the below code (model = "AAA") stands for a model with additive error, additive trend, and additve seasonality.

qcement.hw <- ets(qcement.train, model="AAA")  
autoplot(forecast(qcement.hw))



So when specificying the model type you always specificy the error, trend, then seasonality (hence “ets”). The options you can specify for each component is as follows:

* error: additive (“A”), multiplicative (“M”), unknown (“Z”)
* trend: none (“N”), additive (“A”), multiplicative (“M”), unknown (“Z”)
* seasonality: none (“N”), additive (“A”), multiplicative (“M”), unknown (“Z”)

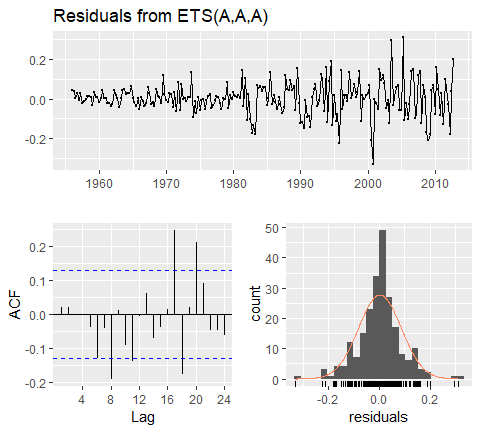
Consequently, if you wanted to apply a Holt’s model where the error and trend were additive and no seasonality exists you would select model = "AAN". If you want to apply a Holt-Winters model where there is additive error, an exponential (multiplicative) trend, and additive seasonality you would select model = “AMA”. If you are uncertain of the type of component then you use “Z”. So if you were uncertain of the components or if you want the model to select the best option, you could use model = “ZZZ” and the “optimal” model will be selected.

If we assess our additive model we can see that , and .

summary(qcement.hw)  
## ETS(A,A,A)   
##   
## Call:  
## ets(y = qcement.train, model = "AAA")   
##   
## Smoothing parameters:  
## alpha = 0.6418   
## beta = 1e-04   
## gamma = 0.1988   
##   
## Initial states:  
## l = 0.4511   
## b = 0.0075   
## s = 0.0049 0.0307 9e-04 -0.0365  
##   
## sigma: 0.0854  
##   
## AIC AICc BIC   
## 126.0419 126.8676 156.9060   
##   
## Training set error measures:  
## ME RMSE MAE MPE MAPE  
## Training set 0.001463693 0.08393279 0.0597683 -0.003454533 3.922727  
## MASE ACF1  
## Training set 0.5912949 0.02150539

If we check our residuals, we see that residuals grow larger over time. This may suggest that a multiplicative error rate may be more appropriate.

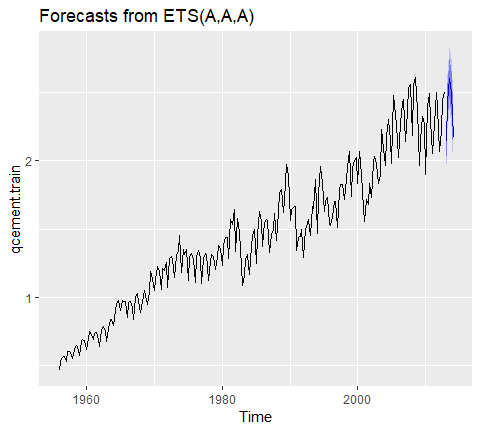
checkresiduals(qcement.hw)



##   
## Ljung-Box test  
##   
## data: Residuals from ETS(A,A,A)  
## Q\* = 20.288, df = 3, p-value = 0.0001479  
##   
## Model df: 8. Total lags used: 11

If we check the predictive accuracy we see that our prediction accuracy is about 2.9% (according to the MAPE).

#forecast the next 5 quarter  
qcement.f1 <- forecast(qcement.hw, h = 5)  
autoplot(qcement.f1)



# check accuracy  
accuracy(qcement.f1, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set 0.001463693 0.08393279 0.05976830 -0.003454533 3.922727  
## Test set 0.031362775 0.07144211 0.06791904 1.115342984 2.899446  
## MASE ACF1 Theil's U  
## Training set 0.5912949 0.02150539 NA  
## Test set 0.6719311 -0.31290496 0.2112428

#### Multiplicative

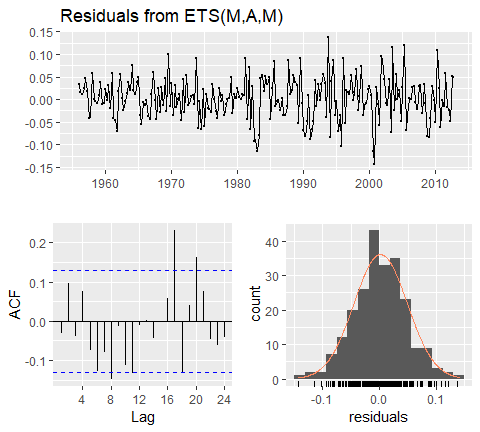
As previously stated, we may have multiplicative features for our Holt-Winters method. If we have multiplicative seasonality, then our equation from changes to:

The level (), trend () and season () are updated through a pair of updating equations, which is where you see the presence of the three smoothing parameters:

$$
L\_t = \alpha y\_t / S\_{t-m} + (1-\alpha)(L\_{t-1}+T\_{t-1})) \\
T\_t = \beta(L\_t - L\_{t-1}) + (1-\beta)T\_{t-1} \\
S\_t = \gamma(y\_t / L\_t) + (1-\gamma)S\_{t-m}
\\
$$

If we apply a multiplicative seasonality model then our model parameter becomes model = "MAM" (here, we are actually applying a multiplicative error and seasonality model). We see that are residuals illustrate less change in magnitude over time. We still have an issue with autocorrelation with errors but we’ll address that in later tutorials.

qcement.hw2 <- ets(qcement.train, model = "MAM")  
checkresiduals(qcement.hw2)



##   
## Ljung-Box test  
##   
## data: Residuals from ETS(M,A,M)  
## Q\* = 23.433, df = 3, p-value = 3.281e-05  
##   
## Model df: 8. Total lags used: 11

To compare the predictive accuracy of our models let’s compare four different models. We see that the first model (additive error, trend and seasonality) results in the lowest RMSE and MAPE on our test data set.

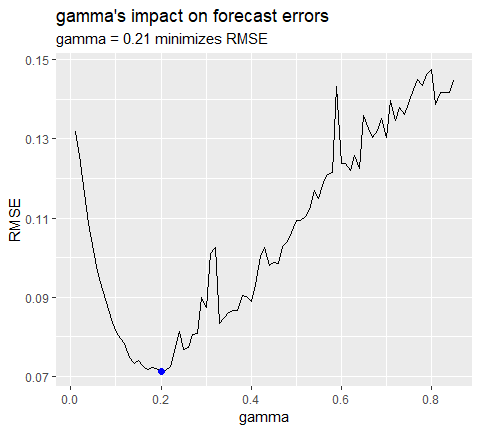
# additive error, trend and seasonality  
qcement.hw1 <- ets(qcement.train, model = "AAA")  
qcement.f1 <- forecast(qcement.hw1, h = 5)  
accuracy(qcement.f1, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set 0.001463693 0.08393279 0.05976830 -0.003454533 3.922727  
## Test set 0.031362775 0.07144211 0.06791904 1.115342984 2.899446  
## MASE ACF1 Theil's U  
## Training set 0.5912949 0.02150539 NA  
## Test set 0.6719311 -0.31290496 0.2112428  
  
# multiplicative error, additive trend and seasonality  
qcement.hw2 <- ets(qcement.train, model = "MAA")  
qcement.f2 <- forecast(qcement.hw2, h=5)  
accuracy(qcement.f2, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set -0.002700842 0.08387446 0.05912640 -0.3689856 3.811958  
## Test set 0.018046198 0.06438073 0.05763575 0.5614035 2.493544  
## MASE ACF1 Theil's U  
## Training set 0.5849446 0.001490172 NA  
## Test set 0.5701974 -0.344335948 0.1722734  
  
# additive error and trend and multiplicative seasonality  
qcement.hw3 <- ets(qcement.train, model = "AAM", restrict = FALSE)  
qcement.f3 <- forecast(qcement.hw3, h = 5)  
accuracy(qcement.f3, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set 0.007542179 0.07818508 0.05640602 0.3985457 3.693964  
## Test set 0.046793468 0.09456029 0.09241462 1.6559245 3.882433  
## MASE ACF1 Theil's U  
## Training set 0.5580315 -0.01243122 NA  
## Test set 0.9142688 -0.05070111 0.2737314  
## ME RMSE MAE MPE MAPE MASE  
## Training set 0.003403387 0.07762833 0.0558983 0.1487162 3.693408 0.5530085  
## Test set 0.027648352 0.08382395 0.0777908 0.8568631 3.304030 0.7695936  
## ACF1 Theil's U  
## Training set -0.02009013 NA  
## Test set -0.06590551 0.2161475  
  
# multiplicative error, additive trend, and multiplicative seasonality  
qcement.hw4 <- ets(qcement.train, model = "MAM")  
qcement.f4 <- forecast(qcement.hw4, h = 5)  
accuracy(qcement.f4, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set 0.0009645886 0.07807481 0.05595930 -0.02540554 3.657772  
## Test set 0.0164979314 0.08297241 0.07162921 0.37381187 3.064455  
## MASE ACF1 Theil's U  
## Training set 0.5536120 -0.03038457 NA  
## Test set 0.7086363 -0.06185361 0.1930288

If we were to compare this to an unspecified model where we let ets select the optimal model, we see that ets selects a model specification of multiplicative error, additive trend, and multiplicative seasonality (“MAM”). This is equivalent to our fourth model above. This model is assumed “optimal” because it minimizes RMSE, AIC, and BIC on the training data set, but does not necessarily minimize prediction errors on the test set.

qcement.hw5 <- ets(qcement.train, model = "ZZZ")  
summary(qcement.hw5)  
## ETS(M,A,M)   
##   
## Call:  
## ets(y = qcement.train, model = "ZZZ")   
##   
## Smoothing parameters:  
## alpha = 0.7664   
## beta = 0.0131   
## gamma = 1e-04   
##   
## Initial states:  
## l = 0.49   
## b = 0.0064   
## s = 1.0298 1.0488 1.0151 0.9063  
##   
## sigma: 0.0477  
##   
## AIC AICc BIC   
## 0.3161397 1.1418277 31.1802503   
##   
## Training set error measures:  
## ME RMSE MAE MPE MAPE  
## Training set 0.0009645886 0.07807481 0.0559593 -0.02540554 3.657772  
## MASE ACF1  
## Training set 0.553612 -0.03038457

As we did in the SES and Holt’s method section, we can optimize the parameter in our Holt-Winters model. Here, we use the additive error, trend and seasonality model that minimized our prediction errors above and identify the parameter that minimizes forecast errors. In this case we see that minimizes the error rate.

gamma <- seq(0.01, 0.85, 0.01)  
RMSE <- NA  
  
for (i in seq\_along(gamma)){  
 hw.expo <- ets(qcement.train, "AAA", gamma = gamma[i])  
 future <- forecast(hw.expo, h = 5)  
 RMSE[i] = accuracy(future, qcement.test)[2,2]  
}  
  
error <- data\_frame(gamma, RMSE)  
minimum <- filter(error, RMSE == min(RMSE))  
ggplot(error, aes(gamma, RMSE)) +  
 geom\_line() +  
 geom\_point(data = minimum, color = "blue", size = 2) +  
 ggtitle("gamma's impact on forecast errors",  
 subtitle = "gamma = 0.21 minimizes RMSE")



If we update our model with this “optimal” parameter we see that we bring our forecasting error rate down from 2.88% to 2.76%. This is a small improvement, but often small improvements can have large business implications.

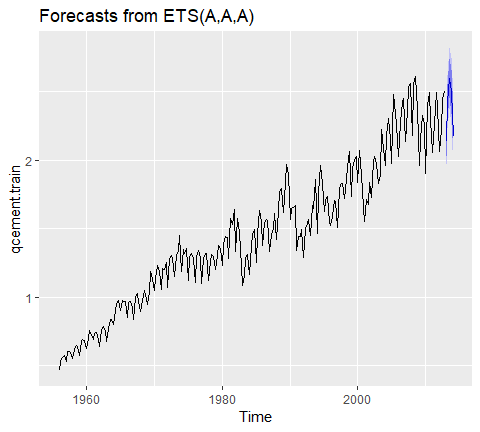
# previous model with additive error, trend and seasonality  
accuracy(qcement.f1, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set 0.001463693 0.08393279 0.05976830 -0.003454533 3.922727  
## Test set 0.031362775 0.07144211 0.06791904 1.115342984 2.899446  
## MASE ACF1 Theil's U  
## Training set 0.5912949 0.02150539 NA  
## Test set 0.6719311 -0.31290496 0.2112428  
## ME RMSE MAE MPE MAPE  
## Training set -0.0004297096 0.08375033 0.05912561 -0.1834617 3.830243  
## Test set 0.0307206913 0.07102359 0.06762179 1.0818090 2.882741  
## MASE ACF1 Theil's U  
## Training set 0.5849367 0.04576105 NA  
## Test set 0.6689904 -0.30067586 0.2084824  
  
# new model with optimal gamma parameter  
qcement.hw6 <- ets(qcement.train, model = "AAA", gamma = 0.21)  
qcement.f6 <- forecast(qcement.hw6, h = 5)  
accuracy(qcement.f6, qcement.test)  
## ME RMSE MAE MPE MAPE  
## Training set -0.001312025 0.08377557 0.05905971 -0.2684606 3.834134  
## Test set 0.033492771 0.07148708 0.06775269 1.2096488 2.881680  
## MASE ACF1 Theil's U  
## Training set 0.5842847 0.04832198 NA  
## Test set 0.6702854 -0.35877010 0.2202448  
## ME RMSE MAE MPE MAPE  
## Training set -0.001206215 0.08425184 0.05975069 -0.154495 3.974104  
## Test set 0.029358296 0.06911376 0.06466821 1.038366 2.761642  
## MASE ACF1 Theil's U  
## Training set 0.5911206 0.03804556 NA  
## Test set 0.6397703 -0.36727559 0.2081361

With this new optimal model we can get our predicted values:

qcement.f6  
## Point Forecast Lo 80 Hi 80 Lo 95 Hi 95  
## 2013 Q1 2.134650 2.025352 2.243947 1.967494 2.301806  
## 2013 Q2 2.427828 2.299602 2.556055 2.231723 2.623934  
## 2013 Q3 2.601989 2.457284 2.746694 2.380681 2.823296  
## 2013 Q4 2.505001 2.345506 2.664496 2.261075 2.748927  
## 2014 Q1 2.171068 1.987914 2.354223 1.890958 2.451179

and also visualize these predicted values:

autoplot(qcement.f6)



### Damping methods

One last item to discuss is the idea of “damping” your forecast. Damped forecasts use a damping coefficients denoted to more conservatively estimate the predicted trend.

Basically, if you belive that your additive or multiplicative trend is or will be slowing down ("flat-lining) in the near future then you are assuming it will dampen.

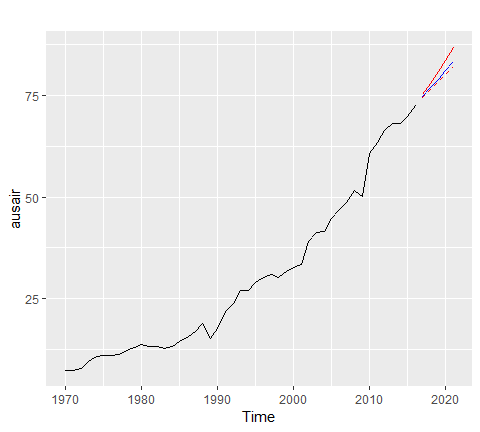
The equation form for an additive model with a damping coefficient is

$$
\hat{h}\_{t+h} = L\_t + ( \phi+\phi^2+...+\phi^h)\beta\_t \\
+\_t = \alpha y\_t + \alpha(1-\alpha)(L\_{t^1}+\phi\beta\_{t-1}), \\
\beta\_t = \beta(L\_t - L\_{t-1}) + (1-\beta)\phi \beta\_{t-1}
$$

where, . When the method is the same as Holt’s additive model. As gets closer to 0, the trend becomes more conservative and flat-lines to a constant in the nearer future. The end result of this method is flat that short-run forecasts are still trendeded while the long-run forecasts are constant.

To illustrate the effect of damped forecast we will use the fpp2::ausair data set. Here, we create several models (additive, additive+damped, multiplicative, multiplicative+damped). In the plot you can see that the dampled models (dashped lines) have more conservative trend lines and if we forecasted these far enough into the future we would see this trend flat-line.

# holt's linear (additive) model  
fit1 <- ets(ausair, model = "ZAN", alpha = 0.8, beta = 0.2)  
pred1 <- forecast(fit1, h = 5)  
  
# holt's linear (additive) model  
fit2 <- ets(ausair, model = "ZAN", damped = TRUE,   
 alpha = 0.8, beta = 0.2)  
pred2 <- forecast(fit2, h = 5)  
  
# holt's exponential (multiplicative) model  
fit3 <- ets(ausair, model = "ZMN" , alpha = 0.8, beta = 0.2)  
pred3 <- forecast(fit3, h = 5)  
  
# holt's exponential multiplicative model damped  
fit4 <- ets(ausair, model = "ZMN", damped = TRUE, alpha = 0.8,  
 beta = 0.2)  
pred4 <- forecast(fit4, h = 5)  
  
autoplot(ausair)+  
 autolayer(pred1$mean, color = "blue")+  
 autolayer(pred2$mean, color = "blue", linetype = "dashed")+  
 autolayer(pred3$mean, color = "red")+  
 autolayer(pred4$mean, color = "red", linetype = "dashed")



The above models were for illustrative purposes only. You would apply the same process as you saw in earlier sections to identify if a damped model predicts more accurately than a non-dampped model. You can even apply the approaches you saw earlier for tuning this parameter to identify the optimal coefficient.