

Time Series Analysis & Forecasting using Python and R

Time Series Analysis & Forecasting using Python and R

ISBN 978-1-716-45113-3

Published by Lulu, Inc.

Copyright© 2020, Jeffrey Strickland

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the author except for the use of brief quotations in a review.

Acknowledgements

I would like to thank Laurie, Mariah, and Evie, my three girls for their unconditional love and support. If there are people who inspire me to write, other than these, I would have to single Joe Kier (my best friend for over 30 years)... and Joe's wife, Debby, who helps me do jigsaw puzzles.

Table of Contents

Acknowledgements	vii
Table of Contents.....	ix
Preface	xvii
Additional Titles by the Author.....	xxii
Chapter 1 – Introduction	1
What is Time Series Data?	1
What Data Formats work Best?.....	1
What is Time Series Analysis?.....	2
What is Time Series Forecasting?	2
Where do we use Time Series Analysis?.....	3
What are the Goals of Time Series Analysis?.....	3
What Techniques are used in Time Series Analysis?	4
What are Smoothing Techniques?.....	4
How do we Load Time Series Data?.....	5
Got Milk (Data)?.....	5
Chapter Review.....	10
Review Exercises	11
Chapter 2 – Component of Time Series Data	13
Trend	15
Seasonality	15
Cycle.....	16
Irregular	17
Other Components	17
Got Milk (Components)?.....	18
Component Analysis Essentials.....	26

Time Series Lag	27
Autocorrelation.....	32
Lags (Lag Operator).....	35
Alternative Time Series Decomposition	36
Chapter Review.....	40
Review Exercises.....	42
Chapter 3 – Moving Averages for Time Series.....	43
Systematic Pattern and Random Noise	43
Two General Aspects of Time Series Patterns	43
Trend Analysis.....	44
Moving Averages	45
Moving Averages using movavg	46
Moving Averages using Filters	50
Moving Averages using fpp::ma	51
Moving averages of moving averages	54
Estimating the trend-cycle with seasonal data.....	56
Electrical equipment manufacturing	57
Weighted moving averages	58
COVID-19.....	59
The Johns Hopkins COVID data.....	59
Calculating rolling averages	60
Symmetry.....	62
CDHE COVID-19 DATA.....	62
Which mean should I use?.....	71
Calculating new cases in each State	71
Moving averages with geofacets	73
Column graph for new cases	73
Tidy dataset of new cases.....	74

Weighted Moving Averages.....	79
Got Milk (Moving Averages)?	80
Got Milk (Forecasting)?.....	81
Chapter review.....	89
Review Exercises	90
Chapter 4 – Exponential Smoothing for Times Series.....	93
Exponential Smoothing Methods	93
Analysis of Seasonality.....	93
Autocorrelation correlogram	94
Examining correlograms	96
Partial autocorrelations	99
Removing serial dependency	99
Decomposing Seasonal Data.....	100
Adjusting Seasonality.....	101
Simple or Single Exponential Smoothing	102
SES Example in R	104
Prediction Intervals.....	105
Double Exponential Smoothing	105
Holt’s linear trend method	107
Example in R: Air Passengers	108
Damped trend methods.....	109
Example In R: Air Passengers (continued)	110
Holt’s Exponential Smoothing.....	114
Triple Exponential Smoothing.....	115
Holt-Winters Exponential Smoothing.....	115
Additive Holt-Winters	116
Got a Job (HW Smoothing)?.....	118
Holt-Winters’ damped method.....	123

Got a Job (HW Dampening)?	123
Exponential Smoothing Taxonomy	126
Smoothing Oscillation.....	127
Plotting the Data.....	127
Seasonal Decomposition.....	128
Single Exponential Smoothing	131
Double Exponential Smoothing	133
Triple Exponential Smoothing.....	135
Accuracy Measurements	137
Chapter Review.....	137
Review Exercises.....	140
Chapter 5 – Exponential Smoothing and Moving Averages in Python	143
Loading and Formatting Non-Native Data.....	143
Where's the Beef (Let them Roll)	146
Working with dates and times with Pandas	148
Aggregation processing	148
Grouping by decade.....	149
User Defined Groups.....	149
Plotting the Data.....	150
Visualizing Time Series Trend	155
Where's the Beef (Exponential Smoothing)?	157
Data with Level and Trend	157
Cattle and Simple Exponential Smoothing	173
Data with Level and Trend Components	177
Chapter Review.....	179
Review Exercises.....	179
Chapter 6 – Stationarity and Differencing	181
Creating a time series in R	181

The Train-Test Split	182
Seasonal Decomposition.....	185
Stationarity and Differencing.....	188
Testing for Stationarity	190
Unit root tests.....	190
Dickey-Fuller Tests	190
Phillips-Perron Test.....	192
Random walk model	195
Differencing a Time Series	199
First Order Differencing	199
Differencing in R.....	199
Second Order Differencing	200
ARIMA Models in R	204
Selecting a Candidate ARIMA Model	205
ARMA versus ARIMA.....	211
Fitting an ARIMA Model.....	213
Assessing Performance and Diagnostics.....	219
Final Model Diagnostics	220
Forecast Evaluation Metrics	225
Forecasting with R.....	231
Out-of-Sample Forecasting	231
Dynamic Plotting.....	232
Chapter Review.....	233
Review Exercises	235
Chapter 7 – ARIMA Modeling	237
Acquiring and Analyzing Formats	237
Reading Time Series Data	238
Seasonal Decomposition.....	241

Decomposing Seasonal Data.....	241
Alternative Seasonal Decomposition.....	243
Seasonal plot.....	243
Month Plot.....	245
Testing for Stationarity	246
Building the Models.....	249
ARIMA(1,0,0).....	249
ARIMA(1,1,0).....	250
ARIMA(0,1,1) without constant	250
ARIMA(1,1,2) without constant.....	251
Developing the Forecast	253
Model Forecast Diagnostics.....	257
Analyzing the Goodness-of-Fit.....	258
Model Accuracy	259
Seasonal ARIMA models	264
Got Milk? (Train-Test Split).....	268
Chapter Review.....	269
Review Exercises.....	269
Chapter 8 – ARIMA Models using Python.....	271
State Space and ARIMA	271
Taxonomy	273
Working with ARIMA(p,q,d) with Python	273
Visualizing multiple time series	279
Defining color palettes.....	281
Correlations between two Variables	289
Correlation and Correlation Matrices.....	290
Visualize Correlation Matrices	291
Clustered heatmaps.....	291

Visual Decomposition	293
Time series forecasting with ARIMA.....	297
Model Diagnostics.....	299
Validating forecasts.....	300
Chapter Review.....	305
Review Exercises	306
Chapter 9 – Structural Models in R	309
Introduction	309
Trend	309
Cycle.....	310
Seasonality	310
Autoregressive Component	310
rucm package.....	311
Modeling the Flow of the Nile River	311
Forecasting with UCM.....	313
Alternative Implementation	314
Model Comparison	319
Chapter Review.....	321
Review Exercises	322
Chapter 10 – Advanced Time Series Topics	323
Time Series Intervention.....	323
Observations	323
Time Series Analysis.....	324
External Predictor Variables	326
Intervention	326
Predictor Contribution	326
Contribution Chart	328
ARIMA Model Parameters	328

The Art of Modeling	329
Conclusion.....	331
Exogenous Regressors in Python	331
Data.....	332
Auto Selecting an ARIMA Model.....	343
Forecasting¶	352
SARIMA Model with Exogenous Regressors	363
Vector Autoregression (VAR) processes¶.....	365
Analyze the time series characteristics.....	368
Test for Causation Among the Series.....	372
Test for Stationarity	376
Find the Optimal Order (K)	381
Prepare Training and Testing Datasets.....	383
Train the VAR model.....	384
Check for Serial Correlation of Residuals (Errors).....	386
Forecast VAR model.....	387
Invert the Transformation to Get the Real Forecast	388
Plot of Forecast vs Actuals	388
Evaluate the Model using the Test Set	389
Chapter Review.....	402
Review Exercises	404
Works Cited	407
Index	413

Preface

I set out on a venture to develop an online course in time series analysis using Python. It turned into a book. Then it expanded, and R became a co-conspirator with Python. Together, the two companions unleashed the power of time series analysis using open source tools.

For many years, SAS was my go-to for predictive modeling, and the only “real” time series analysis I performed relied solely on SAS. However, R and Python revealed the true nature of time series data, its decomposition, and proper model specification. So, here is a book that reveals some of what was manifested in my exploration.

Time Series Analysis with Open Source Tools is my fifth textbook of the kind, following *Data Science Applications using R*, *Text Analytics using Python & R*, *Predictive Analytics using R*, *Logistic Regression Inside-Out*, and *Operations Research using Open Source Tools*.

This book assumes a basic understanding of statistics and mathematical or statistical modeling. Although a little programming experience would be nice, it is not required. There are a few “formulas”, with no theorems or proofs, and calculus never appears.

We use current real-world data, like COVID-19, to motivate times series analysis have three **thread problems** that appear in nearly every chapter: "Got Milk?", "Got a Job?" and "Where's the Beef?" The data comes from dot-gov data sources on the Web, and are assessible at my GitHub site, or can be downloaded from the sources. The thread problems are threads that appear throughout the text to discover the applications of chapter content. Data sets can be downloaded at :

<https://github.com/stricje1/Data>

We emphasize using the right tool for the problem at hand. Sometimes it's Python and sometimes it's R. Occasionally, it's both. We use a lot of graphical methods to visualize the concepts and add plot aesthetics.

We start coding rights away in Chapter 1 and introduce one of our thread problems: Got Milk? loading data in the R-Studio and Jupyter Notebook environments.

In Chapter 2, we introduce the components of a times series: trend seasonality, cycle, and irregularity or error. We also and revisit the "Got Milk?" problem and perform decomposition.

In Chapter 3, we discover moving averages (MAs) and use COVID-19 to apply various MAs. We also further develop the "Got Milk?" problem.

In Chapter 4, we introduce simple exponential smoothing (SES), as well as Holt's and Holt-Winter's double and triple exponential smoothing. We also see out "Got a Job?" problem for the first time.

In Chapter 5, apply Python programming in Jupyter Notebook for the concepts we covered in Chapters 2, 3 and 4. We also begin work with the "Where's the Beef" problem.

In Chapter 6, we discuss stationarity and differencing, including unit root tests. We also introduce the train-test split process and apply it in R-Studio. We also cover model diagnostics, evaluating models, and out-of-sample forecasting.

In Chapter 7, we do a deep-dive into ARIMA and SARMIA (seasonal) modeling and forecast development. We use GLOBAL Land-Ocean Temperature Index as a motivating application.

In Chapter 8, we focus on ARIMA modeling using Python We also complete a complete analysis of the "Where's the Beef" problem.

In Chapter 9, we introduce structural models and analysis using unobserved component models (UCMs) and the annual flow of the *Nile River* in R-Studio. It also covers an alternative structural model approach using the *Nile River* data.

Chapter 10 introduces advanced time series analysis, including time series interventions, exogenous regressors, and vector autoregressive (VAR) processes. We use Python to apply advanced concepts to presidential approval ratings and a *S&P Composite* dataset.

Each chapter ends with a concise review and review exercise for R-Studio and Jupyter, as appropriate including a summary table of definitions introduced in the chapter. Although the tools used here are specified,

the user should feel free to employ R and Python tools they are comfortable with. I use Anaconda for Python, which includes Jupyter.

R may be freely downloaded at the Comprehensive R Archive Network (CRAN) website: <https://cran.r-project.org/>.

R-Studio may be freely downloaded at <https://www.rstudio.com/>.

Anaconda for Python 2.7 and 3.5 may be freely downloaded at <https://www.continuum.io/downloads>. The documentation may be freely accessed at <http://conda.pydata.org/docs/index.html>.

Data can be downloaded at <https://github.com/stricje1/Data>, unless otherwise specified.

R-Studio Notes

R is an excellent language for time series analysis. If you searched the annals of R-dom, you probably could never explore that vast array of times series libraries and functions. But, here are a few tips.

First, to distinguish R Studio input and output, we color coded them.

This is R input

This is R output

Most R functions, like `seasonplot()`, are not part of the R basic packages and have to be loaded (sometimes downloaded first) with a library call:

```
# Using R Studio help, I learned that seasonplot() is part of  
the forecast package.
```

`library(forecast)`

I also used color coded text:

```
# This is a comment  
  
library() # This is a frequently used basic R function  
  
Seasonplot() # this is the basic programming text color  
  
plot(X, col = "blue is aesthetic instructions",
```

```
main = "This is print text for the plot title",
       xlab = 'R does not read fancy ', you have to use plain ")

%<%, =, +      # This purplish color is for special punctuation
(symbols) that I don't want you to miss.

as.array[1, 2, 4] # these numbers are numeric values.

array2 <- as.array[1, 2, 4] # The number 2 in 'array2' is not a
numeric value.
```

To install an R package in R Studio, go to the **Tools** menu and select **Install packages...**, or enter > `install.packages("package name")`.

Jupyter Notebook Notes

The U.S. Bureau of Labor Statistics projects 21 percent growth for programming jobs from 2018 to 2028, which is more than four times the average for all occupations. What's more, the median annual pay for a software programmer is about \$106,000, which nearly three times the median pay for all U.S. workers (Eastwood, 2020). And, the most popular programming language is **Python!** **R** is number 8 in the countdown.

```
In [1]:
This is Python input
```

```
Out [1]:
This is Python output
```

Python functions, like `pyplot()`, are not part of a basic Jupyter package and have to be loaded (sometimes downloaded first) with a library call:

```
# You have to load packages into Jupyter also:
import matplotlib.pyplot as plt
```

I also used color coded text:

```
In [2]:
# This is a comment
import # This is a frequently used basic R function
plt.show() # this is the basic programming text color
plot(X, col = "red is aesthetic instructions",
      main = "This is print text for the plot title",
      xlab = 'R does not read fancy ', you have to use plain " ' )
```

```
::, =, +      # This purplish color is for special punctuation  
(symbols) that I don't want you to miss.  
In [3]:  
np.asarray[1, 2, 4] # these numbers are numeric values.  
array2 <- np.asarray[1, 2, 4] # The number 2 in 'array2' is not  
a numeric value.
```

To install packages on Python from Jupyter Notebook, type:

```
In [4]:  
pip install matplotlib
```

or enter it in a command prompt .

If you want a electronic copy of code, email me at jeff@humalytica.com

Additional Titles by the Author

Jeffrey Strickland (2020). *Data Science Applications using Python and R: Text Analytics*. Lulu.com. ISBN 978-1-716-89644-6.

Jeffrey Strickland (2020). *Data Science Applications using R*. Lulu.com. ISBN 978-0-359-81042-0.

Jeffrey Strickland (2017). Logistic Regression Inside and Out. Lulu.com. ISBN 978-1-365-81915-5.

Jeffrey Strickland (2015). *Data Science and Analytics for Ordinary People*. Lulu.com. ISBN 978-1-329-28062-5

Jeffrey Strickland (2015). *Operations Research using Open-Source Tools*. Lulu.com. ISBN 978-1-329-00404-7

Jeffrey Strickland (2015). *Predictive Analytics using R*. Lulu.com. ISBN 978-1-312-84101-7

Jeffrey Strickland (2015). *Missile Flight Simulation - Surface-to-Air Missiles*, 2nd Edition. Lulu.com. ISBN: 978-1-329-64495-3

Jeffrey Strickland (2014). *Crime Analysis and Mapping*. Lulu.com. ISBN 978-1-312-19311-6

Jeffrey Strickland (2014). *Predictive Modeling and Analytics*. Lulu.com. ISBN 978-1-312-37544-4

Jeffrey Strickland (2010). *Missile Flight Simulation - Surface-to-Air Missiles*, Lulu.com. ISBN: 978-0-557-88553-4

Jeffrey Strickland (2010). *Discrete Event Simulation using ExtendSim 8*, Lulu.com. ISBN 978-1-257-46132-5 (adopted for use at the University of Tennessee, Knoxville)

Jeffrey Strickland (2010). *Systems Engineering Processes and Practice*, Lulu.com. ISBN: 978-1-257-09273-4

Jeffrey Strickland (2011). *Mathematical Modeling of Warfare and Combat Phenomenon*, Lulu.com. ISBN: 978-1-4583-9255-8

Jeffrey Strickland (2011). *Using Math to Defeat the Enemy: Combat Modeling for Simulation*, Lulu.com. ISBN: 978-1-257-83225-5

Chapter 1 – Introduction

What is Time Series Data?

To approach **time series analysis and forecasting**, we must first answer the question regarding what constitutes time series data. A time series is a sequence of data points, typically consisting of successive measurements made over a time interval. Examples of time series are solar activity, ocean tides, stock market behavior, and the spread of disease. Time series are often plotted using line charts. Time series data are found in any domain of applied science and engineering which involves time-based measurements.

Definition 1.1. Time series data are a collection of observations, y_t , each one recorded at time, t . (Time can be discrete, $t = 1, 2, 3, \dots$, or continuous, $t > 0$.)

Most often, the observations are made at regular time intervals. Time series analysis accounts for the fact that data points taken over time may have an internal structure, such as autocorrelation, trend or seasonal variation.

The graph in Figure 1-1 represents simulated data for unsecured consumer loans from January 2002 to September 2011, by month. This is typical time series data.

What Data Formats work Best?

For analysis with open-source tools, comma separated files, such as .CSV and .txt work best. These files are easily read into Python and R. Date indexes are not required, as they can be added when building the time series data in R, for example.

The data represented by Figure 1-1 is stored in a .CSV file and easily imported to R and Python, as well as some other tools.

Unsecured Consumer Loans

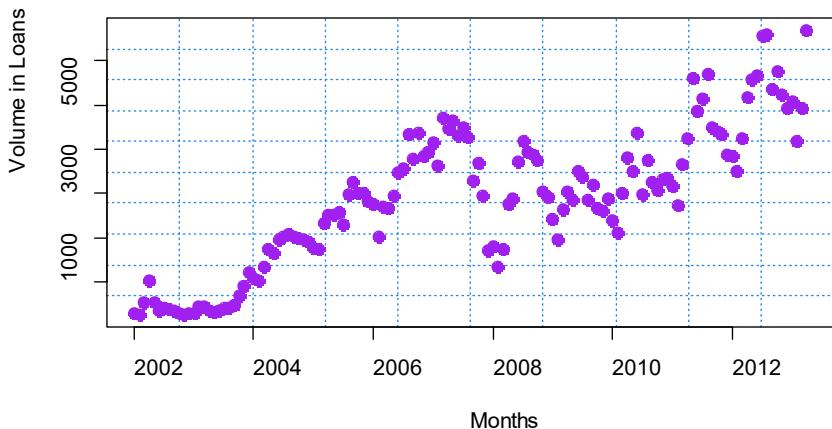


Figure 1-1. Simulated time series data for unsecured consumer loans

What is Time Series Analysis?

Time series analysis comprises methods for analyzing time series data to extract meaningful statistics and other characteristics of time series data. It focuses on comparing values of a single time series or multiple dependent time series at different points in time.

What is Time Series Forecasting?

Time series forecasting is the use of a time series model to predict future values based on previously observed values in the series. In other words, once we construct a time series model, we use the equation that it produces to generate future values. The further into the future we try to project, the less confidence we have in those projections. Although the future time period is relative to the problem, we want to limit those projections. Figure 1-2 show a financial time series with forecast 48 months out. Notice that after about twelve months, the confidence interval (gray area) blows up. We have less confidence in the forecast after twelve months.

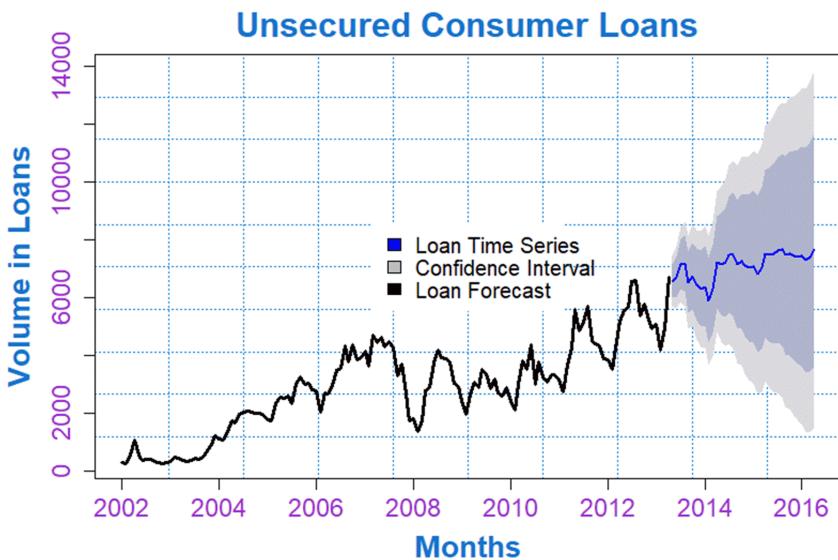


Figure 1-2. Financial time series with forecast

Where do we use Time Series Analysis?

We use time series analysis and forecasting for many applications where pertinent time series data can be collected, such as:

- Budget Analysis
- Financial Market Analysis
- Census Analysis
- Inventory Management
- Economic Forecasting
- Marketing and Sales Forecasting
- Yield Projections
- Seismological Predictions
- Workload Projections
- Military Planning

What are the Goals of Time Series Analysis?

There are two main goals of time series analysis. First, we identify the nature of the phenomenon represented by the sequence of observations in the data. Second, we use the data to forecast or predict future values of the time series variable. Both of these goals require that

we identify the pattern of observed time series data and more or less formally describe it. Once we establish the pattern, we can interpret and integrate it with other data (i.e., use it in our theory of the investigated phenomenon, e.g., seasonal commodity prices). Regardless of the depth of our understanding and the validity of our interpretation of the phenomenon, we can extrapolate the identified pattern to predict future events with this caveat: the further out in time we try to predict, the less accurate is the forecast.

What Techniques are used in Time Series Analysis?

The fitting of time series models can be an ambitious yet ruthless undertaking. It requires much more data preparation than the usual statistical models applied to “ordinary” data—such as response models, uplift models, and so on—where trends and seasonal effects may not be present. For example, unlike data used for standard linear regression, time series data are not necessarily independent and not necessarily identically distributed. One defining characteristic of time series is that this is a list of observations where the ordering matters. Ordering is very important because there is a dependency and changing the order could change the meaning of the data.

There are several different methods for modeling time series data including the following:

- Box-Jenkins ARIMA models
- Box-Jenkins Multivariate Models
- Holt-Winters Exponential Smoothing (single, double, triple)
- Unobserved Components Model (UCM)
- Generalized Linear Autoregressive Moving Average Models

Holt-Winters Exponential Smoothing is often referred to a **moving average** (MA) models. The user’s application and preference will often decide the selection of the appropriate technique. It is beyond the scope of this book to cover all these methods.

What are Smoothing Techniques?

Inherent in the collection of data taken over time is some form of random variation. There are methods for reducing or canceling the

effect due to random variation. “Smoothing” data removes random variation and shows the underlying trends and cyclic components and is an often-used technique in industry.

There are two distinct groups of smoothing methods

- Averaging Methods
- Exponential Smoothing Methods
- Filtering Methods

How do we Load Time Series Data?

It is common to have data, appropriate for time series analysis, that has not been formatted as time series data. We’ll take R for example. The most common issue when using time series data in R is getting it into a format that is easily readable by R and any extra packages we are using. A common format for time series data puts the largest chunk of time first (e.g., year) and gets progressively smaller, like this: 2017-02-25 18:30:45.

This can be generalized to `YYYY-MM-DD HH:MM:SS`. If we can record our data in this format to begin with, analysis will be much easier. If time is not important for our measurements, we can drop the `HH:MM:SS` piece from our records.

Got Milk (Data)?

Got Milk? was an American advertising campaign encouraging the consumption of milk, which was created by the advertising agency Goodby Silverstein & Partners for the California Milk Processor Board in 1993, and was later licensed for use by milk processors and dairy farmers. For example, the data in `milk_monthly` shows the monthly milk production by a herd of cattle between 1962 and 1975. We will load these files (and one that is daily) and check the form of the data set.

```
milk_mon <- read.csv("D:\\Documents\\\\DATA\\\\milk_monthly.txt")
milk_day <- read.csv("D:\\Documents\\\\DATA\\\\milk_daily.txt")
head(milk_mon)
class(milk_mon)
class(milk_mon$month)
head(milk_mon)
```

```
month milk_prod_per_cow_kg
1 1962-01-01      265.05
2 1962-02-01      252.45
3 1962-03-01      288.00
4 1962-04-01      295.20
5 1962-05-01      327.15
6 1962-06-01      313.65
```

```
class(milk_mon)
```

```
[1] "data.frame"
```

```
class(milk_mon$month)
```

```
[1] "factor"
```

The function `head(monthly_milk)` shows us that the month column is in a sensible format (`YYYY-MM-DD`) and contains no time `data`. Also, the function `class(monthly_milk)` shows us that the data is in the form of a data frame, which is ideal for our purposes. However, `class(monthly_milk$month)` shows us that the data is currently being interpreted as a factor. Factors are a data class that can have distinct categories, but infer no sequential order or hierarchy beyond simple alphabetic order, so if we tried to analyze this data in its current form, R would not understand that 1962-01-01 comes before 1962-02-01. Fortunately, R also has a `Date` class, which is much easier to work with. So, we coerce the data to the `Date` class.

```
# Coerce to `Date` class
milk_mon$month_date <- as.Date(milk_mon$month,
                                format = "%Y-%m-%d")
# Check it worked
class(milk_mon$month_date)
```

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

Data in the `Date` class in the conventional `YYYY-MM-DD` format are easier to use with `ggplot2` and various time series analysis packages. In the code above, `format =` tells `as.Date()` what form the original data is in. The symbols `%Y`, `%m`, `%d` etc. are codes understood by many programming languages to define date class data. Note that `as.Date()` requires a year, month, and day somewhere in the original data. So, if the original data doesn't have one of those, you can add them manually

using `paste()`. We will see an example of using `paste()` to add date information later on when we run some forecast models.

Table 1-1 provides an expanded table of date codes, which you can use for reference:

Table 1-1. Date codes used in R.

Name	Code	Example
Long year	%Y	2017
Short year	%y	17
Numeric month	%m	02
Abbreviated month	%b	Feb
Full month	%B	February
Day of the month	%d	25
Abbreviated weekday	%a	Sat
Full weekday	%A	Monday
Day of the week (1-7)	%u	6
Day of the year	%j	56

To transform a Date class object into a character format with an alternative layout, we can use `format()` in conjunction with any of the date codes in the Table 1-1 above. For example, we could transform 2017-02-25 into February - Saturday 25 - 2017. But note that this new character format will not be interpreted as a date by R in analysis. Try a few different combinations of date codes from the table above, using the code below as an example.

```
class(format(milk_mon$month_date, format = "%Y-%B-%u"))
# class is no longer `Date`
```

```
[1] "character"
```

```
format(milk_mon$month_date, format = "%Y-%B-%u")
```

```
[1] "1962-January-1"      "1962-February-4"    "1962-March-4"
[4] "1962-April-7"        "1962-May-2"          "1962-June-5"
[7] "1962-July-7"         "1962-August-3"       "1962-September-6"
[10] "1962-October-1"       "1962-November-4"     "1962-December-6"
[13] "1963-January-2"       "1963-February-5"     "1963-March-5"
[16] "1963-April-1"         "1963-May-3"          "1963-June-6"
[19] "1963-July-1"          "1963-August-4"       "1963-September-7"
[22] "1963-October-2"        "1963-November-5"     "1963-December-7"
[25] "1964-January-3"        "1964-February-6"     "1964-March-7"
[28] "1964-April-3"          "1964-May-5"          "1964-June-1"
[31] "1964-July-3"           "1964-August-6"        "1964-September-2"
[34] "1964-October-4"         "1964-November-7"      "1964-December-2"
[37] "1965-January-5"         "1965-February-1"      "1965-March-1"
[40] "1965-April-4"          "1965-May-6"          "1965-June-2"

[160] "1975-April-2"          "1975-May-4"          "1975-June-7"
[163] "1975-July-2"           "1975-August-5"        "1975-September-1"
[166] "1975-October-3"         "1975-November-6"      "1975-December-1"
```

Dates and times

Sometimes, both the date and time of observation are important. The best way to format time information is to append it after the date in the same column like this:

```
2017-02-25 18:30:45
```

The most appropriate and useable class for this data is the [POSIXct](#) [POSIXt](#) double. To explore this data class, we look at another milk production dataset, this time with higher resolution, showing morning and evening milking times over the course of a few months (we have already loaded it).

```
head(milk_day)
```

	date_time	milk_prod_per_cow_kg
1	1975-01-01 05:00:00	11.21745
2	1975-01-01 17:00:00	10.67182
3	1975-01-02 05:00:00	10.90791
4	1975-01-02 17:00:00	11.03970
5	1975-01-03 05:00:00	12.53303
6	1975-01-03 17:00:00	10.69446

```
class(milk_day$date_time)
```

```
[1] "factor"
```

Again, the date and time are in a sensible format (YYYY-MM-DD HH:MM:SS), but are interpreted by R as being in the factor class. We use functions to convert between character representations and objects of classes "`POSIXlt`" and "`POSIXct`" representing calendar dates and times.

```
milk_day$date_time_posix <- as.POSIXct(milk_day$date_time,  
format = "%Y-%m-%d %H:%M:%S")  
class(milk_day$date_time_posix)
```

```
[1] "POSIXct" "POSIXt"
```

Table 1-2 below is an expanded table of time codes which you can use for reference:

Table 1-2. Expanded table of time codes used in R.

Name	Code	Example
Hour (24 hour)	%H	18
Hour (12 hour)	%I	06
Minute	%M	30
AM/PM (only with %I)	%p	AM
Second	%S	45

Correcting badly formatted date data

If our data are not already nicely formatted it is easy to transform it back into a useable format using `format()`, before we transform it to `Date` class. First, we create some badly formatted date data to look similar to 01/Dec/1975-1, the day of the month, abbreviated month, year and day of the week:

```
milk_mon$bad_date <- format(milk_mon$month_date,  
format = "%d/%b/%Y-%u")  
head(milk_mon$bad_date)
```

```
[1] "01/Jan/1962-1" "01/Feb/1962-4" "01/Mar/1962-4"  
[4] "01/Apr/1962-7" "01/May/1962-2" "01/Jun/1962-5"
```

```
class(milk_mon$bad_date) # Not in `Date` class
```

```
[1] "character"
```

Then to transform it back to the useful YYYY-MM-DD Date format, we just use `as.Date()`, specifying the format that the badly formatted data is in.

```
milk_mon$good_date <- as.Date(milk_mon$bad_date,  
                                format = "%d/%b/%Y-%u")  
head(milk_mon$good_date)
```

```
[1] "1962-01-01" "1962-02-01" "1962-03-01" "1962-04-01"  
[5] "1962-05-01" "1962-06-01"
```

```
class(milk_mon$good_date)
```

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

Now we know how to transform data in to the `Date` class, and how to create character class data from `Date` data. We will use this information in subsequent chapters.

Chapter Review

In this chapter, we looked at what comprises time series data, what constitutes time series analysis, and what techniques might be used for modeling. We also discussed using the models to create forecasts and how we might use them.

Definition Number	Definition Citation
Definition 1.1: Time Series Data	<i>Time series data are a collection of observations, y_t, each recorded at time, t. (Time can be discrete, $t = 1, 2, 3, \dots$, or continuous, $t > 0$.)</i>

Review Exercises

1. Think of some data you have used or observed that might be considered time series data.
 - a. What makes the data a time series?
 - b. How can the data be analyzed or modeled?
 - c. How can the analysis be used?
2. How might a times series model be used in Economic Forecasting?
3. Why are forecasts more accurate the closer they are to the modeling window (the period for which data is available for modeling)?
4. Repeat the milk production example using the daily data.

Chapter 2 – Component of Time Series Data

In Chapter1, we presented some basic concepts and applications for time series models, but I did not write much about time series data. Here we will explore characteristics or components of time series data. . We can represent a time series as follows:

$$y_t = T_t + C_t + S_t + AR_t + \sum_{j=1}^m (B_j x_{jt} + e_t)$$
$$\epsilon_t \sim i.i.d. N(0, \sigma_\epsilon^2)$$

The components, T_t , S_t , and C_t represent the *trend*, *seasonal*, and *cyclical* components, respectively (they are usually represented by Greek letters); the term AR_t represents an unobserved *autoregressive* component; the term $\sum_{j=1}^m (B_j x_{jt} + e_t)$ gives the contribution of *regression* variables with fixed or time varying regression coefficients. The *irregular* or *error terms* ϵ_t are assumed to be independently and identically distributed as Standard Normal distributions.

To visualize the components of time series data, we will look at an example using number of births per month in New York city, from January 1946 to December 1959 shown in Figure 2-1 (Newton, 1988).

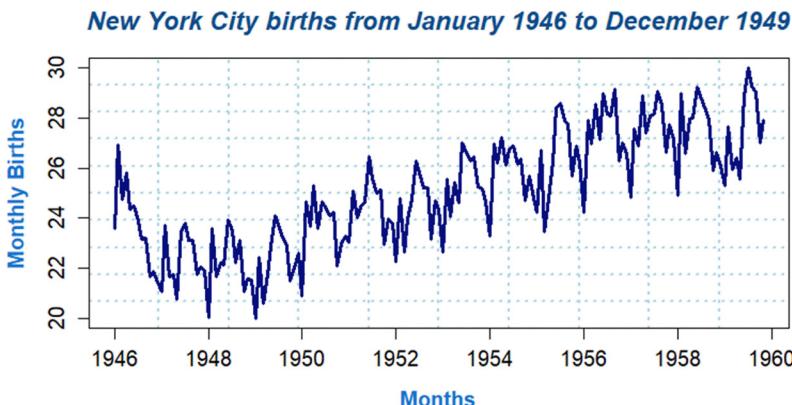


Figure 2-1. New York City births from January 1946 to December 1949.

The components of the New York City (NYC) births time series as shown in Figure 2-2.

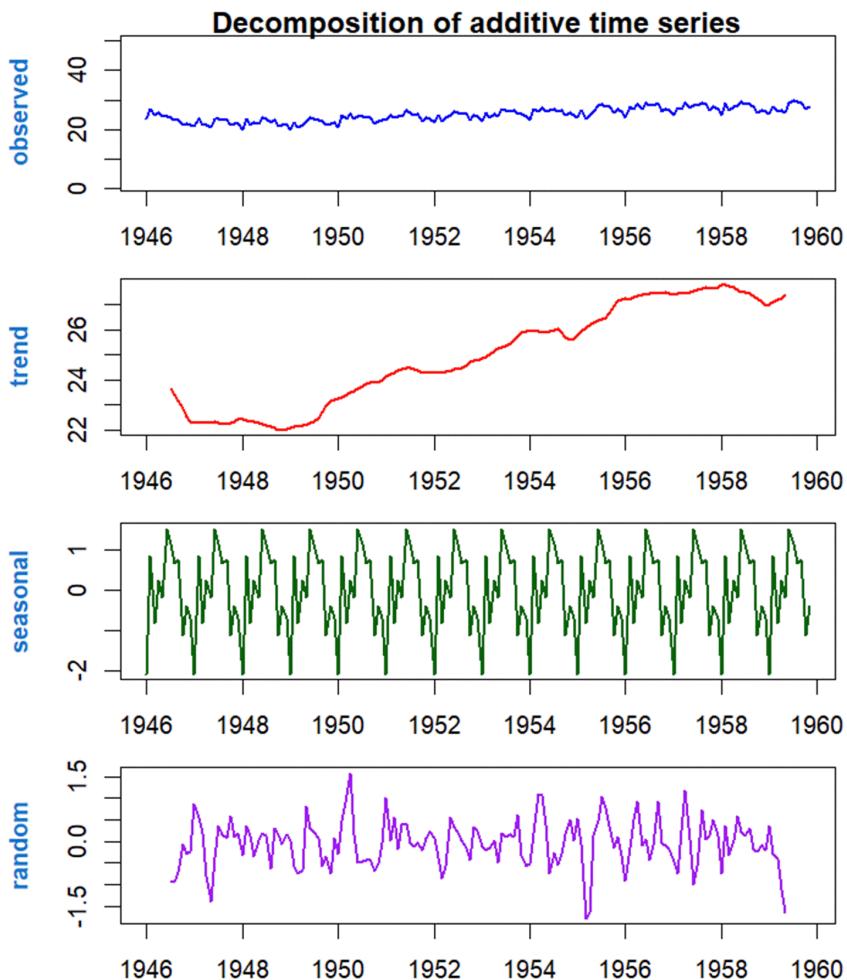


Figure 2-2. Decomposition of time series data into its components.

Definition 2.1. Time series data can contain multiple patterns acting at different temporal scales. **Decomposition** is the process of isolating each of these patterns.

Trend

Trend is the long-term movement in a time series without time or irregular effects and is a reflection of the underlying level. The trend can be increasing or decreasing as well as linear or nonlinear. You might hear terms like *nonseasonal trend*, *additive trend*, and *multiplicative trend*, combined with *linear*, *damped*, *polynomial*, or *exponential trend*. We now define trend:

Definition 2.2. The **trend** shows the general tendency of the data to increase or decrease during a long period of time. A **trend** is a smooth, general, long-term, average tendency. It is not always necessary that the increase or decrease is in the same direction throughout the given period of time.

As an example, suppose we want to forecast the number of households that purchase an LED television on a monthly basis. Every year, the number of households that purchase an LED TV will increase; however, this trend will be damped (e.g., the upward trend will slowly disappear) over time as the market becomes saturated. This would be described as damped nonseasonal trend, assuming that the sales are not seasonal. Another example is the meat data set, which has been made available to us from the U.S. Department of Agriculture. It contains metrics on livestock, dairy, and poultry outlook and production. Each shows an upward trend except for lamb and veal.

There are no proven “automatic” techniques to identify trend components in the time series data; however, as long as the trend is monotonous (consistently increasing or decreasing) that part of data analysis is typically not very difficult. If the time series data contain considerable error, then the first step in the process of trend identification is smoothing. In Chapter 3, we will discuss ways to perform trend analysis, including “smoothing.”

Seasonality

The phenomena that produce time series data exhibit **seasonality**. Seasonality is a component of a time series in which the data experiences regular and predictable changes that recur every unit of time, e.g., calendar or fiscal year. Formally, give a definition.

Definition 2.3. In time series data, **seasonality** is the presence of variations that occur at specific regular intervals less than a year, such as weekly, monthly, or quarterly.

Seasonality may be caused by various factors, such as weather, vacation, and holidays and consists of periodic, repetitive, and generally regular and predictable patterns in the levels of a time series.

Seasonal fluctuations in a time series can be contrasted with cyclical patterns. The latter occur when the data exhibits rises and falls that are not of a fixed period. Such non-seasonal fluctuations are usually due to economic conditions and are often related to the "business cycle"; their period usually extends beyond a single year, and the fluctuations are usually of at least two years.

For instance, summer clothing is sold more in the spring than other seasons and vacation packages sell more in the summer when school is not in session. Similarly, in habitats with cold winters may consume far more heating oil, natural gas, etc. In the insurance industry, times series data comes into play in places susceptible to tropical storms, wildfires, and hail producing storms. The stock market is affected by many of these factors, and it too exhibits seasonality.

Cycle

The **cycle** has often been described as a non-fixed pattern usually of at least two years in duration. The length of the cycle is described as the period. An example of time series data exhibiting cyclic behavior is the harvesting of game or fish.

Definition 2.4. A **cycle** occurs when the data exhibit rises and falls that are not of a fixed frequency. These fluctuations are usually due to economic conditions, and are often related to the "business cycle". The duration of these fluctuations is usually at least 2 years.

For instance, harvesting Georges Bank haddock. The cycle may have a period of length four to five years. However, the haddock population is dependent on herring roe (and external influence) for its survival, in addition to the harvesting rate. Another example is pork retailer prices

in Germany from April 1995 to April 2012 (see Chapter 4 for more details) (USDA, 2016).

Irregular

In addition to the foregoing components, we have an **irregular component**, which is unpredictable. It is the residual time series after the trend, cycle, and the seasonal components have been removed. It results from short-term fluctuations in a series which are not systematic and, in some instances, not predictable.

Definition 2.5. The **irregular component** (or "noise") at time t , or I_t , describes random, irregular influences. It represents the residuals or remainder of the time series after the other components have been removed.

Other Components

Rather than modeling the cyclical nature of a time series via either the deterministic cyclical model or the stochastic cyclical model, one can use the rather straightforward specification called the autoregression component.

Definition 2.6. An **autoregressive component** is a special case of cycle that corresponds to the frequency of zero or π , given by

$$AR_t = \rho AR_{t-1} + v_t, v_t \sim i.i.d. N(0, \sigma_\varepsilon^2),$$

where the AR_t follows a first-order autoregression with $-1 < \rho < 1$, and they represent the "momentum" of the time series as it relates to its past observations.

This autoregression, despite its simplicity, can capture many of the movements in time series data that represent business cycle inertia and that are present in many business and economic time series.

Definition 2.7. *The regressive component*

$$\sum_{j=1}^m (B_j x_{jt} + e_t)$$

includes those terms that we are accustomed to in linear regression and are not unique to time series data.

These regressive component terms are the **explanatory variables** or the **causal factors** that may affect the time series in question.

Definition 2.8. *A stationary time series is one whose properties do not depend on the time at which the series is observed. That is y_t does not depend on y_{t-1}, \dots, y_{t-n} .*

So time series with trends, or with seasonality, are not stationary — the trend and seasonality will affect the value of the time series at different times. On the other hand, a white noise series is stationary — it does not matter when you observe it, it should look much the same at any period. Identifying stationarity will be covered in Chapter 3.

Got Milk (Components)?

In Chapter 1, we loaded data for monthly and daily milk production by a herd of cattle. Now we will examine the components of that time series, as it appears in Figure 2-3. The `color`, `linetype`, and size arguments in `geom_line` provide the aesthetics (`aes`) of the time series. In `scale_x_date`, years are called as the scale and `date_breaks` sets the interval between dates. Notice that putting our entire `ggplot` code in brackets () creates the graph and then shows it in the plot viewer.

```
par(mar = c(5, 5, 3,.25))
par(col.lab = 'dodgerblue3', font.lab = 2, cex.main = 3, cex.axis = 2, cex.lab = 3)
ggplot(milk_day, aes(x = date_time_posix, y = milk_prod_per_cow_kg)) +
  geom_line(color = "purple", linetype = 1, size = 2) +
  scale_x_date(date_labels = "%Y", date_breaks = "1 year") +
  ggtitle("Monthly Milk Production") +
  ylab("Production in kg") +
  xlab("Years") +
  theme(plot.title = element_text(size=24),
```

```
axis.text=element_text(size=14, face="bold"),
axis.title=element_text(size=18, face="bold"))
```

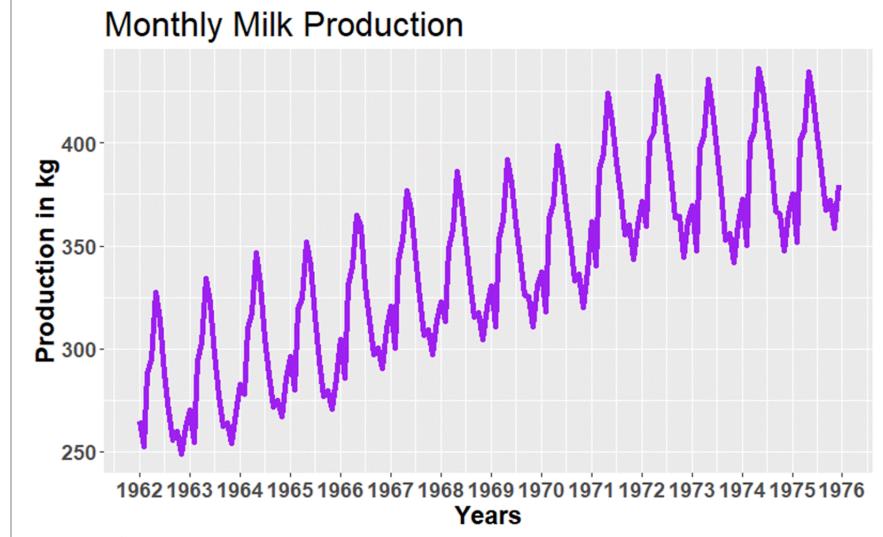


Figure 2-3. Monthly milk production per cow in a herd.

Using `theme_classic()` produces a plot that is a little more aesthetically pleasing than the default options. If you want to learn more about the basics of `ggplot2` and customizing themes are numerous tutorials online, like one at <http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html>.

We can plot the daily data similarly using `scale_x_datetime()`. In Figure 2-4 and the code, notice that we set `date_breaks` to one week in order to keep the horizontal scale legible.

```
ggplot(milk_day, aes(x = date_time_posix, y = milk_prod_per_cow_kg)) +
  geom_line(color = "green3", linetype = 1, size = 2) +
  #scale_x_date(date_labels = "%d", date_breaks = "1 year") +
  ggtitle("Daily Milk Production") +
  ylab("Production in kg") +
  xlab("Days") +
  theme(plot.title = element_text(size=24),
        axis.text=element_text(size=14, face="bold"),
        axis.title=element_text(size=18, face="bold"))
```

Now, we decompose the time series shown in Figure 2-5Figure 2-3.

```
(decomp_1 <- ggplot(milk_mon, aes(x = month_date,
  y = milk_prod_per_cow_kg)) +
  geom_line(color = "dark red", linetype = 1, size = 1.1) +
  scale_x_date(date_labels = "%Y", date_breaks = "1 year") +
  theme_classic())
```

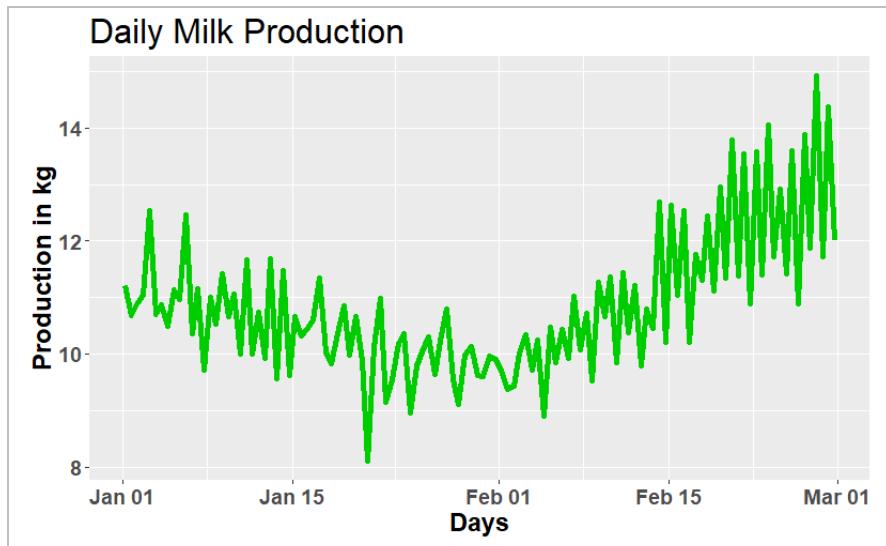


Figure 2-4. Daily milk production per cow in a herd.

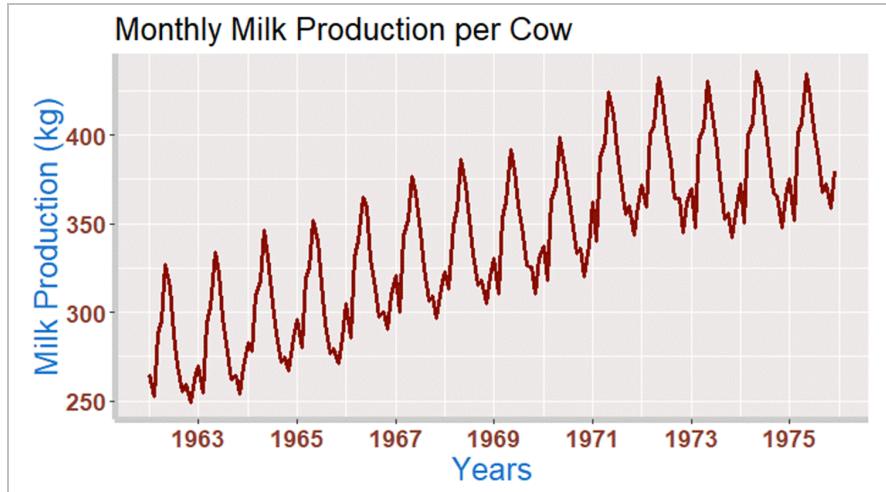


Figure 2-5. Time series of milk production

It looks like there is a general upward trend: more milk is being produced in 1975 than in 1962. This is known as a “smooth” pattern, one that increases or decreases regularly (monotonically) over the course of the time series. We can see this pattern more clearly by plotting a loess regression through the data.

Definition 2.9. Loess regression is a nonparametric technique that uses local weighted regression to fit a smooth curve through points in a scatter plot. Loess curves are can reveal trends and cycles in data that might be difficult to model with a parametric curve.

We use the `geom_smooth()` function with the `method = loess` regression to fit a smooth curve between two variables. The `span` argument in sets the number of points used to plot each local regression in the curve: the smaller the number, the more points are used and the more closely the curve will fit the original data (see *Figure 2-6*).

```
(decomp_2 <- ggplot(milk_mon, aes(x = month_date,
  y = milk_prod_per_cow_kg)) +
  geom_line(color = "dark red", linetype = 1, size = 1.1) +
  geom_smooth(method = "loess", se = FALSE, span = 0.6) +
  theme_classic())
```

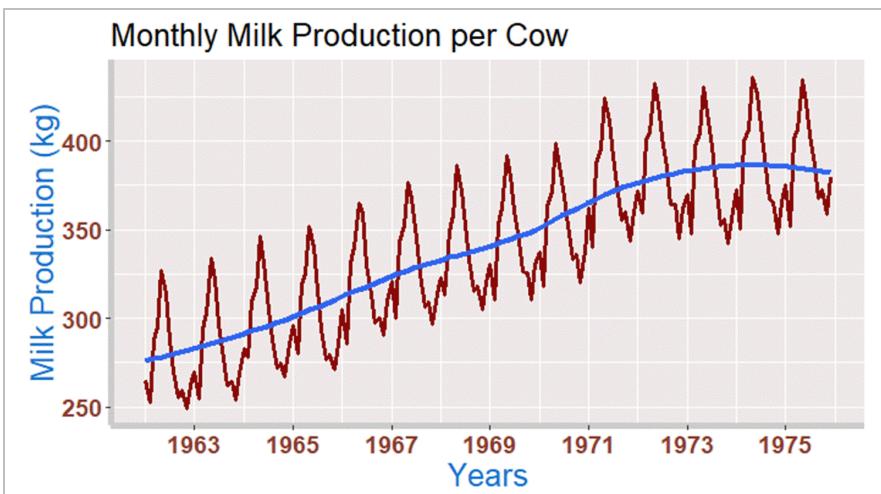


Figure 2-6. Time series of milk production with loess regression smoothing curve

Next, it looks like there are some peaks and troughs that occur regularly in each year. This is a “**seasonal**” pattern. We can investigate this pattern

more by plotting each year as its own line and comparing the different years, as shown in Figure 2-7.

```
# Extract month and year and store in separate columns
milk_mon$year <- format(milk_mon$month_date, format = "%Y")
milk_mon$month_num <- format(milk_mon$month_date, format = "%m")

# Create a color palette using the `colortools` package
year_pal <- sequential(color = "darkturquoise", percentage = 5,
                        what = "value")

# Make the plot
(seasonal <- ggplot(milk_mon, aes(x = month_num,
                                    y = milk_prod_per_cow_kg, group = year)) +
  geom_line(aes(color = year)) +
  theme_classic() +
  scale_color_manual(values = year_pal))
```

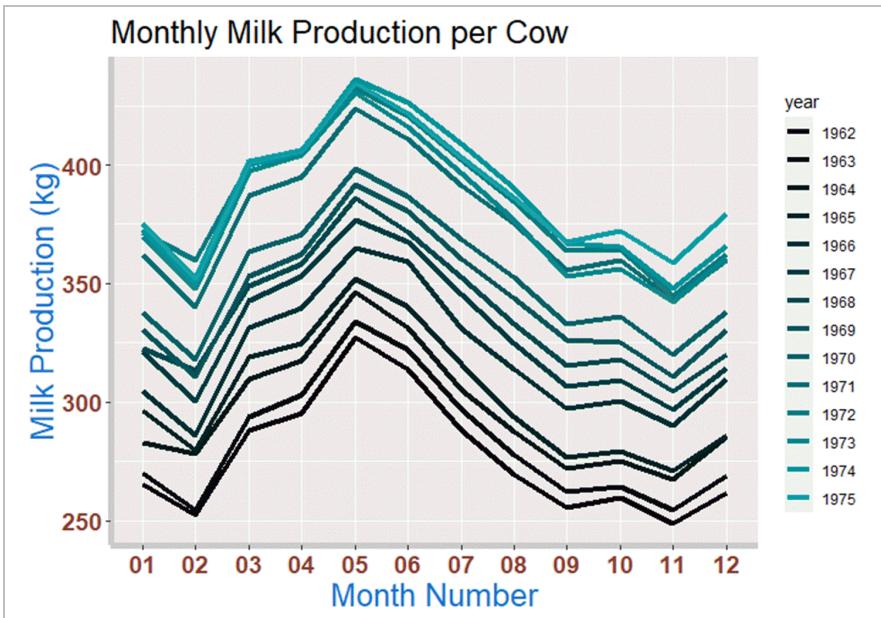


Figure 2-7. Time series of milk production for each year

It's clear from the plot that while milk production is steadily getting higher, the same pattern occurs throughout each year, with a peak in May and a trough in November. Another way of viewing this is to create a 12-month period box plot. The box plot will show the same pattern. For example, take February in . Figure 2-8 shows the low year (1963) the interquartile range, the mean, and the high year (1975).

```
box plot(milk_ts ~ cycle(milk_mon_ts), xlab = "Month",
         ylab = "KG", col = "light blue",
         main = "Monthly Milk Production Per Cow - Boxplot")
```

Monthly Milk Production per Cow - Boxplot

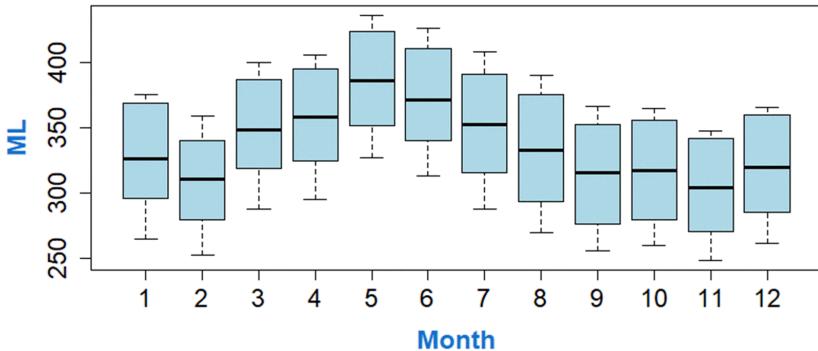


Figure 2-8. Boxplot of seasonal cyclic behavior as reflected in Figure 2-7

“Cyclic” trends are similar to seasonal trends in that they recur over time, but occur over longer time scales. It may be that the general upward trend and plateau seen with the loess regression may be part of a longer decadal cycle related to sunspot activity, but this is impossible to test without a longer time series.

We can use `ggplot2` to convert the time series data frame to a `ts` class object and use them to decompose the time series using `stl()` from the `stats`-package. The plot in Figure 2-9 shows the components of the time series that results from this process.

```
# Transform to `ts` class
# Specify start & end year, measurement frequency (monthly = 12)
milk_mon_ts <- ts(milk_mon$milk_prod, start = 1962, end = 1975,
                  freq = 12)

# Decompose using `stl()`
milk_mon_stl <- stl(milk_mon_ts, s.window = "period")

# Generate plots
# top=original data, second=estimated seasonal, third=estimated
# smooth trend, bottom=estimated irregular element i.e. unaccounted
# for variation
plot(milk_mon_stl)
```

Milk Production Time Series Decomposition

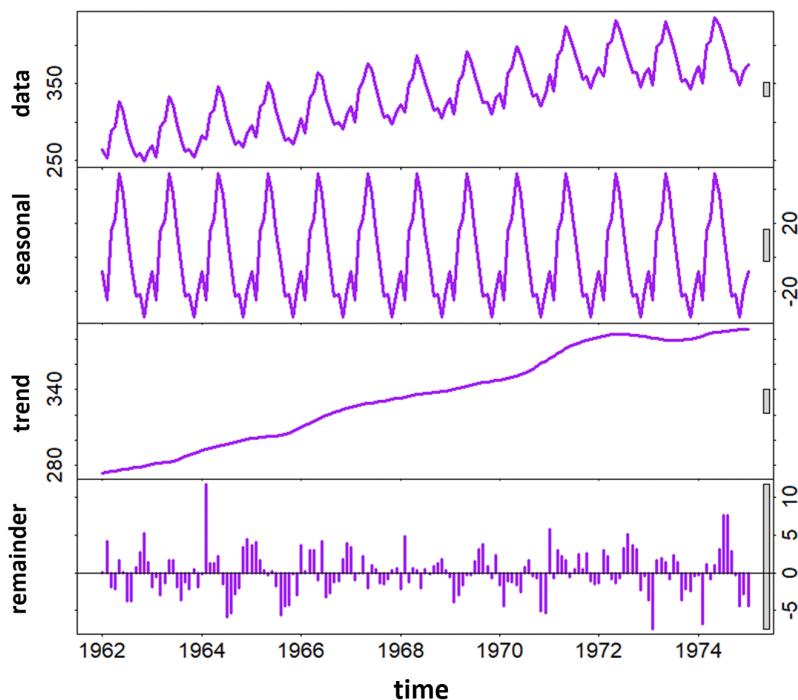


Figure 2-9. Time series component decomposition plot for milk production.

The `monthplot()` function extracts subsamples from a time series and plot them all in one frame, producing a seasonal variation plot as shown in Figure 2-10.

```
# variation in milk production for each month
monthplot(milk_mon_ts, choice = "seasonal", col.base = 1:6,
          lty.base = 1, lwd.base = 3, xlab = "Month",
          ylab = "Milk per Cow (kg)",
          main = "Monthly Milk Production")
```

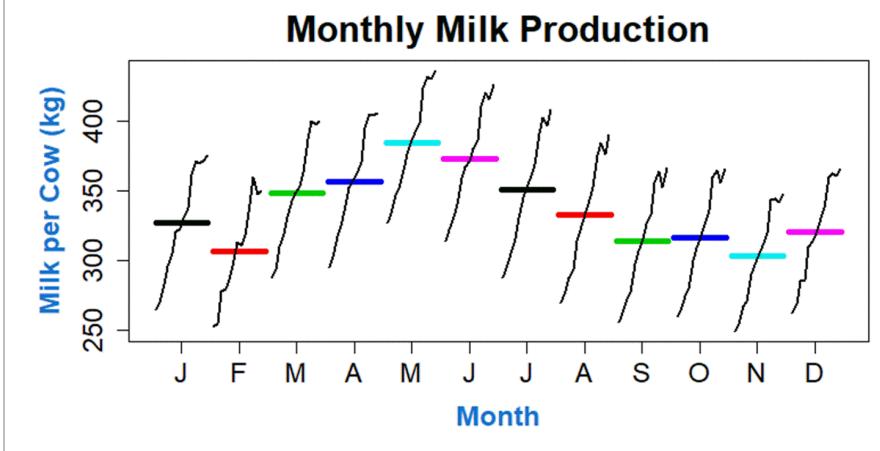


Figure 2-10. Variation in milk production for each month

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. These are exactly the same data as were shown earlier, but now the data from each season are overlapped. A seasonal plot allows the underlying seasonal pattern to be seen more clearly, and is especially useful in identifying years in which the pattern changes.

In Figure 2-11, it is clear that there is a significant increase in milk production during May of each year. Actually, these are probably arrival of Spring (peaking in May). The graph also shows that there was a decline in production during the Winter months.

```
seasonplot(milk_mon_ts, col = 1:6, lty = 1, lwd = 2,
           year.labels = TRUE, year.labels.left = TRUE,
           xlab = "Month", ylab = "Milk per Cow (kg)",
           main = "Monthly Milk Production")
```

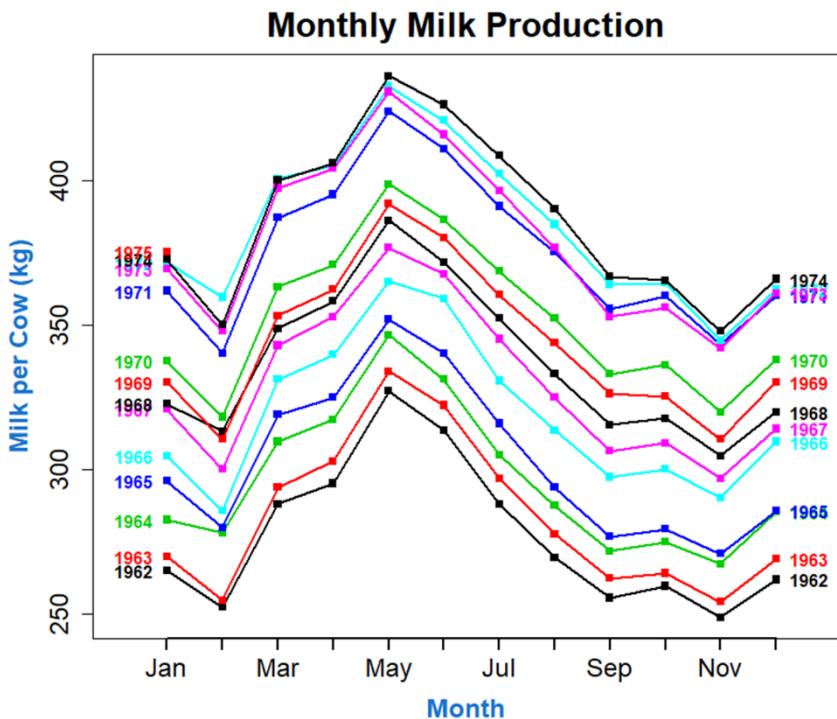


Figure 2-11. Seasonal plot of monthly milk production

Using `ggseasonplot()`, we produce an alternative plot of the data in Figure 2-11, with polar coordinates in a seasonal contour plot shown in Figure 2-12. Notice that the contours are offset with May as the attracting month.

```
ggseasonplot(milk_mon_ts, polar=TRUE) +
  ylab("production in kilograms (kg)") +
  geom_line(linetype = 1, size = 1.05) +
  ggtitle("Monthly milk production per month")
```

Component Analysis Essentials

There are a few ideas that we should review that are essential in time series analysis. The first of these is lag.

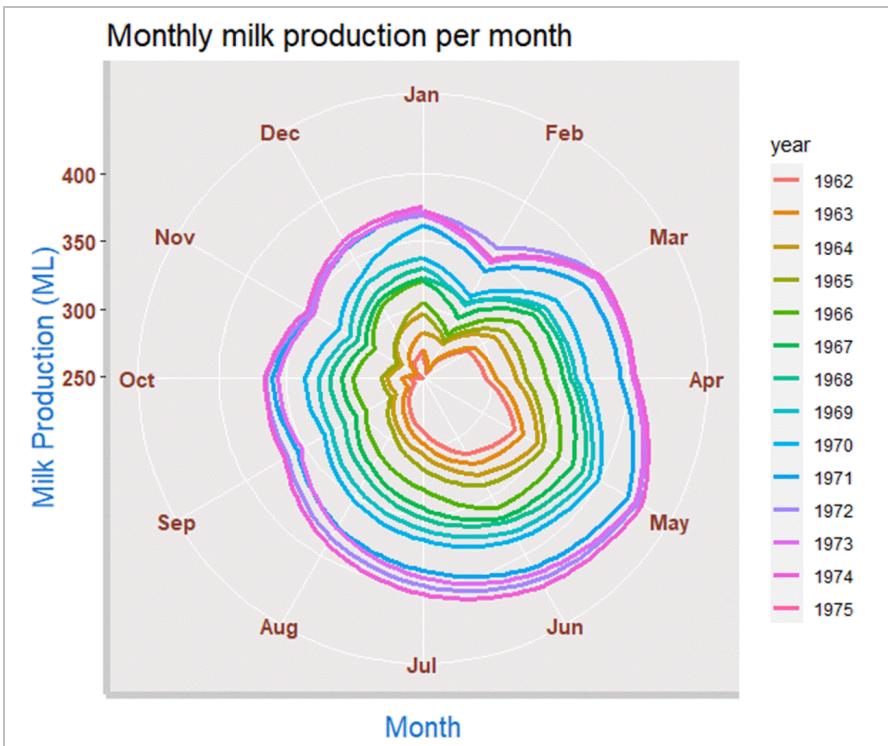


Figure 2-12. Polar seasonal plot of monthly milk production per month.

Time Series Lag

We use lag and lag plots in analysis of time series components, especially seasonality.

Definition 2.10. A *lag* is a fixed amount of passing time, such that the k^{th} lag (lag_k) is the time period that happened k time points before time i :

$$\text{lag}_k(Y_i) = Y_{i-k}$$

The most commonly used lag is 1, called a first-order lag.

For example, $\text{lag}_1(Y_2) = Y_{2-1} = Y_1$ and $\text{lag}_4(Y_9) = Y_{9-4} = Y_5$.

A lag plot is a special type of scatter plot with the two variables (X,Y) **lagged**. One set of observations in a time series is plotted (lagged) against a second, later set of data. Plots with a single plotted lag are the

most common. However, it is possible to create a lag plot with multiple lags with separate groups (typically different colors) representing each lag.

Example. We generate 11 observations of a times series to demonstrate lag plots. Assuming that trend and periodicity are not an issue, we are only looking for autocorrelation. Of course, in practice, you would want more than 11 observations if you were testing for autocorrelation. First, use `c()` to put the data, assumed to be in time order, into an array and use `length()` to check the number of observations. The first call to `plot()` generates a plot of the values of `x` against "Index", that is, the order in the array. The argument `type="b"` means that the points and their connecting lines are both shown in Figure 2-13.

```
x <- c(1,3,4,7,8,6,9,4,3,5,2)  
length(x)
```

```
[1] 11
```

```
plot(x, type = "b", col = "blue", lwd = 2)
```

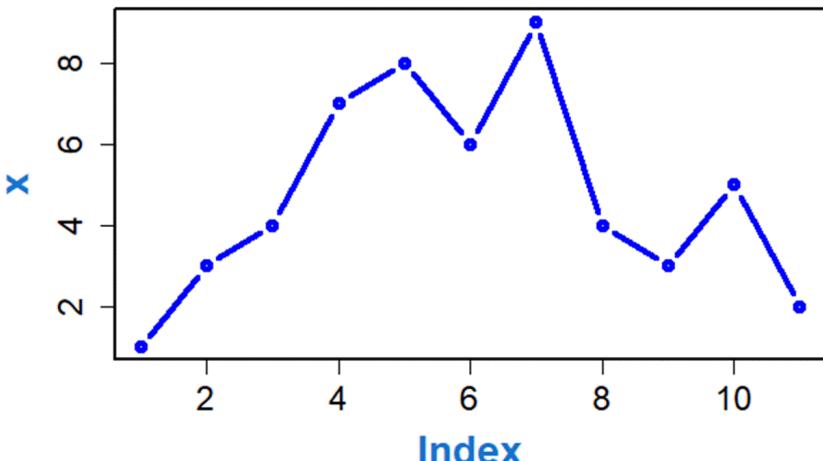


Figure 2-13. Artificial time series to demonstrate lag

To make a lag-1 plot we need to plot x_1 on the ordinate against x_2 on the abscissa, x_2 on the ordinate against x_3 on the abscissa, and so on, up to x_{10} on the ordinate against x_{11} on the abscissa. This is done by

plotting the array x_{11} (that is, the x array with the last observation removed) on the ordinate, against the array x_1 (that is, the x array with the first observation removed) on the abscissa. The call to `abline()` adds a line with 0 intercept and unit slope, while `lty=2` makes it a dashed line as seen in Figure 2-14.

```
plot(x[-1], x[-11], type = "b", col = "blue", lwd = 2)
abline(0, 1, lty = 2)
```

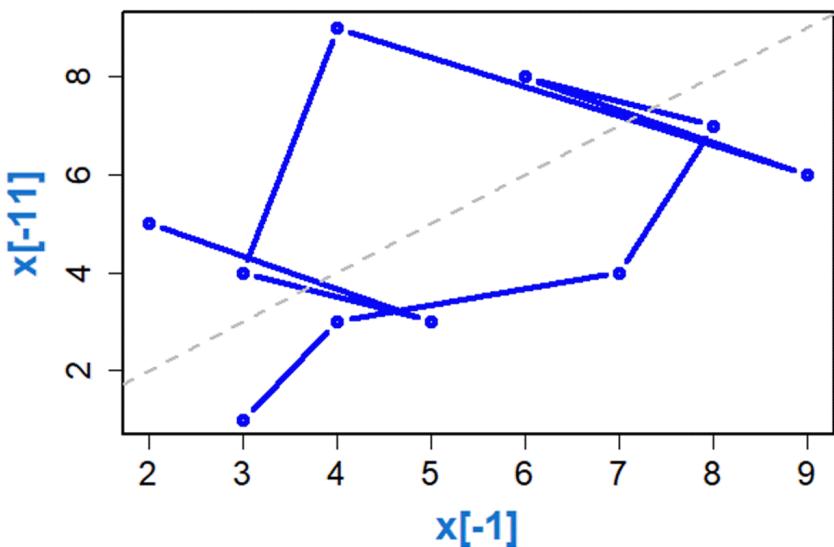


Figure 2-14. A lag plot with lag 1 created manually

If we attach the time series library, we can also use a built-in function `lag.plot()` for making lag plots. Note that `lag.plot()` has several enhancements over the home-made lag plot: better axis labels, a square plot area, a grey dashed line for the diagonal, and the serial order of the points shown explicitly on the graph. These could, of course, be done by adding various options to the `plot()` call but it is easier to use the `lag.plot()` function, as shown in Figure 2-15.

```
lag.plot(x)
```

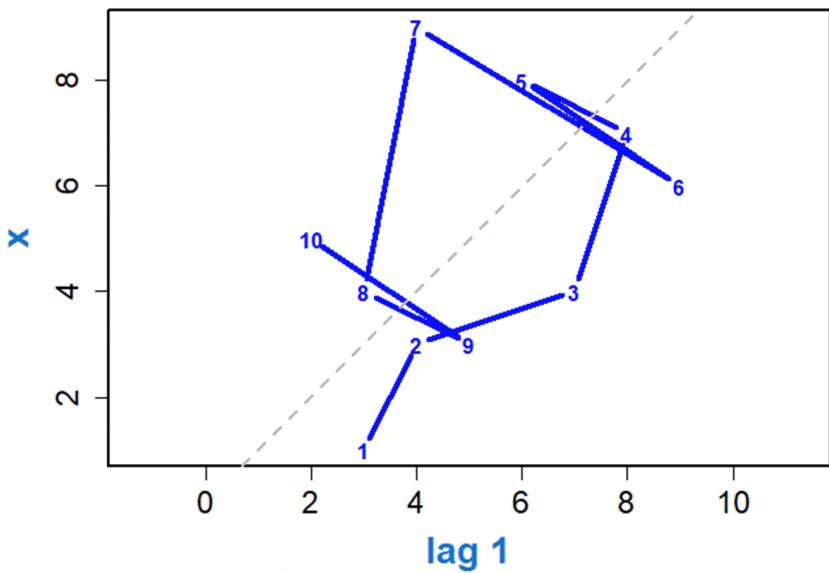


Figure 2-15. A lag plot for lag 1 using the stats-package `lag.plot()`

If the observations were made at successive times, autocorrelation is the most likely alternative. Each pair of points represents a combination of the series observations and their corresponding lagged values. As more points on the lag plot are closer to the 45-degree line, the higher the correlation will be between the series and the corresponding lag.

Figure 2-16 shows lag plots for lag 9 (or lags 1 through 9), recalling that there are 10 points (the 11th point was removed), leaving us with 2 points at lag 9. So, using lags can remove autocorrelations from the time series, resulting in independent observations.

```
lag.plot(x, 9, col = "blue", lwd = 2)
```

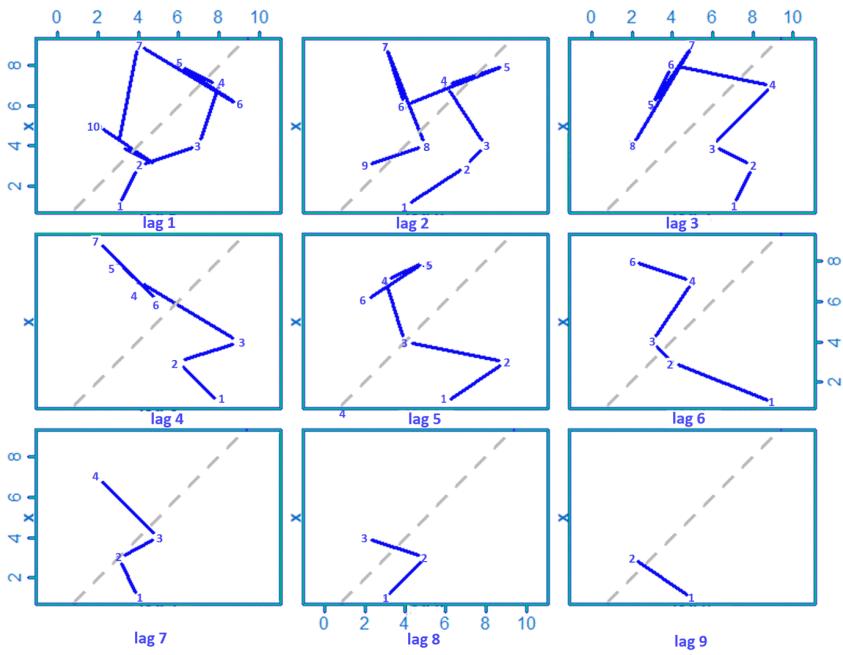


Figure 2-16. Lag plots for lags 1 through 9

Lag plots allow you to check for:

1. **Model suitability.**
2. **Outliers**_(data points with extremely high or low values).
3. **Randomness** (data without a pattern).
4. **Serial correlation** (where error terms in a time series transfer from one period to another).
5. **Seasonality**_(periodic fluctuations in time series data that happens at regular periods).

In Figure 2-17, we plot lag plots for the milk production time series in a more realistic example.

```
ts_lags(milk_mon_ts)
```

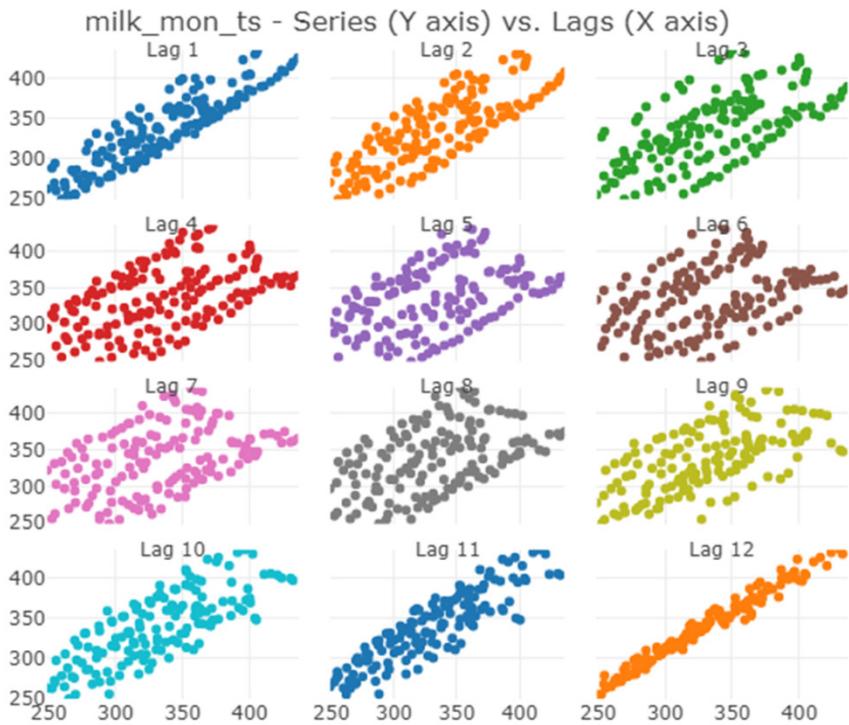


Figure 2-17. Milk production time series lags 1 though 12

At lag 12 in Figure 2-17, we are beginning to see less correlation in the time series.

Lag is essentially delay. Just as correlation shows how much two timeseries are similar, autocorrelation describes how similar the time series is with itself.

Now, consider a discrete sequence of values. For lag 1, we compare our time series with a lagged time series, that is we shift the time series by 1 before comparing it with itself. If we proceed doing this for the entire length of time series by shifting it by 1 every time, we get the autocorrelation function.

Autocorrelation

We briefly described the use of correlograms for analyzing stationarity in Chapter 2. We can also use them to analyze seasonality. After we have

applied a smoothing method, seasonal patterns of time series can be examined via **correlograms**.

Definition 2.11. The autocorrelation coefficient at lag h is given by

$$r_h = \frac{c_h}{c_0}$$

where c_h is the autocovariance function

$$c_h = \frac{1}{N} \sum_{t=1}^{N-h} (Y_t - \bar{Y})(Y_{t+h} - \bar{Y})$$

and c_0 is the variance function

$$c_0 = \frac{1}{N} \sum_{t=1}^N (Y_t - \bar{Y})^2$$

The resulting value of r_h will range between -1 and $+1$.

The correlogram (autocorrelogram) displays graphically and numerically the **autocorrelation function (ACF)**, that is, serial correlation coefficients (and their standard errors) for consecutive lags in a specified range of lags (e.g., 1 through 30). Ranges of two standard errors for each lag are usually marked in correlograms but typically the size of autocorrelation is of more interest than its reliability because we are usually interested only in very strong (and thus highly significant) autocorrelations.

The *autocorrelation function (ACF)* plot is useful for identifying non-stationary time series. For a stationary time series, the ACF will drop to zero relatively quickly, while the ACF of non-stationary data decreases slowly. The ACF of the differenced Unsecured Consumer Loan data (see Figure 2-18) looks just like that from a white noise series.

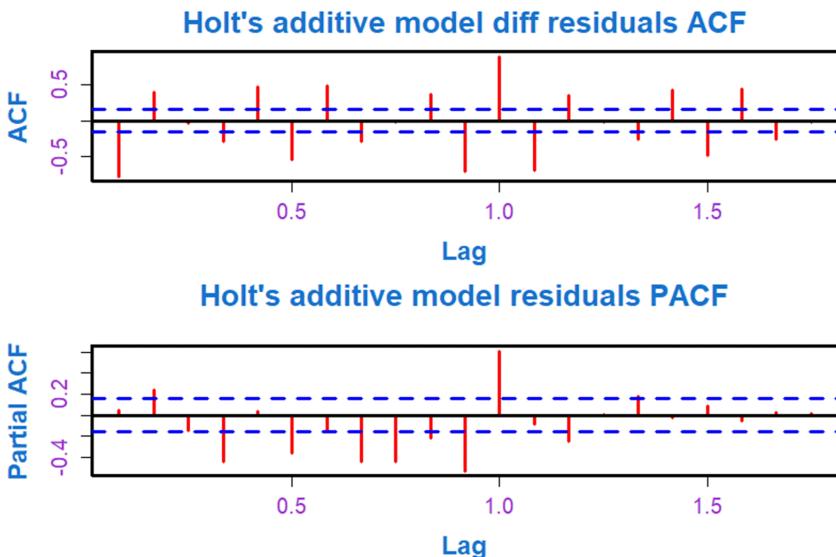


Figure 2-18. ACF and PACF of the differenced Unsecured Consumer Loan data

From the values of an autocorrelation function, we can see how much it correlates with itself. For any time series we will have perfect correlation at lag/delay = 0, since we are comparing same values with each other. As we shift our time series we begin to see the correlation values decreasing. Note that if timeseries comprises of completely random values, we will only have correlation at lag=0, and no correlation everywhere else. In most of time series this is not the case, as values tend to decrease over time, thus having some correlation at low lag values.

Now, consider a long periodic time series, for example outdoor temperature over a few years, sampled hourly. Our time series will correlate with itself on daily basis (day/night temperature drop) as well as yearly (summer/winter temperatures). Let's say our first datapoint is at 1 pm in mid-summer. Lag=1 represents one hour. The autocorrelation function at lag=1 will experience a slight decrease in correlation. At lag=12 you will have the lowest correlation of the day, after what it will begin to increase. If we move forward 6 months to 1 pm, our time series is still somewhat correlated. If we move lag to 6 months and 1 am, we

might see our lowest correlation point in the time series. At lag of 12 months our timeseries is again close to the peak value.

Lags (Lag Operator)

Figure 2-19 illustrates the lag operator (also known as backshift operator) is a function that shifts (offsets) a time series such that the lagged values are aligned with the actual time series. The lags can be shifted any number of units, which simply controls the length of the backshift. illustrates the lag operation for lags 1 and 2.

Date	Value	Value _{t-1}	Value _{t-2}
1/1/2017	200	NA	
1/2/2017	220	200	NA
1/3/2017	215	220	200
1/4/2017	230	215	220
1/5/2017	235	230	215
1/6/2017	225	235	230
1/7/2017	220	225	235
1/8/2017	225	220	225
1/9/2017	240	225	220
1/10/2017	245	240	225

Figure 2-19. Illustration of lag operations for lag 1 and lag 2.

Lags are very useful in time series analysis because of a phenomenon called autocorrelation, which is a tendency for the values within a time series to be correlated with previous copies of itself. One benefit to autocorrelation is that we can **identify patterns within the time series**, which helps in determining seasonality, the tendency for patterns to repeat at periodic frequencies. Understanding how to calculate lags and analyze autocorrelation will be the focus of this post.

Finally, lags and autocorrelation are central to numerous forecasting models that incorporate autoregression, regressing a time series using previous values of itself. Autoregression is the basis for one of the most widely used forecasting techniques, the autoregressive integrated moving average model or **ARIMA** for short. Possibly the most widely used tool for forecasting, the forecast package by Rob Hyndman, implements ARIMA (and a number of other forecast modeling techniques). We'll save autoregression and ARIMA for another day as the subject is truly fascinating and deserves its own focus.

Alternative Time Series Decomposition

An alternative for decomposing a time series is to use the *stlplus package* in R, as follows (Hafner, 2016), using the airpass dataset from the *TSA* package in R (Chan & Ripley, 2012). The *airpass* dataset contains monthly total international airline passengers from 01/1960 – 12/1971.

```
require(TSA)
require(stlplus)
data(airpass)
Air_stl <- stlplus(airpass, t = as.vector(time(airpass)),
                   n.p = 12, l.window = 13, t.window = 19, s.window = 35,
                   s.degree = 1, sub.labels = substr(month.name, 1, 3))
plot(Air_stl, ylab = 'Air Passenger Concentration (ppm)',
      xlab = 'Time (years)')
plot_seasonal(Air_stl)
plot_trend(Air_stl)
plot_cycle(Air_stl)
plot_rembycycle(Air_stl)
```

The above code produces the following charts:

1. In Figure 2-20, we have the time series decomposition by component over time, with the raw data, seasonal component, trend component, and the remainder of variation or noise.
2. In Figure 2-21, we have the plot of Centered Seasonal plus Reminder versus Time per Month (hour, day, week, as appropriate). This shows us the rate or slope of change each month as linear functions that are a subset of the timeseries.
3. In Figure 2-22, we see the Trend and Remainder over Time plot, where the remainder has been removed to show the trend.

- In Figure 2-23, we see the plot of Seasonal over Time by Month (hour, day, week, as appropriate), which is similar to Figure 2-21 but shows a range on seasonality each month.
- In Figure 2-24, we have the plot of the Remainder over Time by Month (hour, day, week, as appropriate).

We will see this time series again in Chapter 6, where we will apply an ARIMA model to it.

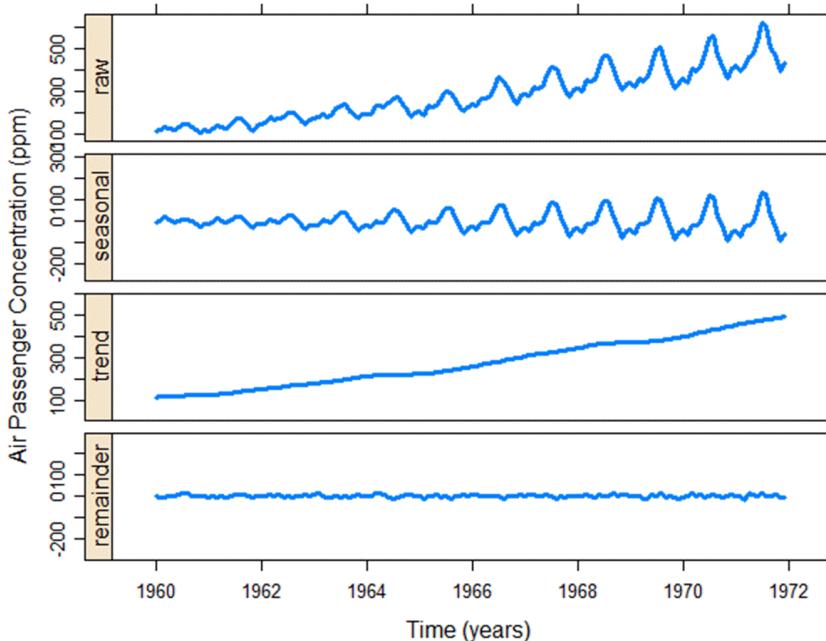


Figure 2-20. Time series decomposition by component over Time

Figure 2-21 presents a **seasonal subseries plot**. Seasonal subseries plots are a graphical tool to visualize and detect seasonality in a time series. Seasonal subseries plots involve the extraction of the seasons from a time series into a subseries. Based on a selected periodicity, it is an alternative plot that emphasizes the seasonal patterns are where the data for each season are collected together in separate mini time plots.

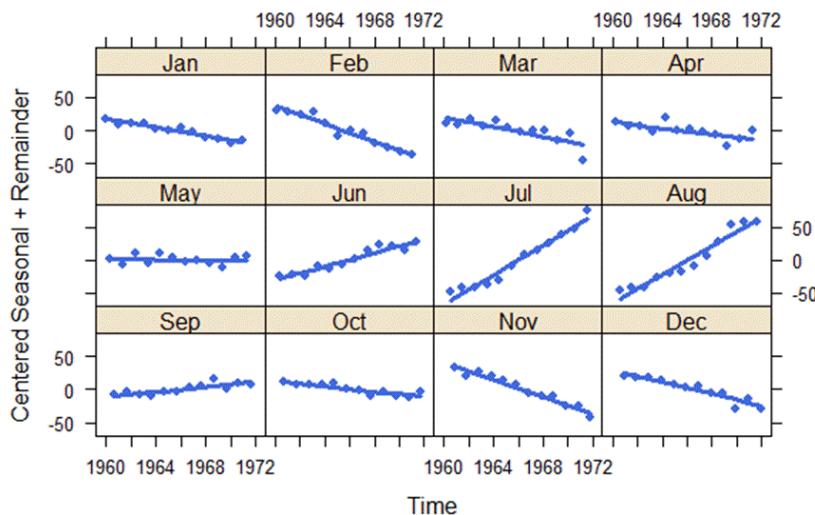


Figure 2-21. Centered Seasonal plus Reminder versus Time per Month

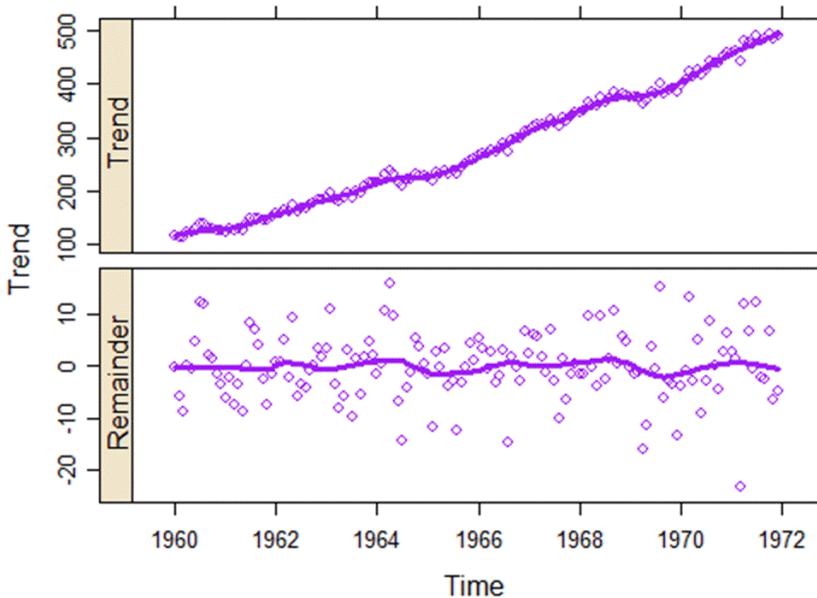


Figure 2-22. Trend and Remainder over Time

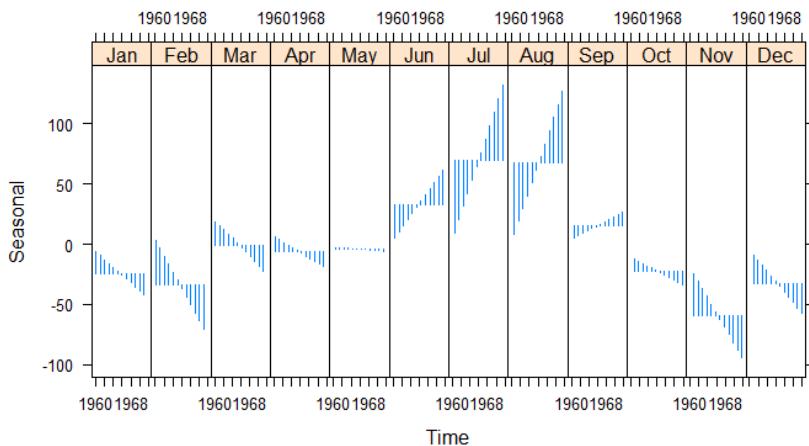


Figure 2-23. Seasonal over Time by Month

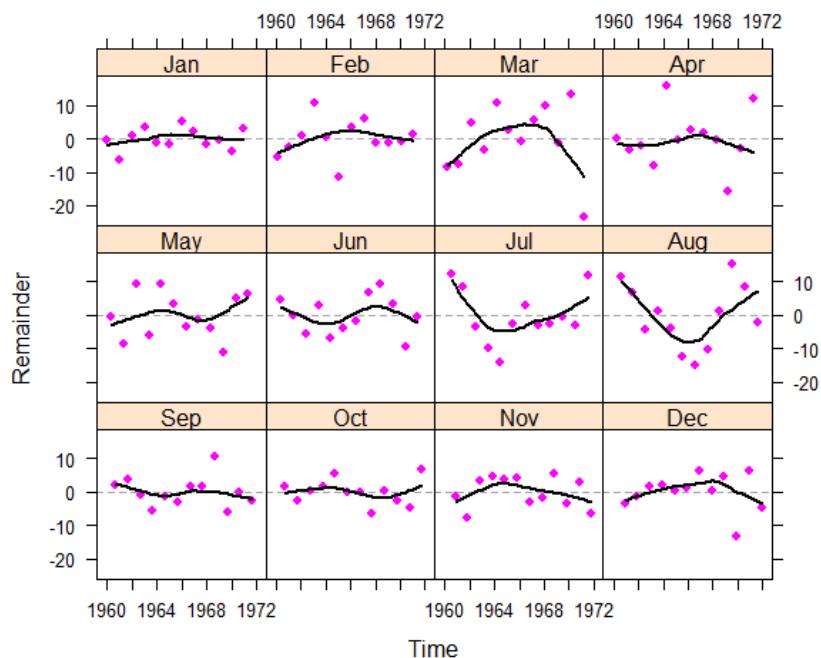


Figure 2-24. Remainder over Time by Month

Chapter Review

In this chapter, we discussed the component of time series data, including trend, seasonality, cycle, and the irregular component. The complexity of these components in times series data render this a challenging analysis method, and when all the components are present in one-time series, there is much work for us to do.

We learned about ACF plots and PACF plots and how they help us understand correlation and autocorrelation. We also discuss the important role played by lag and how to analyze it to improve our interpretation of model suitability, serial correlation and seasonality. In the next chapter, we will use the R programming language to explore how we can apply rolling or moving averages to account for these components of time series.

We also discussed the following definitions:

Definition Number	Definition Citation
Definition 2.1: Decomposition	<i>Time series data can contain multiple patterns acting at different temporal scales. Decomposition is the process of isolating each of these patterns.</i>
Definition 2.2: Trend	<i>The trend shows the general tendency of the data to increase or decrease during a long period of time. A trend is a smooth, general, long-term, average tendency. It is not always necessary that the increase or decrease is in the same direction throughout the given period of time.</i>
Definition 2.3: Seasonality	<i>In <u>time series</u> data, seasonality is the presence of variations that occur at specific regular intervals less than a year, such as weekly, monthly, or quarterly.</i>
Definition 2.4: Cycle	<i>A cycle occurs when the data exhibit rises and falls that are not of a fixed frequency. These fluctuations are usually due to economic conditions, and are often related to the "business cycle". The duration of these fluctuations is usually at least 2 years.</i>
Definition 2.5: Irregular Component	<i>The irregular component (or "noise") at time t, or I_t, describes random, irregular influences. It represents the residuals or remainder of the time series after the other components have been removed.</i>

Definition 2.6: Autoregressive Component	An autoregressive component is a special case of cycle that corresponds to the frequency of zero or π , given by $AR_t = \rho AR_{t-1} + v_t, v_t \sim i.i.d. N(0, \sigma_v^2),$ where the AR_t follows a first-order autoregression with $-1 < \rho < 1$, and they represent the “momentum” of the time series as it relates to its past observations.
Definition 2.7 Regressive Component	The regressive component $\sum_{j=1}^m (B_j x_{jt} + e_t)$ includes those terms that we are accustomed to in linear regression and are not unique to time series data.
Definition 2.8 Stationary Time Series	A stationary time series is one whose properties do not depend on the time at which the series is observed. That is y_t does not depend on y_{t-1}, \dots, y_{t-n} .
Definition 2.9 LOESS Regression	Loess regression is a nonparametric technique that uses local weighted regression to fit a smooth curve through points in a scatter plot. Loess curves are can reveal trends and cycles in data that might be difficult to model with a parametric curve.
Definition 2.10 Lag	A lag is a fixed amount of passing time, such that the k^{th} lag (lag_k) is the time period that happened k time points before time i : $\text{lag}_k(Y_i) = Y_{i-k}$ The most commonly used lag is 1, called a first-order lag.
Definition 2.11 Autocorrelation Coefficient at lag h	The autocorrelation coefficient at lag h is given by $r_h = \frac{c_h}{c_0}$ where c_h is the autocovariance function $c_h = \frac{1}{N} \sum_{t=1}^{N-h} (Y_t - \bar{Y})(Y_{t+h} - \bar{Y})$ and c_0 is the variance function $c_0 = \frac{1}{N} \sum_{t=1}^N (Y_t - \bar{Y})^2$ The resulting value of r_h will range between -1 and +1.

Review Exercises

1. From the fma-Package in R, load the *copper* dataset
 - a. Describe the dataset (refer to the R documentation)
 - b. Set up *copper* as a time series
 - c. Decompose the *copper* time series: generate a decomposition plot like Figure 2-9
 - d. Describe and explore the components of the plot you just created (part c).
 - e. Make a month plot like Figure 2-10. Does it provide additional insight? Discuss.
 - f. Make a season plot like Figure 2-11. Does it provide additional insight? Discuss.
 - g. Generate a lag plot like Figure and check for the following:
 - i. Model suitability.
 - ii. Outliers (data points with extremely high or low values).
 - iii. Randomness (data without a pattern).
 - iv. Serial correlation (where error terms in a time series transfer from one period to another).
 - v. Seasonality (periodic fluctuations in time series data that happens at regular periods).
2. Repeat the milk production example using the daily data.

Chapter 3 – Moving Averages for Time Series

Systematic Pattern and Random Noise

In Chapter 2, we discussed the components of time series data, namely trend, seasonality and cycles. In the next two chapters, starting with this one, we will discuss ways of extracting the trend and seasonality components using moving averages. We will continue to solve time series issues using exponential smoothing in Chapter 4 and differencing in Chapter 6. While moving averages is technically a “smoothing” method, we will treat it as such but reserve more complex smoothing methods, like Holt-Winters, for Chapter 6.

Two General Aspects of Time Series Patterns

Though we have discussed other components of time series data, we can describe most time series patterns in terms of two basic classes of components: **trend and seasonality**. The first represents a general systematic linear or a nonlinear component that changes over time and does not repeat, or, at least, does not repeat within the time range captured by our data (e.g., a plateau followed by a period of exponential growth). The second may have a similar structure (e.g., a plateau followed by a period of exponential growth, but repeats itself in systematic intervals over time. These two general classes of time series components may coexist in real-life data (Dell, 2015). As an example, consider the sales of a company, which can grow rapidly over the years, but they still follow consistent seasonal patterns (e.g., as much as 25% of yearly sales each year are made in December, whereas only 4% in August).

This general pattern is well illustrated in Figure 3-1 by the international passenger data series (G), as mentioned in the textbook *Time Series: Forecast and Control* by Box (Box, Jenkins, & Reinsel, 2008), representing monthly international airline passenger totals (measured in thousands) for twelve consecutive years from 1960 to 1972.

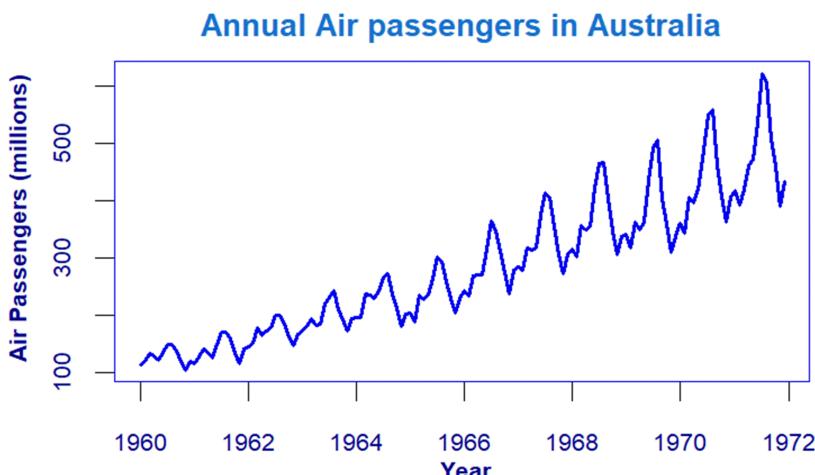


Figure 3-1. International passenger data series (G)

If you plot the successive observations (months) of airline passenger totals, a clear and almost linear trend emerges, indicating that the airline industry enjoyed steady growth over the years (approximately four times more passengers traveled in 1970 than in 1960). At the same time, the monthly figures will follow an almost identical pattern each year (e.g., more people travel during holidays than during any other time of the year). This example data file also illustrates a very common general type of pattern in time series data, where the amplitude of the seasonal changes increases with the overall trend (i.e., the variance is correlated with the mean over the segments of the series). This pattern which is called *multiplicative seasonality* indicates that the relative amplitude of seasonal changes is constant over time, so it is related to the trend.

Trend Analysis

There are no proven “automatic” techniques to identify trend components in the time series data; however, as long as the trend is monotonous (consistently increasing or decreasing) that part of data analysis is typically not very difficult. If the time series data contain considerable error, then the first step in the process of trend identification is smoothing.

In Chapter 2, we got a taste of smoothing when we applied **loess** to the dairy milk problem. **Smoothing** always involves some form of **local**

averaging of data. The term **filter** is sometimes used to describe a smoothing procedure. In R, we use the `filter()` function for smoothing with moving averages. For instance, if the smoothed value for a particular time is calculated as a linear combination of observations for surrounding times, it might be said that we have applied a linear filter to the data. That is not to say that we have fitted the data with a straight line. We merely “averaged out” some of the random noise to more clearly see trends or seasonality.

The classical method of time series decomposition originated in the 1920s and was widely used until the 1950s. It still forms the basis of later time series methods, and so it is important to understand how it works. The first step in a classical decomposition is to use a moving average method to estimate the trend-cycle, so we begin by discussing moving averages.

Moving Averages

The most common technique is **moving average** smoothing which replaces each element of the series by either the simple or weighted average of n surrounding elements, where n is the width of the smoothing "window" (Box & Jenkins, 1976) (Velleman & Hoaglin, 1981).

Formally, a moving average takes a noisy time series and replaces each value with the average value of a neighborhood about the given value. This neighborhood may consist of purely historical data, or it may be centered about the given value. Furthermore, the values in the neighborhood may be weighted using different sets of weights.

Definition 3.1. A **moving average** of order m can be written as

$$\hat{T}_t = \frac{1}{m} \sum_{j=-k}^k y_{t+j},$$

Where $m = 2k + 1$. That is, the estimate of the trend-cycle at time t is obtained by averaging values of the time series within k periods of t .

Observations that are nearby in time are also likely to be close in value, and the average eliminates some of the randomness in the data, leaving

a smooth trend-cycle component. We call this an m -MA meaning a moving average of order m .

Here is an example of an equally weighted three-point moving average, using historical data,

$$T_0 = y_0 T_1 = \frac{y_0 + y_1}{2} T_t = \frac{y_{t-2} + y_{t-1} + y_t}{3} \quad (1)$$

Here, T_t represents the smoothed signal, and y_t represents the noisy time series.

In contrast, to simple moving averages, an exponentially weighted moving average (EWMA) adjusts a value according to an exponentially weighted sum of all previous values. This is the basic idea,

$$T_0 = y_0 T_1 = \alpha y_t + (1 - \alpha)T_{t-1}, \text{ for } t > 0 \quad (2)$$

This is nice because you don't have to worry about having a three-point window, versus a five-point window, or worry about the appropriateness of your weighting scheme. With the EWMA, previous perturbations "remembered," and "slowly forgotten," by the T_{t-1} term in the last equation, whereas with a window or neighborhood with discrete boundaries, a perturbation is forgotten as soon as it passes out of the window.

Moving Averages using movavg

The *pracma* package has several types of available moving averages that we can implement using `movavg()`:

- **s** for "simple", it computes the simple moving average. n indicates the number of previous data points used with the current data point when calculating the moving average.
- **t** for "triangular", it computes the triangular moving average by calculating the first simple moving average with window width of $\text{ceil}(n + 1)/2$; then it calculates a second simple moving average on the first moving average with the same window size.
- **w** for "weighted", it calculates the weighted moving average by supplying weights for each element in the moving window. Here the reduction of weights follows a linear trend.

- **m** for "modified", it calculates the modified moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new value and subtracting the last average from the resulting sum.
- **e** for "exponential", it computes the exponentially weighted moving average. The exponential moving average is a weighted moving average that reduces influences by applying more weight to recent data points () reduction factor $2/(n + 1)$; or
- **r** for "running", this is an exponential moving average with a reduction factor of $1/n$ [same as the modified average?].

We will take a different approach to load our data. We are using ABB stock prices as our data. ABB has four customer-focused, globally leading businesses: Electrification, Industrial Automation, Motion, and Robotics & Discrete Automation, supported by its ABB Ability™ digital platform. We are going to enter stock prices by scanning them from text to our time series, `abbshare`. The data can be found in the `pracma` package documentation.

```
library(pracma)
abbshares <- scan(text =
25.69 25.89 25.86 26.08 26.41 26.90 26.27 26.45 26.49 26.08
26.11 25.57 26.02 25.53 25.27 25.95 25.19 24.78 24.96 24.63
25.68 25.24 24.87 24.71 25.01 25.06 25.62 25.95 26.08 26.25
25.91 26.61 26.34 25.55 25.36 26.10 25.63 25.52 24.74 25.00
25.38 25.01 24.57 24.95 24.89 24.13 23.83 23.94 23.74 23.12
23.13 21.05 21.59 19.59 21.88 20.59 21.59 21.86 22.04 21.48
21.37 19.94 19.49 19.46 20.34 20.59 19.96 20.18 20.74 20.83
21.27 21.19 20.27 18.83 19.46 18.90 18.09 17.99 18.03 18.50
19.11 18.94 18.21 18.06 17.66 16.77 16.77 17.10 17.62 17.22
17.95 17.08 16.42 16.71 17.06 17.75 17.65 18.90 18.80 19.54
19.23 19.48 18.98 19.28 18.49 18.49 19.08 19.63 19.40 19.59
20.37 19.95 18.81 18.10 18.32 19.02 18.78 18.68 19.12 17.79
18.10 18.64 18.28 18.61 18.20 17.82 17.76 17.26 17.08 16.70
16.68 17.68 17.70 18.97 18.68 18.63 18.80 18.81 19.03 18.26
18.78 18.33 17.97 17.60 17.72 17.79 17.74 18.37 18.24 18.47
18.75 18.66 18.51 18.71 18.83 19.82 19.71 19.64 19.24 19.60
19.77 19.86 20.23 19.93 20.33 20.98 21.40 21.14 21.38 20.89
21.08 21.30 21.24 20.55 20.83 21.57 21.67 21.91 21.66 21.53
21.63 21.83 21.48 21.71 21.44 21.67 21.10 21.03 20.83 20.76
20.90 20.92 20.80 20.89 20.49 20.70 20.60 20.39 19.45 19.82
20.28 20.24 20.30 20.66 20.66 21.00 20.88 20.99 20.61 20.45
20.09 20.34 20.61 20.29 20.20 20.00 20.41 20.70 20.43 19.98
19.92 19.77 19.23 19.55 19.93 19.35 19.66 20.27 20.10 20.09
20.48 19.86 20.22 19.35 19.08 18.81 18.87 18.26 18.27 17.91
```

```
17.68 17.73 17.56 17.20 17.14 16.84 16.47 16.45 16.25 16.07")
class(abbshares)
[1] "numeric"
```

Now that we have our data, we will plot it using the `plot()` function preceded by some plotting parameters that we add using the `par()` function. The parameters we want to add are background color (`bg`), font size relative to the plot (`cex`), and plot margins (`mar`).

We will also “embed” our moving averages into the plot using the `lines()` function. That is, we will define the moving average and then call it using `lines()`. We will also generate a legend to help distinguish between the different average types. The completed plot is shown in Figure 3-2

```
par(bg = 'ivory', cex = 1.2, mar = c(4, 3, 3,.15))
plot(abbshares, type = "l", lwd = 2, col = 1, ylim = c(15, 30),
     main = "Types of moving averages", xaxt='n', yaxt='n')
mtext(side = 1, text = "Days", line = 2,
      col = 'deepskyblue3', cex = 1.2)
mtext(side = 2, text = "ABB Shares Price (in USD)",
      col = 'deepskyblue3', line = 2.5, cex = 1.2)
y <- movavg(abbshares, 50, "s"); lines(y, col = 'red2', lwd=2)
y <- movavg(abbshares, 50, "t"); lines(y, col = 'green3', lwd=2)
y <- movavg(abbshares, 50, "w"); lines(y, col = 'blue3', lwd=2)
y <- movavg(abbshares, 50, "m"); lines(y, col = 'cyan3', lwd=2)
y <- movavg(abbshares, 50, "e"); lines(y, col = 'maroon', lwd=2)
y <- movavg(abbshares, 50, "r"); lines(y, col = 'orange', lwd=2)
grid()
legend(120, 29, c("original data", "simple", "triangular",
                 "weighted", "modified", "exponential", "running"),
       col = c('black', 'red2', 'green', 'blue3', 'cyan3',
               'maroon', 'orange'),
       lty = 1, lwd = 2, box.col = 'gray', bg = 'white', bty='n'
     )
```

Types of moving averages

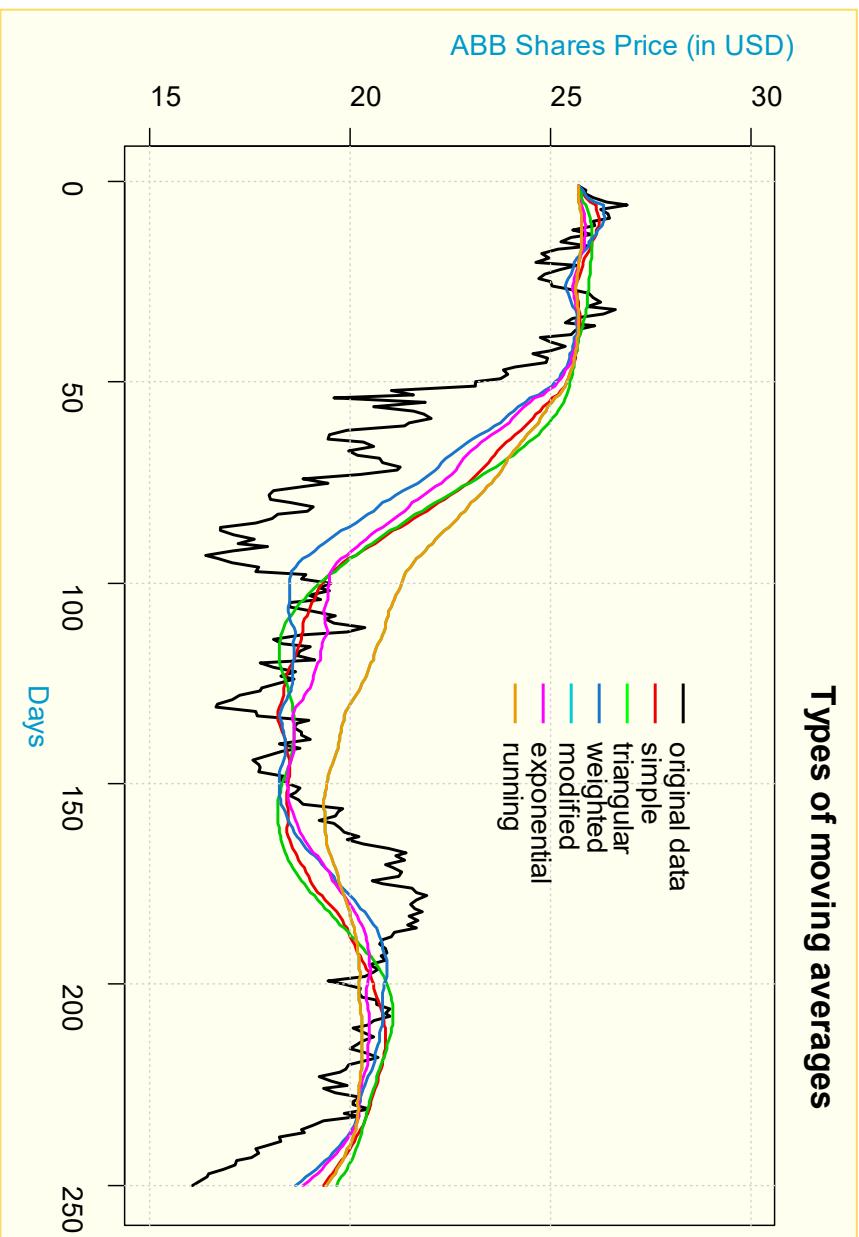


Figure 3-2. Time series plot of ABB stock prices using “movavg” and six different moving average types

Moving Averages using Filters

For quarterly data, we could define a smoothed value for time t as $x_t + x_{t-1} + x_{t-2} + x_{t-3}/4$, the average of this time and the previous 3 quarters, as in Figure 3-3. In R code this will be a one-sided filter.

```
trendpattern = stats::filter(milk_mon_ts,
  c(1/4, 1/4, 1/4, 1/4), sides = 1)
plot(trendpattern, type = "b", col = "blue",
  main = "Quarterly moving average annual trend")
lines(trendpattern, col = "blue", lwd = 2)
```

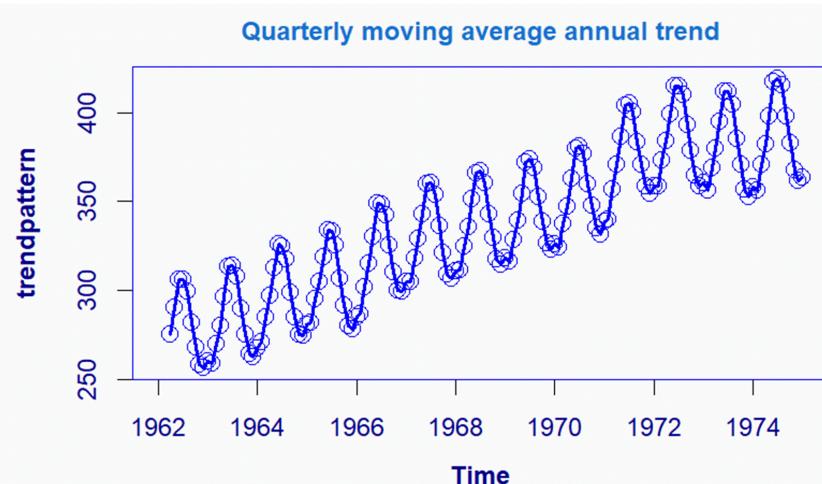


Figure 3-3. Milk TS with moving average smoothing with $n = 4$ (quarters).

Figure 3-4 shows a two-sided moving average for monthly data $n = 12$ surrounding elements with $1/12 + 1/12 + 1/12 + 1/12 + 1/12 + 1/6 + 1/6 + 1/12 + 1/12 + 1/12 + 1/12$ (note that $1/6 + 1/6 = 1/12+1/12+1/12+1/12$).

```
trendpattern = stats::filter(milk_mon_ts,
  c(1/12, 1/12, 1/12, 1/12, 1/6, 1/6, 1/12, 1/12, 1/12, 1/12, 1/12),
  sides = 2)
plot(trendpattern, type = "b", col = 'blue',
  main = "Monthly moving average annual trend")
lines(trendpattern, col = 'blue', lwd = 2)
```

Medians can be used instead of means. The main advantage of median as compared to moving average smoothing is that its results are less biased by outliers (within the smoothing window). Thus, if there are outliers in the data (e.g., due to measurement errors), median

smoothing typically produces smoother or at least more “reliable” curves than moving average based on the same window width.

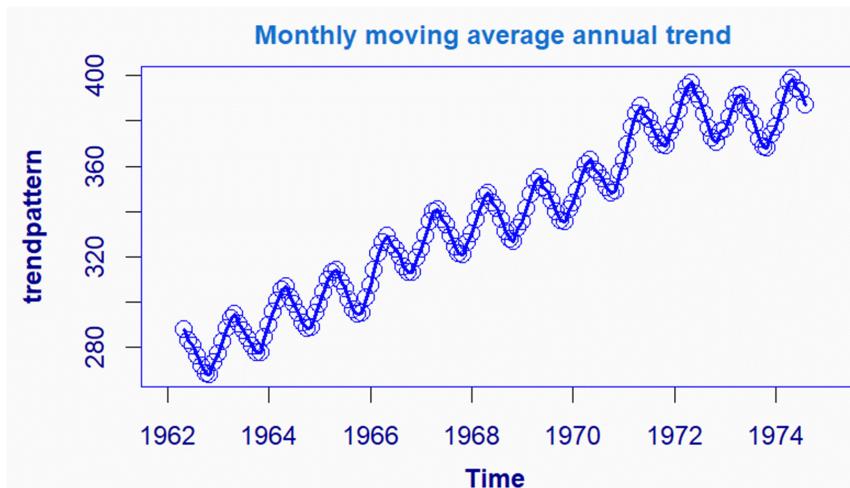


Figure 3-4. Milk TS with moving average smoothing with $n = 12$ (months)

The main disadvantage of median smoothing is that in the absence of clear outliers it may produce more “jagged” curves than moving average and it does not allow for weighting.

Moving Averages using fpp::ma

For example, consider the volume of electricity sold to residential customers in South Australia each year from 1989 to 2008 (hot water sales have been excluded). The `elecsales` dataset are part of the **fpp** package in R (Hyndman, fpp: Data for "Forecasting: principles and practice", 2013). Annual electricity sales for South Australia in GWh.

```
library(fpp)
data(elecsales)
elecsales
```

```
Time Series:
Start = 1989
End = 2008
Frequency = 1
[1] 2354.34 2379.71 2318.52 2468.99 2386.09 2569.47 2575.72
[8] 2762.72 2844.50 3000.70 3108.10 3357.50 3075.70 3180.60
[15] 3221.60 3176.20 3430.60 3527.48 3637.89 3655.00
```

We now construct a moving average for Residential electricity sales and create a plot of with moving average overlaid (see Figure 3-5).

```
elecsales.ma<-ma(elecsales, order = 5)
plot(elecsales, main = "Residential electricity sales",
     ylab = 'GWh',
     xlab = 'Year', col = 'magenta', lwd = 2)
lines(elecsales.ma, col = 'royalblue', lwd = 2)
```

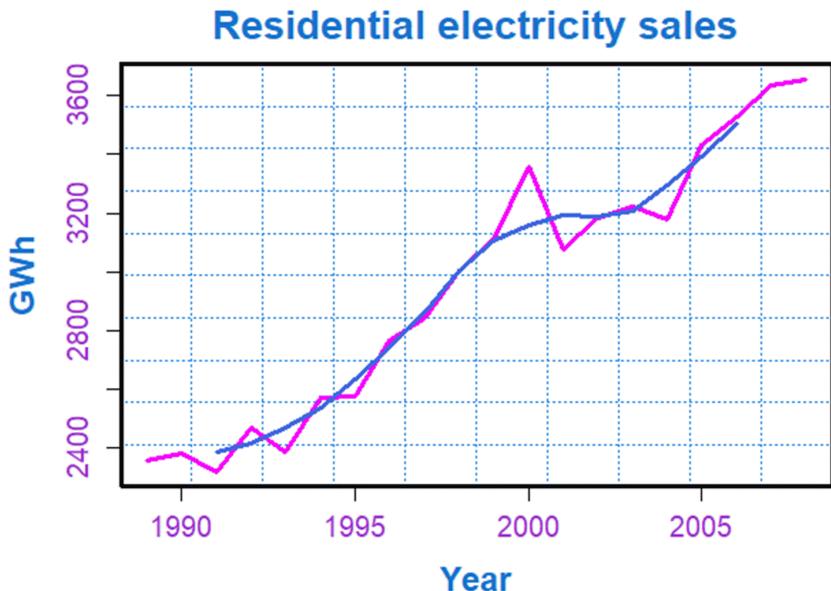


Figure 3-5. Residential electricity sales with moving average overlaid

Notice how the trend (in red) is smoother than the original data and captures the main movement of the time series without all the minor fluctuations. The moving average method does not allow estimates of T_t where t is close to the ends of the series; hence the red line does not extend to the edges of the graph on either side. Later we will use more sophisticated methods of trend-cycle estimation which do allow estimates near the endpoints.

The order of the moving average determines the smoothness of the trend-cycle estimate. In general, a larger order means a smoother curve. The following code and graphs show the effect of changing the order of the moving average for the residential electricity sales data. The plots of

these are shown in Figure 3-6. Note that the code for the first time series and moving average plot is annotated. The other are similar.

```
# Moving averages and their plot code

elecsales.ma3 <- ma(elecsales, order = 4)
elecsales.ma5 <- ma(elecsales, order = 6)
elecsales.ma7 <- ma(elecsales, order = 8)
elecsales.ma9 <- ma(elecsales, order = 10)

old.par <- par(mfrow = c(1, 1), mar = c(0,0,0,0))
# default values for plot window and margins

new.par <- par(mfrow = c(2, 2)) # universal graphics parameter
# for 2 rows with 2 columns of plots (4 plots)

# Moving average 1 (ma3) plot code:
plot(elecsales,
      main = 'Residential electricity sales', # main title
      ylab = 'GWh',    # y-axis label
      xlab = 'Year',   # x-axis label
      col = 'blue',   # series line color
      lwd = 2)        # series line width

lines(elecsales.ma3, col='red', lwd = 2)
# moving avg line color and width

legend('bottomright','MA-3', pch = 1)
# moving avg 3 legend symbol (pch) is
# type 1 = solid line

# Moving average 2 (ma5) plot code:
plot(elecsales, main = 'Residential electricity sales',
      ylab = 'GWh', xlab = 'Year', col = 'blue', lwd = 2)
lines(elecsales.ma5, col = 'red', lwd = 2)
legend('bottomright', 'MA-5', pch = 1)

# Moving average 3 (ma7) plot code:
plot(elecsales, main = 'Residential electricity sales',
      ylab = 'GWh', xlab = 'Year', col = 'blue', lwd = 2)
lines(elecsales.ma7, col = "red", lwd = 2)
legend('bottomright', 'MA-7', pch = 1)

# Moving average 4 (ma9) plot code:
plot(elecsales, main = 'Residential electricity sales',
      ylab = 'GWh', xlab = 'Year', col = 'blue', lwd = 2)
lines(elecsales.ma9, col = "red", lwd = 2)
legend('bottomright','MA-9', pch = 1)
```

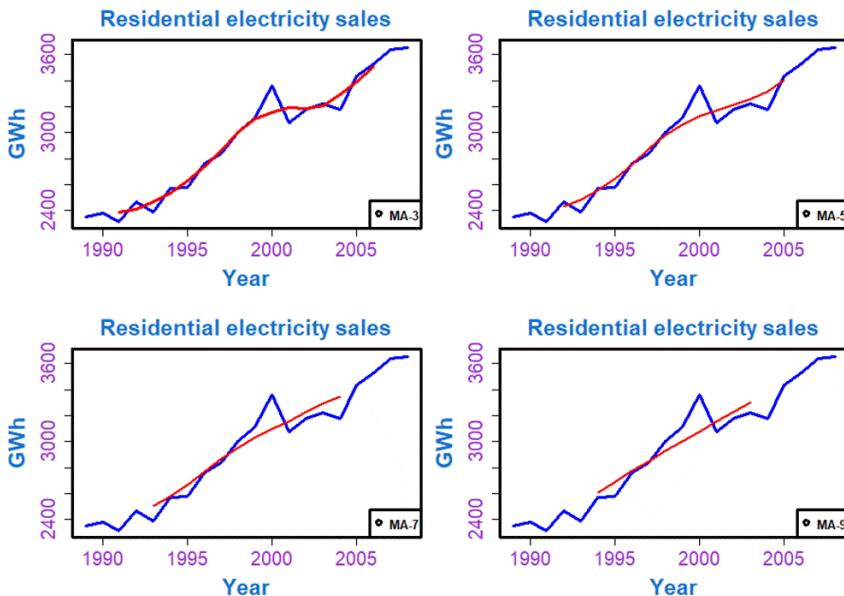


Figure 3-6. Residential electricity sales with moving averages overlaid for orders 3, 5, 7, and 9

Simple moving averages such as these are usually of odd order (e.g., 3, 5, 7, etc.) This is so they are symmetric: in a moving average of order $m = 2k + 1$, there are k earlier observations, k later observations and the middle observation that are averaged. But if m is even, it is no longer symmetric.

Moving averages of moving averages

It is possible to apply a moving average to a moving average. One reason for doing this is to make an even-order moving average symmetric.

```
elecsales2 <- window(elecsales, start = 1989)
# First moving average:
ma4 <- ma(elecsales2 , order = 4, centre = FALSE)
# Second moving average:
ma4x4 <- ma(elecsales2 , order = 4, centre = TRUE)
# Moving average of two moving averages:
tselecsales<-array(c(elecsales, ma4, ma4x4), c(19, 3))
tselecsales # prints results
```

	[,1]	[,2]	[,3]
[1,]	2354.34	3655.000	NA
[2,]	2379.71	NA	NA
[3,]	2318.52	NA	NA

```
[4,] 2468.99 2384.359      NA
[5,] 2386.09 2412.047 2384.359
[6,] 2569.47 2467.917 2412.047
[7,] 2575.72 2536.784 2467.917
[8,] 2762.72 2630.801 2536.784
[9,] 2844.50 2742.006 2630.801
[10,] 3000.70 2862.457 2742.006
[11,] 3108.10 3003.352 2862.457
[12,] 3357.50 3106.600 3003.352
[13,] 3075.70 3157.988 3106.600
[14,] 3180.60 3194.662 3157.988
[15,] 3221.60 3186.188 3194.662
[16,] 3176.20 3207.887 3186.188
[17,] 3430.60 3295.610 3207.887
[18,] 3527.48 3391.006 3295.610
[19,] 3637.89 3502.892 3391.006
```

We now create a plot of the time series with the moving average overlaid (see Figure 3-7)

```
plot(elecsales, main = 'Residential electricity sales',
     ylab = 'GWh', xlab = 'Year' , col = 'blue', lwd = 2)
lines(ma2x4, col = 'red', lwd = 2)
```

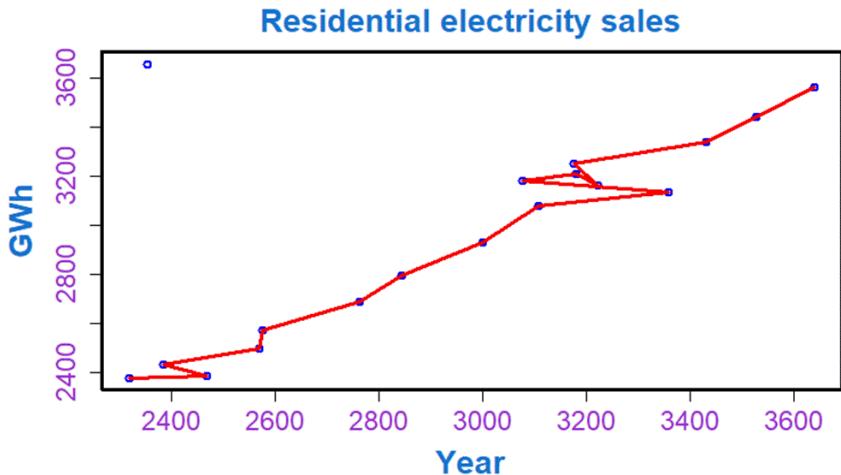


Figure 3-7. Plot of the time series with the moving average overlaid

The notation “2×4-MA” in the last column means a 4-MA followed by a 2-MA. The values in the last column are obtained by taking a moving

average of order 2 of the values in the previous column. For example, the first two values in the 4-MA column are:

$$451.2 = (2379.71 + 2318.52 + 2468.99 + 2386.09)/4 \text{ and}$$

$$2435.77 = (2318.52 + 2468.99 + 2386.09 + 2569.47)/4.$$

The first value in the 2×4 -MA column is the average of these two: $2412.048 = (2388.3275 + 2435.76758)/2$. When a 2-MA follows a moving average of even order (such as 4), it is called a “centered moving average of order 4.” This is because the results are now symmetric. To see that this is the case, we can write the 2×4 -MA as follows:

$$\begin{aligned}\hat{T}_t &= 12[14(y_{t-2} + y_{t-1} + y_t + y_{t+1}) \\ &\quad + 14(y_{t-1} + y_t + y_{t+1} + y_{t+2})] \\ &= 18y_{t-2} + 14y_{t-1} + 14y_t + 14y_{t+1} + 18y_{t+2}.\end{aligned}$$

It is now a weighted average of observations, but it is symmetric. Other combinations of moving averages are also possible. For example, a 3×3 -MA is often used, and consists of a moving average of order 3 followed by another moving average of order 3. In general, an even order MA should be followed by an even order MA to make it symmetric. Similarly, an odd order MA should be followed by an odd order MA.

Estimating the trend-cycle with seasonal data

The most common use of centered moving averages is in estimating the trend-cycle from seasonal data. Consider the 2×4 -MA:

$$\hat{T}_t = 18y_{t-2} + 14y_{t-1} + 14y_t + 14y_{t+1} + 18y_{t+2}.$$

When applied to quarterly data, each quarter of the year is given equal weight as the first and last terms apply to the same quarter in consecutive years. Consequently, the seasonal variation will be averaged out and the resulting values of \hat{T}_t will have little or no seasonal variation remaining. A similar effect would be obtained using a 2×8 -MA or a 2×12 -MA. In general, a $2 \times m$ -MA is equivalent to a weighted moving average of order $m + 1$ with all observations taking weight $1/m$ except for the first and last terms which take weights $1/(2m)$. So if the seasonal period is even and of order m , use a $2 \times m$ -MA to estimate the trend-

cycle. If the seasonal period is odd and of order m , use a m -MA to estimate the trend cycle. In particular, a 2×12 -MA can be used to estimate the trend-cycle of monthly data and a 7-MA can be used to estimate the trend-cycle of daily data. Other choices for the order of the MA will usually result in trend-cycle estimates being contaminated by the seasonality in the data.

Electrical equipment manufacturing

The *fpp* package also contains the *elecequip* dataset, representing the manufacture of electrical equipment: computer, electronic and optical products. The data adjusted by working days covering the European area (16 countries) and an industry new orders index: 2005 = 100.

```
data(elecequip)
elecequip
```

	Jan	Feb	Mar	Apr	May	Jun	
1996	79.43	75.86	86.40	72.67	74.93	83.88	
1997	78.72	77.49	89.94	81.35	78.76	89.59	
1998	81.97	85.26	93.09	81.19	85.74	91.24	
1999	81.68	81.68	91.35	79.55	87.08	96.71	
2000	95.42	98.49	116.37	101.09	104.20	114.79	
2001	100.69	102.99	119.21	92.56	98.86	111.26	
2002	89.66	89.23	104.36	87.17	89.43	102.25	
2003	89.45	86.87	98.94	85.62	85.31	101.22	
2004	89.93	92.73	105.22	91.56	92.60	104.46	
2005	91.73	90.45	105.56	92.15	91.23	108.95	
2006	99.09	99.73	116.05	103.51	102.99	119.45	
2007	103.92	103.97	125.63	104.69	108.36	123.09	
2008	109.35	105.64	121.30	108.62	103.13	117.84	
2009	77.33	75.01	86.31	74.09	74.09	85.58	
2010	79.16	78.40	94.32	84.45	84.92	103.18	
2011	91.47	87.66	103.33	88.56	92.32	102.21	
	Jul	Aug	Sep	Oct	Nov	Dec	
1996	79.88	62.47	85.50	83.19	84.29	89.79	
1997	83.75	69.87	91.18	89.52	91.12	92.97	
1998	83.56	66.45	93.45	86.03	86.91	93.42	
1999	98.10	79.22	103.68	101.00	99.52	111.94	
2000	107.75	96.23	123.65	116.24	117.00	128.75	
2001	96.25	79.81	102.18	96.28	101.38	109.97	
2002	88.26	75.73	99.60	96.57	96.22	101.12	
2003	91.93	77.01	104.50	99.83	101.10	109.16	
2004	96.28	79.61	105.55	99.15	99.81	113.72	
2005	99.33	83.30	110.85	104.99	107.10	114.38	
2006	107.98	90.50	121.85	117.12	113.66	120.35	
2007	108.88	93.98	121.94	116.79	115.78	127.28	

2008	103.62	89.22	109.41	103.93	100.07	101.15
2009	79.84	65.24	87.92	84.45	87.93	102.42
2010	89.42	77.66	95.68	94.03	100.99	101.26
2011	92.80	76.44	94.00	91.67	91.98	

Figure 3-8 shows a 2×12 -MA applied to the electrical equipment orders index. Notice that the smooth line shows no seasonality; it is almost the same as the trend-cycle shown in Figure 5.4 which was estimated using a much more sophisticated method than moving averages. Any other choice for the order of the moving average (except for 24, 36, etc.) would have resulted in a smooth line that shows some seasonal fluctuations.

```
plot(elecequip, ylab = 'New orders index', col = 'gray', lwd = 2,
     main='Electrical equipment manufacturing (Euro area)')
lines(ma(elecequip, order=12), col='red', lwd=2)
```

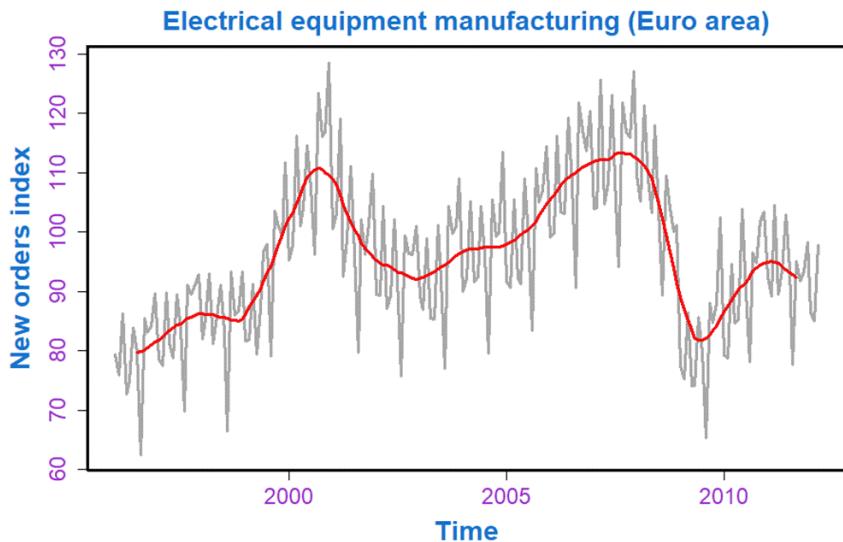


Figure 3-8. Trend-cycle for the manufacture of electrical equipment

Weighted moving averages

Combinations of moving averages result in weighted moving averages. For example, the 2×4 -MA discussed above is equivalent to a weighted 5-MA with weights given by [18, 14, 14, 14, 18].

Definition 3.2. In general, a **weighted m-MA** can be written as

$$\hat{T}_t = \sum_{j=-k}^k a_j y_{t+j},$$

Where $k = (m - 1)/2$ and the weights are given by $[a_{-k}, \dots, a_k]$.

It is important that the weights all sum to one and that they are symmetric so that $a_j = a_{-j}$. The simple m -MA is a special case where all the weights are equal to $1/m$. 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.

COVID-19

Rolling or moving averages are a way to reduce noise and smooth time series data. During the Covid-19 pandemic, rolling averages have been used by researchers and journalists around the world to understand and visualize cases and deaths. This example will cover how to compute and visualize rolling averages for the new confirmed cases and deaths from Covid-19 in the United States.

The Johns Hopkins COVID data

The data we use for demonstrating moving averages is State-level Johns Hopkins (JH) COVID data from the Center for Systems Science and Engineering at the Johns Hopkins Whiting School of Engineering. The JH COVID-19 data includes March through June 2021. We also use state-level data from the Colorado Department of Health and Environment (CDHE) COVID data. The CDHE data includes more months, March to October 2020.

We will develop a data frame that has the following columns.

- `state` – US state
- `state_abbr` – abbreviated state name
- `month_abbr` – month for data reported (with abbreviation)

- date – `as_date()` version of last update

```
library(readr) # read csv, txt, etc into R
library(ggplot2) # graphics
library(zoo) # moving averages
library(tibble) # require for tidyverse packages
library(tidyverse) # all tidyverse packages
library(hrbrthemes) # themes for graphs
library(socviz) # %nin%
library(geofacet) # maps
library(usmap) # latitude and longitude
library(socviz) # for %nin%
library(ggmap) # mapping
```

We import JH COVID-19 data from in R using the `readr()` function. To get the Colorado state data, we downloaded the CSV data from CDHE.

```
JHCovid19States <- readr::read_csv("https://raw.githubusercontent.com/mjfrigaard/storybench/master/drafts/data/jhsph/2020-06-22-JHCovid19States.csv")

utils::head(JHCovid19States)
```

Calculating rolling averages

Two states (Colorado and Georgia) have seen an increase in their death rates. We're going to calculate and visualize the rolling averages for cumulative deaths and new cases in these states and compare them to the other 48 states.

To calculate a simple moving average (over k days), we can use the `rollmean()` function from the `zoo` package. This function takes the parameter k , which is an integer width of the rolling window in days. The code below calculates a 3, 5, 7, 15, and 21-day rolling average for the deaths from COVID-19 in the US.

```
JHCovid19States <- JHCovid19States %>%
  dplyr::arrange(desc(state)) %>%
  dplyr::group_by(state) %>%
  dplyr::mutate(
    death_03da = zoo::rollmean(deaths, k = 3, fill = NA),
    death_05da = zoo::rollmean(deaths, k = 5, fill = NA),
    death_07da = zoo::rollmean(deaths, k = 7, fill = NA),
    death_15da = zoo::rollmean(deaths, k = 15, fill = NA),
    death_21da = zoo::rollmean(deaths, k = 21, fill = NA)) %>%
  dplyr::ungroup()
```

Below is an example of this calculation for the JH COVID-19 data for the state of Colorado. The port of the code “`dplyr::`” indicates the we are performing a function, like `arrange()` from the `dplyr` package.

```
JHCovid19States %>%  
  dplyr::arrange(date) %>%  
  dplyr::filter(state == 'Colorado') %>%  
  dplyr::select(state, date, deaths,  
                death_03da:death_07da) %>%  
  utils::head(7)
```

```
# A tibble: 7 x 6  
  state     date   deaths death_03da death_05da death_07da  
  <chr>    <date>  <dbl>      <dbl>      <dbl>      <dbl>  
1 Colorado 2020-04-12    289        NA        NA        NA  
2 Colorado 2020-04-13    306       307.       NA        NA  
3 Colorado 2020-04-14    327       320.      321       NA  
4 Colorado 2020-04-15    328       337.      338       338  
5 Colorado 2020-04-16    355       352.      354.      357.  
6 Colorado 2020-04-17    372       372       373.      373  
7 Colorado 2020-04-18    389       394.      391.      395.
```

The calculation works by taking the first value in our confirmed deaths, variable, `conf_death_03da` (510.3333), is the average deaths in Colorado from the first date with a data point on either side of it (i.e., the date 2020-04-13 has 2020-04-12 preceding it, and 2020-04-14 following it). We can check our math using the following code.

```
mean(c(461, 499, 571))
```

```
[1] 510.3333
```

The first value in `conf_death_05da` (132.0) is the average deaths in Colorado from the first date with two data points on either side of it (2020-04-14 has 2020-04-12 and 2020-04-13 preceding it, and 2020-04-15 and 2020-04-16 following it). We can check our math below.

```
mean(c(461, 499, 571, 596, 668))
```

```
[1] 559
```

And the first value in `conf_death_07da` (609.7143) is the average death rate in Colorado from the first date with three data points on either side (2020-04-15 has 2020-04-12, 2020-04-13 and 2020-04-14 preceding it,

and 2020-04-16, 2020-04-17, and 2020-04-18 following it). Check our math again:

```
mean(c(461, 499, 571, 596, 668, 725, 748))
```

```
[1] 609.7143
```

Symmetry

It's good practice to calculate rolling averages using an odd number for k (it makes the resulting values symmetrical). Each rolling mean is calculated from the numbers surrounding it. If we want to visualize and compare the three, rolling means against the original deaths data, we can do this with a little pivoting.

CDHE COVID-19 DATA

We decided to go with the state of Colorado data using from the CDHE, rather than trust the JH data. We took this approach assuming that the Colorado's data from thee CDHE is more current and more accurate than the JH (federal) data.

```
dta <- as.data.frame(read_csv("D:/Documents/Data/covid_co.csv"))
```

We create the time series for confirmed deaths in Colorado by extracting the `date` and `confirmed_deaths` fields, or the 1st and 5th columns of the CSV file. Then we used the `ts_xts()` function from the `tsbox` package. The tsbox package offers several time series functions and we are using two of them in this example.

```
dta_conf_death <- dta[,c(1,5)]
conf_death.xts <- tsbox::ts_xts(dta_conf_death)
```

Next, we defined our moving average function using `rollmean()` from the `zoo` package, with several value for k days.

```
conf_death_03da <- zoo::rollmean(conf_death.xts, k= 3, fill = NA)
conf_death_05da <- zoo::rollmean(conf_death.xts, k= 5, fill = NA)
conf_death_07da <- zoo::rollmean(conf_death.xts, k= 7, fill = NA)
conf_death_14da <- zoo::rollmean(conf_death.xts, k=14, fill = NA)
conf_death_21da <- zoo::rollmean(conf_death.xts, k=21, fill = NA)
conf_death_30da <- zoo::rollmean(conf_death.xts, k=31, fill = NA)
```

Having defined our moving averages, we plot the confirmed death time series and the 30-day moving average, which is a smooth line (curve) in Figure 3-9.

```
par(oma = c(3, .15, .15, .15))
plot(conf_death.xts, lwd = 3, col = 'blue3',
      main = 'COVID-19 Confirmed Deaths - Cumulative',
      cex = .85, ylab = "Deaths", las = 2)
lines(conf_death_30da, lwd = 4, col = 'green3')
legend('right', deaths.xts,
       c('Confirmed Deaths', '30-Day MA'),
       fill = c('blue2', "'green2'), cex = .85, box.lty = 0)
```

We should notice that the moving average is smoother than the confirmed deaths data, even though there is not very much noise in the series.

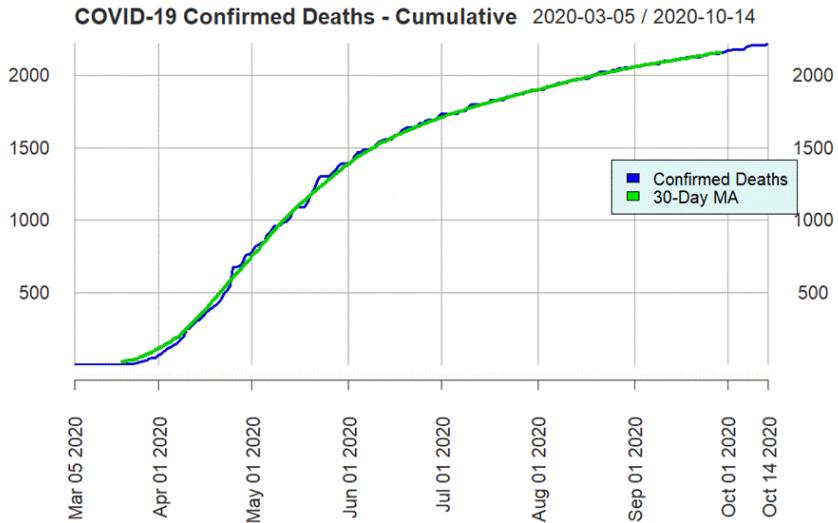


Figure 3-9. COVID-19 confirmed deaths in the state of Colorado

Continuing our examination of COVID-19 in Colorado, it will be instructive to look at the number of new cases per day, the trend. While it is important to look at cumulative growth and decline in any series, we can get a better picture of the COVID-19 pandemic by drilling down to the details.

We start with new deaths each day. As total deaths have risen, the number of new deaths has declined as seen in Figure 3-12Figure 3-11. At

first glance, the series appears as if moving averages would unveil its secrets.

```
library(tsbox)
dta <- as.data.frame(read_csv("D:/Documents/Data/covid_co.csv"))
dta_death <- dta[,c(1,6)]
death.xts <- tsbox::ts_xts(dta_death)

TSstudio::ts_plot(death.xts, line.mode = 'lines', width = 2,
                   dash = NULL, color = 'green2', slider = FALSE,
                   type = 'single', Xtitle = 'Days',
                   Ytitle = 'New Deaths', Xgrid = TRUE, Ygrid = TRUE,
                   title = 'COVID-19 New Deaths - Daily')
```

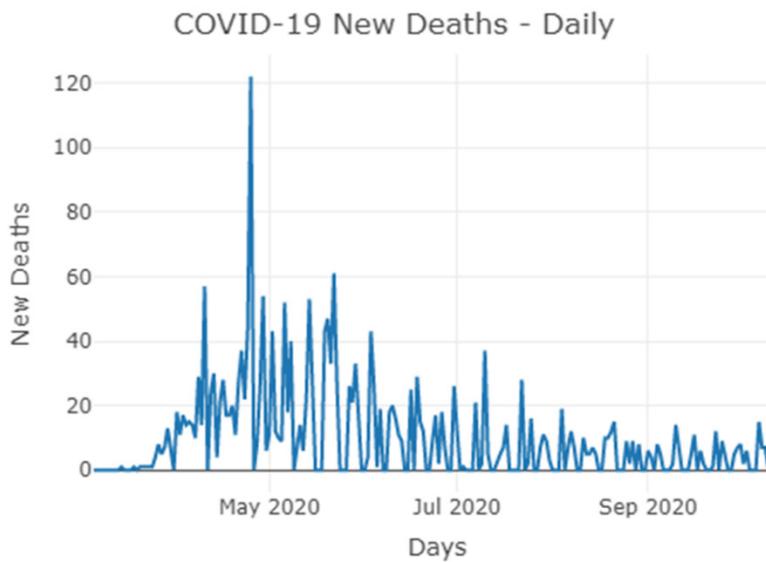


Figure 3-10. COVID-19 daily new deaths in Colorado

We calculate three moving averages. We then look at the trend of the series in Figure 3-11, where we see an initial peak of deaths per day followed by a decrease. This could give an idea of how the fitted MA might look when we calculate and plot MAs.

```
new_death_07da <- zoo::rollmean(death.xts, k = 7, fill = NA)
new_death_14da <- zoo::rollmean(death.xts, k = 14, fill = NA)
new_death_21da <- zoo::rollmean(death.xts, k = 21, fill = NA)
# Plot parameter for margin spacing
```

```

par(oma = c(.15, .15, .15, .15))
# call for the defined officer
op <- options(tsbox.lwd = 3, tsbox.col = 'blue1')
# Time Series plot from tsbox
ts_plot(ts_trend(death.xts),
        title = 'Trend for New Deaths From COVID-19')

```

Daily Trend for New Deaths From COVID-19



Figure 3-11. Trend of new COVID-19 deaths each day in Colorado

Next, we plot the COVID-19 daily new deaths, along with the 7-day and 21-day moving averages. We also add a legend to help us distinguish between the MA plots. The `par()` function will set the margins of our plot. As a reminder, the `cex` parameter is used to size the text of the plot in relation to the plot itself.

```

par(oma = c(3, .15, .15, .15))
plot(death.xts, lwd = 3, col = 'blue3',
      main = 'COVID-19 New Deaths - Daily', cex = .85,
      ylab = 'Deaths', las = 2)
lines(new_death_07da, lwd = 4, col = 'red2')
lines(new_death_21da, lwd = 4, col = 'green3')
legend('center', deaths.xts,
       c('New Deaths', '7-Day MA', '21-Day MA'),
       fill = c('blue2', 'red2', 'green2'), cex = .75,
       box.lty = 0)

```

We see from Figure 3-12 that after an initial peak, the MAs demonstrate an overall decrease in the number of deaths due to COVID-19 (we saw this with the trend line), and the number ultimately assumes a low constant rate of daily deaths.

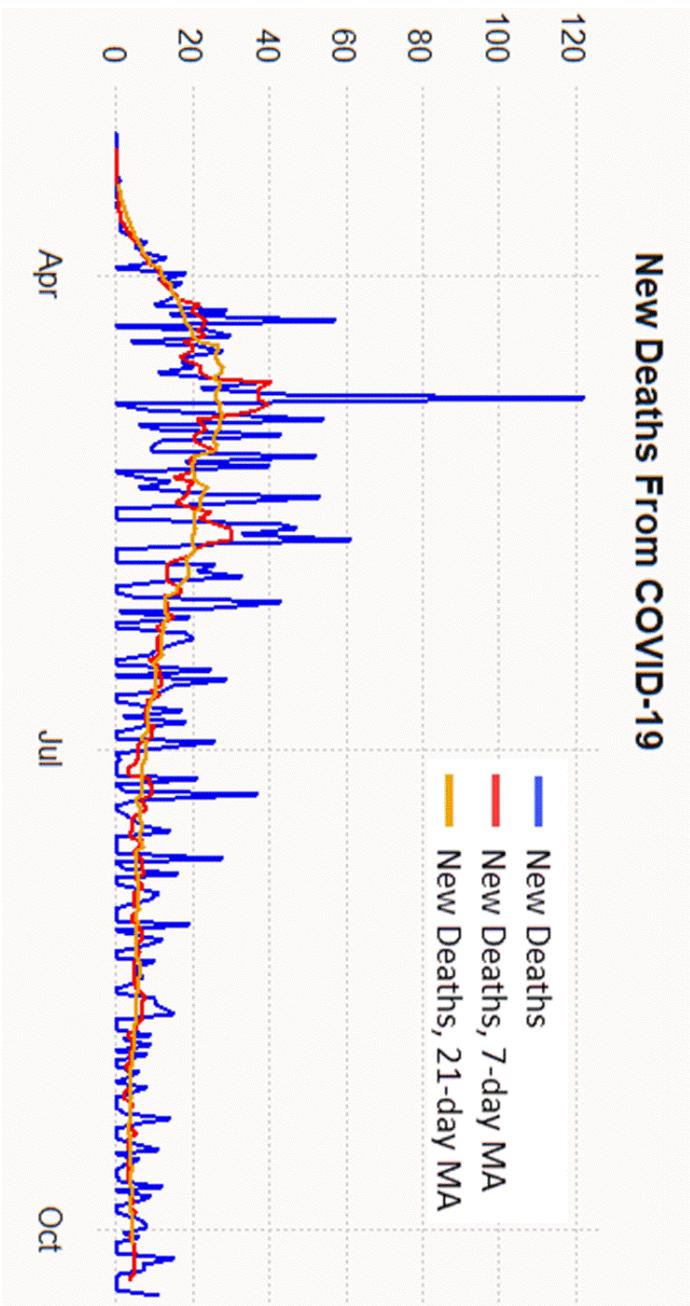


Figure 3-12. COVID-19 deaths per day in Colorado during the specified dates

We use the *PerformanceAnalytics* package to plot the ACF and the PACF of the 21-day moving average. Viewing Figure 3-13, we note that the series is within tolerance after one lag and that the ACF values are decreasing.

```
library(PerformanceAnalytics)
par(bg = 'snow2')
chart.ACFplus(new_death_21da, maxlag = 21, elementcolor = 'red3',
             main = "COVID-19 Daily New Deaths", cex.main = 2,
             col.main = 'slateblue3', xlab = "lags", cex.lab = 1.25,
             col.lab = 'red3')
```

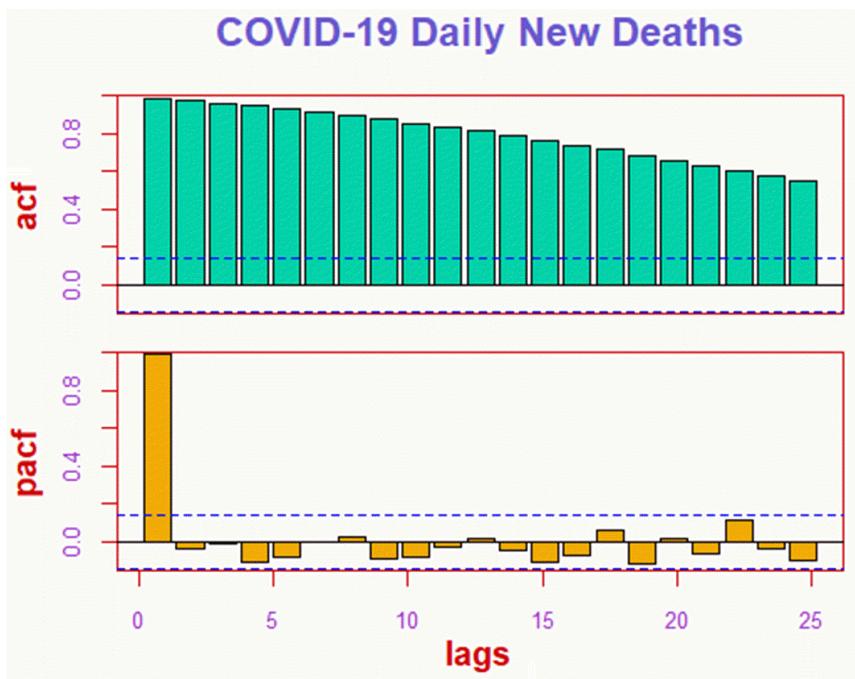


Figure 3-13. Covid-19 21-day moving average plot of the ACF and PACF

We continue this process when looking at new cases per day. Once we generate the MAs for this series, we plot the time series and 21-day MA in Figure 3-14. We also add the 21-day MA that we applied for new deaths per day to give the data its proper perspective. In spite of the bumpy ride of cases, deaths stay low in Colorado and have a practically constant rate.

```
dta_cases <- dta[,c(1,3)]
```

```

cases.xts <- tsbox::ts_xts(dta_cases)

new_case_07da <- zoo::rollmean(cases.xts, k = 7, fill = NA)
new_case_14da <- zoo::rollmean(cases.xts, k = 14, fill = NA)
new_case_21da <- zoo::rollmean(cases.xts, k = 21, fill = NA)

par(oma = c(3, .15, .15, .15))
plot(cases.xts, lwd = 2, col = 'blue3',
      main = 'COVID-19 New Cases vs Deaths - Daily', cex = .85,
      ylab = 'Number of Cases vs Deaths', las = 2)
lines(new_case_21da, lwd = 3, col= 'orange2')
lines(new_death_21da, lwd = 4, col= 'red3')
legend_margin('top', legend = c(''), pch = 0, box.lty = 0)
legend("center", cases.xts,
       c( 'New cases', 'New Cases 21-Day MA',
          'new Deaths 21-Day MA'),
       fill= c('blue2', 'green2', 'red2'), cex = .75, box.lty = 0)

```

Our plotting code introduce two novel parameters. First, we used the `par()` function to customize plot margins. The `oma` parameter is used to set the margins in the outer margin area. We can also use the `mar` parameter to set the margins within the plots margin area. The second parameter we used is the `legend_margin()` function, which places the legend relative to the portion of the margins area. The parameter accepts one-word arguments, for instance, “top”, “right”, “center”, and so on.

In Figure 3-15, we have plotted the ACF and PACF for the 21-MA from above. They reveal that auto correlation decreases rapidly and is within tolerance after lag 1. This indicates a good fit to the time series. Hence, the important relationship is between the number of new cases and the number of new deaths. It appears that the number of deaths is not proportional to the number of cases, which implies that an increase in cases does not result in an increase of deaths, no matter the cause.

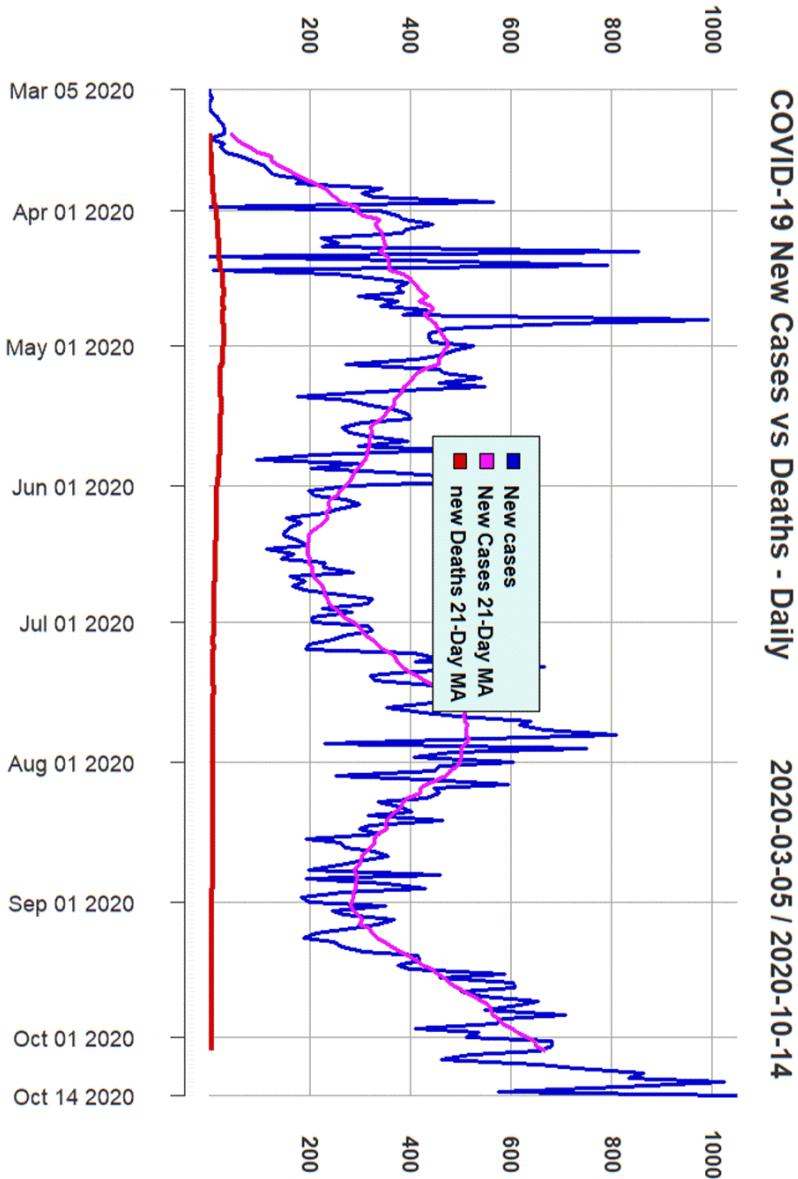


Figure 3-14. COVID-19 new cases each day in the state of Colorado.

```

par(bg = 'lightcyan')
chart.ACFplus(new_case_21da, maxlag = 21, elementcolor = 'red3',
             main = 'COVID-19 Daily New Cases', cex.main = 1.75,
             col.main = 'slateblue3', xlab = 'lags', cex.lab = 1.25,
             col.lab = 'red3', col.axis ='violetred', cex.axis = 1.25)

```

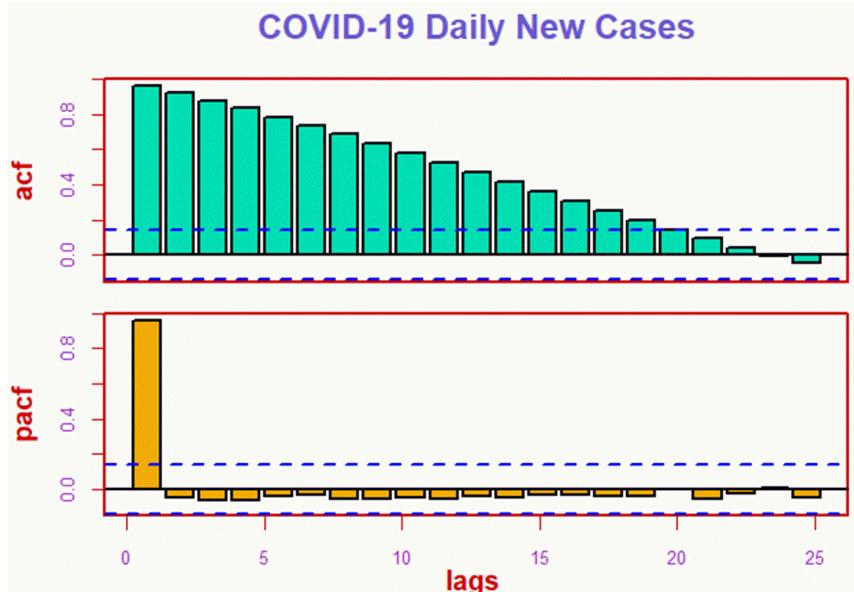


Figure 3-15. COVID-19 new cases with a ACF and PACF for the 21-day moving average and new deaths comparison

Now we tackle COVID-19 deaths for the state of Georgia, using the JH data. In this case, we filter for Georgia in the `state` field, using the `filter()` function from `dplyr` package. We have also embedded the `plot` function within the `plot` function as shown in Figure 3-16.

```

JHCovid19States %>%
  dplyr::filter(state == 'Georgia') %>%
  tidyr::pivot_longer(names_to = 'rolling_mean_key',
                      values_to = 'rolling_mean_value',
                      cols = c(deaths, death_03da, death_21da)) %>%
  dplyr::filter(date >=
                lubridate::as_date('2020-05-15') & # after May 15
                date <= lubridate::as_date('2020-06-20')) %>%
  # before June 20
  ggplot2::ggplot(aes(x = date, y = rolling_mean_value,
                      color = rolling_mean_key)) +
  ggplot2::geom_line(size = 1.25) +
  ggplot2::labs(title =

```

```
'Georgia's rolling average total COVID deaths',
  subtitle = 'Between 2020-05-15 and 2020-06-20',
  y = 'Deaths', color = 'Metric', x = 'Date') +
  hrbrthemes::theme_ipsum_rc()
```

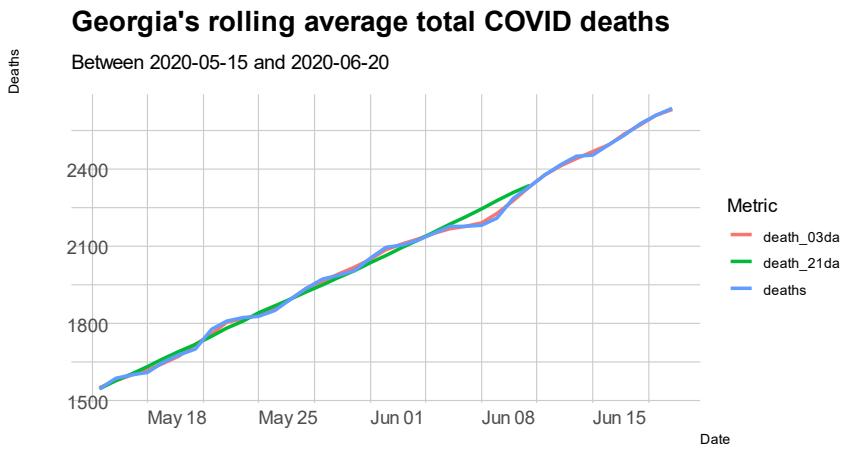


Figure 3-16. COVID-19 deaths in the state of Georgia May-June 2020

Which mean should I use?

The `zoo:::rollmean()` function works by successively averaging each period (k) together. Knowing which period (k) to use in `zoo:::rollmean()` is a judgment call. The higher the value of k , the smoother the line gets, but we are also sacrificing more data. If we compare the 3-day average (`death_3da`) to the 21-day average (`death_21da`), we see the line for deaths gets increasingly smooth.

Calculating new cases in each State

Below we get some help from `dplyr::lag()` to calculate the new cases in each state per day. We join this new calculation back to the JHCovid19States dataset, but renames it `JHCovid19NewCases`.

```
JHCovid19NewCases <- JHCovid19States %>%
  # group this by state and day
  group_by(state, date) %>%
  # get total deaths per day
  dplyr::summarize(
    confirmed_sum = (sum(confirmed, na.rm = TRUE))) %>%
  # calculate 'new deaths' = todays deaths - yesterdays deaths
  mutate(new_confirmed_cases = confirmed_sum -
```

```

dplyr::lag(x = confirmed_sum, n = 1, order_by = date)) %>%
dplyr::select(state, new_confirmed_cases, date) %>%
# join back to JHCovid19
dplyr::left_join(., y = JHCovid19States,
                 by = c('state', 'date')) %>%
# reorganize
dplyr::select(state, state_abbr, date, month_abbr, day,
              confirmed, dplyr::contains('confirm'),
              dplyr::contains('death'), lat,
              long, dplyr::ends_with('rate'))

```

Next, we inspect the data (tibble) to make sure our process worked properly.

```

# check GA
JHCovid19NewCases %>%
  dplyr::filter(state == "Georgia") %>%
  dplyr::select(state_abbr, date, confirmed,
                new_confirmed_cases) %>%
  utils::head()

```

	state	state_abbr	date	confirmed	new_confirmed_cases
1	Georgia	GA	2020-04-12	12452	NA
2	Georgia	GA	2020-04-13	13315	863
3	Georgia	GA	2020-04-14	14578	1263
4	Georgia	GA	2020-04-15	14987	409
5	Georgia	GA	2020-04-16	15669	682
6	Georgia	GA	2020-04-17	17194	1525

We can see this calculation is getting the number of new confirmed cases each day correct. Now we can calculate the rolling mean for the new confirmed cases in each state. We do this by grouping states together for calculating means. When we complete this action, we need to ungroup the states, since we might do more analysis later.

```

JHCovid19NewCases <- JHCovid19NewCases %>%
  dplyr::group_by(state) %>%
  dplyr::mutate(
    new_conf_03da = zoo::rollmean(new_confirmed_cases, k = 3,
                                   fill = NA),
    new_conf_05da = zoo::rollmean(new_confirmed_cases, k = 5,
                                   fill = NA),
    new_conf_07da = zoo::rollmean(new_confirmed_cases, k = 7,
                                   fill = NA),
    new_conf_15da = zoo::rollmean(new_confirmed_cases, k = 15,
                                   fill = NA))

```

```
    fill = NA),
new_conf_21da = zoo::rollmean(new_confirmed_cases, k = 21,
    fill = NA)) %>%
dplyr::ungroup()
```

Moving averages with geofacets

Now, we take a look at the seven-day moving averages of new cases across all states using the *geofacet* package. First, we'll build two plots for Colorado, combine them, and then extend this to the entire country.

Column graph for new cases

We will limit the JHCovid19NewCases data to June 1st – June 21st.

```
JHCovid19NewCasesJun <- JHCovid19NewCases %>%
  dplyr::filter(date >= lubridate::as_date('2020-06-01') &
    # after June 1
    date <= lubridate::as_date('2020-06-20')) # before June 20
```

Then we need to create a `ggplot2::geom_col()` for the `new_confirmed_cases`. We will build these two graphs with `hrbrthemes::theme_modern_rc()`. However, notice that the JH data does not continue after 2020-06-20, and provides a different “picture” of the COVID-19 pandemic. This kind of analysis is often performed, using a limited amount of data, to make bold medical, social, and political points, without regard for the whole story.

```
JHCovid19NewCasesJun %>%
  dplyr::filter(state == "Colorado") %>%
  ggplot2::ggplot(aes(x = day,
    y = new_confirmed_cases)) +
  geom_col(alpha = 1/10) +
  ggplot2::labs(title = 'Colorado's new COVID cases',
    subtitle = 'Moving average between 2020-06-01 and 2020-06-20',
    y = 'New Cases',
    x = 'Day') +
  hrbrthemes::theme_ipsum_rc()
```

Colorado's new COVID cases

Rolling average between 2020-06-01 and 2020-06-20

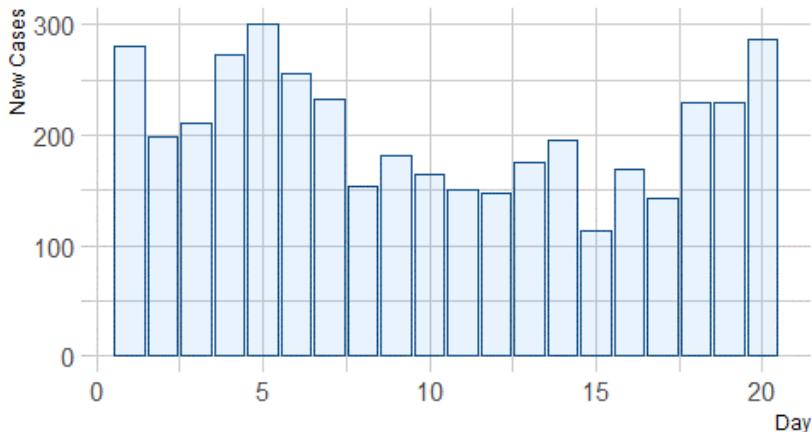


Figure 3-17. Bar chart using incomplete COVID-10 data

Tidy dataset of new cases

Now we want to add lines for the `new_conf_` variables, so we'll use `pivot_longer`.

```
FLNewCasesTidy <- JHCovid19NewCasesJun %>%
  # only Colorado
  dplyr::filter(state == "Colorado") %>%
  # pivot longer
  tidyr::pivot_longer(names_to = 'new_conf_av_key',
                      values_to = 'new_conf_av_value',
                      cols = c(new_conf_03da,
                               new_conf_05da,
                               new_conf_07da)) %>%
  # reduce vars
  dplyr::select(day,
                date,
                state,
                state_abbr,
                new_conf_av_value,
                new_conf_av_key)
```

Now we can combine them into a single plot by adding line graphs for new cases. In this instance, we use `geom_line()` from `ggplot2`.

```
JHCovid19NewCasesJun %>%
  # Colorado new cases
  dplyr::filter(state == "Colorado") %>%
  ggplot2::ggplot(aes(x = day, y = new_confirmed_cases,
    group(date))) +
  geom_col(alpha = 1/10) +
  # add the line with new data
  ggplot2::geom_line(data = FLNewCasesTidy,
    mapping = aes(x = day,
      y = new_conf_av_value,
      color = new_conf_av_key)) +
  ggplot2::labs(title = 'Colorado's new COVID cases',
    subtitle= 'Rolling average between 2020-06-01 and 2020-06-20',
    y = 'New Cases', color = 'Metric', x = 'Day') +
  hrbrthemes::theme_modern_rc()
```

Colorado's new COVID cases

Rolling average between 2020-06-01 and 2020-06-20

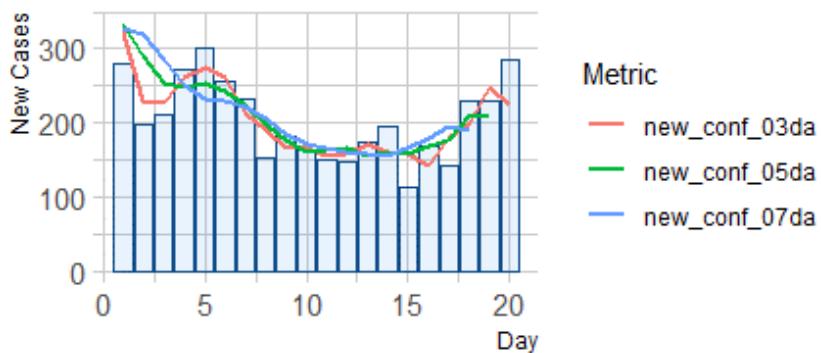


Figure 3-18. Colorado COVID-19 bar chart using incomplete data with fitted moving averages

We can see that the blue (7-day average) of new confirmed cases is definitely the smoothest line. Let's compare it to the 3-day average using a `geofacet` for the other states in the US. Again, we build our tidy data frame of new confirmed case metrics.

```
NewCasesTidy <- JHCovid19NewCasesJun %>%
  # pivot longer
  tidyr::pivot_longer(names_to = 'new_conf_av_key',
    values_to = 'new_conf_av_value',
    cols = c(new_conf_03da, new_conf_07da)) %>%
  # better labels for printing
```

```

dplyr::mutate(new_conf_av_key = dplyr::case_when(
  new_conf_av_key == 'new_conf_03da' ~
    '3-day new confirmed cases',
  new_conf_av_key == "new_conf_07da" ~
    '7-day new confirmed cases',
  TRUE ~ NA_character_)) %>%
# reduce vars
dplyr::select(day, date, state, state_abbr, new_conf_av_value,
  new_conf_av_key)

```

Also, we'll switch the theme to `hrbrthemes::theme_ipsum_tw()`. We also use the min and max to get values for the subtitle. Coupled with `facet_geo()` from the `geofacet` package, this produces Figure 3-19 showing the 3-day and 7-day moving averages for each state, on their corresponding icons.

```

# get min and max for labels
min_date <- min(JHCovid19NewCasesJun$date, na.rm = TRUE)
max_date <- max(JHCovid19NewCasesJun$date, na.rm = TRUE)

library(geofacet)
library(ggplot2)
library(hrbrthemes) # themes for graphs

JHCovid19NewCasesJun %>%
  ggplot2::ggplot(aes(x = day,
                      y = new_confirmed_cases)) +
  geom_col(alpha = 3/10, linetype = 0) +
  ggplot2::geom_line(data = NewCasesTidy,
                      mapping = aes(x = day, y = new_conf_av_value,
                                     color = new_conf_av_key)) +
  geofacet::facet_geo(~ state_abbr, grid = 'us_state_grid2',
                      scales = 'free_y') +
  ggplot2::labs(title =
    'US rolling 3 and 7-day averages of new COVID cases',
    subtitle = 'Between 2020-05-31 and 2020-06-20',
    y = 'New Cases', color = 'Metric:', x = 'Day') +
  hrbrthemes::theme_ipsum_tw() +
  theme(axis.title.x = element_blank(),
        axis.text.x = element_blank(),
        axis.ticks.x = element_blank()) +
  ggplot2::theme(legend.position = 'top')

```

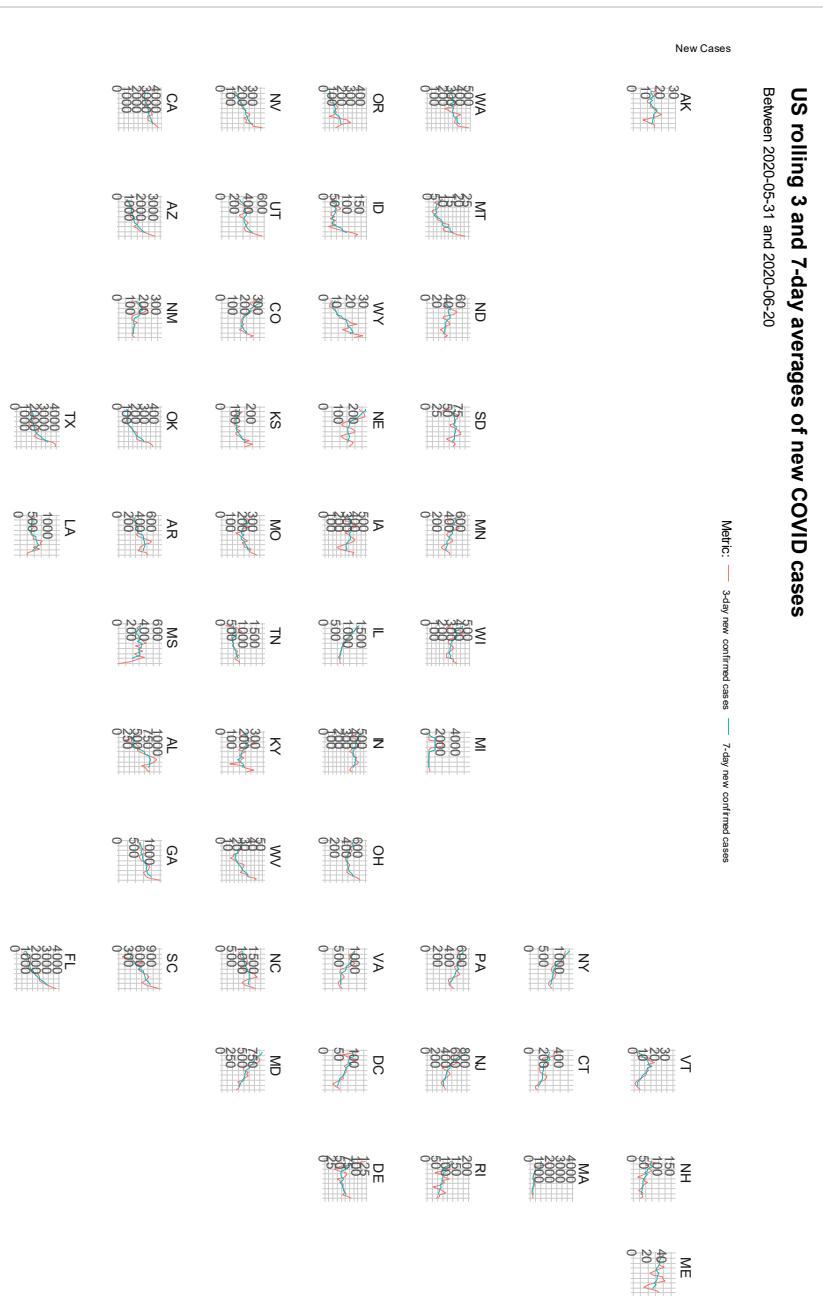


Figure 3-19. US rolling 3 and 7-day moving averages of the COVID-19 cases

```

# get data for only 7-day average
JHCovid19NewCasesJun7da <- JHCovid19NewCasesJun %>%
  dplyr::select(day, new_conf_07da, state, state_abbr)

# get min and max for labels
min_date <- min(JHCovid19NewCasesJun$date, na.rm = TRUE)
max_date <- max(JHCovid19NewCasesJun$date, na.rm = TRUE)

JHCovid19NewCasesJun %>%
  ggplot2::ggplot(aes(x = day,
                      y = new_confirmed_cases)) +
  geom_col(alpha = 2/10, linetype = 0) +
  ggplot2::geom_line(data = JHCovid19NewCasesJun7da,
                      mapping = aes(x = day, y = new_conf_07da,
                                    color = 'darkred'),
                      show.legend = FALSE) +
  geofacet::facet_geo(~ state_abbr,
                      grid = 'us_state_grid1', scales = 'free_y') +
  ggplot2::labs(title =
    'US 7-day rolling average of new COVID cases',
    subtitle = paste0('Between', min_date, ' and ', max_date),
    y = 'New Cases', x = 'Day') +
  hrbrthemes::theme_ipsum_tw() +
  ggplot2::theme(axis.title.x = element_blank(),
                axis.text.x = element_blank(),
                axis.ticks.x = element_blank())

```

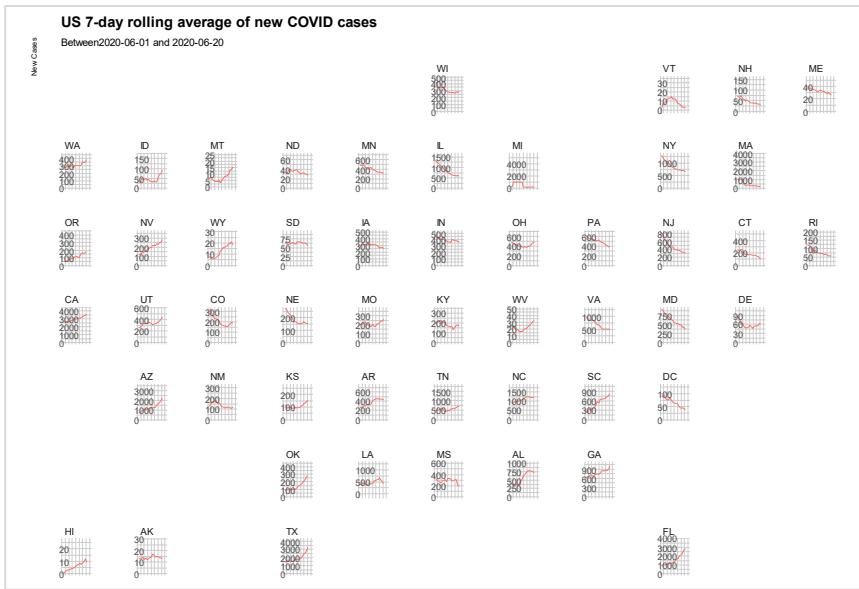


Figure 3-20. US 7-day moving average of the new Covid-19 daily cases

Weighted Moving Averages

The **TTR** package in R provides functions for computing weighted moving averages for constructing technical trading rules. The following functions are available:

- ALMA Arnaud Legoux moving average.
- EMA Exponential moving average.
- SMA Simple moving average
- DEMA Double-exponential moving average
- EVWMA Elastic volume-weighted moving average
- WMA Weighted moving average
- VWA Variable-length moving average.
- ZLEMA Zero lag exponential moving average

```
ema.20 <- EMA(milk_mon_ts, 20)
sma.20 <- SMA(milk_mon_ts, 20)
dema.20 <- DEMA(milk_mon_ts, 20)
evwma.20 <- EVWMA(milk_mon_ts, 20)
zlema.20 <- ZLEMA(milk_mon_ts, 20)
```

The *tstools* package provides some nice features for plotting multiple time series plots in one window. The simple one-line code below produces Figure 3-21.

```
tsplot(ema.20, sma.20, dema.20, evwma.20)
```

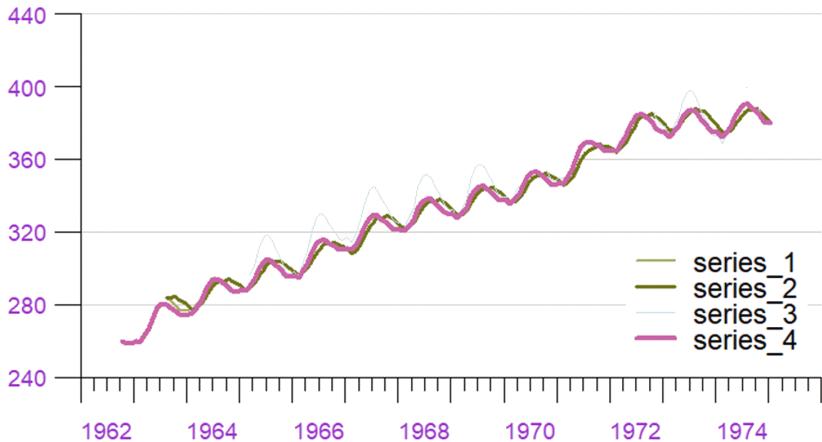


Figure 3-21. *tstools* plot of four moving average models

Got Milk (Moving Averages)?

The next R code here produces a more customized plot of the four moving average models and the milk production time series data, as shown in Figure 3-22, and is still relatively simple. The function `init_tsplot_theme()` is used to set most of the options we use for customization and is a parameter of the `tsplot()` function, which we defined separately and called for it in `tsplot()`.

```
library("tstools")
tt <- init_tsplot_theme(lwd = c(4,2,2,2),
                       line_colors = c('red','purple','blue','green'))
tsplot(list('EMA' = ema.20, 'SMA' = sma.20,'DEMA' = dema.20,
           'EVWMA' = evwma.20), auto_legend = TRUE,
       plot_title ='Milk Production Moving Averages Comparison',
       theme = tt)
```

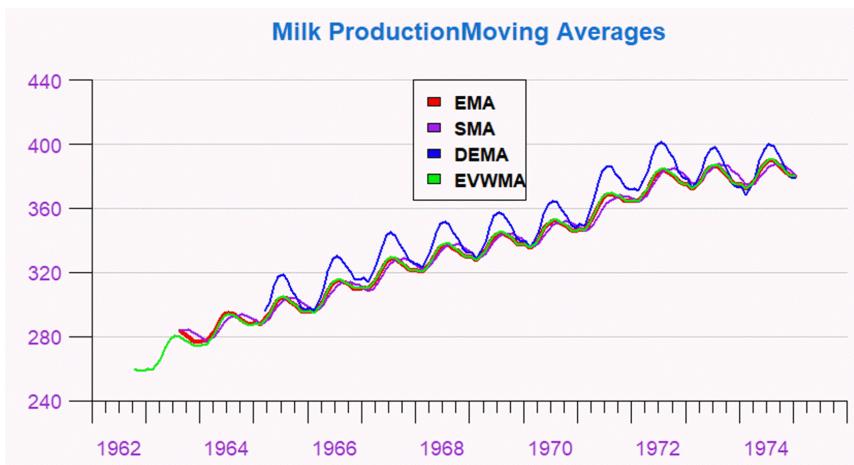


Figure 3-22. Time series data and three weighted moving averages: EMA, SMA, DEMA, and EVWMA

Got Milk (Forecasting)?

Often time series data are used to predict what might happen in the future, given the patterns seen in the data. This is known as forecasting. There are many methods used to forecast time series data, and they vary widely in complexity, but this should serve as a brief introduction to the commonly used method, known as Error-Trend-Seasonality (ETS) models. ETS models are a type of Exponential Smoothing State Space model. ETS models are used for modelling how a single variable will change over time by identifying its underlying trends, not taking into account any other variables. ETS models differ from a simple moving average (we will see these in Chapter 5) by weighting the influence of previous points on future time points based on how much time is between the two points. i.e., over a longer period of time it is more likely that some unmeasured condition has changed, resulting in different behavior of the variable that has been measured. In Chapter 6, we will cover another important group of forecast models called the ARIMA models, autoregressive models which describe autocorrelations in the data rather than trends and seasonality.

The model type is usually a three-character string identifying method using the framework terminology of Hyndman et al. (Hyndman, Koehler, Snyder, & Grose, 2002) and Hyndman et al (Hyndman, Koehler, Ord, & Snyder, 2008). The first letter denotes the error type ("A", "M" or "Z"); the second letter denotes the trend type ("N", "A", "M" or "Z"); and the third letter denotes the season type ("N", "A", "M" or "Z"). In all cases, "N"=none, "A"=additive, "M"=multiplicative and "Z"=automatically selected. So, for example, "ANN" is simple exponential smoothing with additive errors, "MAM" is multiplicative Holt-Winters' method with multiplicative errors, and so on.

It is also possible for the model to be of class "`ets`", and equal to the output from a previous call to `ets`. In this case, the same model is fitted to `y` without re-estimating any smoothing parameters.

Choosing which model to use to forecast our data can be difficult and requires using our own intuition, as well as looking at test statistics. To test the accuracy of a model, we have to compare it to data that has not been used to generate the forecast, so we will create some data subsets from the `milk_mon_ts` time series object - one for generating the model (`milk_mon_model`) and one for testing the model's accuracy (`milk_mon_test`). The function `window()` is a function similar to `subset()` or `filter()`, subsetting an object based on arguments, but it is used especially for time series (`ts`) objects. `window()` takes the original time series object (`x`) and the `start` and `end` points of the subset. If `end` is not included, the subset extends to the end of the time series.

```
milk_mon_model <- window(x = milk_mon_ts, start = c(1962),
                           end = c(1970))
milk_mon_test <- window(x = milk_mon_ts, start = c(1970))
```

We will first compare model forecasts of different ETS models visually by extracting forecast data from forecast objects and plotting it using `ggplot()` against the test data. The code below creates the ETS models we will compare.

```
# Creating model objects of each type of ets model
milk_ets_auto <- ets(milk_mon_model)
milk_ets_mmm <- ets(milk_mon_model, model = 'MMM')
milk_ets_zzz<- ets(milk_mon_model, model = 'ZZZ')
milk_ets_mmm_damped <- ets(milk_mon_model, model = 'MMM',
```

```
damped = TRUE)
```

Now, we create the forecast objects using the code below. In this code, `h = 60` means that the forecast will be 60 time periods long, and in our case a time period is one month

```
milk_ets_fc <- forecast(milk_ets_auto, h = 60)
milk_ets_mmm_fc <- forecast(milk_ets_mmm, h = 60)
milk_ets_zzz_fc <- forecast(milk_ets_zzz, h = 60)
milk_ets_mmm_damped_fc <- forecast(milk_ets_mmm_damped, h = 60)
```

Now, we convert the individual forecast to data frames. This ensures we have all the information needed for the forecasts and the metrics used to compare them. For each model type, we create the data frame, remove whitespaces from the column names (otherwise R would not recognize them), prepend the day of the month to the date field, and add a column for the model type (for comparison purposes).

```
# Creating a data frame for each model
# the default ETS model
milk_ets_fc_df <- cbind(
    'Month' = rownames(as.data.frame(milk_ets_fc)),
    as.data.frame(milk_ets_fc))

# Removing whitespace from column names
names(milk_ets_fc_df) <- gsub(' ', '_', names(milk_ets_fc_df))

# prepending day of month to date
milk_ets_fc_df$Date <- as.Date(paste('01-',
    milk_ets_fc_df$Month, sep = ''), format = "%d-%b %Y")

# Adding column of model type
milk_ets_fc_df$Model <- rep('ets')

# the MMM model
milk_ets_mmm_fc_df <- cbind(
    'Month' = rownames(as.data.frame(milk_ets_mmm_fc)),
    as.data.frame(milk_ets_mmm_fc))
names(milk_ets_mmm_fc_df) <- gsub(' ', '_', names(
    milk_ets_mmm_fc_df))
milk_ets_mmm_fc_df$Date <- as.Date(paste("01-",
    milk_ets_mmm_fc_df$Month, sep = ''), format = "%d-%b %Y")
milk_ets_mmm_fc_df$Model <- rep('ets_mmm')

# the ZZZ model
milk_ets_zzz_fc_df <- cbind(
    'Month' = rownames(as.data.frame(milk_ets_zzz_fc)),
    as.data.frame(milk_ets_zzz_fc))
```

```

names(milk_ets_zzz_fc_df) <- gsub(' ', '_', names(
    milk_ets_zzz_fc_df))
milk_ets_zzz_fc_df$Date <- as.Date(paste('01-',
    milk_ets_zzz_fc_df$Month, sep = ''), format = "%d-%b %Y")
milk_ets_zzz_fc_df$Model <- rep('ets_zzz')

# the MMM damped model
milk_ets_mmm_damped_fc_df <- cbind(
    'Month' = rownames(as.data.frame(milk_ets_mmm_damped_fc)),
    as.data.frame(milk_ets_mmm_damped_fc))
names(milk_ets_mmm_damped_fc_df) <- gsub(' ', '_', names(
    milk_ets_mmm_damped_fc_df))
milk_ets_mmm_damped_fc_df$Date <- as.Date(paste('01-',
    milk_ets_mmm_damped_fc_df$Month, sep = ''),
    format = "%d-%b %Y")
milk_ets_mmm_damped_fc_df$Model <- rep('ets_mmm_damped')

```

The next snippet of code combines all the forecast objects in one data frame.

```

forecast_all <- rbind(milk_ets_fc_df, milk_ets_mmm_fc_df,
    milk_ets_zzz_fc_df, milk_ets_mmm_damped_fc_df)

```

Now, we plot the time series forecasts for each model in one graph, as shown in Figure 3-23.

```

(forecast_plot <- ggplot() +
    geom_line(data = milk_mon[1:108,], aes(x = month_date,
        y = milk_prod_per_cow_kg), lwd=1.05) +
    geom_line(data = forecast_all, aes(x = Date,
        y = Point_Forecast, color = Model), lwd = 1.1) +
    ggtitle('Monthly Milk Production Forecast
        (Jan 1971 - Dec-1975)') +
    xlab('Years') +
    ylab('Milk Production per Cow per month (kg)') +
    guides(color = guide_legend(title = 'Monthly forecasts')) +
    theme(legend.position = 'bottom')
)

```

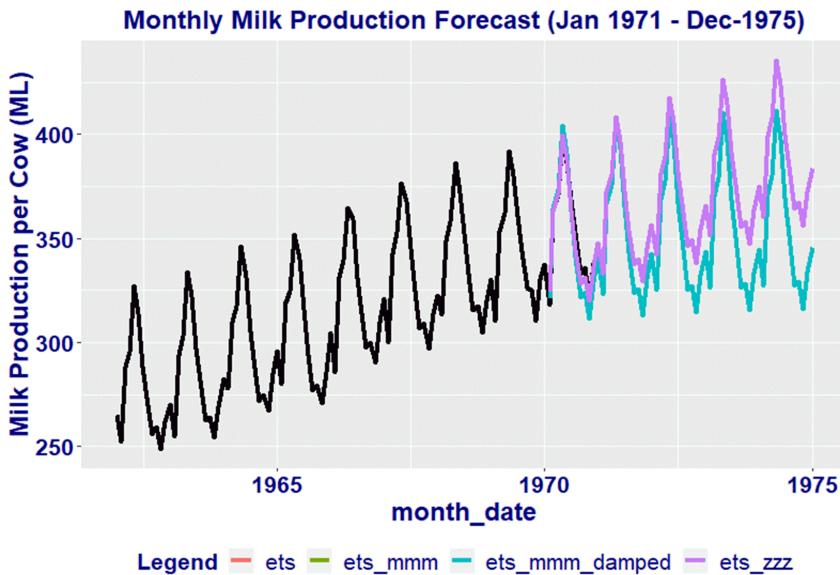


Figure 3-23. Comparison forecasts of the milk production per month

We can also numerically compare the accuracy of different models to the data we excluded from the model (`monthly_milk_test`) using `accuracy()`.

```
# default ETS model accuracy
accuracy(milk_ets_fc, milk_mon_test)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	0.0186	2.7264	2.0941	0.0012	0.6752	0.2190	0.0070
Test set	6.4972	10.8703	8.6438	1.6876	2.2924	0.9041	0.8164
	Theil's U						
Training set	NA						
Test set	0.4875						

```
# MMM ETS model accuracy
accuracy(milk_ets_mmm_fc, milk_mon_test)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	0.3733	3.1087	2.3070	0.1121	0.7331	0.2413	0.0104
Test set	23.9447	26.5710	24.4991	6.3609	6.5099	2.5623	0.8777
	Theil's U						
Training set	NA						
Test set	1.201562						

```
# ZZZ ETS model accuracy
accuracy(milk_ets_zzz_fc, milk_mon_test)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	0.0187	2.7264	2.0941	0.0012	0.6752	0.2190	0.0070
Test set	6.4972	10.8703	8.6438	1.6876	2.2924	0.9041	0.8164
Theil's U							
Training set	NA						
Test set	0.4875						

```
# MMM damped ETS model accuracy
accuracy(milk_ets_mmm_damped_fc, milk_mon_test)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	0.3733	3.1087	2.3070	0.1121	0.7331	0.2413	0.0104
Test set	23.9447	26.5710	24.4991	6.3609	6.5099	2.5624	0.8777
Theil's U							
Training set	NA						
Test set	1.2016						

Now, we will briefly examine the accuracy statistics provided by the previous output. We will cover these metrics and train-test splits thoroughly in Chapter 6. We provide definitions of the metrics we use most often here. Why they are often used will come later.

Root Mean Squared Error (RMSE). Take each difference between the model and the observed values, square it, take the mean (the expected value), then apply the square root it as follows.

***Definition 3.3.** The Root Mean Squared Error (RMSE) of an estimator $\hat{\theta}$ with respect to an estimated parameter θ is defined as the square root of the mean square error:*

$$RMSE(\hat{\theta}) = \sqrt{MSE(\hat{\theta})} = \sqrt{E((\hat{\theta} - \theta)^2)}$$

where $E(X)$ is the expected value of X . The Mean Squared Error (MSE) is the term under the radical.

Mean Error (ME). the mean difference between modelled and observed values.

Mean Absolute Error (MAE). The same as ME, but all errors are transformed to positive values so positive and negative errors don't cancel each other out.

Mean Percentage Error (MPE). Similar to ME, but each error is expressed as a percentage of the forecast estimate. Percentage Errors are not scale dependent so they can be used to compare forecast accuracy between datasets.

Mean Absolute Percentage Error (MAPE). The same as MPE, but all errors are transformed to positive values so positive and negative errors don't cancel each other out.

***Definition 3.4.** The Mean Absolute Percentage Error (MAPE) is a measure of prediction accuracy of forecasting methods and is usually expressed as a ratio (or percentage) defined by the formula:*

$$MAPE = \frac{1}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|$$

where A_t is the actual value and F_t is the forecast value.

Mean Absolute Scaled Error (MASE). Compares the MAE of the forecast with the MAE produced by a naive forecast. A naive forecast is one which simply projects a straight line into the future, the value of which is the final value of the time series used to construct the model. A $MASE > 1$ tells us that the naive forecast fit the observed data better than the model, while a $MASE < 1$ tells us that the model was better than the naive model.

Auto-Correlation Function at lag 1 (ACF1). How correlated are data points with data points directly after them, where $ACF = 1$ means points are fully correlated and $ACF = 0$ means points are not at all correlated.

Theil's U. Compares the forecast with results from a model using minimal data. Errors are squared to give more weight to large errors. A $U < 1$ means the forecast is better than guessing, while a $U > 1$ means the forecast is worse than guessing.

MAPE is the most commonly used measure of forecast accuracy, probably due to it being easy to understand conceptually. However, MAPE becomes highly skewed when observed values in the time series are close to zero and infinite when observations equal zero, making it unsuitable for some time series that have low report values. MAPE also gives a heavier penalty to positive deviations than negative deviations, which makes it useful for some analyses, e.g. economic forecasts which don't want to run the risk of over-estimating the value of a commodity. MASE is suggested here as an alternative which avoids the shortcomings of MAPE while remaining interpretable.

Training set denotes values that were gathered from comparing the forecast to the data that was used to generate the forecast (notice how the Mean Error (ME) is very small).

Test set denotes values that were gathered from comparing the forecast to the test data which we deliberately excluded when training the forecast.

By comparing the **MAPE** and **MASE** statistics of the four models in the Test set row, we can see that the `monthly_milk_ets_fc` and `monthly_milk_ets_zzz_fc` models have the lowest values. Looking at the graphs for this forecast and comparing it visually to the test data, we can see that this is the forecast which best matches the test data. So, we can use that forecast to project into the future.

Now that we have identified the best forecast model(s), we can use these models to find out what milk production will be like in the year 1975! Use the code below to extract a predicted value for a given year from our forecasts. This is as simple as subsetting the forecast data frame to extract the correct value. I'm using functions from the `dplyr` package, with pipes (`%>%`), but you could use any other method of subsetting such as the `[]` square bracket method using base R.

```
milk_ets_fc_df %>%
  filter(Month == 'Jan 1975') %>%
  select(Month, Point_Forecast)
```

Month	Point_Forecast
1 Jan 1975	383.7897

```
milk_ets_zzz_fc_df %>%
  filter(Month == 'Jan 1975') %>%
  select(Month, Point_Forecast)
```

Month	Point_Forecast
1 Jan 1975	383.7897

This output shows us that either the default or the ZZZ model is most accurate for forecasting monthly milk production.

Chapter review

In this chapter we discussed and demonstrated performing time series decomposition using R. This included: (1) setting up data into a time series and plotting it; (2) performing trend analysis; (3) performing seasonality analysis; (4) smoothing data using moving (or rolling) averages. We also demonstrated how to perform times series decomposition and analysis using the `stlplus()` function, which provides better diagnostics. We will perform decomposition of all of the remaining chapters are we use the results to specify time series models.

Transformations such as logarithms can help to stabilize the variance of a time series. Differencing can help stabilize the mean of a time series by removing changes in the level of a time series, and thus eliminating trend and seasonality.

We also covered the following definitions:

Definition Number	Definition
Definition 3.1: Moving Average	A moving average of order m can be written as $\hat{T}_t = \frac{1}{m} \sum_{j=-k}^k y_{t+j}$ <p>Where $m = 2k + 1$. That is, the estimate of the trend-cycle at time t is obtained by averaging values of the time series within k periods of t.</p>
Definition 3.2: Weighted Moving Average	In general, a weighted m-MA can be written as $\hat{T}_t = \sum_{j=-k}^k a_j y_{t+j}$ <p>Where $k = (m - 1)/2$ and the weights are given by $[a_{-k}, \dots, a_k]$.</p>

Definition 3.3: <i>Root Mean Squared Error (RMSE)</i>	<p>The Root Mean Squared Error (RMSE) of an estimator $\hat{\theta}$ with respect to an estimated parameter θ is defined as the square root of the mean square error:</p> $RMSE(\hat{\theta}) = \sqrt{MSE(\hat{\theta})} = \sqrt{E((\hat{\theta} - \theta)^2)}$ <p>where $E(X)$ is the expected value of X. The Mean Squared Error (MSE) is the term under the radical.</p>
Definition 3.4: <i>Mean Absolute Percentage Error (MAPE)</i>	<p>The Mean Absolute Percentage Error (MAPE) is a measure of prediction accuracy of forecasting methods and is usually expressed as a ratio (or percentage) defined by the formula:</p> $MAPE = \frac{1}{n} \sum_{t=1}^n \left \frac{A_t - F_t}{A_t} \right $ <p>where A_t is the actual value and F_t is the forecast value.</p>

Review Exercises

- Consider the `usmelec` dataset, the total net generation of electricity (in billion kilowatt hours) by the U.S. electric industry (monthly for the period January 1973 – June 2013). In general, there are two peaks per year: in mid-summer and mid-winter (<https://github.com/stricje1/Data> or included with `fpp2`).
 - Examine the 12-month moving average of this series to see what kind of trend is involved.
 - Do the data need transforming? If so, find a suitable transformation.
- From the `fma`-Package in R, load the `copper` dataset
 - Describe the dataset (refer to the R documentation)
 - Set up copper as a time series
 - Decompose the copper time series
 - Describe and explore the components
- This exercise uses the `cangas` data (monthly Canadian gas production in billions of cubic metres, January 1960 – February 2005).
 - Plot the data using `autoplot()`, `ggsubseriesplot()` and `ggseasonplot()` to look at the effect of the changing seasonality over time. What do you think is causing it to change so much?

- b. Do an STL decomposition of the data. You will need to choose `s.window` to allow for the changing shape of the seasonal component.
- 4. Repeat the milk production example using the daily data.

Chapter 4 – Exponential Smoothing for Times Series

Exponential smoothing is a time series forecasting method for univariate data that can be extended to support data with a systematic trend or seasonal component. It was proposed in the late 1950s (Brown, 1956) (Holt, 2004) (R., 1960), and has motivated some of the most successful forecasting methods.

In this chapter, we present the mechanics of the most important exponential smoothing methods, and their application in forecasting time series with various characteristics. This helps us develop an intuition to how these methods work. In this setting, selecting and using a forecasting method may appear to be somewhat ad hoc. The selection of the method is generally based on recognizing key components of the time series (trend and seasonality) and the way in which these enter the smoothing method (e.g., in an additive, damped, or multiplicative manner). In detail, we examine the seasonality of time series data, reviewing how to perform decomposition, and look at statistical ways of detecting seasonality. We also discuss adjusting for seasonality and why we would be concerned with doing so.

Exponential Smoothing Methods

Forecasts produced using exponential smoothing methods are weighted averages of past observations, with the weights decaying exponentially as the observations get older. In other words, the more recent the observation the higher the associated weight. This framework generates reliable forecasts quickly and for a wide range of time series, which is a great advantage and of major importance to applications in industry.

Analysis of Seasonality

In Chapter 3, we spent some time discussing trend analysis using moving averages. Here, we will begin discussing seasonality analysis as we apply exponential smoothing. Seasonal dependency (**seasonality**) is another general component of the time series pattern. The concept was illustrated in the example of the airline passenger's data. In Chapter 2,

we gave an intuitive definition of seasonality (see Definition 2.3). We now provide a formal definition.

Definition 4.1. *Seasonality is the correlational dependency of order k between each k th element of the series and the $(i - k)$ th element (Kendall, 1976) and measured by autocorrelation (i.e., a correlation between the two terms); k is usually called the lag.*

If the measurement error is not too large, seasonality can be visually identified in the series as a pattern that repeats every k elements.

Autocorrelation correlogram

We briefly described the use of correlograms for analyzing stationarity in Chapter 2. We can also use these tools to analyze seasonality. After we have applied a smoothing method, seasonal patterns of time series can be examined via **correlograms**.

Definition 4.2. *The autocorrelation coefficient at lag h is given by*

$$r_h = \frac{c_h}{c_0}$$

where c_h is the autocovariance function

$$c_h = \frac{1}{N} \sum_{t=1}^{N-h} (Y_t - \bar{Y})(Y_{t+h} - \bar{Y})$$

and c_0 is the variance function

$$c_0 = \frac{1}{N} \sum_{t=1}^N (Y_t - \bar{Y})^2$$

The resulting value of r_h will range between -1 and $+1$.

The correlogram (auto-correlogram) displays graphically and numerically the **autocorrelation function (ACF)**, that is, serial correlation coefficients (and their standard errors) for consecutive lags in a specified range of lags (e.g., 1 through 30). Ranges of two standard errors for each lag are usually marked in correlograms but typically the size of autocorrelation is of more interest than its reliability because we

are usually interested only in very strong (and thus highly significant) autocorrelations.

Definition 4.3. The **autocorrelation function** (ACF) is a time domain measure of the stochastic process memory, and does not reveal any information about the frequency content of the process. Generally, for an error signal, e_t , the ACF is defined as,

$$\rho_k = \frac{\text{Cov}(e_t, e_{t+k})}{\sqrt{\text{Var}(e_t)\text{Var}(e_{t+k})}}$$

The ACF plot is useful for identifying non-stationary time series. For a stationary time series, the ACF will drop to zero relatively quickly, while the ACF of non-stationary data decreases slowly. The ACF of the smoothed Consumer Confidence Index (CCI) data (see Figure 4-1) looks just like that from a white noise series. We will cover this example fully under Holt-Winters later in this chapter.

Values between -0.2 and 0.2 demonstrate that the series data points are not highly correlated. That is the observations at t_n are not dependent on values at t_{n-1} . If the values are correlated, as seen at lag 1 in Figure 4-1, we can remove the dependencies by performing operations like making the series stationary (we cover the details in Chapter 6). There is only one autocorrelation lying just outside the 95% limits, and the **Ljung-Box Q^*** statistic has a p-value of $< 2.2\text{e-}16$ (for $h = 10$). This suggests that the *monthly change* in the CCI data is essentially a random amount uncorrelated with previous days.

```
data(unemp.cci)
cci <- ts(unemp.cci[, 'cci'])
cci <- window(unemp.cci[, 'cci'], start = 1997)
cci.fit<-hw(cci, seasonal = 'additive')
cci.acf <- acf(cci.fit$fitted, 36)
cci.pacf <- pacf(cci.fit$fitted, 36)
Box.test(f1$fitted, lag = 36, type = 'Ljung-Box')
par(mfrow = c(2,1))
plot.acf(cci.acf)
title(main = 'CCI ACF fit with Holt_Winters Additive method',
      cex.main = 1.5, font.main = 2, col.main = 'blue',
      col.lab = 'darkblue')
plot.acf(cci.pacf)
title(main = 'CCI PACF fit with Holt_Winters Additive method',
      cex.main = 1.5, font.main = 2, col.main = 'blue',
```

```
col.lab = 'darkblue')
```

```
Box-Ljung test  
data: f1$fitted  
X-squared = 1369.2, df = 48, p-value < 2.2e-16
```

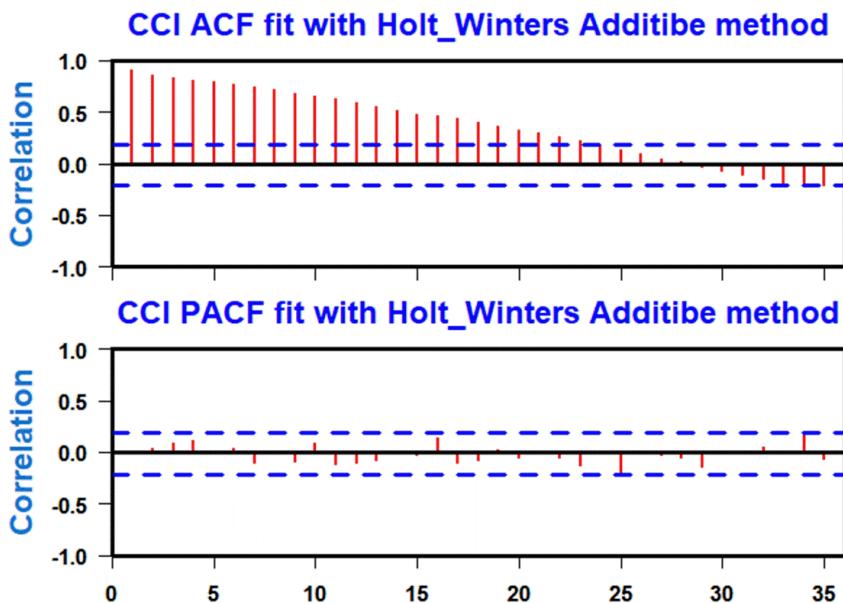


Figure 4-1. ACF and PACF of the differenced Consumer Confidence Index (CCI) data

Examining correlograms

While examining correlograms, you should keep in mind that autocorrelations for consecutive lags are formally dependent. Consider the following example of monthly beer sales in millions of barrels, from January 1975 to December 1990 (see Figure 4-2), which is available as part of the TSA package in R (Chan & Ripley, 2012). If the first element is closely related to the second, and the second to the third, then the first element must also be somewhat related to the third one, etc. This implies that the pattern of serial dependencies can change considerably after removing the first order autocorrelation (i.e., after differencing the series with a lag of 1).

```
require(TSA)
```

```
#Read the Beer data
data(beersales)
beersales
#Build a time series
beer2.ts <- ts(beersales, start = c(1991, 1), end = c(1995, 12),
               frequency = 12)
plot(beer2.ts, col = 4)
```

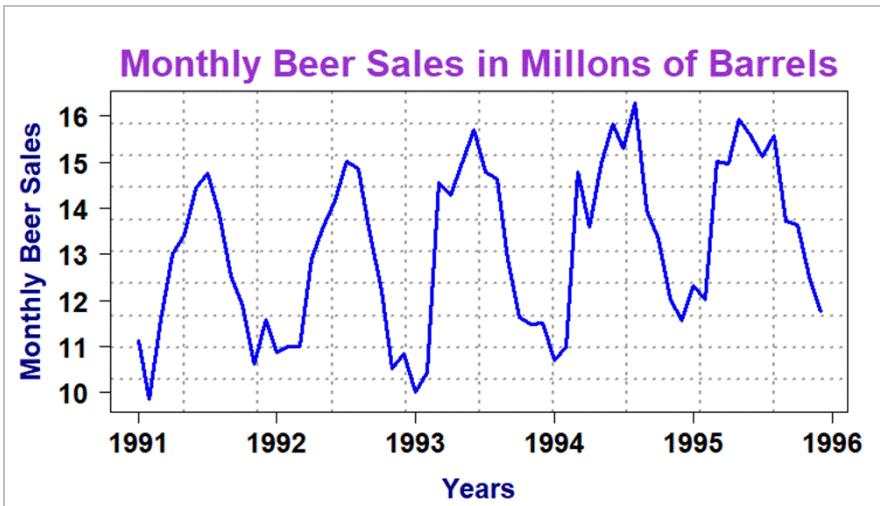


Figure 4-2. Beer Sales Time series data

In Figure 4-3, notice that the top correlogram shows residuals that are outside of the tolerance past lag 20. However, in the lower correlogram, after one differencing, only one at the 12th lag is out of tolerance. Similarly, in Figure 4-3, there are five autocorrelations either out of or close to tolerance before differencing. This is reduced to two after differencing. The R code used to develop the ACFs and PACFs is:

We now apply exponential smoothing.

```
#Apply exponential smoothing
beer.fit <- HoltWinters(beersales, beta = FALSE, gamma = FALSE)
#Build a forecast
beer.for <- forecast(beer.fit, h = 48)
#Develop the ACF and PACF
acf(beer.for$residuals, lag.max = 48)
pacf(beer.for$residuals, lag.max = 48)
```

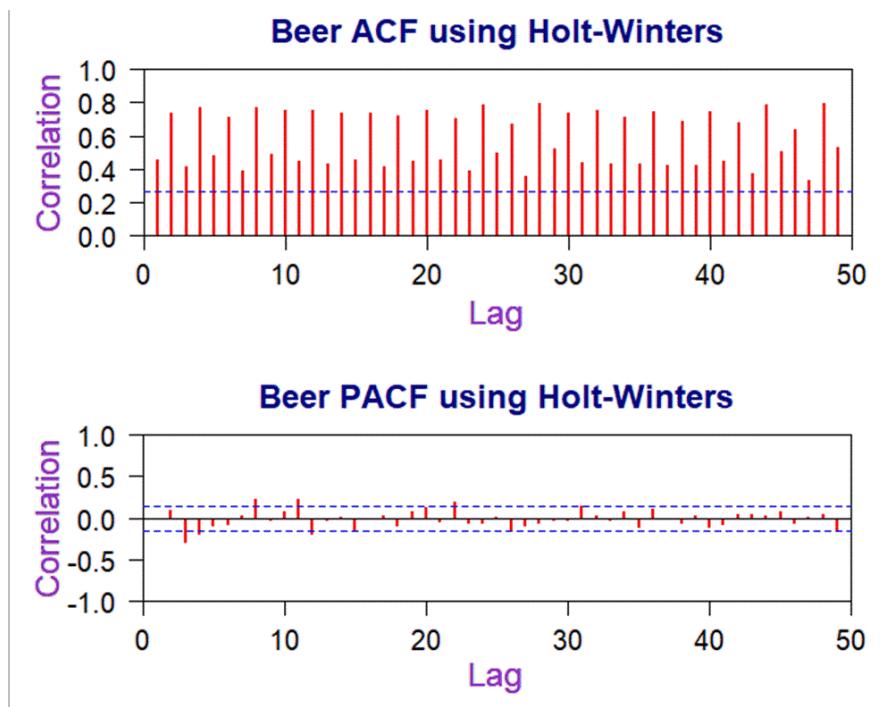


Figure 4-3. Plotted ACF and PACF for the Beer times series data

We now difference the times series data and develop a forecast (see Figure 4-4).

```
# Analyze effect of Differencing
beer.diff1<-diff(beer2.ts, differences = 2)
# ACF of the differenced beer data
beer.acf1 <- acf(beer.diff1, lag.max = 36)
# PACF of the differenced beer data
beer.pacf1 <- pacf(beer.diff1, lag.max = 36)
# correlograms of the differenced beer data
plot.acf(beer.acf1)
title(main = 'Beer ACF using differencing = 2',
      cex.main = 1.25,
      font.main = 2, col.main = 'blue', col.lab = 'darkblue')
plot.acf(beer.pacf1)
title(main = 'Beer PACF using differencing = 2',
      cex.main = 1.25,
      font.main = 2, col.main = 'blue', col.lab = 'darkblue')
```

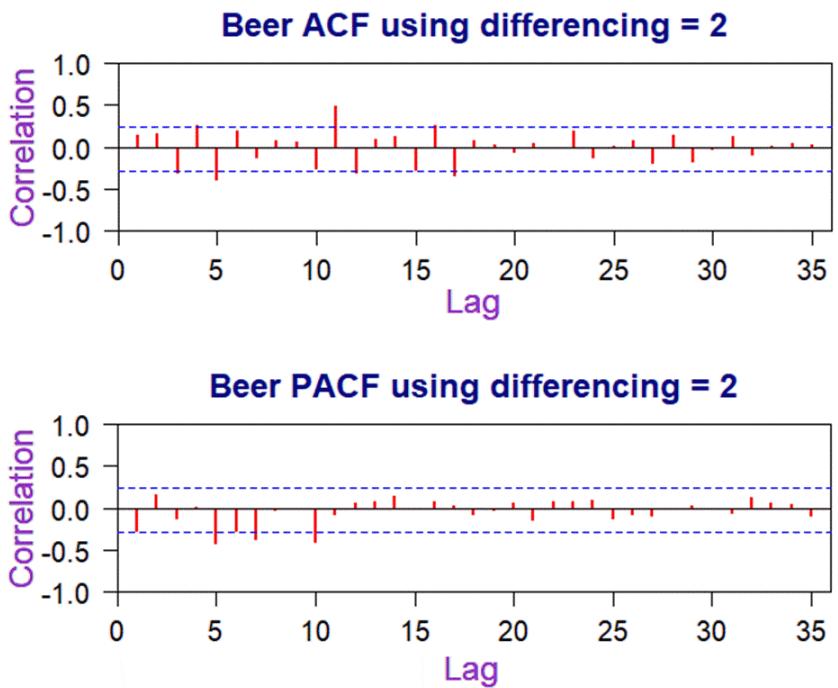


Figure 4-4. Plotted ACF and PACF for the fitted beer model residuals

Partial autocorrelations

Another useful method to examine serial dependencies is to examine the partial autocorrelation function (*PACF*) - an extension of autocorrelation, where the dependence on the intermediate elements (those *within* the lag) is removed. In other words the partial autocorrelation is similar to autocorrelation, except that when calculating it, the (auto) correlations with all the elements within the lag are partialled out (Box & Jenkins, Time Series Analysis: Forecasting and Control, 1976). If a lag of one is specified (i.e., there are no intermediate elements within the lag), then the partial autocorrelation is equivalent to auto correlation. In a sense, the partial autocorrelation provides a "cleaner" picture of serial dependencies for individual lags (not confounded by other serial dependencies).

Removing serial dependency

Serial dependency for a particular lag of k can be removed by differencing the series, which is converting each i^{th} element of the

series into its difference from the $(i - k)^{th}$ element. There are two major reasons for such transformations.

First, we can identify the hidden nature of seasonal dependencies in the series. Remember that, as mentioned in the previous paragraph, autocorrelations for consecutive lags are interdependent. Therefore, removing some of the autocorrelations will change other autocorrelations, that is, it may eliminate them, or it may make some other seasonality more apparent.

The other reason for removing seasonal dependencies is to make the series **stationary** which is necessary for **ARIMA** and other techniques.

Decomposing Seasonal Data

A seasonal time series consists of a trend component, a seasonal component, and an irregular component. Decomposing the time series means separating the time series into these three components: that is, estimating these three components.

To estimate the trend component and seasonal component of a seasonal time series that can be described using an additive model, we can use the `decompose()` function in R. This function estimates the trend, seasonal, and irregular components of a time series that can be described using an additive model.

The function `decompose()` returns a list object as its result, where the estimates of the seasonal component, trend component and irregular component are stored in named elements of that list objects, called “seasonal”, “trend”, and “random” respectively.

For example, as discussed above, the time series of the monthly live births (adjusted) in thousands for the United States is seasonal with a peak every summer and trough every winter, and can probably be described using an additive model since the seasonal and random fluctuations seem to be roughly constant in size over time:

```
beertimeseriescomponents <- decompose(beer2.ts)
```

We plot the estimated trend, seasonal, and irregular components of the time series by using the `plot()` function, for example, in Figure 4-5.

```
plot(beertimeseriescomponents, col = 'blue', lwd = 2)
```

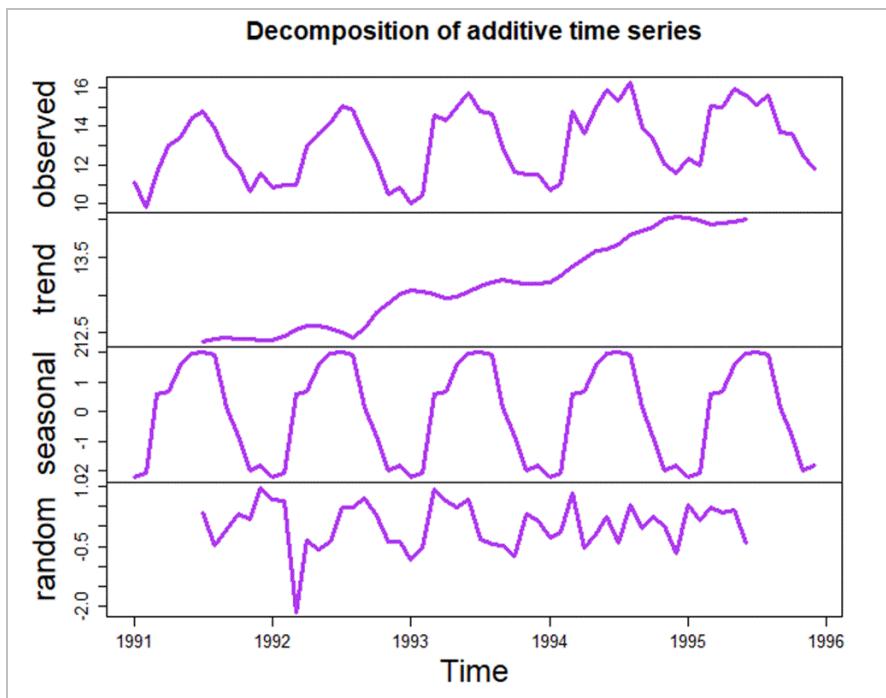


Figure 4-5. Decomposition of Beer Sales data

The plot below shows the original time series (top), the estimated trend component (second from top), the estimated seasonal component (third from top), and the estimated irregular component (bottom). We see that the estimated trend component shows an increase from about 12 in 1991 to about 14 in 1996.

Adjusting Seasonality

If you have a seasonal time series that can be described using an additive model, you can seasonally adjust the time series by estimating the seasonal component, and subtracting the estimated seasonal component from the original time series. We can do this using the estimate of the seasonal component calculated by the `decompose()` function. For example, to seasonally adjust the time series of the beer sales data, we can estimate the seasonal component using `decompose()`, and then subtract the seasonal component from the original time series as seen in Figure 4-6.

```

beertimeseries <- ts(beersales, start = c(1991, 1),
                      end = c(1995, 12), frequency = 12)
beertimeseriescomponents <- decompose(beertimeseries)
beertimeseriesseasonallyadjusted <- beertimeseries -
  beertimeseriescomponents$seasonal
plot(beertimeseriesseasonallyadjusted, col = 4, lwd = 2,
     main = 'Beer Seasonally Adjusted')

```

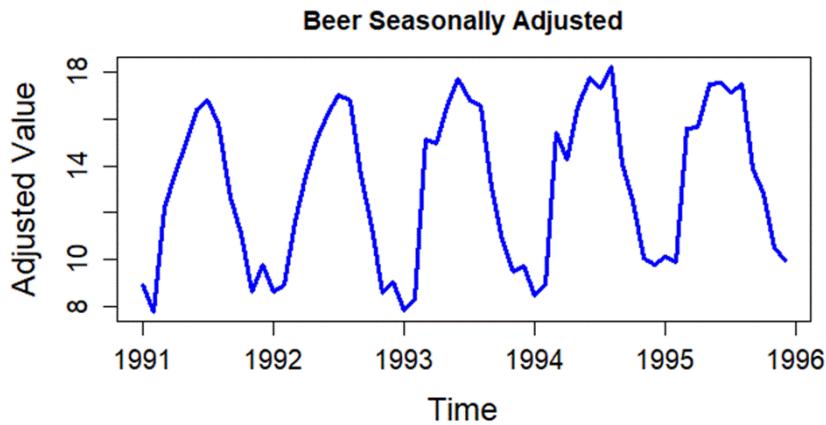


Figure 4-6. Seasonally adjusted Beer Sales data

You can see that the seasonal variation has been removed from the seasonally adjusted time series. The seasonally adjusted time series now just contains the trend component and an irregular component.

Simple or Single Exponential Smoothing

The simplest of the exponentially smoothing methods is naturally called **simple or single exponential smoothing (SES)**. This method is suitable for forecasting data with no clear trend or seasonal pattern. For example, the data in Figure 4-7 do not display any clear trending behavior or any seasonality. (There is a rise in the last few years, which might suggest a trend. We will consider whether a trended method would be better for this series later in this chapter.)

```

autoplot(oil) +
  ylab('Oil (millions of metric tons)') + xlab('Year') +
  geom_line(color = 'purple', linetype = 1, size = 1.1)

```

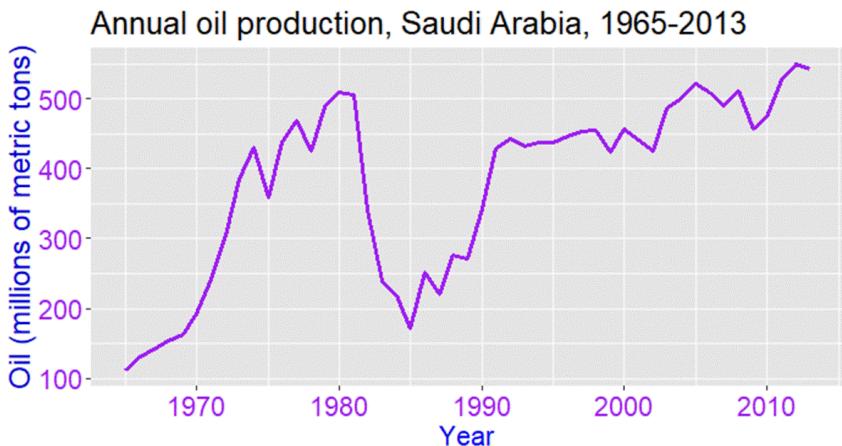


Figure 4-7. Annual oil production (millions of metric tons), Saudi Arabia, 1965-2013

Definition 4.4. *Simple exponential smoothing (SES) is calculated using weighted averages, where the weights decrease exponentially as observations come from further in the past — the smallest weights are associated with the oldest observations, such that*

$$y_{t+h} = \alpha y_t + \alpha(1 - \alpha)y_{t-1} + \alpha(1 - \alpha)^2 y_{t-2} + \dots$$

Where y_t is the smoothed statistic and $0 \leq \alpha \leq 1$ is the **smoothing parameter**. Then the forecasted values of y_t are estimates by $\hat{y}_{t+h|T}$, where all future forecasts are equal to a simple average of the observed data:

$$\hat{y}_{t+h|T} = \frac{1}{T} \sum_{t=1}^T y_t$$

for $h = 1, 2, \dots$. Hence, the average method assumes that all observations are of equal importance, and gives them equal weights when generating forecasts.

The one-step-ahead forecast for time $t + 1$ is a weighted average of all of the observations in the series s_1, \dots, s_t . The rate at which the weights decrease is controlled by the parameter α . Thus, the simplest form of exponential smoothing is given by the formula:

$$y_t = \alpha y_t + \alpha(1 - \alpha)y_{t-1} = y_{t-1} + \alpha(x_t - y_{t-1})$$

where α is the *smoothing factor*, and $0 < \alpha < 1$. The component form is given by $\ell_t = \alpha y_t + (1 - \alpha)\ell_{t-1}$, where ℓ_t is the level (or the smoothed value) of the series at time t . Simple exponential smoothing is easily applied, and it produces a smoothed statistic as soon as two observations are available.

The exponential smoothing methods constitute a broad family of approaches to univariate time series forecasting. Although these have been around for many decades. It was not until the twenty-first century that they became part of a systematic framework. Hyndman, et all (2015) developed the State Space approach, which capture the exponential smoothing methods. Well known models such as simple or single exponential smoothing, Holt's linear trend method, and the Holt-Winters seasonal method can be shown to be special cases of a single family.

SES Example in R

We take the oil times series we plotted in Figure 4-7, Annual oil production (millions of metric tons), Saudi Arabia, 1965-2013, and apply simple exponential smoothing, `ses()`, with a five-year (`h=5`) forecast.

```
# Starts in 1996 and ends in 2013
oildata <- window(oil, start = 1996)
# Estimate parameters
fc <- ses(oildata, h = 5)
# Accuracy of one-step-ahead training errors
round(accuracy(fc), 2)
```

ME	RMSE	MAE	MPE	MAPE	MASE	ACF1	
Training set	6.4	28.12	22.26	1.1	4.61	0.93	-0.03

The `fc` model gives parameter estimates $\alpha = 0.83$ and $\ell_0 = 446.6$, obtained by minimizing SSE over periods $t = 1, 2, \dots, 18$, subject to the restriction that $0 \leq \alpha \leq 1$.

```
fc$model
```

Simple exponential smoothing
Call:
<code>ses(y = oildata, h = 5)</code>
Smoothing parameters:

```

alpha = 0.8339
Initial states:
l = 446.5868
sigma: 29.8282
AIC      AICc      BIC
178.1430 179.8573 180.8141

```

The five-year forecast given by `fc` is:

`fc`

	Point Forecast	Lo 80	Hi 80	Lo 95	Hi 95
2014	542.6806	504.4541	580.9070	484.2183	601.1429
2015	542.6806	492.9073	592.4539	466.5589	618.8023
2016	542.6806	483.5747	601.7864	452.2860	633.0752
2017	542.6806	475.5269	609.8343	439.9778	645.3834
2018	542.6806	468.3452	617.0159	428.9945	656.3667

We now plot the oil data forecast. The purple line in Figure 4-8 is a plot of the data, which shows a changing level over time. The red line is the represents the one-step-ahead fitted values alongside the data over the period 1996–2013. The forecasts for the period 2014–2018 are plotted in Figure 4-8 as the wide band on the right side of the plot. The large value of α in this example is reflected in the large adjustment that takes place in the estimated level ℓ_t at each time. A smaller value of α would lead to smaller changes over time, and so the series of fitted values would be smoother.

Prediction Intervals

The prediction intervals shown here are calculated using the ETS method described later in this chapter. The prediction intervals show that there is considerable uncertainty in the future values of oil production over the five-year forecast period. So, interpreting the point forecasts without accounting for the large uncertainty can be very misleading.

```

autoplot(fc) +
  autolayer(fitted(fc), series = 'Fitted', color= 'red',
            size = 1.2) +
  ylab('Oil (millions of metric tons)') + xlab('Year')

```

Double Exponential Smoothing

Double Exponential Smoothing is an extension to Simple Exponential Smoothing that explicitly adds support for trends in the univariate time series. In addition to the α (alpha) parameter for controlling smoothing

factor for the level, an additional smoothing factor is added to control the decay of the influence of the change in trend called β (beta).

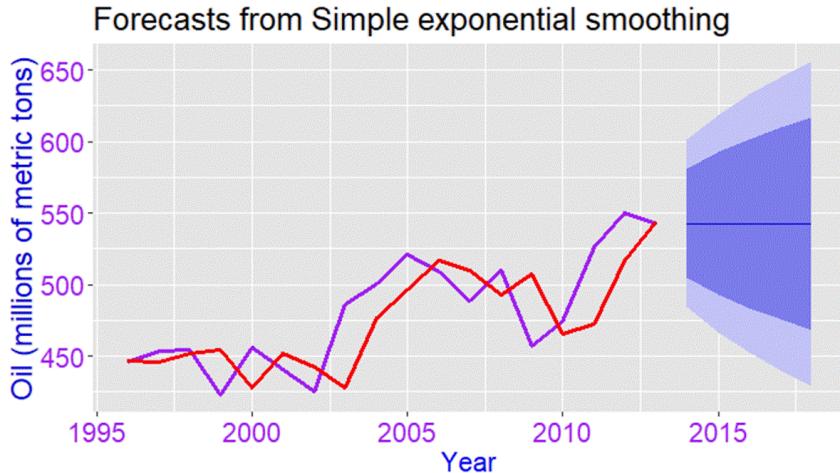


Figure 4-8. Oil time series forecast using simple exponential smoothing

The method supports trends that change in different ways: an additive and a multiplicative, depending on whether the trend is linear or exponential respectively.

Definition 4.5. Given a raw data sequence of observations represented by Y_t , beginning at time $t = 0$, we use ℓ_t to represent the smoothed **level** value for time t , and b_t is our best estimate of the **trend**, β , at time t . Then output of the estimate of the value of y_{t+m} at time $m > 0$ based on the raw data up to time t is now written as \hat{Y}_{t+m} . Then, **double exponential smoothing** is given by

$$\ell_t = \alpha y_t + (1 - \alpha)[\ell_{t-1} + b_{t-1}]$$

with

$$b_t = \gamma [\ell_t - \ell_{t-1}] + (1 - \gamma)b_{t-1}$$

such that

$$\hat{Y}_{t+m} = \ell_{t+m-1} + b_{t+m-1}$$

Is the forecasted estimate for the series Y_t .

Double Exponential Smoothing with an additive trend is classically referred to as Holt's linear trend model, named for the developer of the method, Charles Holt.

Holt's linear trend method

Holt (1957) extended simple exponential smoothing to allow the forecasting of data with a trend. **Holt's Linear Trend** is a type of double exponential smoothing. It computes an evolving trend equation through the data using a special weighting function that places the greatest emphasis on the most recent time periods. This method involves a forecast equation and two smoothing equations (one for the level and one for the trend). This is a form of **double exponential smoothing**.

Definition 4.6. *Holt's Linear Trend* assumes a raw data sequence of observations represented by y_t , beginning at time $t = 0$. We use ℓ_t to represent the smoothed **level** value for time t , and b_t is our best estimate of the **trend** at time t , such that:

$$y_{t+h} = \ell_t + hb_t$$

where ℓ_t and b_t are given by:

$$\ell_t = \alpha y_t + (1 - \alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$$

where α is the smoothing parameter for the level, $0 \leq \alpha \leq 1$, and β is the smoothing parameter for the trend, $0 \leq \beta \leq 1$.

As with simple exponential smoothing, the level equation here shows that ℓ_t is a weighted average of observation y_t and the one-step-ahead training forecast for time t , here given by $\ell_{t-1} + b_{t-1}$. The trend equation shows that b_t is a weighted average of the estimated trend at time t based on $\ell_t - \ell_{t-1}$ and b_{t-1} , the previous estimate of the trend.

The forecast function is no longer flat but trending. The h -step-ahead forecast is equal to the last estimated level plus h times the last estimated trend value. Hence the forecasts are a linear function of h .

For longer range (multi-step) forecasts, the trend may continue on unrealistically. As such, it can be useful to dampen the trend over time.

Dampening means reducing the size of the trend over future time steps down to a straight line (no trend).

Example in R: Air Passengers

Total annual air passengers (in millions) including domestic and international aircraft passengers of air carriers registered in Australia, 1970-2016.

```
air <- window(ausair, start = 1990)
fc <- holt(air, h = 5)
```

The smoothing parameters, α and β^* , and the initial values ℓ_0 and b_0 are estimated by minimizing the SSE for the one-step training error. We can examine these value estimates using `fc$model`, as `alpha`, `beta`, `l`, and `b`, respectively.

```
fc$model
```

```
Holt's method
Call:
holt(y = air, h = 5)
Smoothing parameters:
alpha = 0.8302
beta  = 1e-04
Initial states:
l = 15.5715
b = 2.1017
sigma: 2.3645
      AIC     AICC      BIC
141.1291 143.9863 147.6083
```

The very small value of $\beta^* = 1e-04 = 0.0001$, means that the slope hardly changes over time. We can observe this in Figure 4-9.

```
autoplot(air) +
  ylab('Passengers (in millions)') + xlab('Year') +
  geom_line(color = 'dark red', linetype = 1, size = 1.25)
```

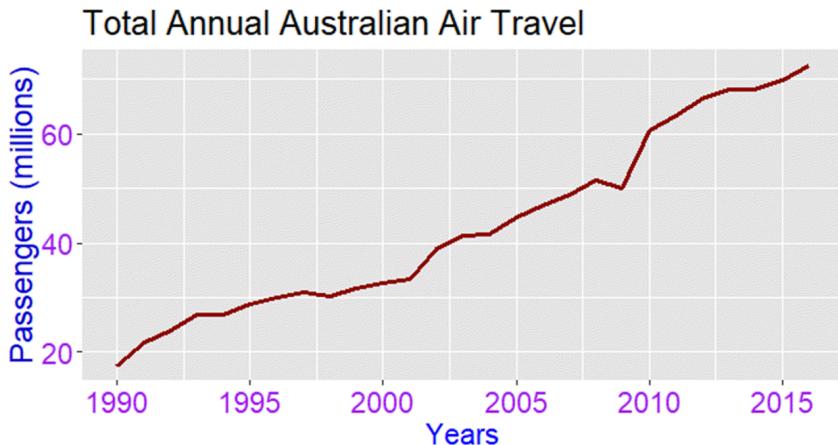


Figure 4-9. Holt smoothing for air passenger time series

fc

	Point Forecast	Lo 80	Hi 80	Lo 95	Hi 95
2017	74.60130	71.57106	77.63154	69.96695	79.23566
2018	76.70304	72.76440	80.64169	70.67941	82.72668
2019	78.80478	74.13092	83.47864	71.65673	85.95284
2020	80.90652	75.59817	86.21487	72.78810	89.02494
2021	83.00826	77.13343	88.88310	74.02348	91.99305

Damped trend methods

The forecasts generated by Holt's linear method display a constant trend (increasing or decreasing) indefinitely into the future. Empirical evidence indicates that these methods tend to over-forecast, especially for longer forecast horizons. Motivated by this observation, Gardner & McKenzie (1985) introduced a parameter that "dampens" the trend to a flat line sometime in the future. Methods that include a damped trend have proven to be very successful, and are arguably the most popular individual methods when forecasts are required automatically for many series.

A damping coefficient φ (*phi*) is used to control the rate of trend dampening. The method supports trend dampening for additive (linear) and multiplicative (exponential) models. Since we are still dealing with trend, this is also **double exponential smoothing**.

Definition 4.7. The **additive damped trend method**, with the smoothing parameters α and β^* with values between 0 and 1 extends Holt's method by including a damping parameter $0 < \varphi < 1$, such that

$$y_{t+h|t} = \ell_t + (\varphi + \varphi^2 + \cdots + \varphi^h)b_t$$

where ℓ_t and b_t are given by:

$$\ell_t = \alpha y_t + (1 - \alpha)(\ell_{t-1} + \varphi b_{t-1})$$

$$b_t = \beta^{(\ell_t - \ell_{t-1})} + (1 - \beta)\varphi b_{t-1}$$

If $\varphi = 1$, the method is identical to Holt's linear method. For values between 0 and 1, φ dampens the trend so that it approaches a constant some time in the future.

In fact, the forecasts converge to $\ell_t + \varphi b_t / (1 - \varphi)$ as $h \rightarrow \infty$ for any value $0 < \varphi < 1$. This means that short-run forecasts are trended while long-run forecasts are constant.

In practice, φ is rarely less than 0.8 as the damping has a very strong effect for smaller values. Values of φ close to 1 will mean that a damped model is not able to be distinguished from a non-damped model. For these reasons, we usually restrict φ to a minimum of 0.8 and a maximum of 0.98.

Example In R: Air Passengers (continued)

Figure 4-10 shows the forecasts for years 2017–2031 generated from Holt's linear trend method and the damped trend method. The parameters of `holt()` are `phi`, the damping parameter and `h`, the forecast period.

```
fc <- holt(air, h = 15)
fc2 <- holt(air, damped = TRUE, phi = 0.9, h = 15)
autoplot(air, size = 1.2, color = 'purple') +
  autolayer(fc, series = 'Holt's method', PI = FALSE,
            color = 'blue', size = 1.2) +
  autolayer(fc2, series = 'Damped Holt's method', PI = FALSE,
            color = 'red', size = 1.2) +
  ggtitle('Forecasts from Holt's method') + xlab('Year') +
  ylab('Air passengers in Australia (millions)') +
  guides(colour = guide_legend(title = 'Forecast'))
```

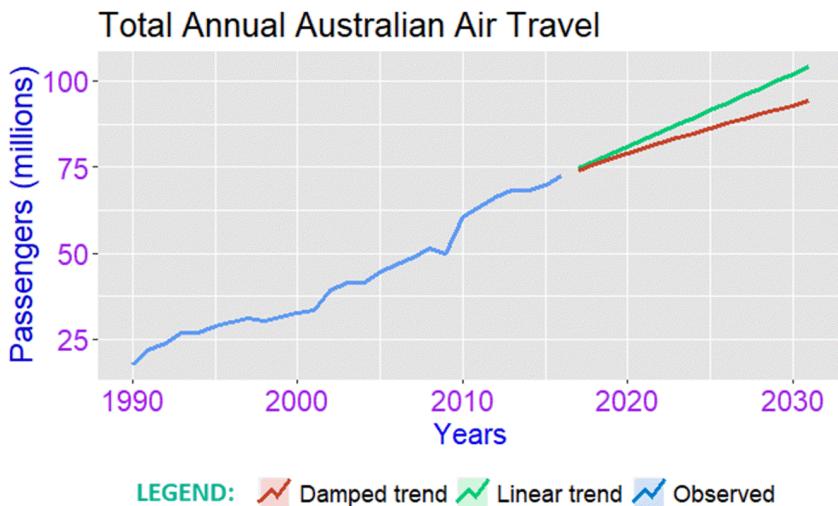


Figure 4-10. Forecasting total annual passengers of air carriers registered in Australia (millions of passengers, 1990–2016). For the damped trend method, $\phi=0.90$.

We have set the damping parameter to a relatively low number ($\phi = 0.90$) to exaggerate the effect of damping for comparison. Usually, we would estimate ϕ along with the other parameters. We have also used a rather large forecast horizon ($h = 15$) to highlight the difference between a damped trend and a linear trend. In practice, we would not normally want to forecast so many years ahead with only 27 years of data.

In this example, we compare the forecasting performance of the three exponential smoothing methods that we have considered so far in forecasting the sheep livestock population in Asia. The data spans the period 1961–2007 and is shown in Figure 4-11.

```
autoplott(livestock) +
  autolayer(livestock, color = 'blue', size = 1.2) +
  xlab('Year') + ylab('Livestock, sheep in Asia (millions)')
```

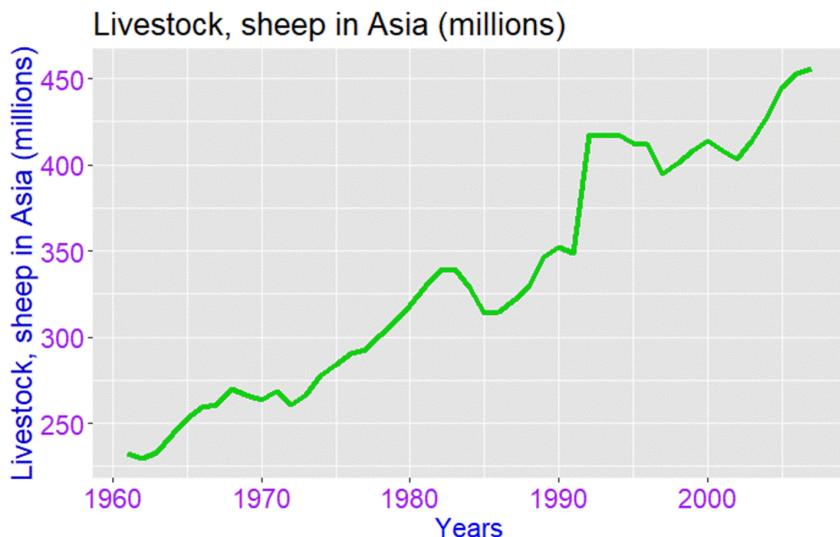


Figure 4-11. Annual sheep livestock numbers in Asia (in million head)

We will use time series cross-validation to compare the one-step forecast accuracy of the three methods.

```
e1 <- tsCV(livestock, ses, h = 1)
e2 <- tsCV(livestock, holt, h = 1)
e3 <- tsCV(livestock, holt, damped=TRUE, h = 1)
# Compare MSE:
mean(e1^2, na.rm = TRUE), mean(e2^2, na.rm = TRUE),
mean(e3^2, na.rm = TRUE)
```

```
[1] 178.2531
[2] 173.365
[3] 162.6274
```

```
# Compare MAE:
mean(abs(e1), na.rm = TRUE), mean(abs(e2), na.rm = TRUE),
mean(abs(e3), na.rm = TRUE)
```

```
[1] 8.53246
[2] 8.803058
[3] 8.024192
```

Damped Holt's method is best whether you compare MAE or MSE values. So, we will proceed with using the damped Holt's method and apply it to the whole data set to get forecasts for future years.

```
fc <- holt(livestock, damped = TRUE)
# Estimated parameters:
fc[['model']]
```

```
Damped Holt's method
Call:
holt(y = livestock, damped = TRUE)
Smoothing parameters:
alpha = 0.9999
beta = 3e-04
phi = 0.9798
Initial states:
l = 223.35
b = 6.9046
sigma: 12.8435
      AIC     AICC      BIC
427.6370 429.7370 438.7379
```

The smoothing parameter for the slope is estimated to be essentially zero, indicating that the trend is not changing over time. The value of α is very close to one, showing that the level reacts strongly to each new observation. We plot the forecast as shown in Figure 4-12.

```
autoplot(fc) +
  autolayer(livestock, color = 'blue', size = 1.2) +
  xlab('Year') + ylab('Livestock, sheep in Asia (millions)')
```

The resulting forecasts look sensible with increasing trend, and relatively wide prediction intervals reflecting the variation in the historical data. The prediction intervals are calculated using the methods described later in the chapter.

In this example, the process of selecting a method was relatively easy as both MSE and MAE comparisons suggested the same method (damped Holt's). However, sometimes different accuracy measures will suggest different forecasting methods, and then a decision is required as to which forecasting method we prefer to use. As forecasting tasks can vary by many dimensions (length of forecast horizon, size of test set, forecast error measures, frequency of data, etc.), it is unlikely that one method will be better than all others for all forecasting scenarios. What we require from a forecasting method are consistently sensible forecasts, and these should be frequently evaluated against the task at hand.



Figure 4-12. Forecasting livestock, sheep in Asia: comparing forecasting performance of non-seasonal method.

In this example, the process of selecting a method was relatively easy as both MSE and MAE comparisons suggested the same method (damped Holt's). However, sometimes different accuracy measures will suggest different forecasting methods, and then a decision is required as to which forecasting method we prefer to use. As forecasting tasks can vary by many dimensions (length of forecast horizon, size of test set, forecast error measures, frequency of data, etc.), it is unlikely that one method will be better than all others for all forecasting scenarios. What we require from a forecasting method are consistently sensible forecasts, and these should be frequently evaluated against the task at hand.

Holt's Exponential Smoothing

If you have a time series that can be described using an **additive model** with **increasing or decreasing trend** and **no seasonality**, you can use Holt's exponential smoothing to make short-term forecasts. **Holt's exponential smoothing** estimates the level and slope at the current time point. Smoothing is controlled by two parameters, α , for the estimate of the level at the current time point, α_t , and b for the estimate of the slope β of the trend component at the current time point, β_t . As with simple

exponential smoothing, the parameters α and β have values between 0 and 1, and values that are close to 0 means that little weight is placed on the most recent observations when making forecasts of future values.

```
# simple exponential - models level
fit1 <- HoltWinters(airpass, beta = FALSE, gamma = FALSE)
# predict next 12 future values
for1 <- forecast::forecast(fit1, h = 12)
# predictive accuracy
accuracy(for1)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train	2.2379	33.7110	25.8606	1.5169	9.1519	0.8064	0.3029

```
# double exponential - models level and trend
fit2 <- HoltWinters(airpass, gamma = FALSE)
# predict next 12 future values
for2 <- forecast::forecast(fit2, h = 12)
# predictive accuracy
accuracy(for2)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train	-3.0685	33.9463	25.9337	-0.6434	8.9179	0.8079	0.3025

```
# triple exponential - models level, trend, and seasonal
# components
fit3 <- HoltWinters(airpass)
# predict next 12 future values
for3 <- forecast::forecast(fit3, h = 12)
# predictive accuracy
accuracy(for3)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train	-3.0685	33.9463	25.9337	-0.6434	8.9179	0.8079	0.3025

Triple Exponential Smoothing

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

Holt-Winters Exponential Smoothing

Holt-Winters exponential smoothing estimates the level, slope and seasonal component at the current time point. Smoothing is controlled by three parameters: alpha, beta, and gamma, for the estimates of the level, slope b of the trend component, and the seasonal component,

respectively, at the current time point. The parameters alpha, beta and gamma all have values between 0 and 1, and values that are close to 0 mean that relatively little weight is placed on the most recent observations when making forecasts of future values.

Now that we see what smoothing does, we will perform it on data with more noise than the monthly milk time series. Moreover, we will use a different smoothing method, called Holt-Winters. There are two types of Holt-Winters smoothing, additive and multiplicative. Additive is the default in R.

Additive Holt-Winters

The additive method (**Definition 4.8**) is preferred when the seasonal variations are roughly constant through the series. With the additive method, the seasonal component is expressed in absolute terms in the scale of the observed series, and in the level equation the series is seasonally adjusted by subtracting the seasonal component. Within each year, the seasonal component will add up to approximately zero. This is called triple exponential smoothing. One form of this smoothing method is the Holt-Winters additive model.

Definition 4.8. The **additive Holt-Winters (HW)** prediction function, for time series with period length h , is

$$\hat{y}_{t+h} = \ell_t + hb_t + s_{t+h-m(k+1)},$$

where ℓ_t , b_t and s_t are given by:

$$\ell_t = \alpha(y_t - s_{t-m}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$$

$$s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1 - \gamma)s_{t-m}$$

where k is the integer part of $(h - 1)/m$, which ensures that the estimates of the seasonal indices used for forecasting come from the final year of the sample.

The level equation shows a weighted average between the seasonally adjusted observation $(y_t - s_{t-m})$ and the non-seasonal forecast $(\ell_{t-1} + b_{t-1})$ for time t . The trend equation is identical to Holt's linear method.

The seasonal equation shows a weighted average between the current seasonal index, $(y_t - \ell_{t-1} - b_{t-1})$, and the seasonal index of the same season last year (i.e., m time periods ago).

The multiplicative method (**Definition 4.9**) is preferred when the seasonal variations are changing proportional to the level of the series. With the multiplicative method, the seasonal component is expressed in relative terms (percentages), and the series is seasonally adjusted by dividing through by the seasonal component. Within each year, the seasonal component will sum up to approximately m .

Definition 4.9. *The multiplicative Holt-Winters prediction function (for time series with period length h) is*

$$\hat{y}_{t+h} = (\ell_t + h b_t) s_{t+h-m(k+1)},$$

where ℓ_t , b_t and s_t are given by:

$$\ell_t = \alpha \left(\frac{y_t}{s_{t-m}} \right) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$$

$$s_t = \gamma \left(\frac{y_t}{\ell_{t-1} - b_{t-1}} \right) + (1 - \gamma)s_{t-m}$$

The data in x are required to be non-zero for a multiplicative model, but it makes most sense if they are all positive. The function tries to find the optimal values of α and/or β^* and/or γ by minimizing the squared one-step prediction error if they are **NULL** (the default). **optimize** will be used for the single-parameter case, and **optim** otherwise.

For seasonal models, start values for ℓ , b and s are inferred by performing a simple decomposition in trend and seasonal component using moving averages (recall the **decompose()** function) on the **start.periods** first periods (a simple linear regression on the trend component is used for starting level and trend). For level/trend-models (no seasonal component), start values for ℓ and b are y_2 and $y_2 - y_1$, respectively. For level-only models (ordinary exponential smoothing), the start value for a is y_1 .

Got a Job (HW Smoothing)?

Smoothing of time series data is often used in economics, so we will explore a time series from the Consumer Confidence Index (CCI) on Unemployment.

The data we will use for smoothing is comprised of 100 monthly observations on the consumer confidence index (CCI) and seasonally adjusted civilian unemployment (`unemp`) in the US, covering the period June 1997 – September 2005. The third column is a "terrorism" indicator variable taking value "1" from September 2001. The `unemp.cci` dataset is part of the `expsmooth` package in R (Hyndman, expsmooth: Data Sets from "Forecasting with Exponential Smoothing", 2015). First, we merely plot the data (see Figure 4-13).

```
data(unemp.cci)
cci <- ts(unemp.cci[, 'cci'], start = c(1997))
plot.ts(cci, main = 'Consumer Cost Index', col = 2)
```



Figure 4-13. The plotted time series data for Consumer Cost Index (CCI)

Using the `forecast package` in R, we enter the following code for simple exponential smoothing (Hyndman, et al., 2015), which implements Holt-Winters exponential smoothing without trend and seasonal component.

```
cci.smooth <- HoltWinters(cci, beta = FALSE, gamma = FALSE)
cci.smooth
```

```

Call:
HoltWinters(x = cci, beta = FALSE, gamma = FALSE)

Smoothing parameters:
alpha: 0.9939455
beta : FALSE
gamma: FALSE

Coefficients:
[,1]
a 87.60891

```

In Figure 4-14, we produce a plot of the smooth, fitted data.

```
plot(cci.smooth$fitted, col = 4)
```

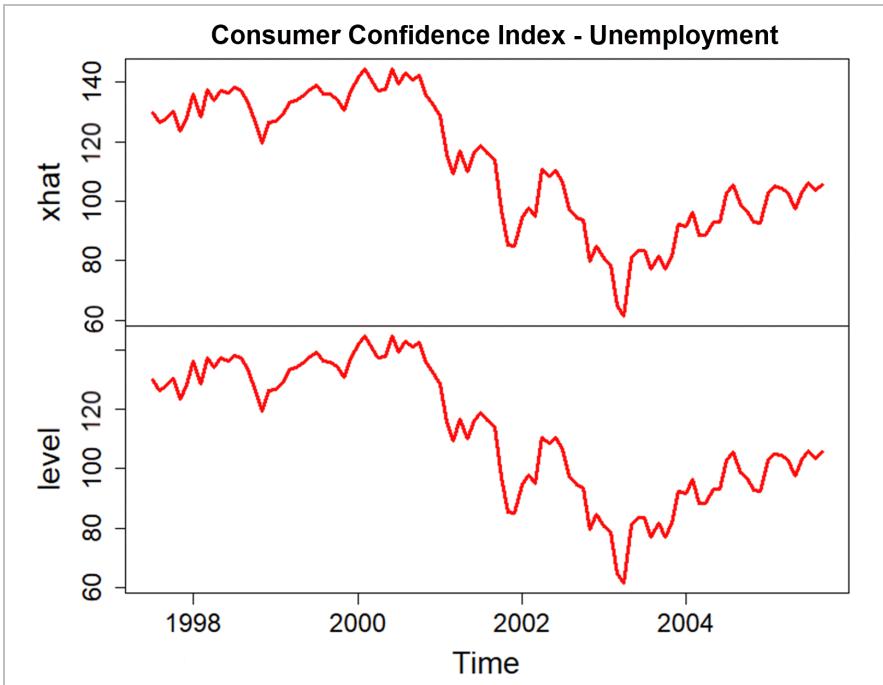


Figure 4-14. Smooth fitted plot of the CCI data

The following R code performs double-exponential smoothing:

```

cci.smother <- HoltWinters(cci, gamma = FALSE)
cci.smother

```

Holt-Winters exponential smoothing with trend and without seasonal component.

```
Call:  
HoltWinters(x = cci, gamma = FALSE)  
Smoothing parameters:  
alpha: 1  
beta : 0.04122861  
gamma: FALSE  
Coefficients:  
[,1]  
a 87.5000000  
b -0.5619308
```

In Figure 4-15, we plot the smooth, fitted data from the double exponential smoothing.

```
plot(cci.smother$fitted, col = 'purple')
```

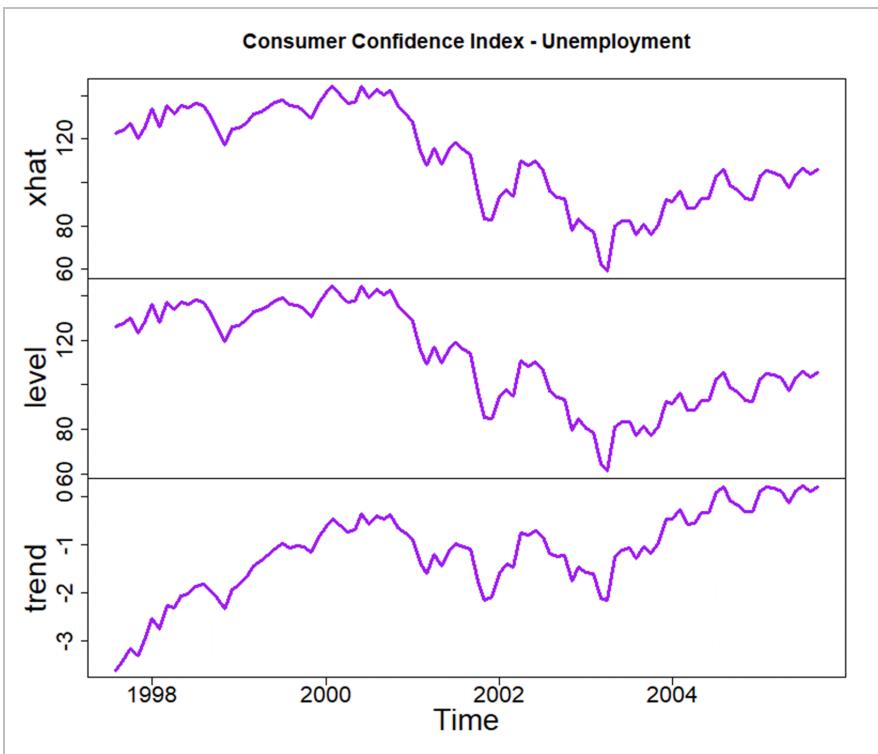


Figure 4-15. Smooth, fitted data from double exponential smoothing

To improve our confidence in this result, we check the residuals as follows:

```
fit1 <- HoltWinters(diff(loans.ts, differences=1), beta=FALSE,  
gamma = FALSE)
```

```
for1<-forecast(fit1, h = 48)
acf(for1$residuals, lag.max = 48)
pacf(for1$residuals, lag.max = 48)
Box.test(for1$residuals, lag = 48, type = 'Ljung-Box')
```

```
Box-Ljung test
```

```
data: for1$residuals
X-squared = 186.5272, df = 48, p-value < 2.2e-16
```

Now, we will use the `ses` package to compare the Holt-Winters additive and multiplicative forecasts.

```
cci<-window(unemp.cci[, 'cci'], start = 1997)
fit_1<-hw(cci, seasonal = 'additive')
fit_2<-hw(cci, seasonal = 'multiplicative')
fit_1$model
```

```
Holt-Winters' additive method
```

```
Call:
hw(y = cci, seasonal = "additive")
Smoothing parameters:
alpha = 0.9541
beta = 1e-04
gamma = 1e-04
Initial states:
l = 140.123
b = -0.3125
s = 3.7955 0.9928 -0.9429 -2.7646 -0.4567 0.1421
-2.9483 -7.3985 -1.5967 3.4772 3.5556 4.1444
sigma: 6.1632
      AIC     AICc      BIC
840.8015 848.2649 885.0894
```

```
fit_2$model
```

```
Holt-Winters' multiplicative method
```

```
Call:
hw(y = cci, seasonal = "multiplicative")
Smoothing parameters:
alpha = 0.4226
beta = 1e-04
gamma = 1e-04
Initial states:
l = 140.9633
b = -0.1169
s = 1.0378 1.0095 0.9931 0.9863 1.0044 1.0125
0.9701 0.9328 0.9852 1.014 1.0132 1.041
sigma: 0.0729
```

AIC	AICc	BIC
895.2973	902.7607	939.5852

Finally, we compare the forecasts, as seen in Figure 4-16.

```
autoplot(cci) +
  autolayer(cci, color = 'blue', size = 1.25) +
  autolayer(fit_1,series= 'Additive', PI = FALSE,size = 1.25) +
  autolayer(fit_2,series = 'Multiplicative', PI = FALSE,
            size = 1.25) +
  xlab('Year') +
  ylab('Unemployment Index') +
  ggtitle('Consumer Confidence Index (CCI)') +
  guides(colour = guide_legend(title = 'HW Forecast'))
```

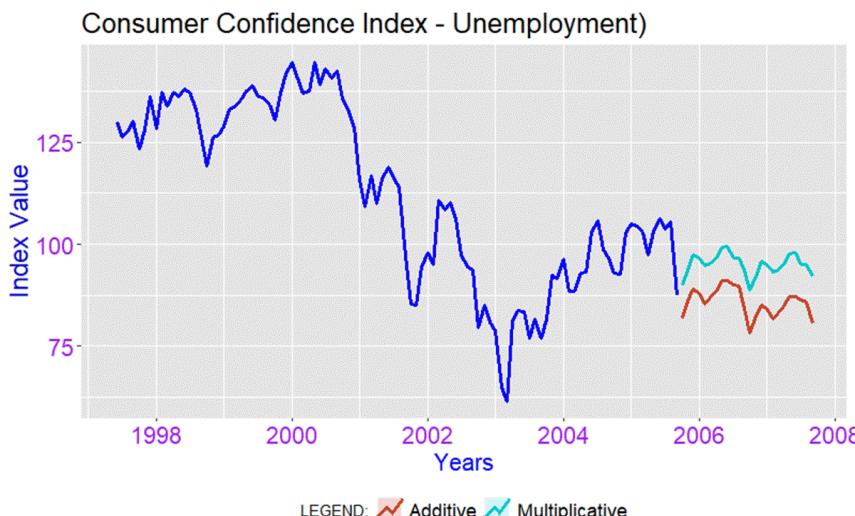


Figure 4-16. Forecasting the Consumer Confidence Index for Unemployment using the Holt-Winters method with both additive and multiplicative seasonality.

Since both models have the same number of parameters to estimate, compare the root mean square error (RMSE) from both models.

```
rmse(fit_1$fitted, fit_1$lower)
```

```
[1] 88.21825
```

```
rmse(fit_2$fitted, fit_2$lower)
```

```
[1] 79.6556
```

The RMSE comparison shows that the multiplicative model is more appropriate.

Holt-Winters' damped method

Damping is possible with both additive and multiplicative Holt-Winters' methods. A method that often provides accurate and robust forecasts for seasonal data is the Holt-Winters method with a damped trend and multiplicative seasonality.

Definition 4.10. The **damped Holt-Winters prediction function** (for time series with period length h) with damping factor ϕ is

$$Y_{t+h} = [\ell_t + (\phi + \phi^2 + \cdots + \phi^h)b_t]s_{t+h-m(k+1)}$$

where ℓ_t , b_t , and s_t are given by:

$$\ell_t = \alpha \left(\frac{y_t}{s_{t-m}} \right) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$$

$$b_t = \beta^{*(\ell_t - \ell_{t-1})} + (1 - \beta^*)\phi b_{t-1}$$

$$s_t = \gamma \frac{y_t}{\ell_{t-1} + \phi b_{t-1}} + (1 - \gamma)s_{t-m}.$$

Got a Job (HW Dampening)?

Smoothing of time series data is often used in economics, so we will explore a time series from the Consumer Confidence Index (CCI) on Unemployment. We can compare the six exponential smoothing methods we have studied so far.

```
Cci<-window(unemp.cci[, "cci"], start = 1998)
f1<-hw(cci, seasonal = "additive")
f2<-hw(cci, seasonal = "multiplicative")
f3<-hw(cci, damped = TRUE, seasonal = "multiplicative")
f4<-hw(cci, damped = TRUE, seasonal = "additive")
f5<-holt(cci, h = 24, damped = TRUE, exponential = TRUE)
f6<-holt(cci, h = 24, initial = "optimal")
```

Plotting these smoothed series together, as seen in Figure 4-17, allows us to compare forecasts as we look at the general trend of the original time series.

```
autoplott(cci) +
```

```

autolayer(cci, color = "blue", size = 1.25) +
autolayer(f1, series = "HW Additive", PI = FALSE,
          size = 1.25) +
autolayer(f2, series = "HW Multiplicative", PI = FALSE,
          size = 1.25) +
autolayer(f3, series = "HW Multiplicative damped", PI = FALSE,
          size = 1.25) +
autolayer(f4, series = "HW Additive damped", PI = FALSE,
          size = 1.25) +
autolayer(f5, series = "Holt damped", PI = FALSE,
          size = 1.25) +
autolayer(f6, series = "Holt", PI = FALSE, size = 1.25) +
xlab("Year") +
ylab("Unemployment Index") +
ggtitle("Consumer Confidence Index (CCI)") +
guides(colour=guide_legend(title = "LEGEND")) +
theme(legend.position = "bottom")

```

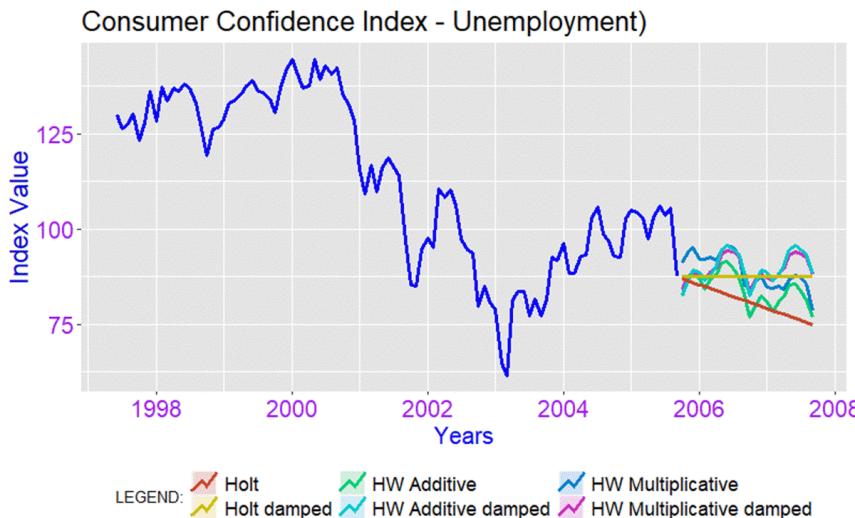


Figure 4-17. Unemployment forecasts comparison of six exponential smoothing models.

We can also compare the forecast by their RMSE values.

```

r1 <- rmse(f1$fitted,f1$lower)
r2 <- rmse(f2$fitted,f2$lower)
r3 <- rmse(f3$fitted,f3$lower)
r4 <- rmse(f4$fitted,f4$lower)
r5 <- rmse(f5$fitted,f5$lower)
r6 <- rmse(f6$fitted,f6$lower)
print(paste0("HW Additive RMSE = ", r1))

```

```

print(paste0("HW Multiplicative RMSE = ", r2))
print(paste0("HW Multiplicative damped RMSE = ", r3))
print(paste0("HW Additive damped RMSE = ", r4))
print(paste0("Holt damped RMSE = ", r5))
print(paste0("Holt RMSE = ", r6))

```

```

[1] "HW Additive RMSE = 84.889212388019"
[2] "HW Multiplicative RMSE = 89.0760357507311"
[3] "HW Multiplicative damped RMSE = 82.83636863179"
[4] "HW Additive damped RMSE = 83.0661493743828"
[5] "Holt damped RMSE = 83.1382755315174"
[6] "Holt RMSE = 88.6627637742808"

```

Output line 3 should make clear that the Holt-Winters multiplicative damped model provides the most suitable forecast, given RMSEs. Moreover, we can look at the “complete” set of accuracy metrics and make comparisons.

```
print(paste0("HW Additive RMSE:")); round(accuracy(f1),4)
```

```

[1] "HW Additive RMSE:"
      ME      RMSE      MAE      MPE      MAPE      MASE      ACF1
Train set -0.0678  5.6593  4.3944 -0.2331  4.3225  0.3062 -0.0030

```

```
print(paste0("HW Multiplicative RMSE:")); round(accuracy(f2),4)
```

```

[1] "HW Multiplicative RMSE:"
      ME      RMSE      MAE      MPE      MAPE      MASE      ACF1
Train set  0.2529  7.0144  5.5838  0.0704  5.5014  0.3891  0.4167

```

```
print(paste0("HW Multiplicative damped:"));round(accuracy(f3),4)
```

```

[1] "HW Multiplicative damped:"
      ME      RMSE      MAE      MPE      MAPE      MASE      ACF1
Train set -0.6309  5.7683  4.4830 -0.7383  4.4150  0.3124  0.0635

```

```
print(paste0("HW Additive damped RMSE:"));round(accuracy(f4),4)
```

```

[1] "HW Additive damped RMSE:"
      ME      RMSE      MAE      MPE      MAPE      MASE      ACF1
Train set -0.5832  5.6259  4.3615 -0.7036  4.3013  0.3039 -0.0086

```

```
print(paste0("Holt damped RMSE:")); round(accuracy(f5), 4)
```

```

[1] "Holt damped RMSE:"
      ME      RMSE      MAE      MPE      MAPE      MASE      ACF1
Train set -0.4323  6.4137  4.8564 -0.6362  4.7808  0.3384  0.0117

```

```
print(paste0("Holt RMSE:")); round(accuracy(f6), 4)
```

```
[1] "Holt RMSE:"  
      ME    RMSE     MAE     MPE     MAPE     MASE     ACF1  
Train set -0.0219 6.4033  4.8449 -0.2295  4.7518 0.3376  0.0066
```

In this instance, the Holt-Winters multiplicative damped model and the Holt-Winters additive damped model seem to be equally appropriate for the forecast. Notice that these two models nearly lay on top of one another in Figure 4-17.

Exponential Smoothing Taxonomy

So far, we have presented six exponential smoothing methods. Each method has specific use cases where they may be appropriate. We can name these methods using a pair of letters (T, S) defining the type of ‘Trend’ and ‘Seasonal’ components. For example, (A, N) is the method with an additive trend and no seasonality. Table 4-1 summarizes the taxonomy for methods we have covered. Notice that the two-letter nomenclature (shorthand) is more descriptive than the method name, in that it provides an indication

Table 4-1. Exponential smoothing naming taxonomy

Abbr.	Method	Components
(N,N)	Simple exponential smoothing	Level only
(A,N)	Holt’s linear trend method	Level & Trend
(A _d ,N)	Additive damped trend method	Level & Trend
(A,A)	Additive Holt-Winters’ method	Level, Trend, & Season
(A,M)	Multiplicative Holt-Winters’ method	Level, Trend, & Season
(A _d ,A)	Holt-Winters’ Additive damped method	Level, Trend, & Season
(A _d ,M)	Holt-Winters’ Multiplicative damped method	Level, Trend, & Season

There are two additional methods, no trend and additive seasonal (N,A) and no trend and multiplicative seasonal (N,M), that we will not cover. However, there are still more methods which different nomenclature schemes, including the Double-exponential moving average (DEMA) and the Zero lag exponential moving average (ZLEMA). There are also additional moving average methods, such as the Elastic volume-weighted moving average (EVWMA), the Volume-weighed moving average (VWMA), the Volume-weighted average price (VWAP), the

Variable-length moving average (VWA), the Hull moving average (HMA), and the Arnaud Legoux moving average (ALMA).

There is also another taxonomy for smoothing models that follow the Error, Trend, Seasonal (ETS) nomenclature, which we will look at in Chapter 5.

Smoothing Oscillation

This example uses the *aTSA package* for exponential smoothing. We begin by loading the *aTSA* and the *soi* dataset.

```
library(astsa)
library(forecast)
data("soi")
```

Configure the data into a time series from Jan 1950 (i.e., `c(1950, 1)`) to December 1995 (i.e., `c(1995, 12)`), and a subset of the time series from January 1985 to December 1995 as indicated in the R code below. We will use the subset (*myts2*) for our analysis.

```
# from Jan 1950 to Dec 1995 as a time series object
Myvector <- soi
soi1.ts <- ts(myvector, start = c(1950, 1), end = c(1995, 12),
               frequency = 12)
# subset the time series (Jan 1985 to December 1995)
soi2.ts <- window(myvector, start = c(1985, 1),
                   end = c(1995, 12))
myts2 <- window(soi2.ts, start = c(1985, 1),
                  end = c(1995, 12))
```

Plotting the Data

Now we plot the time series data (see Figure 4-18).

```
autoplott(soi2.ts) +
  geom_line(series=soi2.ts, lwd=1.25, col='magenta4') +
  ggtitle('Southern Oscillation Index (SOI)') +
  xlab('Year') + ylab("Oscillation") +
  theme(plot.title=element_text(size=rel(1.5))) +
  theme(axis.title.x=element_text(siz=rel(1.2),color='blue')) +
  theme(axis.title.y=element_text(siz=rel(1.2),color='blue')) +
  theme(panel.background=element_rect(fill='grey90', size=1.1),
        axis.text=element_text(color='purple', size=rel(1.05))) +
  theme(axis.line=element_line(size=2, color='grey80'))
```

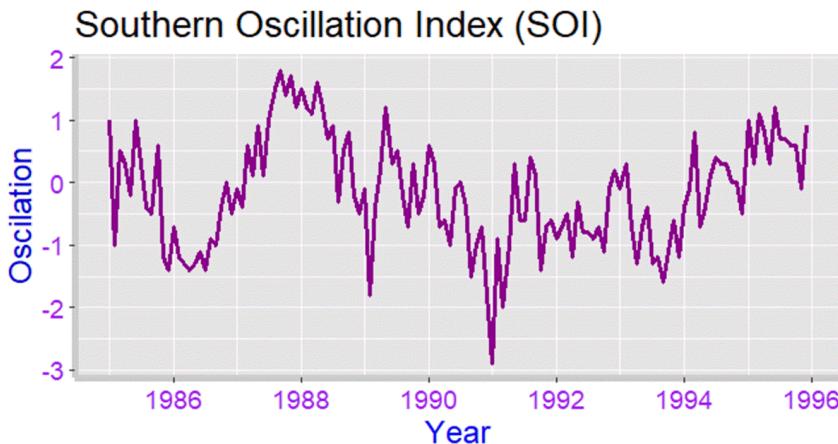


Figure 4-18. Southern Oscillation Index (SOI) time series data

If we consider the plots we have made thus far, there are several commonalities when it comes to aesthetics. We can eliminate the need to rewrite the same code over and over by define our aesthetic parameters.

Let's take our most recent plot and configure all of the theme code as one plotting object, `theme_01`, and use it in subsequent plots when appropriate:

```
Theme_01<- (theme(plot.title=element_text(size=rel(1.5))) +
  theme(axis.title.x=element_text(siz=rel(1.2),color='blue')) +
  theme(axis.title.y=element_text(siz=rel(1.2),color='blue')) +
  theme(panel.background=element_rect(fill='grey90', size=1.1),
    axis.text=element_text(color='purple', size=rel(1.05))) +
  theme(axis.line=element_line(size = 2, color = 'grey80')))
```

Seasonal Decomposition

The data appears to have seasonality in it based on our plot. We will use the `stl()` function to show the components of the time series data (see Figure 4-19). This is a good place to incorporate `theme_01`. This replaces five or six lines of code we would have to rewrite.

```
fit1 <- stl(soi2.ts, s.window = 'period')
autoplot(fit1) +
  geom_line(series = soi2.ts, lwd = 1, col = 'magenta4') +
  ggtitle('Southern Oscillation Index (SOI)') +
  theme_01
```

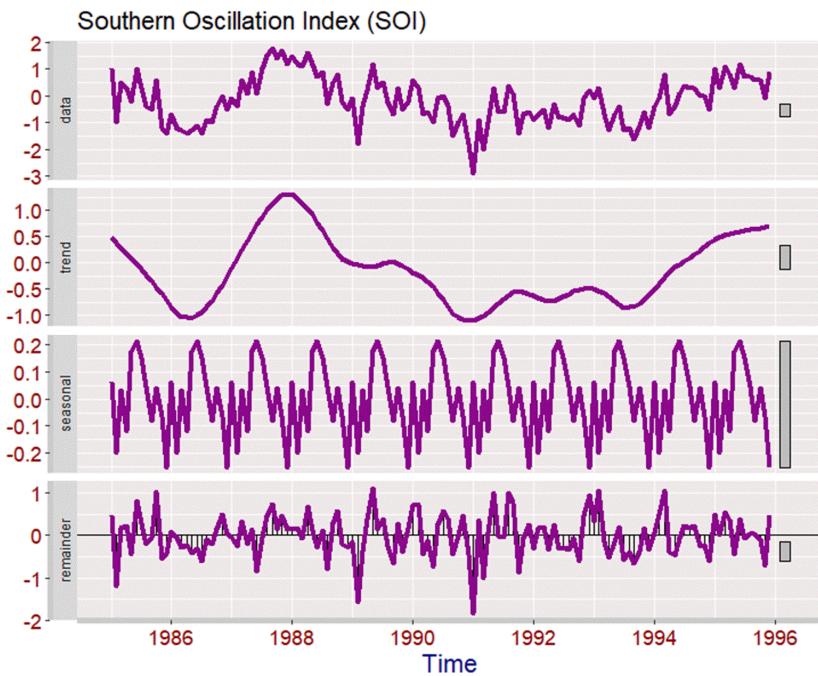


Figure 4-19. Southern Oscillation Index (SOI) seasonal decomposition

The plot shows the seasonality in the data, so we use other plots to examine this. In Figure 4-20, we use a `ggmonthplot()` from the `ggplot2` package. To create this plot, we used an abundance of enhancements using the `theme()` function, as we did with Figure 4-19.

```
library('ggplot2')
ggmonthplot(soi2.ts, col=brewer.pal(n = 12, name = 'Paired'),
year.labels=TRUE) + geom_line(lwd = 1, col = 'slateblue3') +
  ggtitle('Southern Oscillation Index (SOI) Monthly Averages') +
  ylab('Oscillation Values') +
  theme(panel.background = element_rect(fill = 'azure'),
    panel.grid =element_line(color = 'dodgerblue2',
      linetype = 3),
    plot.title =element_text(color = 'slateblue4',
      size = rel(1.25), face = 'bold'),
    axis.title.x = element_text(color='slateblue4',
      size = rel(1.15), face = 'bold'),
    axis.title.y = element_text(color='slateblue4',
      size = rel(1.15), face = 'bold'),
    axis.text.x = element_text(color = 'dodgerblue3',
      size = rel(1), face = 'bold'),
    axis.text.y = element_text(color = 'dodgerblue3',
```

```
size = rel(1), face = 'bold'),  
axis.line = element_line(color = 'dodgerblue4'))
```

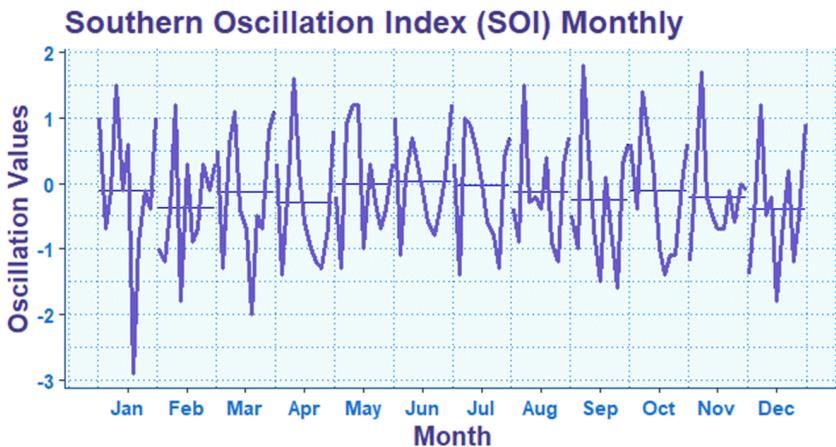


Figure 4-20. Southern Oscillation Index (SOI) monthly plot

The month plot (top figure) displays the subseries for each month (all January values connected, all February values connected, and so on), along with the average of each subseries. From observing the mean values in Figure 4-20, it appears that the months appear to fluctuate in a random manner. Additionally, there is no pattern in the monthly maxes and minima from month to month.

The season plot in Figure 4-21 displays the subseries by year. Again, we see no apparent pattern, with increases and decreases in oscillations each year. So, it looks like there is no seasonal pattern.

```
library('RColorBrewer')  
ggseasonplot(soi2.ts, col=brewer.pal(n = 12, name = 'Set3'),  
             year.labels = TRUE, font = 2) +  
  ggtitle('Southern Oscillation Index (SOI) Monthly Values') +  
  geom_line(lwd = 1.25) + ylab("Oscillation Values") +  
  theme(text=element_text(face = 'bold'),  
        panel.background = element_rect(fill = 'royalblue2'),  
        panel.grid = element_line(color = 'lightblue1',  
                                  linetype = 3))
```

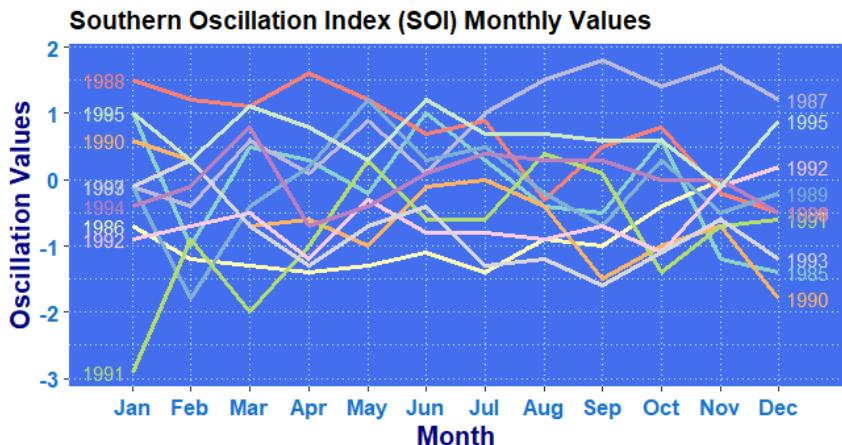


Figure 4-21. Southern Oscillation Index (SOI) seasonal plot

The season plot shows much variation over the years, with no particular pattern. The analysis of the seasonal decomposition of oscillation data shows a lot of variance among the years, so a simple approach is to try various levels of exponential smoothing, including single, double and triple.

Single Exponential Smoothing

Recall that we perform single exponential smoothing (SES) on a time series with univariate data and without a trend or seasonality. It requires a single parameter, called alpha (α), also called the smoothing factor or smoothing coefficient.

This parameter controls the rate at which the influence of the observations at prior time steps decay exponentially. Alpha is often set to a value between 0 and 1. Large values mean that the model pays attention mainly to the most recent past observations, whereas smaller values mean more of the history is taken into account when making a prediction.

```
# simple exponential: models have a level component
fit1 <- HoltWinters(soi2.ts, beta = FALSE, gamma = FALSE)
# predict next 12 future values & summarize the forecast
for1 <- forecast(fit1, 12)
summary(for1)
```

Forecast method: HoltWinters

Model Information:

Holt-Winters exponential smoothing without trend and without seasonal component.

Call:

```
HoltWinters(x = soi2.ts, beta = FALSE, gamma = FALSE)
```

Smoothing parameters:

```
alpha: 0.4649429  
beta : FALSE  
gamma: FALSE
```

Coefficients:

```
[,1]  
a 0.5769332
```

Error measures:

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	-0.00695	0.64913	0.49972	NaN	Inf	0.50689	0.0012032

Forecasts:

	Point Forecast	Lo 80	Hi 80	Lo 95	Hi 95
Jan 1996	0.5769332	-0.2581038	1.411970	-0.7001456	1.854012
Feb 1996	0.5769332	-0.3439471	1.497813	-0.8314316	1.985298
Mar 1996	0.5769332	-0.4224438	1.576310	-0.9514819	2.105348
Apr 1996	0.5769332	-0.4952086	1.649075	-1.0627662	2.216632
May 1996	0.5769332	-0.5633395	1.717206	-1.1669634	2.320830
Jun 1996	0.5769332	-0.6276230	1.781489	-1.2652765	2.419143
Jul 1996	0.5769332	-0.6886455	1.842512	-1.3586024	2.512469
Aug 1996	0.5769332	-0.7468580	1.900724	-1.4476307	2.601497
Sep 1996	0.5769332	-0.8026164	1.956483	-1.5329058	2.686772
Oct 1996	0.5769332	-0.8562070	2.010073	-1.6148655	2.768732
Nov 1996	0.5769332	-0.9078646	2.061731	-1.6938691	2.847735
Dec 1996	0.5769332	-0.9577845	2.111651	-1.7702149	2.924081

Now, we plot the series with forecast in Figure 3-22 using the `autoplot()` function.

```
autoplot(fit1$x, size = 1, color = 'purple') +  
  autolayer(for1, color = 'blue', size = 1.2) +  
  ggtitle('Southern Oscillation Index (SOI)') +  
  xlab('Year') + ylab('Oscillation Values') +  
  theme(panel.background = element_rect(fill = 'snow2'),  
        plot.title = element_text(size = rel(1.2),  
                                  color = 'navy', face = 'bold'),  
        axis.title.x = element_text(size = rel(1.2), color='navy'),  
        axis.title.y = element_text(size = rel(1.2), color='navy'),  
        axis.text = element_text(colour = 'red4', size= rel(0.95)),  
        axis.line = element_line(size = 2, colour = 'grey80'))
```

Southern Oscillation Index (SOI) Forecast Single Smoothing

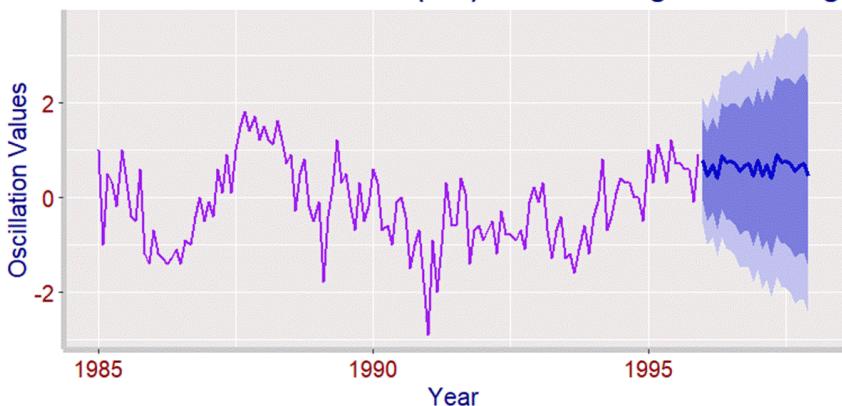


Figure 4-22. SOI forecast using single exponential smoothing

Double Exponential Smoothing

Simple exponential smoothing does not do well when there is a **trend** in the data, which is inconvenient. In such situations, several methods were devised under the name “**double exponential smoothing**” or “second-order exponential smoothing,” which is the recursive application of an exponential filter twice, thus being termed “double exponential smoothing”. This nomenclature is similar to quadruple exponential smoothing, which also references its recursion depth. The basic idea behind double exponential smoothing is to introduce a term to take into account the possibility of a series exhibiting some form of trend. This slope component is itself updated via exponential smoothing. Double exponential smoothing employs a level component and a trend component at each period. Double exponential smoothing uses two weights, (also called smoothing parameters), to update the components at each period.

```
# double exponential: models level and trend components
fit2 <- HoltWinters(soi2.ts, gamma=FALSE)
for2 <- forecast(fit2, 12)
```

Forecast method: HoltWinters

Model Information:

Holt-Winters exponential smoothing without trend and without seasonal component.

```

Call:
HoltWinters(x = soi2.ts, beta = FALSE, gamma = FALSE)

Smoothing parameters:
alpha: 0.4649429
beta : FALSE
gamma: FALSE

Coefficients:
[ ,1]
a 0.5769332

Error measures:
      ME     RMSE     MAE MPE MAPE     MASE     ACF1
Training set -0.006946 0.64913 0.49972 NaN Inf 0.50689 0.001203

Forecasts:
    Point Forecast     Lo 80     Hi 80     Lo 95     Hi 95
Jan 1996      0.5769332 -0.2581038 1.411970 -0.7001456 1.854012
Feb 1996      0.5769332 -0.3439471 1.497813 -0.8314316 1.985298
Mar 1996      0.5769332 -0.4224438 1.576310 -0.9514819 2.105348
Apr 1996      0.5769332 -0.4952086 1.649075 -1.0627662 2.216632
May 1996      0.5769332 -0.5633395 1.717206 -1.1669634 2.320830
Jun 1996      0.5769332 -0.6276230 1.781489 -1.2652765 2.419143
Jul 1996      0.5769332 -0.6886455 1.842512 -1.3586024 2.512469
Aug 1996      0.5769332 -0.7468580 1.900724 -1.4476307 2.601497
Sep 1996      0.5769332 -0.8026164 1.956483 -1.5329058 2.686772
Oct 1996      0.5769332 -0.8562070 2.010073 -1.6148655 2.768732
Nov 1996      0.5769332 -0.9078646 2.061731 -1.6938691 2.847735
Dec 1996      0.5769332 -0.9577845 2.111651 -1.7702149 2.924081

```

We now plot the forecast for double exponential smoothing model (see Figure 4-23).

```

autoplot(fit2$x, size= 1, color= 'purple') +
  autolayer(for2, color='blue', size = 1.2) +
  ggtitle('Southern Oscillation Index (SOI)') +
  xlab('Year') + ylab('Oscillation Values') +
  theme(panel.background = element_rect(fill = 'snow2'),
        plot.title = element_text(size = rel(1.2),
                                   color='navy', face='bold'),
        axis.title.x = element_text(size = rel(1.2), color='navy'),
        axis.title.y = element_text(size = rel(1.2), color='navy'),
        axis.text = element_text(colour = 'red4', size= rel(0.95)),
        axis.line = element_line(size = 2, colour = 'grey80'))

```

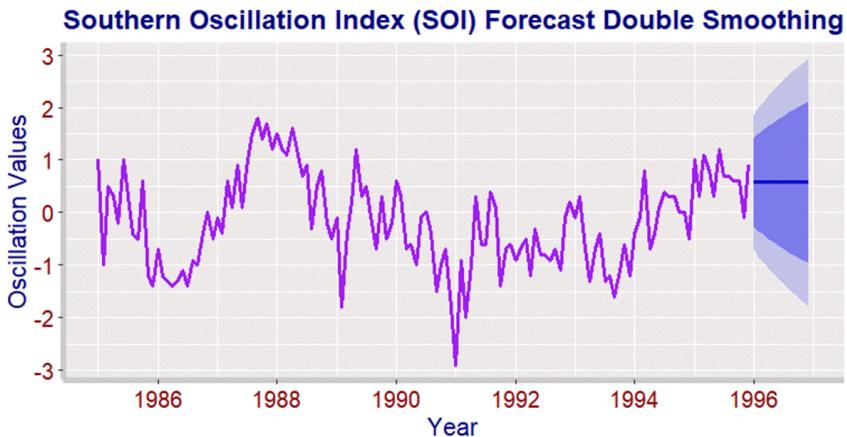


Figure 4-23. Forecast for the double exponential smoothing model.

Triple Exponential Smoothing

As we have already mentioned, [Definition 4.9](#), the Holt-Winters exponential smoothing estimates the level, slope and seasonal component where we can apply our R script to the oscillation problem.

```
fit3 <- HoltWinters(soi2.ts)
for3 <- forecast(fit3, 12)
```

```
Forecast method: HoltWinters
Model Information:
Holt-Winters exponential smoothing with trend and without seasonal component.
Call:
HoltWinters(x = soi2.ts, gamma = FALSE)
Smoothing parameters:
alpha: 0.6881499
beta : 0.2981243
gamma: FALSE
Coefficients:
[,1]
a 0.60104607
b 0.02699741

Error measures:
          ME      RMSE      MAE MPE MAPE      MASE      ACF1
Training set 0.076003 0.80883 0.6196 NaN  Inf 0.62850 -0.0013022

Forecasts:
          Point Forecast      Lo 80      Hi 80      Lo 95      Hi 95
Jan 1996       0.6280435 -0.4079140 1.664001 -0.9563168 2.212404
```

Feb 1996	0.6550409	-0.7340665	2.044148	-1.4694156	2.779497
Mar 1996	0.6820383	-1.1136687	2.477745	-2.0642586	3.428335
Apr 1996	0.7090357	-1.5378241	2.955896	-2.7272399	4.145311
May 1996	0.7360331	-2.0010100	3.473076	-3.4499131	4.921979
Jun 1996	0.7630306	-2.4994808	4.025542	-4.2265500	5.752611
Jul 1996	0.7900280	-3.0305013	4.610557	-5.0529673	6.633023
Aug 1996	0.8170254	-3.5919579	5.226009	-5.9259327	7.559983
Sep 1996	0.8440228	-4.1821477	5.870193	-6.8428416	8.530887
Oct 1996	0.8710202	-4.7996544	6.541695	-7.8015282	9.543569
Nov 1996	0.8980176	-5.4432721	7.239307	-8.8001481	10.596183
Dec 1996	0.9250150	-6.1119546	7.961985	-9.8371014	11.687131

We now plot the forecast for Holt-Winter's model (see Figure 4-24).

```
# triple exponential: models level, trend, & seasonal components
autoplot(fit3$x, size= 1, color= 'purple') +
  autolayer(for3, color = 'blue', size = 1.2) +
  ggtitle('Southern Oscillation Index (SOI)') +
  xlab('Year') + ylab('Oscillation Values') +
  theme(panel.background = element_rect(fill = 'snow2'),
    plot.title = element_text(size = rel(1.2),
      color='navy', face='bold'),
    axis.title.x = element_text(size = rel(1.2), color='navy'),
    axis.title.y = element_text(size = rel(1.2), color='navy'),
    axis.text = element_text(colour = 'red4',
      size = rel(0.95)),
    axis.line = element_line(size = 2, colour = 'grey80'))
```

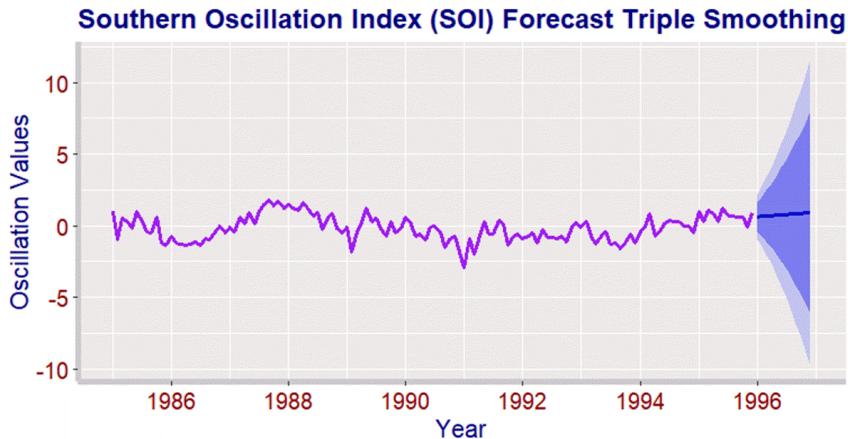


Figure 4-24. Forecast for triple exponential smoothing model

Accuracy Measurements

We now want to determine the accuracy of the predictions produced by exponential smoothing. We will compare the single and triple exponential smoothing models here.

```
# predictive accuracy for single exponential smoothing  
accuracy(forecast(fit2, 12))
```

	ME	RMSE	MAE	MPE	MAPE	MASE
Training set	0.0139	1.2248	0.9704	-88.9679	405.1278	0.5229
			ACF1			
Training set	0.0024					

```
# predictive accuracy for double exponential smoothing  
accuracy(forecast(fit3, 12))
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train set	-0.1051	1.4205	1.1229	-93.8576	363.0845	0.6018	0.0058

```
# predictive accuracy for triple exponential smoothing  
accuracy(forecast(fit4, 12))
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train set	0.1406	1.3448	1.0492	-41.3478	306.5387	0.5493	0.0370

Although the triple exponential smoothing is a little better, neither are very good with a minimum Mean Absolute Percent Error of 306.5387. Consequently, we will perform differencing of the time series data.

Chapter Review

Exponential smoothing has motivated some of the most successful forecasting methods. Forecasts produced using exponential smoothing methods are weighted averages of past observations, with the weights decaying exponentially as the observations get older. In other words, the more recent the observation the higher the associated weight.

This framework generates reliable forecasts quickly and for a wide range of time series, which is a great advantage and of major importance to applications in industry. This is one of the most popular methods for time series analysis. Smoothing takes a noisy time series and replaces each value with the average value of a neighborhood about the given value. We learned to use R apply various methods.

We also discussed several key definitions including:

Definition Number	Definition Name
Definition 4.1: Seasonality	Seasonality is the correlational dependency of order k between each k^{th} element of the series and the $(i-k)^{th}$ element (Kendall, 1976) and measured by autocorrelation (i.e., a correlation between the two terms); k is usually called the lag.
Definition 4.2: Autocorrelation Coefficient	The autocorrelation coefficient at lag h is given by $r_h = \frac{c_h}{c_0}$, where c_h is the autocovariance function $c_h = \frac{1}{N} \sum_{t=1}^{N-h} (Y_t - \bar{Y})(Y_{t+h} - \bar{Y})$, and c_0 is the variance function $c_0 = \frac{1}{N} \sum_{t=1}^N (Y_t - \bar{Y})^2$. The resulting value of r_h will range between -1 and +1.
Definition 4.3 Autocorrelation Function (ACF)	The autocorrelation function (ACF) is a time domain measure of the stochastic process memory, and does not reveal any information about the frequency content of the process. Generally, for an error signal, e_t , the ACF is defined as, $\rho_k = \frac{\text{Cov}(e_t e_{t+k})}{\sqrt{\text{Var}(e_t) \text{Var}(e_{t+k})}}$.
Definition 4.4 Simple Exponential Smoothing (SES)	Simple exponential smoothing (SES) is calculated using weighted averages, where the weights decrease exponentially as observations come from further in the past — the smallest weights are associated with the oldest observations, such that $y_{t+h} = \alpha y_t + \alpha(1 - \alpha)y_{t-1} + \alpha(1 - \alpha)^2 y_{t-2} + \dots$, where y_t is the smoothed statistic and $0 \leq \alpha \leq 1$ is the smoothing parameter. Then the forecasted values of y_t are estimates by $\hat{y}_{t+h T}$, where all future forecasts are equal to a simple average of the observed data: $\hat{Y}_{t+h T} = \frac{1}{T} \sum_{t=1}^T y_t$, for $h = 1, 2, \dots$. Hence, the average method assumes that all observations are of equal importance, and gives them equal weights when generating forecasts.

Definition 4.5: Double Exponential Smoothing	Given a raw data sequence of observations represented by Y_t , beginning at time $t = 0$, we use ℓ_t to represent the smoothed level value for time t , and b_t is our best estimate of the trend, β , at time t . Then output of the estimate of the value of y_{t+m} at time $m > 0$ based on the raw data up to time t is now written as \hat{Y}_{t+m} . Then, double exponential smoothing is given by $\ell_t = \alpha y_t + (1 - \alpha)[\ell_{t-1} + b_{t-1}]$, with $b_t = \gamma [\ell_t - \ell_{t-1}] + (1 - \gamma)b_{t-1}$, such that, $\hat{Y}_{t+m} = \ell_{t+m-1} + b_{t+m-1}$, is the forecasted estimate for the series Y_t .
Definition 4.6: Holt's Linear Trend	Holt's Linear Trend assumes a raw data sequence of observations represented by y_t , beginning at time $t = 0$. We use ℓ_t to represent the smoothed level value for time t , and b_t is our best estimate of the trend at time t , such that: $y_{t+h} = \ell_t + hb_t$, where ℓ_t and b_t are given by: $\ell_t = \alpha y_t + (1 - \alpha)(\ell_{t-1} + b_{t-1})$, and $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$, where α is the smoothing parameter for the level, $0 \leq \alpha \leq 1$, and β is the smoothing parameter for the trend, $0 \leq \beta \leq 1$.
Definition 4.7: Additive damped trend	The additive damped trend method , with the smoothing parameters α and β^* with values between 0 and 1 extends Holt's method by including a damping parameter $0 < \varphi < 1$, such that that $y_{t+h t} = \ell_t + (\varphi + \varphi^2 + \dots + \varphi^h)b_t$, where ℓ_t and b_t are given by: $\ell_t = \alpha y_t + (1 - \alpha)(\ell_{t-1} + \varphi b_{t-1})$, and $b_t = \beta^{(\ell_t - \ell_{t-1})} + (1 - \beta)\varphi b_{t-1}$. If $\varphi = 1$, the method is identical to Holt's linear method. For values between 0 and 1, φ dampens the trend so that it approaches a constant sometime in the future.
Definition 4.8: Additive Holt-Winters (HW)	Definition 4.11. The additive Holt-Winters (HW) prediction function , for time series with period length h , is: $\hat{y}_{t+h} = \ell_t + hb_t + s_{t+h-m(k+1)}$, where ℓ_t , b_t and s_t are given by: $\ell_t = \alpha(y_t - s_{t-m}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$ $s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1 - \gamma)s_{t-m}$ where k is the integer part of $(h - 1)/m$, which ensures that the estimates of the seasonal indices used for forecasting come from the final year of the sample.

Definition 4.9: multiplicative Holt-Winters	<p>The multiplicative Holt-Winters prediction function (for time series with period length h) is</p> $\hat{y}_{t+h} = (\ell_t + hb_t)s_{t+h-m(k+1)},$ <p>where ℓ_t, b_t and s_t are given by:</p> $\ell_t = \alpha \left(\frac{y_t}{s_{t-m}} \right) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$ $s_t = \gamma \left(\frac{y_t}{\ell_{t-1} + b_{t-1}} \right) + (1 - \gamma)s_{t-m}$
--	--

Definition 4.10: damped Holt- Winters	<p>The damped Holt-Winters prediction function (for time series with period length h) with damping factor ϕ is</p> $Y_{t+h} = [\ell_t + (\phi + \phi^2 + \dots + \phi^h)b_t]s_{t+h-m(k+1)}$ <p>where ℓ_t, b_t, and s_t are given by:</p> $\ell_t = \alpha \left(\frac{y_t}{s_{t-m}} \right) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$ $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)\phi b_{t-1}$ $s_t = \gamma \frac{y_t}{\ell_{t-1} + \phi b_{t-1}} + (1 - \gamma)s_{t-m}.$
--	--

Review Exercises

1. Fit the *copper* dataset from the `fma` package using moving averages of orders 3, 7, 11, and 15.
 - a. Describe what happens as each moving average if applied
 - b. Fit the data with an exponential moving average (EMA) and describe the fits comparison with that of order 15 in part (a).
 - c. Fit the data with a weighted moving average (WMA) and describe the fits comparison with that of order 15 in part (a).
 - d. Fit the data with a Zero lag exponential moving average (ZLEMA) and describe the fits comparison with that of order 15 in part (a).
 - e. Fit the data with a Arnaud Legoux moving average (ALMA) and describe the fits comparison with that of order 15 in part (a).
2. Using the *advsales* dataset from the `fma` package:
 - a. Plot the difference between Sales volume and Advertising expenditure
 - b. Apply a moving average of order 3 and describe its behavior
3. Using the *chicken* dataset from the `fma` package:
 - a. Set up a time series and plot it

- b. Fit the data in (a) with moving averages of 3, 5, 7 and 9; describe the results.
 - c. Set up a new time series that runs from 1970 to 1990
 - d. Fit the data in (b) with moving averages of 3, 5, 7 and 9; describe the results.
- 4. Fit the *bicoal* dataset from the [fma](#) package using moving averages of orders 3, 5, 7, and 95.
 - a. Describe what happens as each moving average if applied
 - b. Fit the data with an elastic, volume-weighted moving average (EVWMA) and describe the fits comparison with that of order 5 in part (a).

Chapter 5 – Exponential Smoothing and Moving Averages in Python

For this example, we'll be using the meat dataset which has been made available by the U.S. Department of Agriculture (USDA, 2016). It contains metrics on livestock, dairy, and poultry outlook and production. Before we get there, however, let's take a look at loading nonstandard series formats into Jupyter Notebook.

Loading and Formatting Non-Native Data

Loading data from outside sources can be tricky business. In Chapter 3 we worked with Unemployment Rate data that was native to R. Here we want to load that set into Jupyter and format it as time series data. First, we copy and paste it to a .CSV file. We can do this with a text editor or Excel. Then we use pandas to load the data. First we want to split the data into two comma separated value files, one containing the index (date) and the dependent variable, Volume, with the other containing the index (date) and the independent variables `uRate` and `iRate`. I have named them "`unemp`" and "`extpred`", respectively.

```
In [1]:  
dta1 = pd.read_csv(  
    "C:/Users/Strickland/Documents/Python Scripts/unemp.csv")  
dta1= pd.read_csv(  
    "C:/Users/Strickland/Documents/PythonScripts/extpred.csv")
```

Next, we want to prepare the two sets of time series data so that the indexes are dates from 31 June 1997 to 30 September 2005.

```
dta1.index =  
pd.Index(sm.tsa.datetools.dates_from_range('1997m6', '2005m9'))  
del dta1["date"]  
dta2.index =  
pd.Index(sm.tsa.datetools.dates_from_range('1997m6', '2005m9'))  
del dta2["date"]
```

To see that our import and formatting worked, we plot the time series data as shown in Figure 5-1.

```
In [2]:
```

```
%matplotlib inline  
dta1.plot(figsize=(12,8));
```

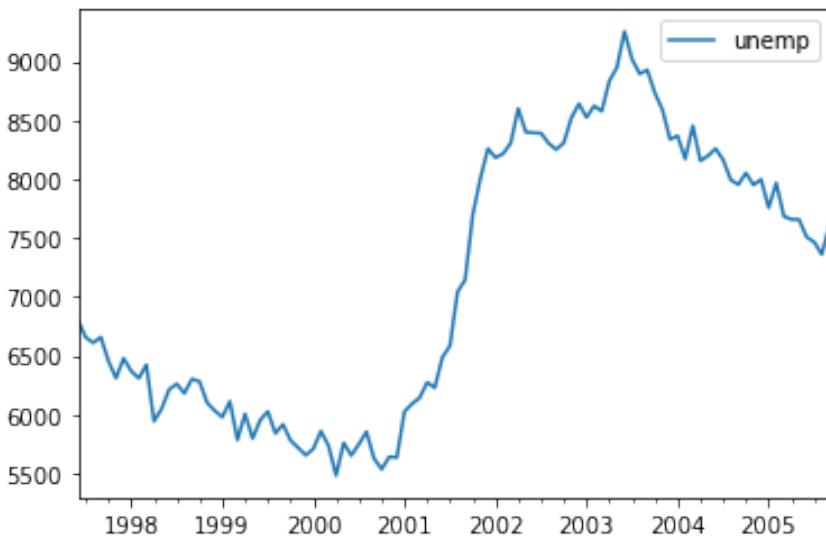


Figure 5-1. Time series for unemployment rate

West Texas Intermediate (WTI) refers to a grade or a mix of crude stream produced in Texas and southern Oklahoma which serves as a reference or "marker" for pricing a number of other crude streams and which is traded in the domestic spot market at Cushing, Oklahoma. Cushing OK WTI Spot Price FOB.csv (shortened here as "cushing.csv") contains the daily spot prices from January 2, 1986 to June 19, 2020. The file holds two columns of data, 'Day' and 'Spot_Price'. The 'Day' field is used as the index.

```
In [3]:  
# Load to the dependencies  
from matplotlib import pyplot as plt  
import pandas as pd  
from scipy import interpolate  
import matplotlib as mpl  
mpl.rcParams['figure.dpi'] = 300  
%matplotlib inline  
# Load the West Texas Intermediate (WTI) Crude Oil Spot Price  
df = pd.read_csv(  
    'D:\\\\Documents\\\\Data\\\\cushing.csv', index_col = 'Day',  
    parse_dates = True)
```

We limit the data set and aggregate its values from daily to monthly. This is not necessary but for this data set it easier to visualize the averages.

```
In [4]:  
# Limit the data set and aggregate its values to monthly.  
df.head()  
df = df.loc['2015-12-12':]  
df = df.resample('M').sum()  
df.head(2)
```

```
Out [1]  
      Spot_Price  
Day  
2015-12-31    346.26
```

Next, we apply five different moving averages using the `rolling(window = j).mean` function for $j = -k \dots k$, and plot them together with the series of spot prices in Figure 5-2. We used the default coverings scheme and provided a legend. Recall in Chapter 3 that we used R's `rollmean()` function from the `zoo` package and its similarity to the `rolling().mean` function here.

```
In [5]:  
# Create and visualize the rolling average  
plt.figure(num = None, figsize=[7, 4])  
df['Spot_Price'].plot(  
    label = 'Spot Price FOB Dollars per Barrel')  
df['Spot_Price'].rolling(window=7).mean().plot(  
    label = '7-day Moving Average')  
df['Spot_Price'].rolling(window=14).mean().plot(  
    label = '14-day Moving Average')  
df['Spot_Price'].rolling(window=21).mean().plot(  
    label = '21-day Moving Average')  
df['Spot_Price'].rolling(window=30).mean().plot(  
    label = '30-day Moving Average')  
df['Spot_Price'].rolling(window=60).mean().plot(  
    label = '60-day Moving Average')  
plt.title('Cushing OK WTI Spot Price FOB')  
plt.xlabel('Days')  
plt.ylabel('Dollars per Barrel')  
plt.legend(loc = 'upper left')  
plt.show()
```

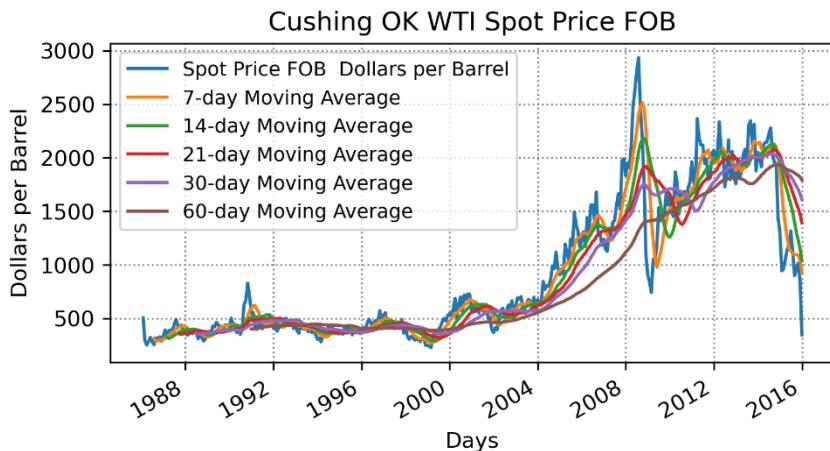


Figure 5-2. Spot prices of WTI crude oil with five moving averages

Where's the Beef (Let them Roll)

For this example, we get the Meat dataset in either the [ggplot](#) package or the [pandasql](#)-package, both of which are installed via [pip](#).

```
$ pip install -U ggplot
$ pip install -U pandasql
```

```
In [1]:
from pandasql import sqldf, load_meat, load_births
pysqldf = lambda q: sqldf(q, globals())
meat = load_meat()
births = load_births()
print pysqldf('SELECT * FROM meat LIMIT 10;').head()
```

Out [1]

					lamb and			
	date	beef	veal	pork	mutton	broilers	chicken	turkey
0	1944-01-01	751.0	85.0	1280.0	89.0	None	None	None
1	1944-02-01	713.0	77.0	1169.0	72.0	None	None	None
2	1944-03-01	741.0	90.0	1128.0	75.0	None	None	None
3	1944-04-01	650.0	89.0	978.0	66.0	None	None	None
4	1944-05-01	681.0	106.0	1029.0	78.0	None	None	None

We can get information about the data that will help us understand the data's attributes using the `info()` function.

```
In [2]:  
meat.info()
```

```
Out [2]  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 827 entries, 0 to 826  
Data columns (total 8 columns):  
 #   Column            Non-Null Count  Dtype     
---  --     
 0   date              827 non-null    datetime64[ns]   
 1   beef               827 non-null    float64   
 2   veal               827 non-null    float64   
 3   pork               827 non-null    float64   
 4   lamb_and_mutton   827 non-null    float64   
 5   broilers          635 non-null    float64   
 6   other_chicken     143 non-null    float64   
 7   turkey             635 non-null    float64  
dtypes: datetime64[ns](1), float64(7)  
memory usage: 51.8 KB  
  
import numpy as np
```

Alternatively, we can import the Meat dataset from the `ggplot` package.

```
In [3]  
import numpy as np  
import pandas as pd  
from ggplot import *  
meat = meat.dropna(thresh=800, axis=1) # drop columns that have  
fewer than 800 observations  
meat_ts = meat.set_index(['date'])
```

```
Out [3]  
beef    veal    pork    lamb_and_mutton  
1944-01-01    751.0  85.0    1280.0  89.0  
1944-02-01    713.0  77.0    1169.0  72.0  
1944-03-01    741.0  90.0    1128.0  75.0  
1944-04-01    650.0  89.0    978.0   66.0  
1944-05-01    681.0  106.0   1029.0  78.0  
...      ...      ...      ...  
2012-07-01    2200.8  9.5    1721.8  12.5  
2012-08-01    2367.5  10.1   1997.9  14.2  
2012-09-01    2016.0  8.8    1911.0  12.5  
2012-10-01    2343.7  10.3   2210.4  14.2  
2012-11-01    2206.6  10.1   2078.7  12.4
```

```
827 rows × 4 columns
```

Using the `info()` function again, we can see the structure of `meat_ts`, and note its different content. Instead of 8 columns (at `Out[1]`), we have 4 using the data from ggplot.

```
In [4]  
meat_ts.info()
```

```
Out [4]  
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 827 entries, 1944-01-01 to 2012-11-01  
Data columns (total 4 columns):  
 #   Column      Non-Null Count  Dtype     
 ---  --          --          --       --  
 0   beef        827 non-null    float64  
 1   veal        827 non-null    float64  
 2   pork        827 non-null    float64  
 3   lamb_and_mutton  827 non-null  float64  
dtypes: float64(4)  
memory usage: 32.3 KB
```

Working with dates and times with Pandas

Pandas has some excellent out of the box functionality for aggregating date and time-based data. There are quite a few ways to manipulate data using dates and times.

Aggregation processing

Before we go further, let's change the structure of the data by aggregating months to years. By using the `groupby` and `sum` functions, it is possible to easily calculate the result of the aggregation per year.

```
In [5]:  
meat_ts.groupby(meat_ts.index.year).sum().head(10)
```

```
Out[5]:
```

	beef	veal	pork	lamb_and_mutton
1944	8801.0	1629.0	11502.0	1001.0
1945	9936.0	1552.0	8843.0	1030.0
1946	9010.0	1329.0	9220.0	946.0
1947	10096.0	1493.0	8811.0	779.0
1948	8766.0	1323.0	8486.0	728.0
1949	9142.0	1240.0	8875.0	587.0
1950	9248.0	1137.0	9397.0	581.0
1951	8549.0	972.0	10190.0	508.0

1952	9337.0	1080.0	10321.0	635.0
1953	12055.0	1451.0	8971.0	715.0

Grouping by decade

If we're only interested in one or more specific decades, we can accomplish that using the date and time slicing functionality baked-in to *pandas*. Here we selected a slice of the data corresponding to the 1940s.

Since we indexed our data on a date-time column (date), we can group by the year and take the sum of the columns pretty easily. But what if we're keen to look at the sums over the decades?

```
In [6]:  
the1940s = ts.groupby(ts.index.year).sum().index[1950:1959]
```

User Defined Groups

We can specify a user-defined function to groupby. In the following example, we calculate the year, truncated from the year in units of 10 using the `floor_decade` function. We can sum the column or columns we are interested in to get the total for the decade of interest.

But what if you need to look at all the decades? One quick way is to use Python's explicit floor division operator, `//`, in a definition, `def()`.

```
In [7]:  
def floor_decade(date_value):  
    "Takes a date. Returns the decade."  
    return (date_value.year // 10) * 10  
pd.to_datetime('2013-10-9')
```

```
Out[7]:  
Timestamp('1970-01-01 00:00:00.000001994')
```

Now, we run our newly defined function.

```
In [8]:  
floor_decade(_)
```

```
Out[8]:  
1970
```

Now, we can ask for an easy summation for every ten years using our `floor_decade`.

```
In [9]:  
by_decade = meat_ts.groupby(floor_decade).sum()
```

Out [9]				
	beef	veal	pork	lamb_and_mutton
1940	55751.0	8566.0	55737.0	5071.0
1950	119161.0	12693.0	98450.0	6724.0
1960	177754.0	8577.0	116587.0	6873.0
1970	228947.0	5713.0	132539.0	4256.0
1980	230100.0	4278.0	150528.0	3394.0
1990	243579.0	2938.0	173519.0	2986.0
2000	260540.7	1685.3	208211.3	1964.7
2010	76391.5	371.9	66491.2	455.6

Plotting the Data

We use the `ggplot()` to display the value of beef of by decade to bar graph seen in Figure 5-4. Note that `ggplot` is a plotting system for Python based on R's `ggplot2` and the Grammar of Graphics (Wilkinson, Wills, Rope, Norton, & Dubbs, 2005). It is built for making professional looking plots quickly with minimal code. For this example, we are going to use `plotnine`, which is a Python implementation inspired by the interface of the `ggplot2` package from R.

Basic building blocks of a plot, according to the grammar of graphics, are:

1. **Data:** The data + a set of aesthetic mappings that describing variables mapping
2. **Geom:** Geometric objects, represent what you actually see on the plot: points, lines, polygons, etc.
3. **Stats:** Statistical transformations, summarize data in many useful ways.
4. **Scale:** The scales map values in the data space to values in an aesthetic space
5. **Coord:** A coordinate system, describes how data coordinates are mapped to the plane of the graphic.
6. **Facet:** A faceting specification describes how to break up the data into subsets for plotting individual set.

To produce a plot like Figure 5-3 with the `ggplot` class from `plotnine`, we must provide three things:

1. A data frame containing our data.
2. How the columns of the data frame can be translated into positions, colors, sizes, and shapes of graphical elements ("aesthetics").
3. The actual graphical elements to display ("geometric objects").

```
In [10]
```

```
ggplot(meat_yr, aes(x = 'year', weight = 'beef')) + \
  geom_bar(fill = 'blue', color = 'steelblue', width = 5) + \
  ggtitle('Head of Cattle Slaughtered by Decade')
```

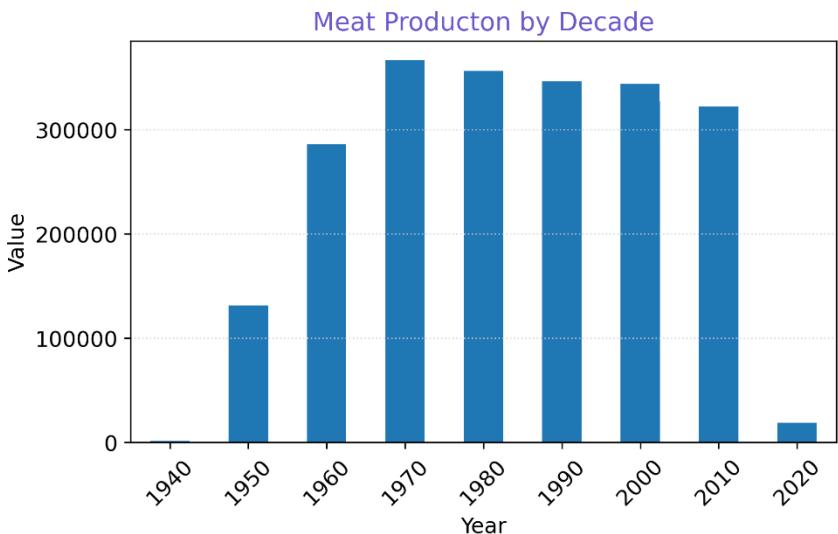


Figure 5-3. Value of beef of by decade

We can generate a similar bar chart using `pyplot`, like Figure 5-4, which we abbreviated as `plt` using: `import matplotlib.pyplot as plt`

```
In [11]:
```

```
matplotlib.rcParams['figure.dpi'] = 300
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(meat_yr['year'], meat_yr['beef'], width = 7,
       color = 'dodgerblue')
ax.set_xlabel('Year', size = 14)
ax.set_ylabel('Value', size = 14)
ax.set_title('Head of Cattle Slaughtered by Decade', size = 16)
ax.grid(color = 'lightblue')
plt.show()
```

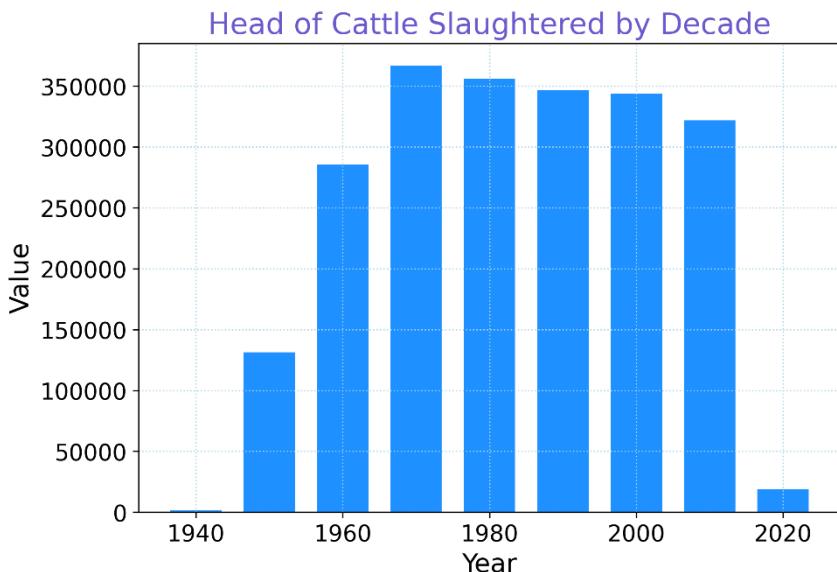


Figure 5-4. Value of beef of by decade

We can also use the melt function is set to ID the year in `id_vars`, set the column name to the variable column, and then converted to a format that you set the meat type as a variable and its corresponding value in the value column.

```
In [2]:  
by_decade_long = pd.melt (by_decade, id_vars = "year");  
by_decade_long.head()
```

	year	variable	value
0	1940	beef	55751
1	1950	beef	119161
2	1960	beef	177754
3	1970	beef	228947
4	1980	beef	230100.0

Setting the data up as just described makes it is possible to specify a variable to color, producing a bar graph that has been sorted by type of meat and colored accordingly, as seen in Figure 5-5.

```
In [14]:  
meat_yr.plot.bar(ylim = 0)  
plt.grid(axis = 'y', alpha = 0.75)
```

```

n = 8; ind = np.arange(n)
plt.ylabel('Value', fontsize = 12)
plt.xlabel('Year', fontsize = 12)
plt.title('Meat Production by Livestock', fontsize = 14)
plt.show()

```

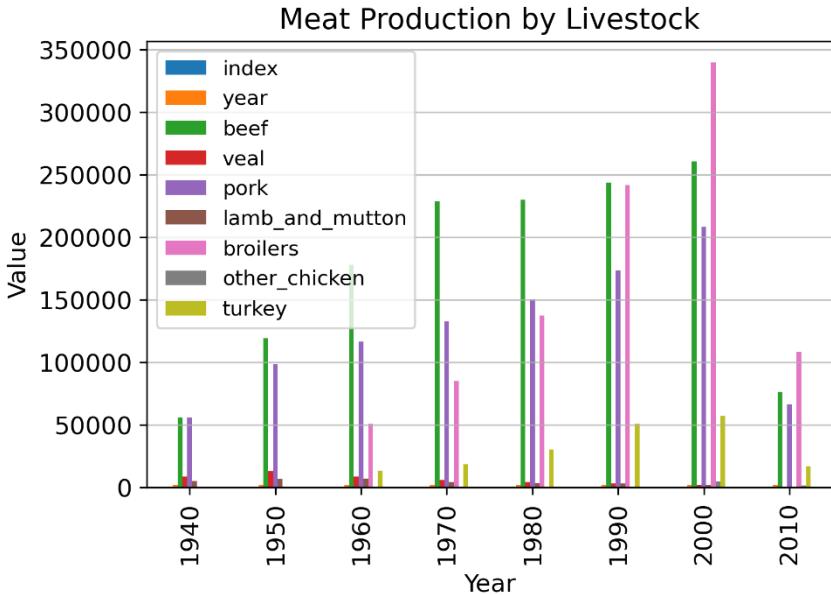


Figure 5-5. Bar chart of meat source by decade

We can use the plot in Figure 5-5 to see the meats that occur most frequently, say the top 4, and produce a bar plot of those, as we do in Figure 5-6.

```

In [15]:
meat_yr_top = meat_yr[['beef','broilers','pork','turkey']]
plt.figure(figsize = [10,6], dpi = 300)
meat_yr_top.plot.bar(ylim = 0)
plt.grid(axis = 'y', alpha = 0.75)
n = 8; ind = np.arange(n)
plt.xticks(ind,['1940','1950','1960','1970','1980','1990','2000',
,'2010'], fontsize = 12)
plt.yticks(fontsize = 12)
plt.ylabel('Value', fontsize = 12)
plt.xlabel('Year', fontsize = 12)
plt.title('Meat Production by Livestock', fontsize = 14)

```

```
plt.show()
```

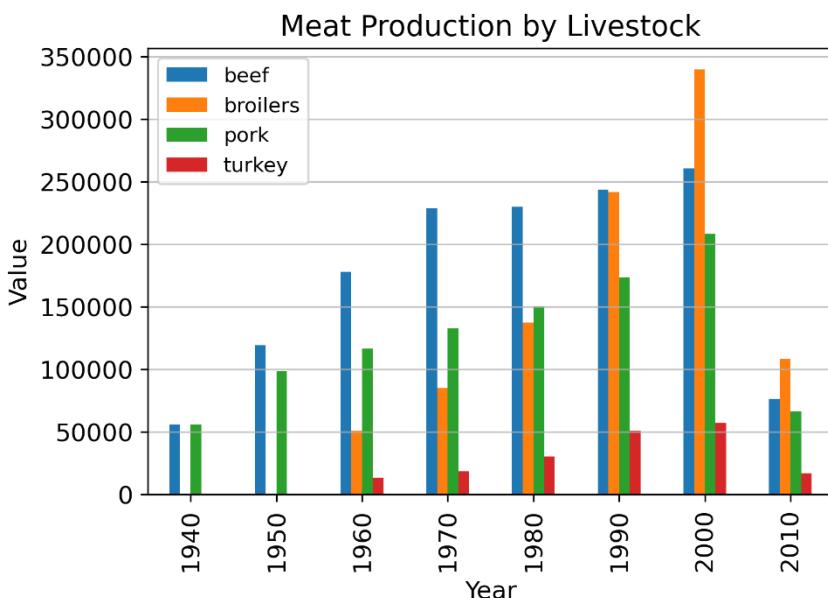


Figure 5-6. Bar graph with a bar for each category per year

The next snippet of code creates the stacked bar chart in Figure 5-7, using `plotly` in conjunction with `ggplot`. We had to load `plotly` using `pip install plotly` either in Jupyter or using a CMD.exe prompt. When we load a library into our notebook, we will typically give the function an alias. For example:

```
from matplotlib import pyplot as plt
```

Then when we want to call on `pyplot`, we merely write `plt`. Also, if we want all the functionality a a particur libray, we call the with a wildcard vale, `*`, as we see in In[16].

```
In [16]:
from ggplot import*
from plotnine import *
from plotly.tools import mpl_to_plotly as ggplotly
by_decade = meat_yr[['year','beef','broilers','pork','turkey']]
by_decade_lg = pd.melt(by_decade, id_vars = "year")
plt.figure(figsize = [10,6], dpi = 300)
p2 = ggplot(aes(x = 'year', weight = 'value',
                 fill = 'variable'), data = by_decade_lg) + \
geom_bar(colour ='black', width = 4) + \
theme(text = element_text(size = 16)) + \
ggtitle('Meat Production by Decade')
ggplotly(p2.draw())
plt.show(p2.draw())
```

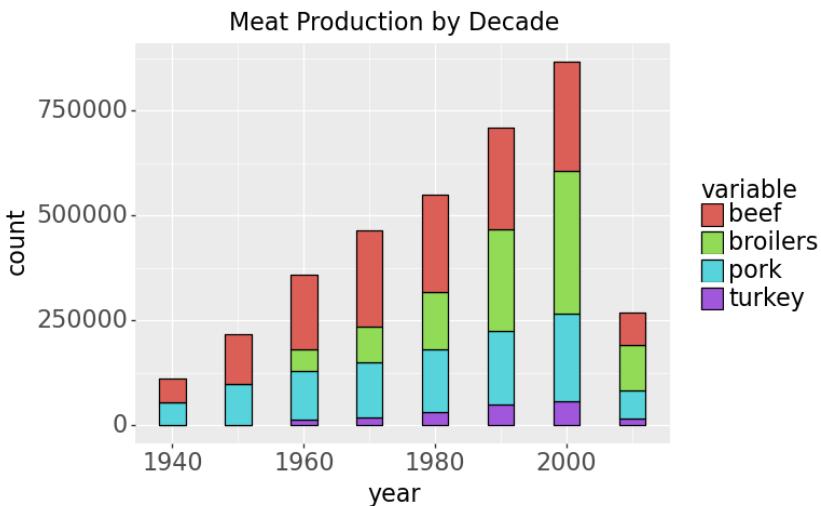


Figure 5-7. Bar graph that has been stacked up by type of meat

Visualizing Time Series Trend

Now, let's look at the trend in the time series of meat data. We reload the meat data, and plot in a time series using `geom_line`, as seen in Figure 5-8.

```
In [17]:
ts.plot(figsize=(8,8), linewidth=2, fontsize=14)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Value', fontsize=14)
plt.title('Meat production', fontsize=18)
```

```
plt.legend(fontsize=12)
plt.grid(True)
pyplot.show()
```

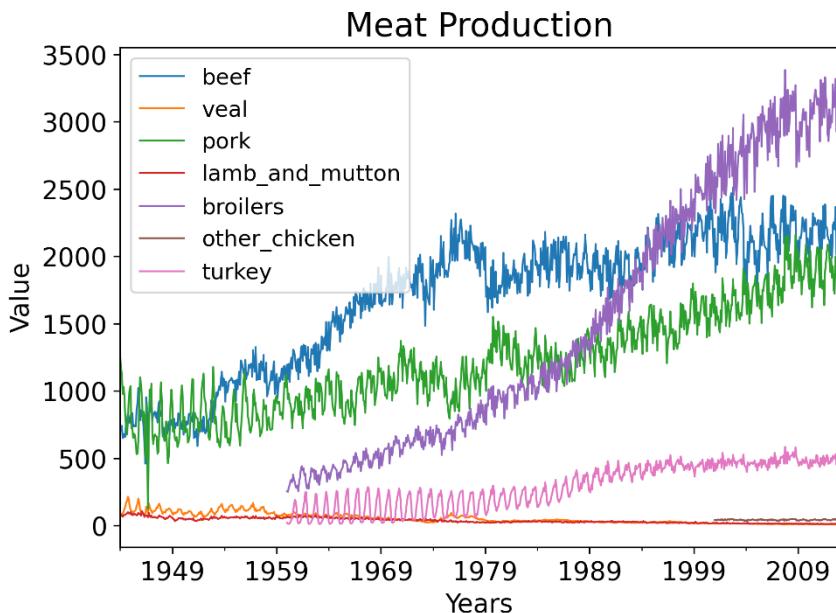


Figure 5-8. Time series plots of the meat data

Ok, so this plot looks a bit cluttered. We've got way too much zigging and zagging. The colors are nice but it is a bit overwhelming. Instead of getting rid of our data, we are going to apply a smoothing function so that we can see the trend instead of the noise (see Figure 5-9).

```
In [18]:
ggplot(aes(x = 'date', y = 'value', color = 'variable'),
data=meat_lng) + \
  stat_smooth(span=0.10) + \
  theme(text = element_text(size = 14),
axis_text_x = element_text(angle = 90, hjust = 1)) + \
  ggtitle('Smoothed Livestock Production')
```

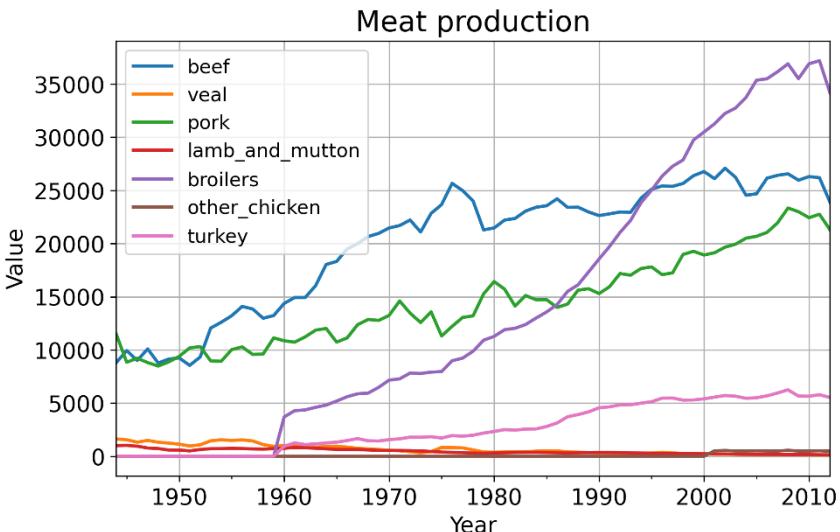


Figure 5-9. Meat data with a smoothing function applied to cancel the noise

Where's the Beef (Exponential Smoothing)?

Toward the end of each chapter, we have been asking the question: "Got milk?" Since we have been talking about "meat" in this chapter, let's ask: Where's the beef? "Where's the beef?" is a catchphrase in the United States and Canada, introduced as a slogan for the fast food chain Wendy's in 1984. Since then it has become an all-purpose phrase questioning the substance of an idea, event, or product.

The strategy behind the campaign was to distinguish competitors' (McDonald's and Burger King) big name hamburgers (Big Mac and Whopper respectively) from Wendy's 'modest' Single by focusing on the large bun used by the competitors and the larger beef patty in Wendy's hamburger. In the ad, titled "Fluffy Bun", actress Clara Peller receives a burger with a massive bun from a fictional competitor, which uses the slogan "Home of the Big Bun". The small patty prompts Peller angrily to exclaim, "Where's the beef?"

Data with Level and Trend

We will use the meat data that we introduced in this chapter, focusing on our favorite meat: beef. Since each meat is a separate column in the

data, we can easily call it out in Python, using `meat["beef"]`. While doing this, we also want to show some different ways to represent the time series, and we start with colors. Here, we define a set of “color-blind+ friendly color. (Kassambara, 2018).

```
In [19]:  
# The palette with grey:  
cbp1=( '#999999', '#E69F00', '#56B4E9', '#009E73',  
      '#F0E442', '#0072B2', '#D55E00', '#CC79A7')  
  
# The palette with black:  
cbp2 <- c('#000000', '#E69F00', '#56B4E9', '#009E73',  
          '#F0E442', '#0072B2', '#D55E00', '#CC79A7')
```

The difference between the two sets in Figure 5-10 are the colors gray and black. A color can be called from a set using, for example, `cbp1[0]`, which calls the color gray, or `cbp2[6]` which calls orange (`#D55E00`). We point out that the index is from 0 to 7.

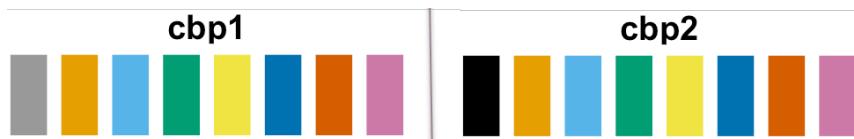


Figure 5-10. Color-blind friendly color palettes in Python

Now, we plot the time series in three parts. First, we plot the points of the series in Figure 5-11.

```
In [20]:  
p = ggplot(aes(x = 'date', y = 'beef'), data = meat)  
p + geom_point(color = cbp1[7]) +\n    theme(text = element_text(size = 14),  
axis_text_x = element_text(angle = 45, hjust = 1, size = 12)) +\n    ggttitle('Cattle Slaughtered for Beef Production') +\n    xlab('Date') +\n    ylab('Head of Cattle')
```

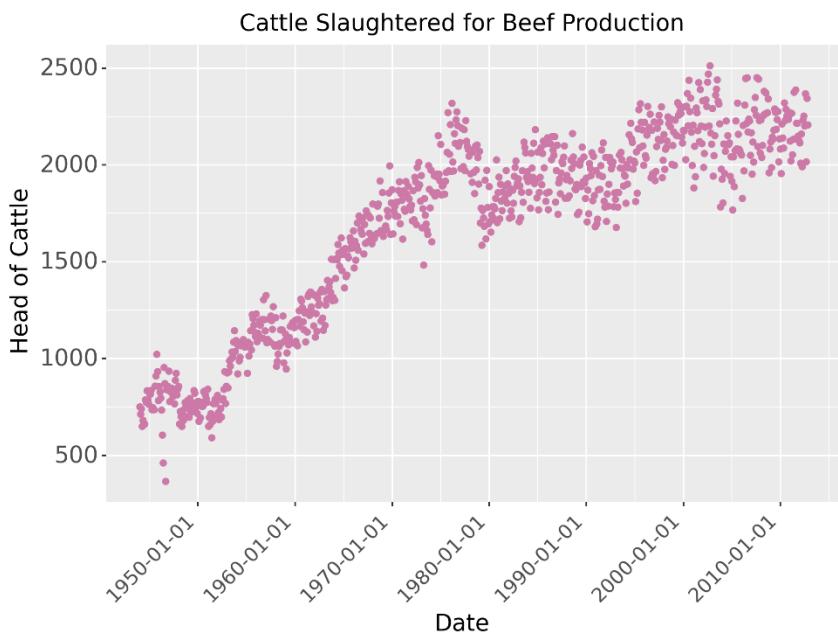


Figure 5-11. Time series plot of the points for beef production

Second, we add an overlying plot of lines that join the points of the series and show this in Figure 5-12. Although this is not absolutely required, it gives a clearer picture on the "noise" embedded in the time series. We should always make it a point to examine a scatterplot of times series before we delve into analysis.

```
In [21]:
p = ggplot(aes(x = 'date', y = 'beef'), data = meat)

p + geom_point(color = cbp1[7]) + geom_line(color=cbp1[1]) +\
    theme(text = element_text(size=10), axis_text_x = \
element_text(angle = 90, hjust = 1)) +\
    ggtitle("Beef Production")
```

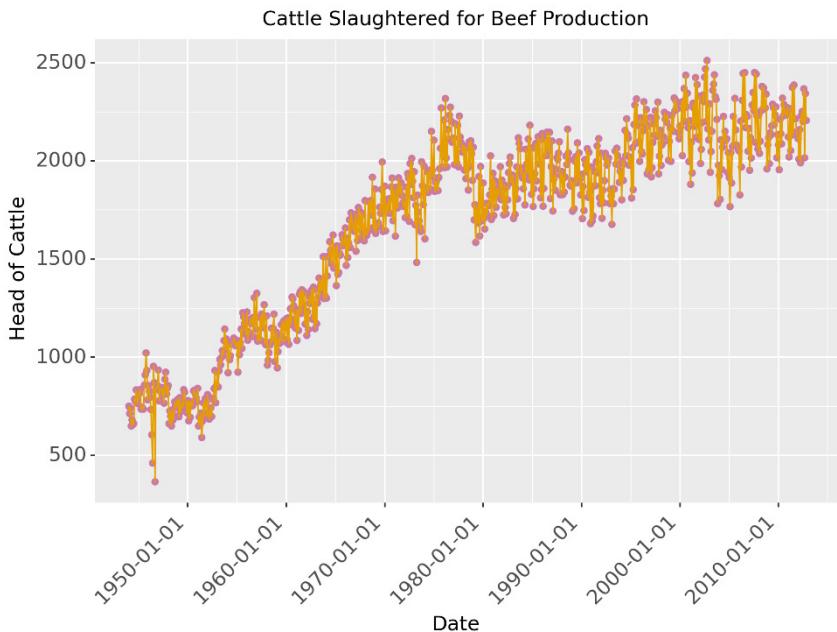


Figure 5-12. Beef production time series with line-connected points.

Finally, we use **locally weighted scatterplot smoothing** (LOWESS), the default method in `stat_smooth()`, to fit a line through the points as shown in Figure 5-13. Nonparametric smoothers like LOWESS try to find a curve of best fit without assuming the data must fit some distribution shape, but we use it sparingly with time series as it can misrepresent the data. We plotted this in Figure 5-13. Most algorithms use classical methods, such as linear and nonlinear least squares regression.

```
In [22]:
p + geom_point(color = cbp1[7]) + geom_line(color=cbp1[1]) +
  stat_smooth(color = cbp1[5]) +\
  theme(text = element_text(size=10),
        axis_text_x = element_text(angle = 90, hjust = 1)) +\
  ggtitle("Smoothed Beef Production")
```

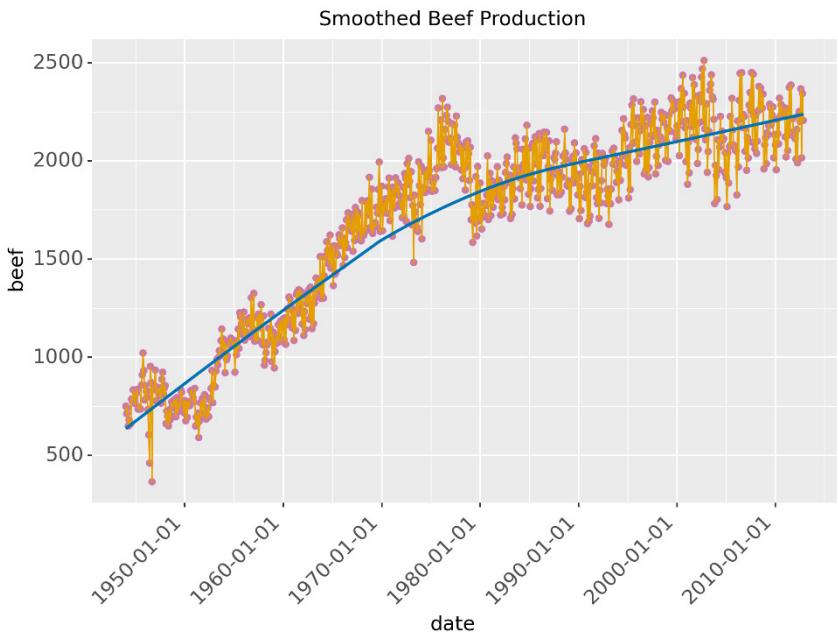


Figure 5-13. Beef production with lowess smoothing.

When we view a times series, like Figure 5-13, it can be difficult to sort out the noise and visualize what the data is telling us. Using a box plot to clear the noise can be instructive. The box plot in Figure 5-14 shows the beef production for each year, along with the means, interquartile ranges, first and fourth percentiles, as well as outliers.

In [23]:

```
from pandas import read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
groups = beef['1965':'1980'].groupby(Grouper(freq = 'A'))
date = DataFrame()
for name, group in groups:
    date[name.year] = group.values
date.boxplot(figsize = (7,4), rot = 90, fontsize = 12)
plt.title('Annual Number of Slaughtered Cattle', size = 16)
plt.xlabel('Year', size = 14)
plt.ylabel('Count in Thousands', size = 14)
pyplot.show()
```

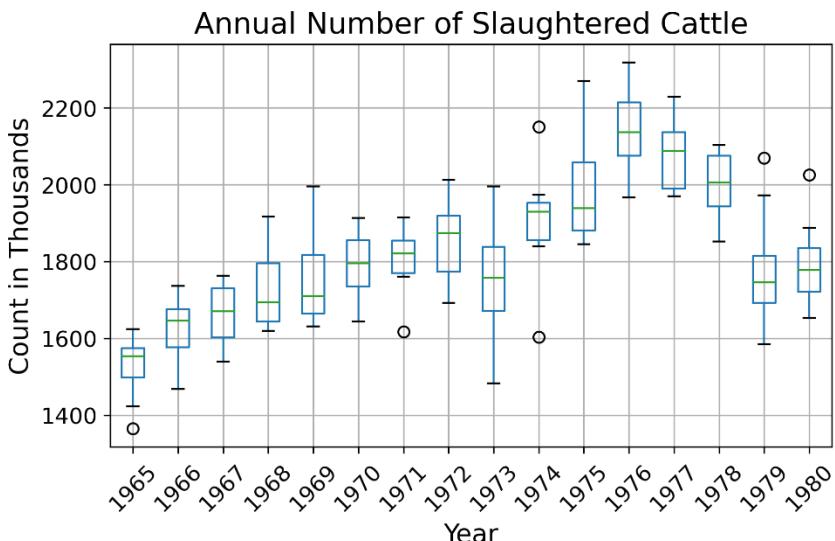


Figure 5-14. Boxplot of beef production per year (1965-1980)

In Figure 5-14, we only look at 15 years, rather than the full data set, for clarity. We also used the default setting for box plot. Now, we show how to customize a box plot and we have chosen to concentrate on colors alone. There are more aesthetic options we could apply but we do not want devote too much time to making “pretty plots.”

Notice that the code producing Figure 5-14 only has two line of code for the graphics. The first four lines load the libraries we will use, but these could have been loaded at the beginning of our project. The next two four lines set up the data to be plotted. To produce Figure 5-15, we only wrote the code necessary for the customized colors.

```
In [24]:  
bp1 = date.boxplot(figsize = (8,4), rot = 90, fontsize = 12,  
    boxprops = dict(color = 'blue'),  
    capprops = dict(color = 'green'),  
    whiskerprops = dict(color = 'purple'),  
    medianprops = dict(color = 'red'),  
    flierprops = dict(color = 'yellow',  
        markeredgecolor = 'orange'))  
plt.title('Annual Number of Slaughtered Cattle', size = 16)  
plt.xlabel('Year', size = 14)  
plt.ylabel('Count in Thousands', size = 14)
```

```
pyplot.show()
```

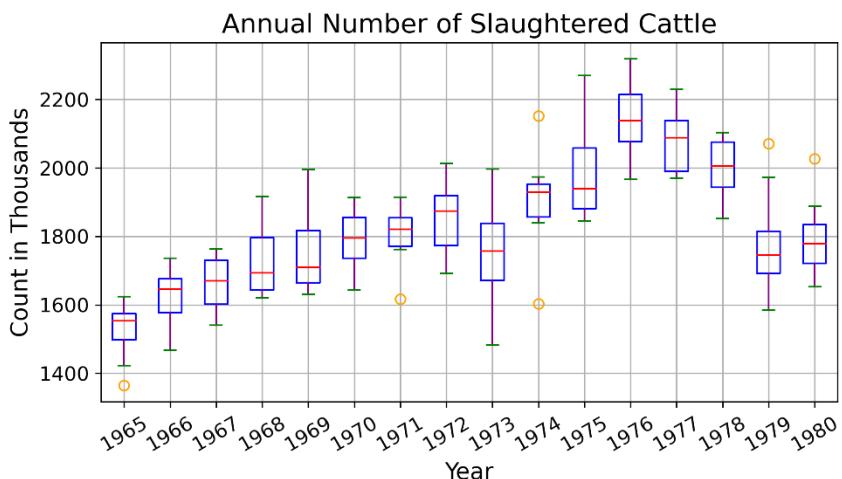


Figure 5-15. Customize color box plot of annual beef production.

The work we have performed on the beef data, we have not yet transformed it to time series data that can be manipulate by moving averages, exponential smoothing and the like. So, we will look at the data and determine how to proceed. The data frame contains a date and beef column with an $0, 1, \dots, n$ index. We need that index to be dates for it to be a time series.

```
In [25]:
```

```
df_beef = meat[['date', 'beef']]  
df_beef.head(5)
```

year	date	beef
0	1944-01-01	751.0
1	1944-02-01	713.0
2	1944-03-01	741.0
3	1944-04-01	650.0
4	1944-05-01	681.0

If this data frame had more column, like the meat data frame we extracted it from, we would need to look more closely at its structure. In the interest of good methodology, we will do that here using `dataframe.info()`, an operation unique to data frames in Python.

```
In [26]:  
df_beef.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 827 entries, 0 to 826  
Data columns (total 2 columns):  
 #   Column  Non-Null Count  Dtype     
---  --  -----  --  
 0   date    827 non-null    datetime64[ns]  
 1   beef     827 non-null    float64  
dtypes: datetime64[ns](1), float64(1)  
memory usage: 13.0 KB
```

Here, we see the two columns and their data types. If the date type was not a date format, say text, then we would have to convert it. Since, it is already in a date format, we will make it the new index using `dataframe.set_index()`, another operation we can perform on data frames.

```
In [27]:  
# set year column as index  
df_beef.set_index('date', inplace = True)  
# translate index name into English  
df_beef.index.name = 'month_year'  
df_beef.mean(axis = 1)
```

```
month_year  
1944-01-01    751.0  
1944-02-01    713.0  
1944-03-01    741.0  
1944-04-01    650.0  
1944-05-01    681.0  
...  
2012-07-01    2200.8  
2012-08-01    2367.5  
2012-09-01    2016.0  
2012-10-01    2343.7  
2012-11-01    2206.6  
Length: 827, dtype: float64
```

Examining the data after running the code, we can see that the `month_year` is now the index.

For the purpose of our analysis, we will transform this time series to years and look at the average beef production each year. We do this

using the `dataframe.mean()` function, and we will name the aggregated column, `avg_beef`.

```
In [28]:  
# calculate the yearly average head of cattle slaughtered  
df_beef['avg_beef'] = df_beef.mean(axis=1)  
# drop columns containing monthly values  
df_beef = df_beef[['avg_beef']]  
# visualize the first 2 columns  
df_beef.head()
```

year	avg_beef
1944-01-01	751.0
1944-02-01	713.0
1944-03-01	741.0
1944-04-01	650.0
1944-05-01	681.0

Now, we will plot the new time series, shown in Figure 5-16, using the `seaborn` library for statistical data visualization. We will also customize the font sizes, colors, and labels.

```
In [29]:  
beef_ts['beef'].plot(linewidth = 1, figsize = (8,4),  
                     fontsize = 12)  
plt.title('The yearly average Cattle slaughter in the US',  
          fontsize = 16)  
plt.xlabel('Year', fontsize = 14)  
plt.ylabel('Count in Thousands of Head', fontsize = 14)
```

We are calling the y-axis “production in kilograms,” but it is really the number of cattle slaughtered in thousands for beef products. That is the price for feeding a nation.

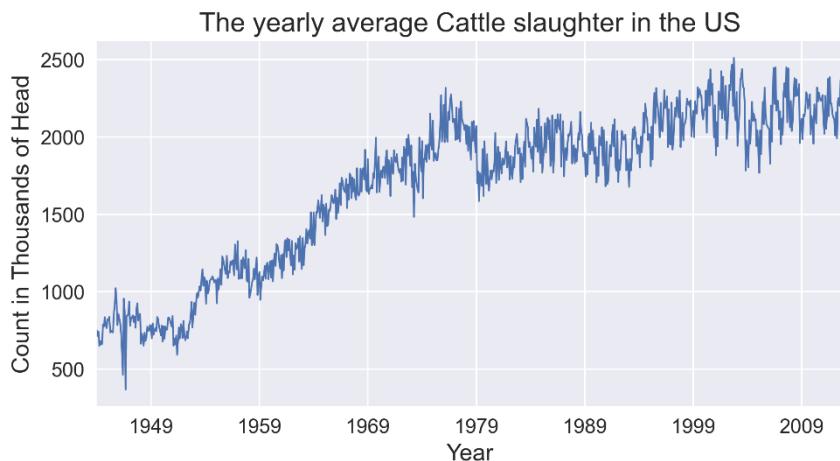


Figure 5-16. Yearly average beef production (1945 – 2010)

Next, we will fit two moving averages to the time series and examine their effect. We will use a simple moving average (SMA) or the *pandas* function `rolling()`.

```
In [30]:
# Average number of head of cattle slaughtered annually
# the simple moving average over a period of 10 years
df_beef['SMA_10'] = df_beef.avg_beef.rolling(10,
    min_periods = 1).mean()
# the simple moving average over a period of 20 year
df_beef['SMA_20'] = df_beef.avg_beef.rolling(20,
    min_periods = 1).mean()
```

Using some very simple code `timeseries.plot()` and the figure size parameter, we get a pretty nice plot in Figure 5-17.

```
In [31]:
df_beef.plot(figsize = (12,6))
```

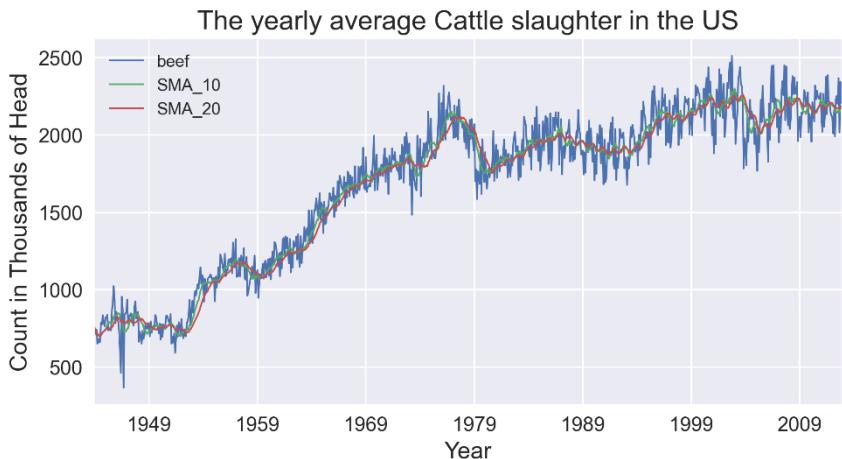


Figure 5-17. Beef production time series with two moving average models: SMA_10 and SMA_20

Now, we redraw the plot in Figure 5-17 using customized aesthetics in Figure 5-18.

```
In [32]:
matplotlib.rcParams['figure.dpi'] = 300
plt.figure(num=None, figsize = [7, 4])
beef_ts['beef'].plot(label = 'Beef', linewidth = 1,
                     color = 'lightblue')
beef_ts['beef'].rolling(10, min_periods = 1).mean().plot(
    label = '10-day Moving Average', linewidth = 1,
    color = 'dodgerblue')
beef_ts['beef'].rolling(20, min_periods = 1).mean().plot(
    label = '20-day Moving Average', linewidth = 1,
    color = 'magenta')
plt.title('Annual Number of Slaughtered Cattle')
plt.xlabel('Year')
plt.ylabel('Count in Thousands')
plt.legend(loc='upper left')
plt.show()
```

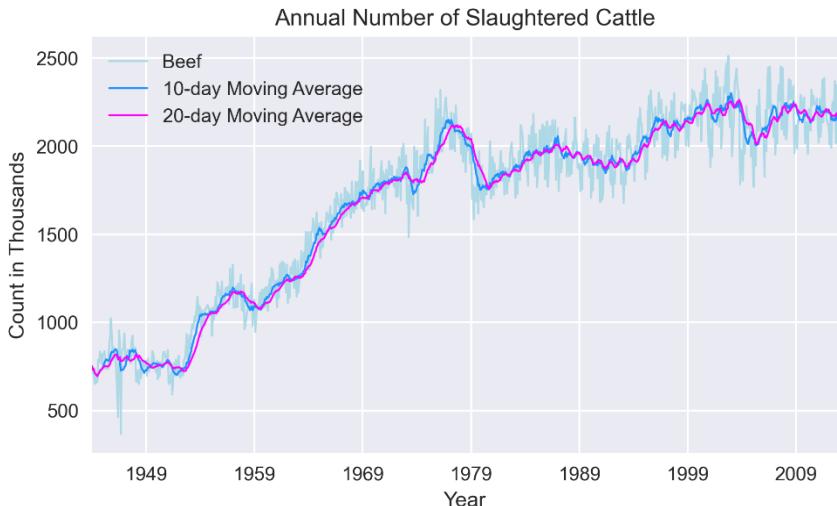


Figure 5-18. Customizes plot of beef production time series with two moving average models: SMA_10 and SMA_20

Recall that we can calculate the correlation for time series observations with observations from previous time steps, called lags. Because the correlation of the time series observations is calculated with values of the same series at previous times, this is called a **serial correlation**, or an autocorrelation.

Also remember that a plot of the **autocorrelation** (Definition 4.2) of a time series by lag is called the **AutoCorrelation Function**, or **ACF** (Definition 4.3). This plot is sometimes called a **correlogram** or an **autocorrelation plot**. We know that the ACF describes the autocorrelation between an observation and another observation at a prior time step that includes direct and indirect dependence information.

Consider a time series we just generated by a moving average (MA) process with a lag of k . Remember that the moving average process is an autoregression model of the time series of residual errors from prior predictions. Another way to think about the moving average model is that it corrects future forecasts based on errors made on recent forecasts.

We would expect the ACF for the $MA(k)$ process to show a strong correlation with recent values up to the lag of k , then a sharp decline to low or no correlation. By definition, this is how the process was generated. For the PACF, we would expect the plot to show a strong relationship to the lag and a trailing off of correlation from the lag onwards.

Here we use the `plot_acf()` function from the from `statsmodels` to calculate and draw our plot in Figure 5-19.

```
In [33]:
```

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(df_beef['SMA_10'])
pyplot.show()
```

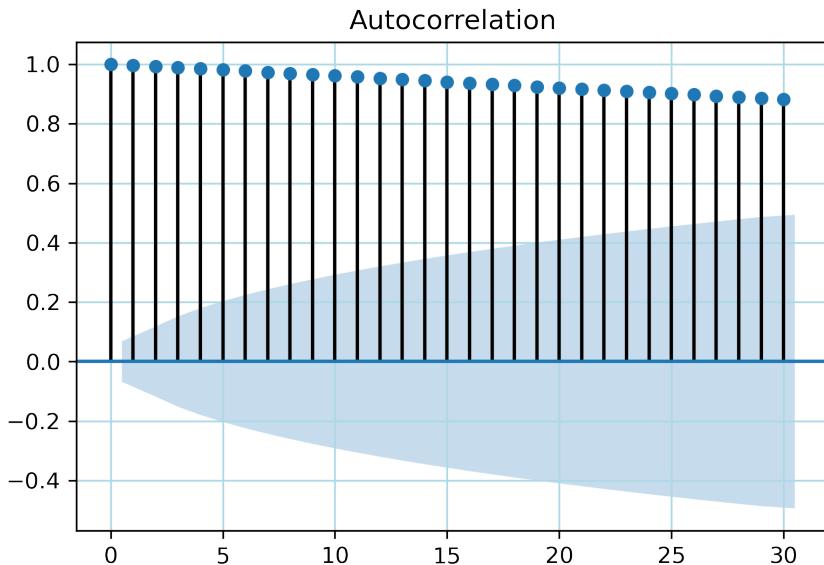


Figure 5-19. ACF correlogram for the lag 10 simple moving average

Before we examine the outcome, we'll make the ACF for the other moving average model for comparison in Figure 5-20.

```
In [34]:
```

```
plot_acf(df_beef['SMA_20'])
pyplot.show()
```

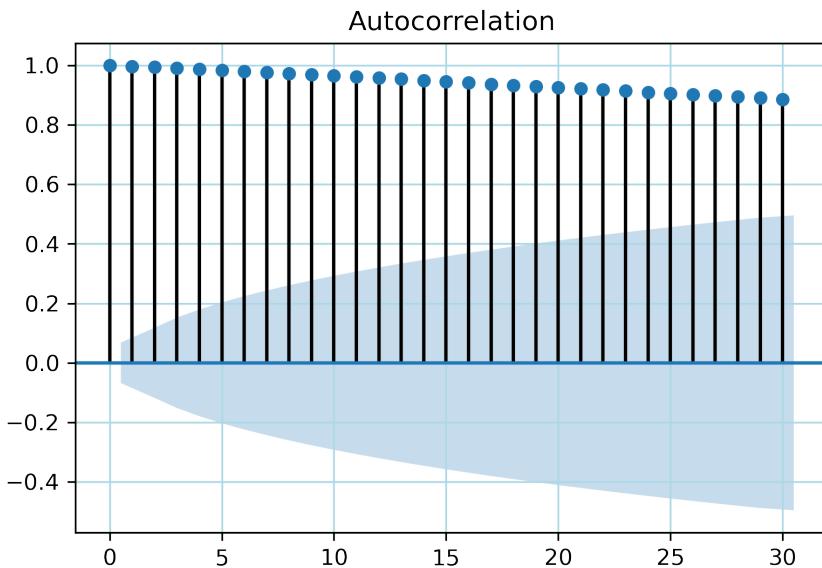


Figure 5-20. ACF correlogram for the lag 20 simple moving average

We can easily see that the two ACF plots are nearly identical, and both show a considerable amount of correlation. Recall that correlation values range from -1 to $+1$, or negatively to positively correlated, respectively. So, we'll see if the PACFs provide any additional information.

In Figure 5-21, we plot the SMA-10 PACF using the `plot_pacf()` function from `statsmodels`. By default, all lag values are printed, which makes the plot noisy. We can limit the number of lags on the x -axis to 50 to make the plot easier to read. We will also plot the SMA-20 PACF in Figure 5-22 for comparison.

```
In [35]:
```

```
from statsmodels.graphics.tsaplots import plot_pacf
plot_pacf(df_beef['SMA_10'], lags = 50)
pyplot.show()
```

```
In [36]:
```

```
plot_pacf(df_beef['SMA_20'], lags = 50); pyplot.show()
```

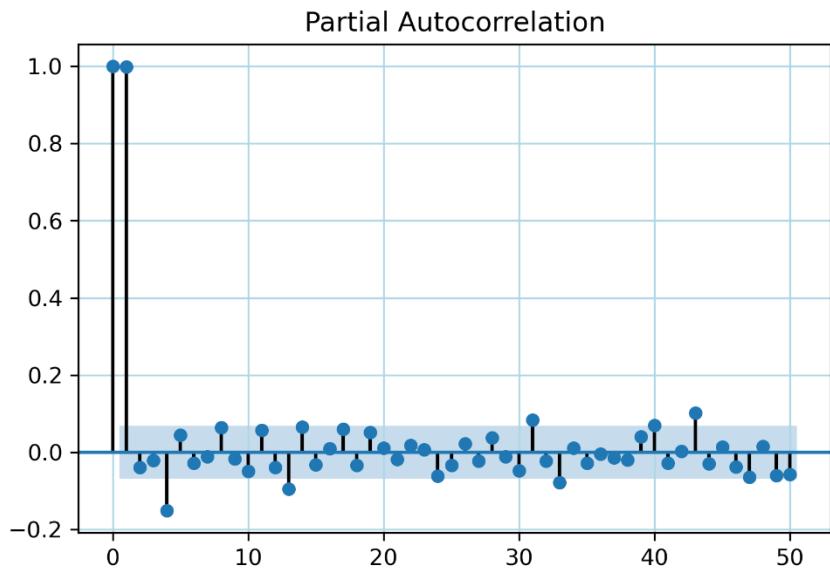


Figure 5-21. PACF correlogram for the lag 10 simple moving average

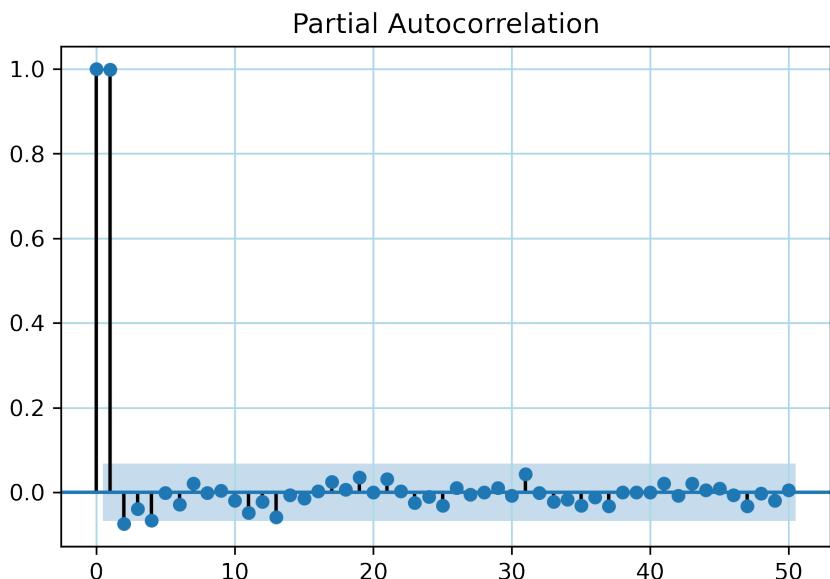


Figure 5-22. PACF correlogram for the lag 20 simple moving average

Notice that the autocorrelation is much diminished in the residuals for the SMA_20 model, suggesting that the 20-lag simple moving average model may be best here, without other diagnostic tests.

We can apply statistical tests and Augmented Dickey-Fuller test is the widely used one. The null hypothesis of the test is time series has a unit root, meaning that it is non-stationary. We interpret the test result using the p-value of the test. If the p-value is lower than the threshold value (5% or 1%), we reject the null hypothesis and time series is stationary. If the p-value is higher than the threshold, we fail to reject the null hypothesis and time series is non-stationary.

```
In [37]:  
from statsmodels.tsa.stattools import adfuller  
dftest = adfuller(df_beef['SMA_20'])  
dfoutput = pd.Series(dftest[0:4], index= ['Test Statistic',  
    'p-value','#Lags Used','Number of Observations Used'])  
for key, value in dftest[4].items():  
    dfoutput['Critical Value (%s)'%key] = value  
print(dfoutput)
```

Test Statistic	-1.748185
p-value	0.406473
#Lags Used	21.000000
Number of Observations Used	805.000000
Critical Value (1%)	-3.438499
Critical Value (5%)	-2.865137
Critical Value (10%)	-2.568685
dtype:	float64

P-value is greater than the threshold value, we fail to reject the null hypothesis and time series is non-stationary, it has time dependent component. These approaches suggest we have non-stationary data. So, we need to find a way to make it stationary, which we will do in Chapter 6.

Another task we performed in R is decomposition. We can do this in Python using `seasonal_decompose()` from `statsmodels`, the results of which as seen in Figure 5-23.

```
In [38]:  
import statsmodels.api as sm  
from statsmodels.tsa.seasonal import seasonal_decompose
```

```
decomposition = seasonal_decompose(df_beef['avg_beef'], freq=12)
decomposition.plot()
plt.show()
```

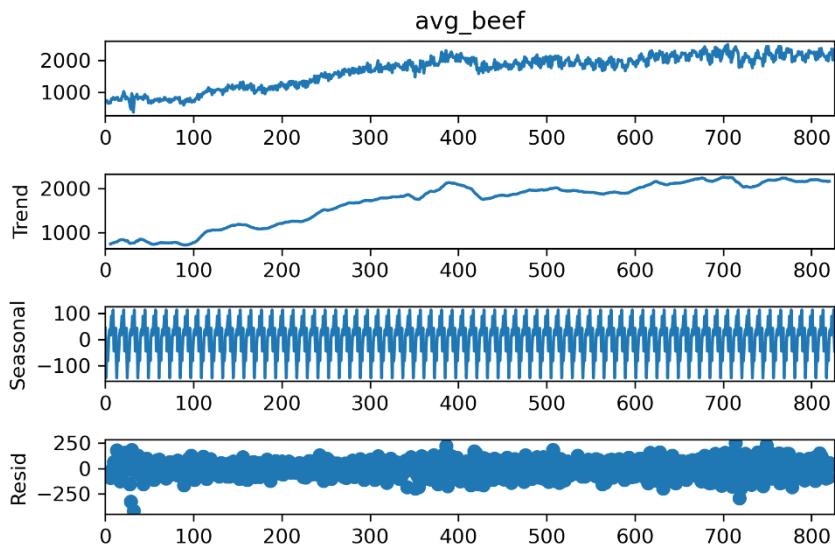


Figure 5-23. Seasonal decomposition of the beef production time series

Cattle and Simple Exponential Smoothing

We also performed SES on time series in Chapter 4. We will now perform SES on a modification of the meat data, comprised of cattle, hogs, sheep, and poultry. The data is obtained from Livestock & Meat Domestic Data containing current and historical data on pork, beef, veal, and poultry, including production, supply, utilization, and farm prices. We generated all the data for Meat production Tuesday, August 25, 2020. Updates of this data can be found at the USDA, ERS Livestock and Meat Domestic Data page: <https://www.ers.usda.gov/data-products/livestock-meat-domestic-data/>

Our first step is to read the data into a Jupyter Notebook using `pandas' read_csv()` function, and put the data into a data frame. We could have done this in one step, but we want the process to be obvious.

```
In [1]:
import requests
import pandas as pd
import json
```

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from statsmodels.tsa.holtwinters import SimpleExpSmoothing, Holt
import numpy as np
%matplotlib inline
meat = pd.read_csv("D:/Documents/Data/meat_min.csv")
meat_df = pd.DataFrame(meat)
meat_df.head(6)
```

Out [1]:

	date	Cattle	Hogs	Sheep	Poultry
0	Dec-2005	2667.1	9234.5	230.8	3408.58
1	Nov-2005	2668.1	9133.2	224.6	3477.92
2	Oct-2005	2679.2	9118.1	228.2	3583.07
3	Sep-2005	2775.9	8871.1	231.1	3520.18
4	Aug-2005	2992.5	8994.2	229.2	3660.83
5	Jul-2005	2717.5	7662.2	199.9	3298.77

After reading the data into Python, we take a subset of the data for our example. We also examine its structure.

In [2]:

```
cattle = meat[['date','Cattle']]
cattle.info()
```

Out [2]:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 696 entries, 0 to 695
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype  
 ---  -- 
 0   date     696 non-null    object 
 1   Cattle   696 non-null    float64 
dtypes: float64(1), object(1)
memory usage: 11.0+ KB
```

Nest, we use the `set_index()` function to format the data as a time series and observe the first few rows of output.

In [3]:

```
cattle_ts = pd.DataFrame(meat,
                         columns = ['date','Cattle']).set_index ('date')
cattle_ts.head(8)
```

Out [3]:

date	Cattle
Dec-2005	2667.1

Nov-2005	2668.1
Oct-2005	2679.2
Sep-2005	2775.9
Aug-2005	2992.5
Jul-2005	2717.5
Jun-2005	2937.6
May-2005	2793.3

In step 4, we use the `train_test_split` function to generate subsets for our training and testing data.

```
In [4]:
from sklearn.model_selection import train_test_split
train, test = train_test_split(cattle_ts, test_size=0.30,
                               shuffle = False)
model = SimpleExpSmoothing(np.asarray(train['Cattle']))
```

Next, we perform SES on the data, using various values for alpha, for instance, $\alpha = 0.3, 0.2, and 0.5.$

```
In [5]:
model = SimpleExpSmoothing(np.asarray(train['Cattle']))
model._index = pd.to_datetime(train.index)
model._index
```

Now that we have a SES model, we will use it to fit our data at the three values for alpha.

```
In [6]:
fit1 = model.fit(smoothing_level = 0.3)
pred1 = fit1.forecast(9)
fit1.summary()
```

SimpleExpSmoothing Model Results		
Dep. Variable:	endog	No. Observations: 487
Model:	SimpleExpSmoothing	SSE 2014556.630
Optimized:	True	AIC 768.882
Trend:	None	BIC 773.517
Seasonal:	None	AICC 769.453
Seasonal Periods:	None	Date: Thu, 22 Oct 2020
Box-Cox:	False	Time: 22:08:10
	coeff	code optimized
smoothing_level	0.300000	alpha True
initial_level	2699.4714	1.0 True

```
In [7]:
fit2 = model.fit(smoothing_level = 0.2)
```

```
pred2 = fit2.forecast(9)
fit2.summary()
```

```
SimpleExpSmoothing Model Results
Dep. Variable: endog           No. Observations:    75
Model: SimpleExpSmoothing    SSE: 2020457.626
Optimized: True                AIC: 769.101
Trend: None                  BIC: 773.736
Seasonal: None                 AICC: 769.672
Seasonal Periods: None        Date: Thu, 22 Oct 2020
Box-Cox: False                Time: 22:08:11

                  coeff      code      optimized
smoothing_level 0.2000000   alpha     False
initial_level    2717.3104    1.0      True
```

```
In [8]:
fit3 = model.fit(smoothing_level = 0.5)
pred3 = fit3.forecast(9)
fit3.summary()
```

```
SimpleExpSmoothing Model Results
```

```
Dep. Variable: endog           No. Observations:    75
Model: SimpleExpSmoothing    SSE: 2169575.278
Optimized: True                AIC: 774.442
Trend: None                  BIC: 779.077
Seasonal: None                 AICC: 775.013
Seasonal Periods: None        Date: Thu, 22 Oct 2020
Box-Cox: False                Time: 22:08:12

                  coeff      code      optimized
smoothing_level 0.5000000   alpha     False
initial_level    2708.9537    1.0      True
```

Finally, we plot the data on one chart in Figure 5-24 and compare the results.

```
In [9]:
matplotlib.rcParams['figure.dpi'] = 300
fig, ax = plt.subplots(figsize=(6,3))
ax.plot(train.index[5:], train.values[5:])
for p, f, c in zip((pred1, pred2, pred3),
                    (fit1, fit2, fit3), ('#ff7823','#3c763d','cyan')):
    ax.plot(train.index[5:], f.fittedvalues[5:],
            label = str(f.params['smoothing_level'])[:3])
ax.plot(test.index[-11:], p, color = c, )
```

```

plt.xticks(rotation = 45)
plt.xticks([0, 12, 24, 36, 48, 60, 72, 84],
           ['Jan 1969', 'Jan 1970', 'Jan 1971', 'Jan 1972',
            'Jan 1973', 'Jan 1974', 'Jan 1975', 'Jan 1976'])
plt.title('Simple Exponential Smoothing')
plt.legend();

plt.savefig('02_ses_ts.png', figsize = (7,6), dpi = 300,
bbox_inches = 'tight')
plt.show()

```

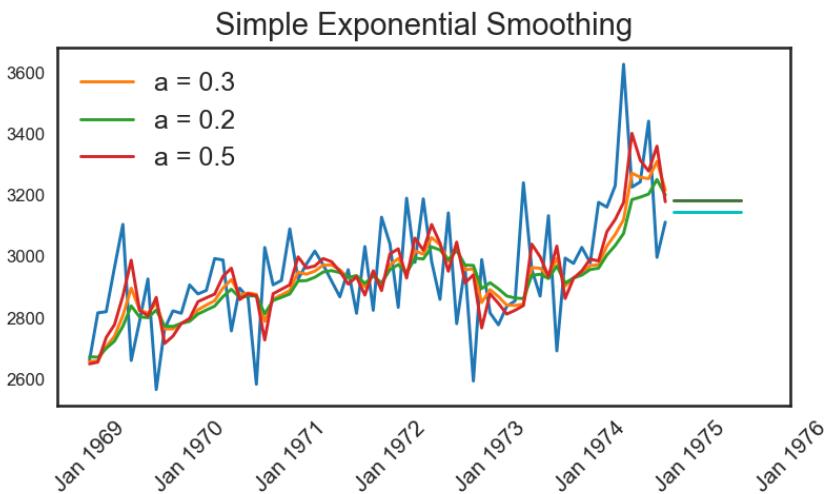


Figure 5-24. Simple Exponential Smoothing of meat data

Data with Level and Trend Components

When we look at Figure 5-24, we see the forecasts from all three methods yield horizontal lines or predictions that remain constant. We can try Holt's (double exponential) smoothing to account for trend. We practically use the same code, with a different model. The results are shown in Figure 5-25.

```

In [10]:
model = Holt(np.asarray(train['Cattle']))
fit4 = model.fit(smoothing_level = 0.3, smoothing_slope = 0.1)
pred4 = fit4.forecast(9)
fit5 = model.fit(smoothing_level = 0.3, smoothing_slope = 0.2)
pred5 = fit5.forecast(9)
fit6 = model.fit(optimized=True)

```

```

pred6 = fit6.forecast(9)plt.style.use('seaborn-white')
matplotlib.rcParams['figure.dpi'] = 300
fig, ax = plt.subplots(figsize = (8,4))
ax.plot(train.index[5:], train.values[5:])
for p, f, c in zip((pred4, pred5, pred6),(fit4, fit5, fit6),
    ('#ff7823','#3c763d','c')):
    ax.plot(train.index[5:], f.fittedvalues[5:],
        label = str(f.params['smoothing_level'])[:3]) +
        str(f.params['smoothing_slope'])[:3])
    ax.plot(test.index[-11:], p, color = c, )
plt.xticks(rotation = 45)
plt.xticks([0, 12, 24, 36, 48, 60, 72, 84],
    ['Jan 1969', "Jan 1970", "Jan 1971", 'Jan 1972',
     'Jan 1973', "Jan 1974", "Jan 1975", 'Jan 1976'])
plt.title("Holt's Exponential Smoothing")
plt.legend(loc = 2);

```

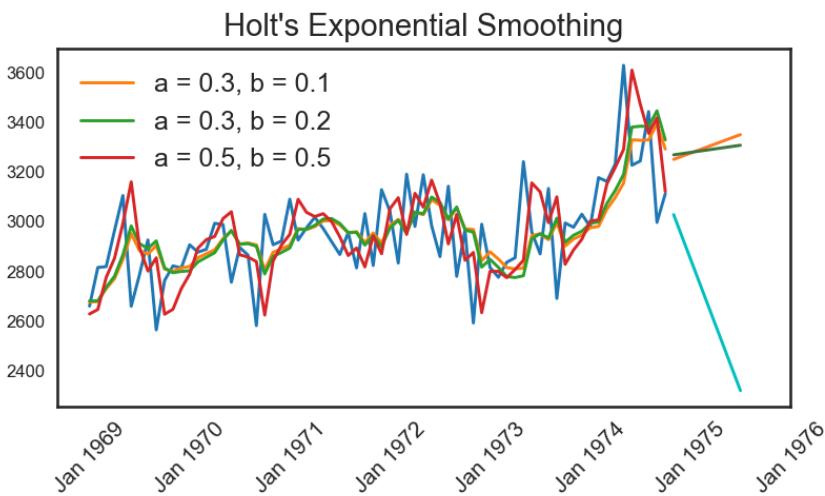


Figure 5-25. Holt's Exponential Smoothing of meat data

Holt's model appears to handle the trend properly and should be a better fit for the series. With the optimized estimates, as the level factor and trend factor get closer to zero, less emphasis is placed on older point in time, so the trend goes down since the most previous values impact the forecasts more. For the forecasts with $(a = 0.3, b = 0.1)$ and $(a = 0.3, b = 0.2)$, the trend is generally going up, since the earlier values are indicating an upward trend.

Chapter Review

In this chapter, we applied the methods we learned in Chapter 4, using Jupyter Notebook. We explored the Cushing stock price problem and went from milk to meat with, "Where's the beef?"

We learned more about preparing data in a time series format using Python. This included entering the data and formatting it as time series data, i.e., working with dates. We also practiced aggregating, plotting and observing the seasonality and trend in the time series data.

We studied the processes for applying simple moving averages and exponential smoothing to time series data, as well as plot the results, using Matplotlib. We also examined the process of producing ACFs and PACFs for the smoothed data, as well as how to construct bar plot and box plots.

Review Exercises

Note on tools. You may use any python tool you prefer for these exercises, but we used Jupyter Notebook and its inputs and output for these exercises.

1. Load the *copper2* dataset from (<https://github.com/stricje1/Data>) into Jupyter and set it up as a time series.
 - a. Plot the *copper* time series
 - b. Print out the *copper* time series data in a table
 - c. Smooth “*copper2*” time series and plot the smoothed data.
 - d. Construct a histogram of the annual copper prices
2. The data set books, contains one year (52 weeks) of historical sales records for book sales by the same store (download at <https://github.com/stricje1/Data>, not the same file native to R) The sales are for sales of paperback and hardcover books. The task is to forecast the next month sales for paperback and hardcover books.
 - a. Plot the series and discuss the main features of the data.
 - b. Use the `ses()` function to forecast each series, and plot the forecasts.
 - c. Compute the RMSE values for the training data in each case.
 - d. Apply Holt’s linear method to the paperback and hardback series and compute four-day forecasts in each case.

- e. Compare the RMSE measures of Holt's method for the two series to those of simple exponential smoothing in the previous question. (Remember that Holt's method is using one more parameter than SES.) Discuss the merits of the two forecasting methods for these data sets.
 - f. Compare the forecasts for the two series using both methods. Which do you think is best?
 - g. Calculate a 95% prediction interval for the first forecast for each series, using the RMSE values and assuming normal errors. Compare your intervals with those produced using ses and holt.
3. For this exercise use data set eggs, the price of a dozen eggs in the United States from 1900–1993. Experiment with the various options in the `holt()` function to see how much the forecasts change with damped trend, or with a Box-Cox transformation. Try to develop an intuition of what each argument is doing to the forecasts.
- a. [Hint: use `h=100` when calling `holt()` so you can clearly see the differences between the various options when plotting the forecasts.]
 - b. Which model gives the best RMSE?

Chapter 6 – Stationarity and Differencing

ARIMA models are the most general category of models for forecasting a time series where the data can be made to be “stationary” by differencing (if necessary). If necessary, this may occur in conjunction with nonlinear transformations such as lagging or deflating. We have seen that a random variable is a time series is stationary if its statistical properties are all constant over time. A stationary series has no trend, its variations around its mean have a constant amplitude, and it varies in a consistent fashion, i.e., its short-term random time patterns always look the same in a statistical sense. The latter condition means that its autocorrelations remain constant over time, or equivalently, that its power spectrum remains constant over time. A random variable of this form can be viewed as a combination of signal and noise, and the signal, if one is apparent. The time series could also have a seasonal component. An ARIMA model can be viewed as a “filter” that tries to separate the signal from the noise, and the signal is then extrapolated into the future to obtain forecasts.

R has extensive facilities for analyzing time series data. This section describes the creation of a time series, seasonal decomposition, modeling with exponential and ARIMA models, and forecasting with the forecast package.

Creating a time series in R

We keep revisiting this topic, because getting the data in the correct format is crucial to times series analysis. The `ts()` function from the `stats` package will convert a numeric vector into a R time series object. The format is `ts(vector, start=, end=, frequency=)` where start and end are the times of the first and last observation and frequency is the number of observations per unit time (1=annual, 4=quarterly, 12=monthly, etc.). The default is one year, but sometimes researchers collect time series data at regular intervals less than one year, for example, monthly or quarterly. In this case, you can specify the number of times that data was collected per year by using the ‘frequency’ parameter in the `ts()` function. For monthly time series data, you set

`frequency=12`, while for quarterly time series data, you set `frequency=4`.

A misgiving about the `ts()` function is that it is a black-box, that is, we cannot see how the time series is being constructed from the raw data. However, we will explore this process in Chapter 7 with Python, which requires us to know the details of creating a time series.

We can also specify the first year that the data was collected, and the first interval of that year by using the ‘start’ parameter in the `ts()` function. For example, if the first data point corresponds to the second quarter of 1986, you would set `start=c(1986, 2)`.

The Train-Test Split

In some of the output we have viewed, there is a reference to testing and training sets, such as:

	ME	RMSE	MAE	MPE
Training set	"0.3478"	"7.7272"	"5.6433"	"0.1754"
Test set	"0.0069"	"0.0576"	"0.0468"	"0.1143"

As we delve into forecasting as a general purpose of time series analysis, it is important to “hold-back” some of our data for the purpose of testing our predictions about the future, or put plainly: to test “the future” that has already occurred. If we can get that right, then we should have more confidence in the future and has not yet occurred. So, let’s defined the training and test sets.

Training set denotes values that were gathered from comparing the forecast to the data that was used to generate the forecast.

Test set denotes values that were gathered from comparing the forecast to the test data which we deliberately excluded when training the forecast.

To explore the structure of time series data, we will use the *airpass* dataset from Chapter 3, which we have plotted below in Figure 6-1:

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1960	112	118	132	129	121	135	148	148	136	119	104	118
1961	115	126	141	135	125	149	170	170	158	133	114	140
1962	145	150	178	163	172	178	199	199	184	162	146	166

```

1963 171 180 193 181 183 218 230 242 209 191 172 194
1964 196 196 236 235 229 243 264 272 237 211 180 201
1965 204 188 235 227 234 264 302 293 259 229 203 229
1966 242 233 267 269 270 315 364 347 312 274 237 278
1967 284 277 317 313 318 374 413 405 355 306 271 306
1968 315 301 356 348 355 422 465 467 404 347 305 336
1969 340 318 362 348 363 435 491 505 404 359 310 337
1970 360 342 406 396 420 472 548 559 463 407 362 405
1971 417 391 419 461 472 535 622 606 508 461 390 432

```

```

# plot series
plot(airpass, col='blue', lwd=2)

```

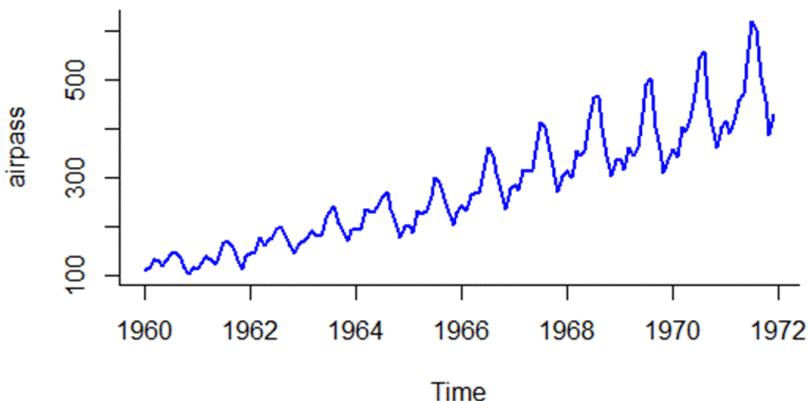


Figure 6-1. Air Passenger Times Series Data

There are a number of methods we could use for this purpose. We provide one method in the next code snippet.

```

set.seed(123)
data("airpass")
airpass
sample = sample.split(airpass, SplitRatio = 0.75) # splits the da
ta in the ratio mentioned in SplitRatio. After splitting marks t
hese rows as logical TRUE and the the remaining are marked as lo
gical FALSE
train_set = subset(airpass, sample == TRUE) # creates a training
dataset named train1 with rows which are marked as TRUE
test_set = subset(airpass, sample == FALSE)
train_set
test_set

```

```

# train_set
[1] 112 132 135 148 136 119 118 115 126 141 125 149 170 133
[15] 114 145 150 178 163 172 178 184 146 166 180 193 181 183

```

```
[29] 218 230 242 209 191 172 194 196 236 235 243 264 272 237
[43] 211 201 204 188 235 227 264 229 203 229 242 233 267 269
[57] 270 315 364 347 312 274 237 284 277 374 413 405 355 306
[71] 271 306 301 356 348 355 422 465 404 336 340 318 348 363
[85] 491 505 404 310 337 360 342 406 396 420 548 559 463 407
[99] 362 417 391 419 461 606 508 461 390 432
# test_set
[1] 118 129 121 148 104 135 170 158 140 199 199 162 171 196
[15] 229 180 234 302 293 259 278 317 313 318 315 467 347 305
[29] 362 435 359 472 405 472 535 622
```

This method seems to work well with most time series formats. In Python we use two methods:

```
In [1]
# Method 1
nobs = 18
df_train, df_test = exogx[0:-nobs], exogx[-nobs:]

# Check size
print(df_train.shape)
print(df_test.shape)

In [2]
# Method 2
df= pd.read_csv('D:\\Documents\\DATA\\airpass.csv')
train = df[:int(0.75*(len(df)))]
valid = df[int(0.75*(len(df))):]

# Check size
print(train.shape)
print(test.shape)
```

Ideally, we would analyze our time series data for stationarity, seasonality, etc., before sampling or partitioning it into training and test sets, rather than perform various data wrangling tasks on two subsets. So, we would perform seasonal decomposition on the entire set of air passenger data, first and the finding we make would also apply to the subsets.

How much to hold out is a judgement call and practical call. The more data we have, the more we can hold out. Moreover, we should know enough about the data from exploratory analysis as to whether 60% or

80% of the data should be holdout. No matter how much data we decide to hold out, we need to do it randomly.

Seasonal Decomposition

A time series with additive trend, seasonal, and irregular components can be decomposed using the `stl()` function. Note that a series with multiplicative effects can often be transformed into series with additive effects through a log transformation (i.e., `newts <- log(myts)`). The code below produces the plot in Figure 6-2.

```
# Seasonal decomposition
fit <- stl(airpass, s.window = "period")
plot(fit, col = 'purple', lwd = 2)
```

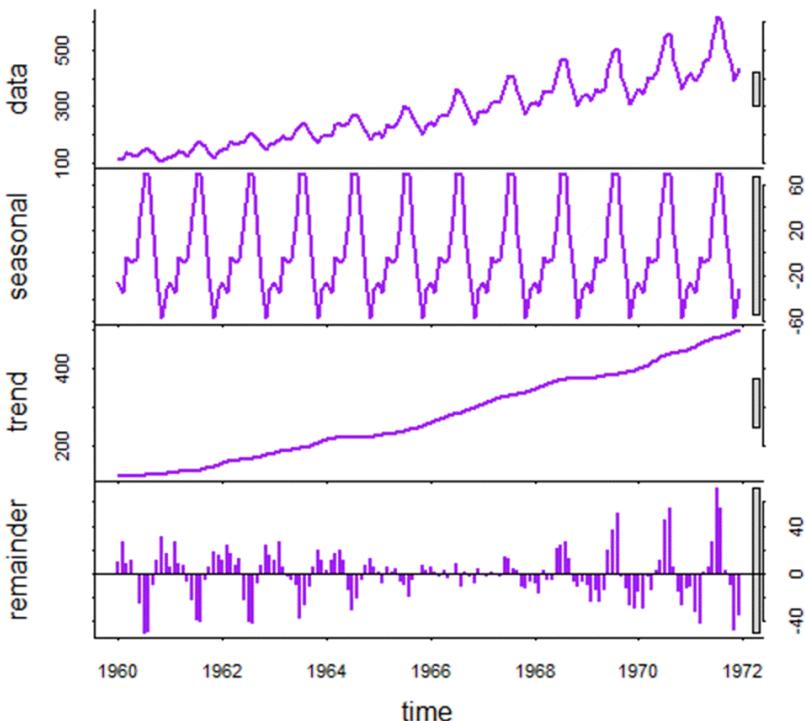


Figure 6-2. Seasonal decomposition of the Air Passenger data

A numerical test can be performed using the `trend.test` function from the `aTSA package` (Qiu, 2015), as follows:

```
trend.test(airpass, plot = TRUE)
```

```
Approximate Cox-Stuart trend test  
data: AirPassengers  
D- = 72, p-value < 2.2e-16  
alternative hypothesis: data have a decreasing trend
```

This verifies our conclusion that the trend is increasing.

R offers additional plots to examine components. Next, we make a `monthplot` (see Figure 6-3) and a `seasonplot` (see Figure 6-5).

```
# additional plots  
monthplot(AirPassengers, choice = 'seasonal', col.base = 'red',  
          col= 'blue'", lwd=2,  
          lty.base = 1, lwd.base = 3, xlab = 'Month',  
          ylab = 'Number of Passengers',  
          main = 'Monthly AirPassengers')
```

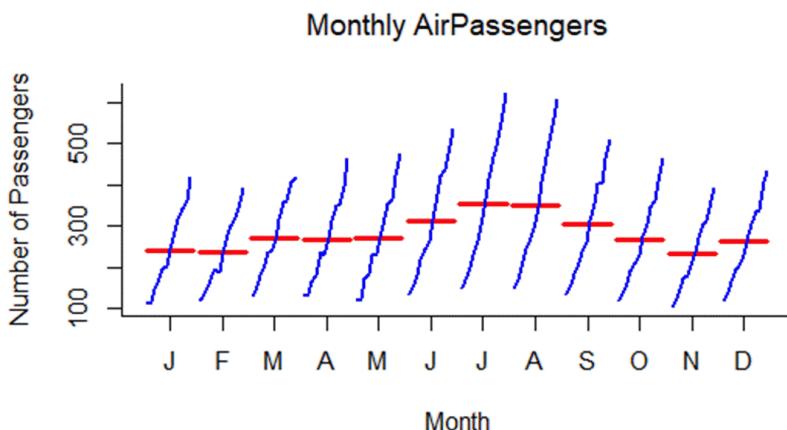


Figure 6-3. This plot depicts the mean air passengers per month over the span of years.

In Figure 6-5, we construct a seasonal plot with each year of the series on the vertical axis and each month of the year on the horizontal axis. If all horizontal lines were at the same level of air passenger, then the data would be stationary. For the plot aesthetics, we use colors from `RColorBrewer` palettes, which may require installation. The palettes can

be viewed using `display.brewer.all()`. The palette we are using can be viewed using:

```
display.brewer.pal(n = 12, name = 'Paired').
```



```
library("RColorBrewer")  
  
ggseasonplot(airpass,  
             col=brewer.pal(n = 12, name = '"Paired'),  
             year.labels=TRUE) +  
  geom_line(lwd=1.1) +  
  theme_gray()
```

The importance of identifying seasonality is critical at this point. We have seen trend and seasonality recently, and we discussed the need for triple exponential smoothing to deal with level, trend, and seasonality. Soon we will learn how to detect and remedy non-stationarity, including testing for it and removing it from time series data. This will become a requirement for modeling with Autoregressive Integrated Moving Average (ARIMA) models later in this chapter. Figure 6-4 shows some basic colors in Matplotlib.

Base Colors

b	c	k
g	m	w
r	y	

Tableau Palette

tab:blue	tab:brown
tab:orange	tab:pink
tab:green	tab:gray
tab:red	tab:olive
tab:purple	tab:cyan

Figure 6-4. Basic and Tableau colors in Python

Seasonal plot: airpass

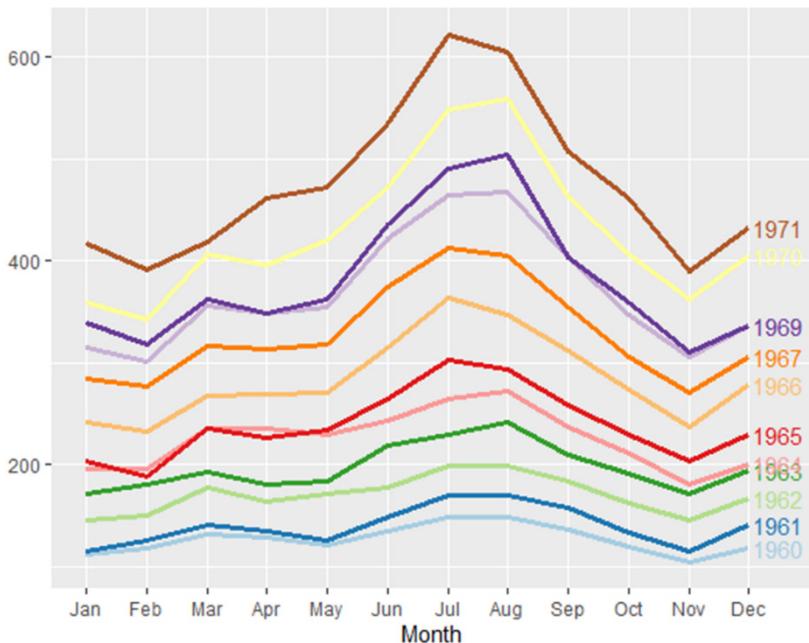


Figure 6-5. This plot depicts years by month

Stationarity and Differencing

In general, a stationary time series is one whose properties do not depend on the time at which the series is observed. So time series with trends, or with seasonality, are not stationary — the trend and seasonality will affect the value of the time series at different times. On the other hand, a white noise series is stationary — it does not matter when you observe it, as it should look much the same at any period.

Consider the nine series plotted in Figure 6-6. Which of these do you think are stationary?

Obvious seasonality rules out series *home sales* (e) and *beer* (c). Trends and changing levels rule out series *stock price* (g), *strikes* (f), *eggs* (d), and *pigs* (a). That leaves only *Stock % Change* (h) and *lynx* (b) as stationary series.

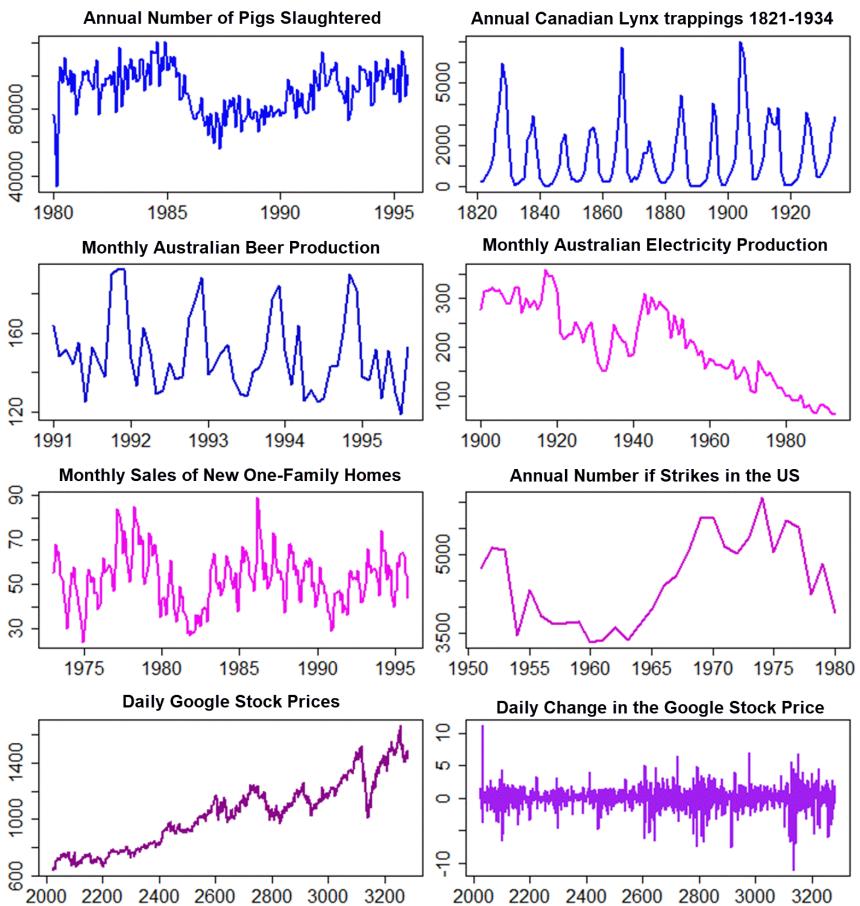


Figure 6-6. Which of these series are stationary? From top left to bottom right:
 (a) Monthly total of pigs slaughtered; (b) Annual total of lynx trapped; (c)
 Monthly Australian beer production; (d) Annual price of a dozen eggs; (e)
 Monthly sales of new one-family houses; (f) Annual number of strikes; (g)
 Google stock price; (h) Daily change in the Google stock (*Yahoo!, 2020*)

At first glance, the strong cycles in series (b) might appear to make it non-stationary. But these cycles are aperiodic — they are caused when the lynx population becomes too large for the available feed, so that they stop breeding and the population falls to low numbers, then the regeneration of their food sources allows the population to grow again,

and so on. In the long-term, the timing of these cycles is not predictable. Hence the series is stationary.

Testing for Stationarity

Unit root tests

One way to determine more objectively if differencing is required is to use a *unit root test*. These are statistical hypothesis tests of stationarity that are designed for determining whether differencing is required.

Dickey-Fuller Tests

The **Dickey-Fuller test** was the first statistical test developed to test the null hypothesis that a unit root is present in an autoregressive model of a given time series and that the process is thus not stationary. The original test treats the case of a simple lag-1 AR model.

Definition 6.1. Given a time series sample, an **augmented Dickey–Fuller test (ADF)** tests the null hypothesis that a unit root is present by estimating the regression model

$$Y_t' = \hat{\alpha}y_{t-1}' + \hat{\beta}_1y_{t-1}' + \hat{\beta}_2y_{t-2}' + \cdots + \hat{\beta}_ky_{t-k}',$$

where y_t' denotes the first-differenced series, $y_t' = y_t - y_{t-1}$, where $\hat{\alpha}$ is a constant, $\hat{\beta}$ the coefficient on a time trend, and k is the number of lags to include in the regression.

If the original series, y_t , needs differencing, then the coefficient $\hat{\alpha}$ should be approximately zero. If y_t is already stationary, then $\hat{\alpha} < 0$. In other words, the existence of a unit root means the series is not stationary and can cause unpredictable behavior.

The test has three versions that differ in the model of unit root process they test for.

Test for a unit root:

$$\Delta y_i = \delta y_{i-1} + u_i$$

Test for a unit root with trend:

$$y_i = a_0 + \delta y_{i-1} + u_i$$

Test for a unit root with trend and deterministic time trend:

$$\Delta y_i = a_0 + a_1^* t + \delta y_{i-1} + u_i$$

The choice of which version to use—which can significantly affect the size and power of the test—can use prior knowledge or structured strategies for series of ordered tests, allowing the discovery of the most fitting version.

Extensions of the test were developed to accommodate more complex models and data. These include the **Augmented Dickey-Fuller (ADF)** that (see Error! Reference source not found.), and the Phillips-Perron test (PP), which enhances robustness to unspecified autocorrelation and heteroscedasticity. First, we will use the ADF test in R. The fact that the p-value is less than 0.05 gives use cause to reject the null hypothesis (the series has a unit root) and treat the series as stationary. We can also compare the calculated DF_T statistic with a tabulated critical value. If the DF_T statistic is more negative than the table value, reject the null hypothesis of a unit root. Note: The more negative the DF test statistic, the stronger the evidence for rejecting the null hypothesis of a unit root. Existence of a unit root implies nonstationarity. Two values of the test statistic are given, one for unit root with trend, and the other for a unit root with trend and deterministic time trend.

```
adf.test(dta.ts)
```

```
Augmented Dickey-Fuller Test
```

```
data: dta.ts
Dickey-Fuller = -4.2592, Lag order = 11, p-value = 0.01
alternative hypothesis: stationary
```

In R, the default value of k is set to $\left\lfloor (T - 1)^{\frac{1}{3}} \right\rfloor$ where T is the length of the time series and $\lfloor x \rfloor$ means the largest integer not greater than x . The half-square brackets $\lfloor \quad \rfloor$ indicate the floor function.

Since the null-hypothesis for an ADF test is that the data are non-stationary, large p-values are indicative of non-stationarity, and small p-values suggest stationarity. Using the usual 5% threshold, differencing is required if the p-value is greater than $\alpha = 0.05$.

Another version of the ADF test comes from the [urca](#) package or the *Unit Root and Cointegration Tests for Time Series Data* package., which provides a little more information regarding test results. We can set the test to run with trend (drift) or without trend.

```
df=ur.df(dta.ts, type = 'drift', lags = 1)
summary(df)
```

```
#####
# Augmented Dickey-Fuller Test Unit Root Test #
#####

Test regression drift

Call:
lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.46750 -0.06836  0.00199  0.06864  0.40861 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.002452  0.002761   0.888   0.375    
z.lag.1     -0.033539  0.007646  -4.386 1.22e-05 ***  
z.diff.lag   -0.334089  0.023091 -14.468 < 2e-16 ***  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1 

Residual standard error: 0.1123 on 1675 degrees of freedom
Multiple R-squared:  0.1334,          Adjusted R-squared:  0.1324 
F-statistic: 129 on 2 and 1675 DF,  p-value: < 2.2e-16 

Value of test-statistic is: -4.3865 9.6919

Critical values for test statistics:
      1pct  5pct 10pct
tau2 -3.43 -2.86 -2.57
phi1  6.43  4.59  3.78
```

Phillips-Perron Test

Compared with the Augmented Dickey-Fuller test, **Phillips-Perron test** makes correction to the test statistics and is robust to the unspecified autocorrelation and heteroscedasticity in the errors.

Definition 6.2. The Phillips-Perron tests assess the null hypothesis of a unit root in a univariate time series y . The test regression for the PP tests is:

$$Y_t = c + \delta_t + \alpha y_{t-1} + e_t.$$

The null hypothesis restricts $\alpha = 1$.

Variants of the test, appropriate for series with different growth characteristics, restrict the drift and deterministic trend coefficients, c and δ , respectively, to be 0. The tests use modified Dickey-Fuller statistics (see `adftest`) to account for serial correlations in the innovations process e_t .

There are two types of test statistics, Z_ρ and Z_τ , which have the same asymptotic distributions as Augmented Dickey-Fuller test statistic, ADF. In order to calculate the test statistic, we consider three types of linear regression models. The first type (type1) is the one with no drift and linear trend with respect to time:

$$y_t = \alpha y_{t-1} + e_t,$$

where e_t is an error term. The second type (type2) is the one with drift but no linear trend:

$$y_t = \mu + \alpha y_{t-1} + e_t.$$

The third type (type3) is the one with both drift and linear trend from Definition 6.2:

$$y_t = c + \delta t + \alpha y_{t-1} + e_t,$$

where c is the drift coefficient, δ is the deterministic trend coefficient and α is the AR(1) coefficient ($\alpha < 1$). The p-value is calculated by the interpolation of test statistics from the critical values tables. The null hypothesis is y_t is a non-stationary time series.

```
pp.test(dta.ts)
```

```
Phillips-Perron Unit Root Test
```

```
data: dta.ts
```

```
Dickey-Fuller Z(alpha) = -258.52, Truncation lag parameter = 8,  
p-value = 0.01  
alternative hypothesis: stationary
```

Since the series appears to be stationary, we can turn to modeling it using ARIMA.

A third test for stationarity is the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test.

Definition 6.3. Given a times series sample, the **Kwiatkowski-Phillips-Schmidt-Shin** (KPSS) test partitions the series into three parts: a deterministic trend (βt), a random walk (r_t), and a stationary error (ε_t), with the regression equation:

$$y_t = r_t + \beta t + \varepsilon_t.$$

If the data is stationary, it will have a fixed element for an intercept or the series will be stationary around a fixed level (Wang, 2006, p. 33). The test uses OLS find the equation, which differs slightly depending on whether you want to test for level stationarity or trend stationarity (Kocenda & Cerný, 2007, p. 21). A simplified version, without the time trend component, is used to test level stationarity. We normally log-transform the data before running the KPSS test, to turn any exponential trends into linear ones.

To run the test, we use the `kpss.test()` function from the `tseries` package in R (Trapletti, Hornik, & LeBaron, 2015). This reverses the hypotheses, so the null-hypothesis is that the data are stationary. In this case, small p-values (e.g., less than $\alpha = 0.05$) suggest that differencing is required, i.e., the time series is not stationary.

```
kpss.test(diff(dta.ts, 1), null = "Trend")
```

```
KPSS Test for Trend Stationarity
```

```
data: diff(dta.ts, 1)  
KPSS Trend = 0.005404, Truncation lag parameter = 8, p-value=0.1
```

We prefer the `stationary.test()` from the `aTSA` package. This function combines the existing functions `adf.test()`, `pp.test()` and `kpss.test()` for testing the stationarity of a univariate time series x

```
stationary.test(ts(dta.ts), method="kpss")
```

```
KPSS Unit Root Test
alternative: nonstationary
```

```
Type 1: no drift no trend
```

```
lag stat p.value
 9 3.68    0.01
```

```
-----
```

```
Type 2: with drift no trend
```

```
lag stat p.value
 9 6.97    0.01
```

```
-----
```

```
Type 1: with drift and trend
```

```
lag stat p.value
 9 1.58    0.01
```

```
Note: p.value = 0.01 means p.value <= 0.01
      : p.value = 0.10 means p.value >= 0.10
```

In general, a stationary time series will have no predictable patterns in the long-term. Time plots will show the series to be roughly horizontal (although some cyclic behavior is possible) with constant variance. Definition 6.4 is an alternate version of Definition 2.8.

Definition 6.4. A *stationary time series* is one whose statistical properties such as mean, variance, autocorrelation, etc. are all constant over time has **statistical stationarity**.

Most business and economic time series are far from stationary when expressed in their original units of measurement, and even after deflation or seasonal adjustment they will typically still exhibit trends, cycles, random-walking, and other non-stationary behavior.

Definition 6.5. A series is said to be **trend-stationary** if it has a stable long-run trend and has a tendency to revert to the trend line following a disturbance

Random walk model

The differenced series is the *change* between consecutive observations in the original series, and can be written as

$$Y_t' = y_t - y_{t-1}.$$

The differenced series will have only $T - 1$ values since it is not possible to calculate a difference y_1' for the first observation.

When the differenced series is white noise, the model for the original series can be written as

$$y_t - y_{t-1} = e_t \text{ or } y_t = y_{t-1} + e_t.$$

A random walk model is very widely used for non-stationary data, particularly finance and economic data. Random walks typically have:

- long periods of apparent trends up or down
- sudden and unpredictable changes in direction.

The forecasts from a random walk model are equal to the last observation, as future movements are unpredictable, and are equally likely to be up or down. Thus, the random walk model underpins naïve forecasts.

A closely related model allows the differences to have a non-zero mean. Then

$$y_t - y_{t-1} = c + e_t \text{ or } y_t = c + y_{t-1} + e_t.$$

The value of c is the average of the changes between consecutive observations. If c is positive, then the average change is an increase in the value of y_t . Thus y_t will tend to drift upwards. But if c is negative, y_t will tend to drift downwards.

Here is a simulated Random Walk in R with plots shown in Figure 6-7, Figure 6-8, and Figure 6-9.

```
set.seed(1)
x <- w <- rnorm(1000)
for (t in 2:1000) x[t] <- x[t-1] + w[t]
layout(1:2)
plot(x, type="l", col="orange", lwd=2, main="Random Walk")
acf(x, col="red", lwd=3, main="Lag Plot of Random Walk")
```

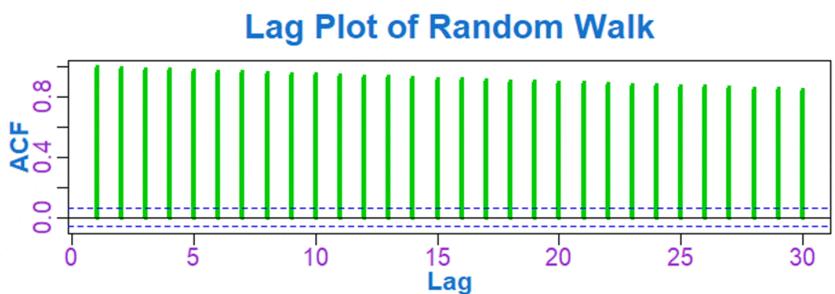
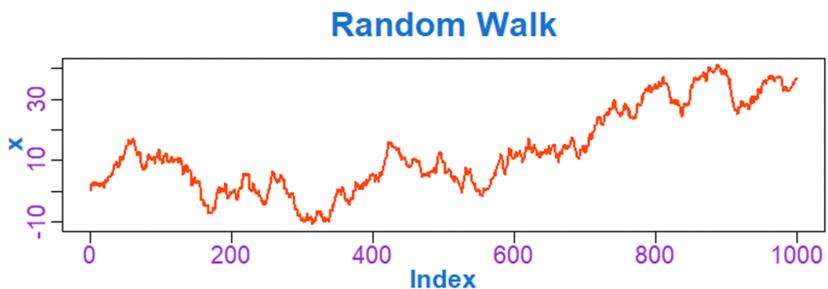


Figure 6-7. Time series and lag plots of the Random Walk simulated data

```
acf(diff(x), col = 'magenta', lwd = 3,
     main = 'Random Walk First-Order Differencing')
```

Note that since the first-order differences of a random walk are a white noise series, the correlogram of the series of differences can be used to judge whether a given series is reasonably modeled as a random walk.

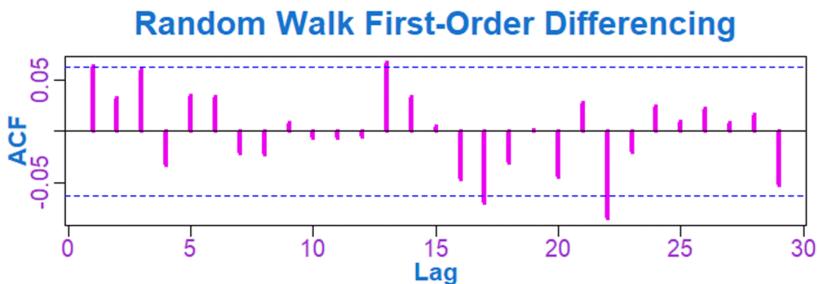


Figure 6-8. ACF lag plot of the Random Walk simulated data

In the following example, the random walk model is used to fit a time series of exchange rate (see Figure 6-9)

```
z <- read.csv("D:\\Documents\\DATA\\pounds_nz.txt", header=T)
```

```

Z.ts <- ts (Z,st=1991, fr=4)
plot(Z.ts, col = 'purple', lwd = 2, main = 'Pounds Random Walk')
acf(diff(Z.ts), col = 'red', lwd = 2,
     main = 'First-Order Difference Random Walk')

```

As can be seen from the correlogram of the time series difference, a significant value occurs at lag 1, suggesting random walk model is not adequate, and a more complex model may be needed.

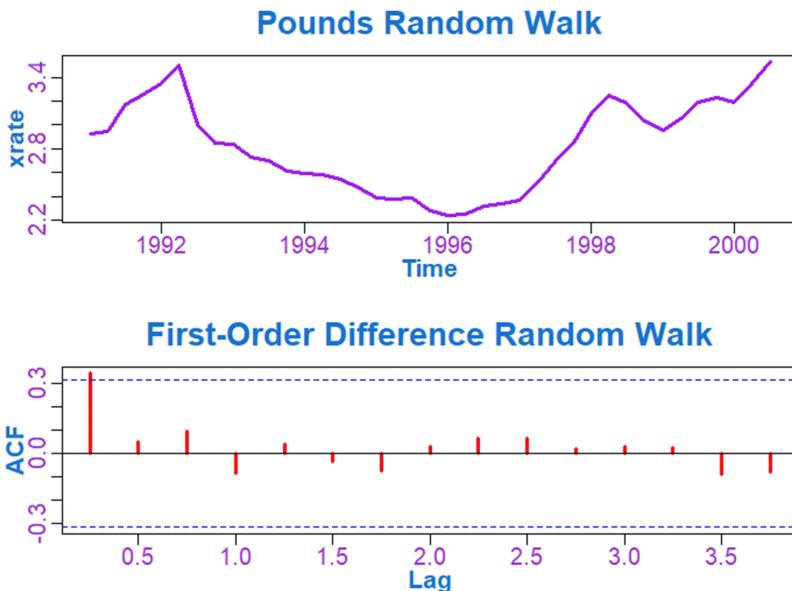


Figure 6-9. Plots of the Random Walk model for exchange rates

We may be able to **stationarize** a trend stationary series may be possible to it by de-trending (e.g., by fitting a trend line and subtracting it out prior to fitting a model), perhaps in conjunction with lagging or deflating. However, sometimes even de-trending is not sufficient to make the series stationary, in which case it may be necessary to transform it into a series of period-to-period and/or season-to-season differences.

Definition 6.6. A series is said to be **difference-stationary** if the mean, variance, and autocorrelations of the original series are not constant in time, even after detrending.

We may find that the statistics of the changes in a difference-stationary series between periods or between seasons will be constant. Sometimes it can be hard to tell the difference between a series that is trend-stationary and one that is difference-stationary, and a so-called **unit root test** may be used to get a more definitive answer. We will return to this topic later in the course.

Differencing a Time Series

ARIMA models are defined for stationary time series. Therefore, if you start off with a **non-stationary** time series, you will first need to '**difference**' the time series until you obtain a stationary time series. If you have to difference the time series d times to obtain a stationary series, then you have an ARIMA(p,d,q) model, where d is the order of differencing used.

First Order Differencing

The **first difference** of a time series is the series of changes from one period to the next.

Definition 6.7. If Y_t denotes the value of the time series Y at period t , then the **first difference** of Y at period t is equal to $Y_t - Y_{t-1}$. We define the difference operator as

$$y'_t = y_t - y_{t-1}$$

For n observations, the differenced series will only have $n - 1$ observations as it is not possible to calculate a difference for the first element of a series. In other words, every time differencing occurs an element is lost.

Differencing in R

In R we can use the **diff()** function for differencing a time series, which requires 3 arguments: **x** (the data), **lag** (the lag at which to difference), and **differences** (the order of differencing or k in y_t^k). We can

calculate the difference of the `loans` time series as shown in the R-code below and observe its effect on the time series data (see Figure 6-10):

```
library(forecast)
loans <- read.csv("D:\\Documents\\DATA\\loans.csv")
myvector=loans[,2]
# Transform loan data to a times series
loans.ts <- ts(myvector, start = c(2002, 1), end = c(2013, 4),
frequency = 12)
plot.ts(loans.ts, col = 4, lwd = 2, main = 'Loans Time Series')
# Apply differences = 1
loandiff1 <- diff(loans.ts, differences = 1)
plot.ts(loandiff1, col = 4, lwd = 2,
main = 'Loans Times Series Differencing')
```

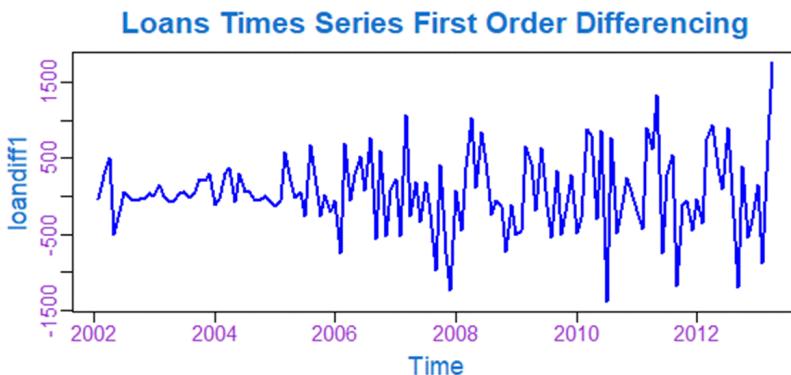


Figure 6-10. Result of differencing the time series data

Though the variance increases over time, the mean is fairly constant, so stationarity is achieved by differencing the series once. However, there are instances when the first difference is not enough to render the series stationary. In that event, we consider **second order differencing**.

Second Order Differencing

Occasionally the differenced data will not appear stationary, and it may be necessary to difference the data a second time to obtain a stationary series:

$$Y_t'' = y_t' - y_{t-1}' = (y_t - y_{t-1}) - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}.$$

In this case, y_t'' will have $T - 2$ values. Then we would model the “change in the changes” of the original data. In practice, it is almost never necessary to go beyond second-order differences.

Definition 6.8. If Y_t denotes the value of the time series Y at period t , then the **second difference** of Y at period t is equal to $Y'_t - Y'_{t-1}$. We define the difference operator as

$$\begin{aligned}y_t'' &= y'_t - y'_{t-1} \\&= (y_t - y_{t-1}) - (y_{t-1} - y_{t-2}) \\&= y_t - 2y_{t-1} + y_{t-2}\end{aligned}$$

Here the change in changes would be modelled, as well as there being two less data points belonging to the series. In most cases second order differencing is sufficient to make a series stationary, it is widely recommended to never go beyond seconder differencing.

The R code that follows uses the `differences` parameter to instruct the `diff()` function to use second order differencing. Figure 6-11 indicates that the resulting series is stationary.

```
# Second order differencing
loandiff2 <- diff(loans.ts, differences = 2)
plot.ts(loandiff1, col = 4, lwd = 2,
       main="Loans Times Series First Order Differencing")
plot.ts(loandiff2, col = 4, lwd = 2,
       main="Loans Times Series Second Order Differencing")
```

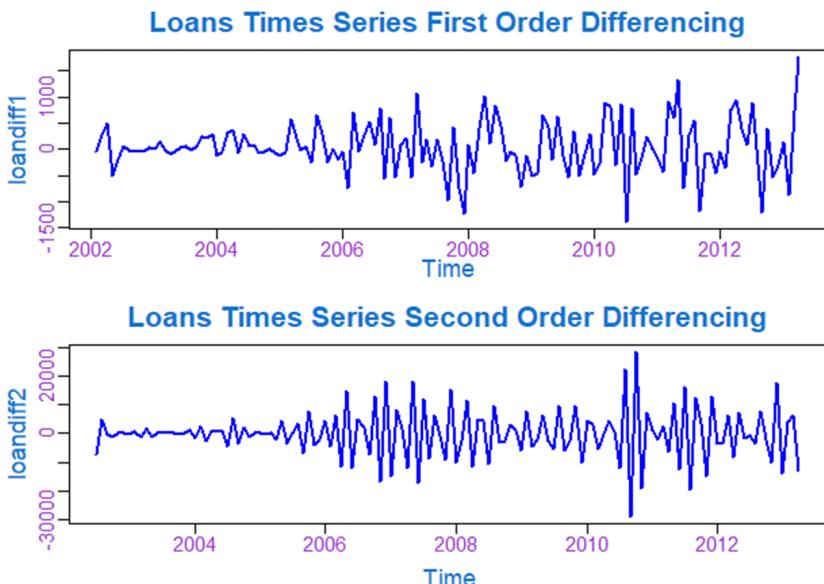


Figure 6-11. Twice differenced Unsecured Consumer Loan data

We can difference the time series (which we stored in “airpass”) once, and plot the differenced series (see Figure 6-12), by typing:

```
airdiff1 <- diff(airpass, differences = 1)
plot.ts(airdiff1 , lwd = 2, col = 'red')
```

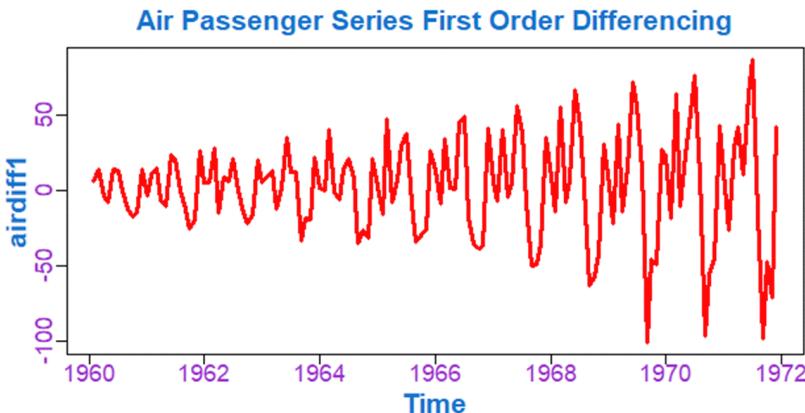


Figure 6-12. The first-differenced Air Passenger data

By applying a second difference, we should look for a drastic improvement. Otherwise, we stick with one difference. We can see from Figure 6-13 that we have not significantly improved matter with the second difference.

```
airdiff2 <- diff(airpass, differences = 2)
plot.ts(airdiff2 , lwd = 2, col = 'darkblue')
```

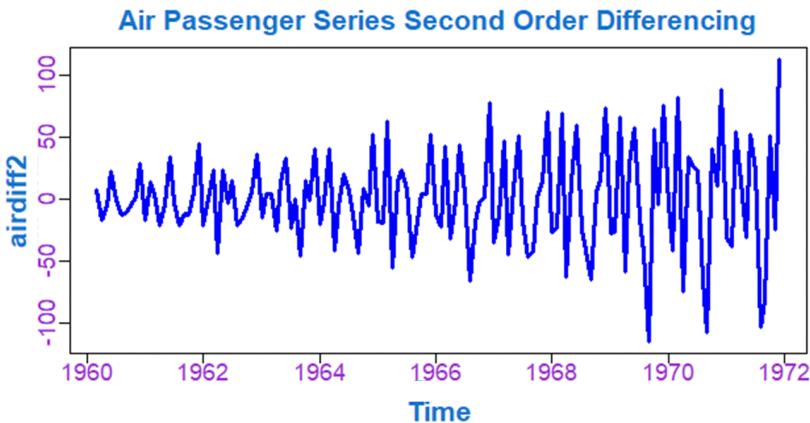


Figure 6-13. Air Passenger data with two differences applied

Now, we show both plots on one plot output Figure 6-14 so that we can compare them.

```
p1<-plot.ts(airdiff1, lwd = 2, col = 'red', xaxt = 'n',
              yaxt = 'n')
par(new=TRUE)
p2<-plot.ts(airdiff2, lwd = 2, col = 'blue')
```

Now we plot the time series with forecast for the first exponential smoothing model we constructed (see Figure 6-15). We will use the residual from this forecast to determine how we should specify the ARIMA(p,d,q) model, that is, how to select a candidate model.

```
#forecast using the ARIMA model
plot(for1, col = 'purple', lwd = 2)
```

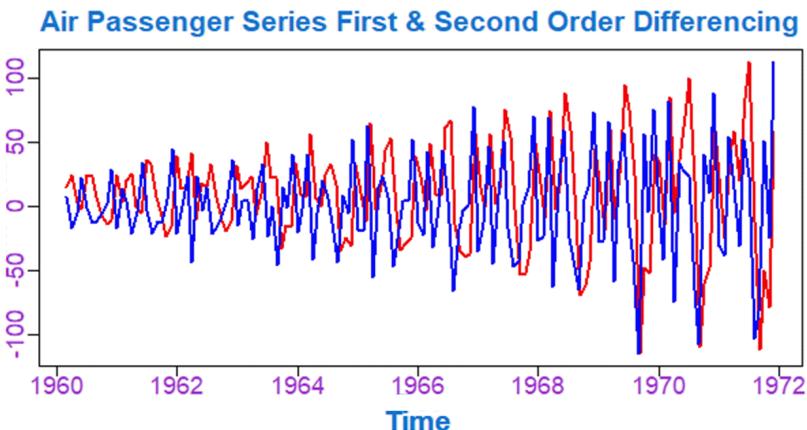


Figure 6-14. Both differenced series in one plot for comparison

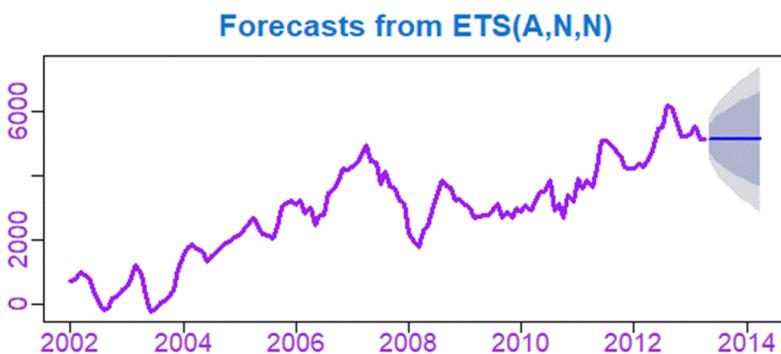


Figure 6-15. The air passenger forecast based on the single exponential smoothing model

ARIMA Models in R

Exponential smoothing methods are useful for making forecasts and make no assumptions about the correlations between successive values of the time series. However, if you want to make prediction intervals for forecasts made using exponential smoothing methods, the prediction intervals require that the forecast errors are uncorrelated and are normally distributed with mean zero and constant variance.

While exponential smoothing methods do not make any assumptions about correlations between successive values of the time series, in some cases you can make a better predictive model by taking correlations in

the data into account. **Autoregressive Integrated Moving Average (ARIMA)** models include an explicit statistical model for the irregular component of a time series, which allows for non-zero autocorrelations in the irregular component.

The `arima()` function can be used to fit an autoregressive integrated moving averages model. Other useful functions include:

<code>lag(ts, k)</code>	lagged version of time series, shifted back k observations
<code>diff(ts, differences=d)</code>	difference the time series d times
<code>ndiffs(ts)</code>	Number of differences required to achieve stationarity (from the <code>forecast</code> package)
<code>acf(ts)</code>	autocorrelation function
<code>pacf(ts)</code>	partial autocorrelation function
<code>adf.test(ts)</code>	Augmented Dickey-Fuller test. Rejecting the null hypothesis suggests that a time series is stationary (from the <code>tseries</code> -Package)
<code>Box.test(x, type="Ljung-Box")</code>	Pormanteau test that observations in vector or time series x are independent

Note that the `forecast` package has somewhat nicer versions of `acf()` and `pacf()` called `Acf()` and `Pacf()` respectively.

Selecting a Candidate ARIMA Model

If your time series is **stationary**, or if you have transformed it into a stationary time series by **differencing d times**, the next step is to select the appropriate ARIMA model, which means finding the values of most appropriate values of p and q for an ARIMA(p,d,q) model. To do this, you usually need to examine the correlogram and partial correlogram of the stationary time series. We can test for stationarity using the R function `stationary.test()` from the `aTSA` package (Qiu, 2015), as follows:

```
stationary.test(ts(airpass))
```

```
Augmented Dickey-Fuller Test
alternative: stationary
```

```
Type 1: no drift no trend
      lag      ADF p.value
[1,]  0   0.04712  0.657
[2,]  1  -0.35240  0.542
[3,]  2  -0.00582  0.641
[4,]  3   0.26034  0.718
[5,]  4   0.82238  0.879
Type 2: with drift no trend
      lag      ADF p.value
[1,]  0  -1.748  0.427
[2,]  1  -2.345  0.194
[3,]  2  -1.811  0.402
[4,]  3  -1.536  0.509
[5,]  4  -0.986  0.701
Type 3: with drift and trend
      lag      ADF p.value
[1,]  0  -4.64   0.01
[2,]  1  -7.65   0.01
[3,]  2  -7.09   0.01
[4,]  3  -6.94   0.01
[5,]  4  -5.95   0.01
----
```

```
Note: in fact, p.value = 0.01 means p.value <= 0.01
```

Based on the output from the Augmented Dickey-Fuller Test, with drift and trend we will require at least of difference.

To plot a **correlogram** and **partial correlogram**, we can use the `acf()` and `pacf()` functions in R, respectively. To get the actual values of the autocorrelations and partial autocorrelations, we set "`plot=FALSE`" in the `acf()` and `pacf()` functions.

To plot the partial correlogram for lags 1-20 for the once differenced time series of the Volume of Unsecured Consumer Loans, and get the values of the partial autocorrelations, we use the `pacf()` function.

The in-sample forecast errors are stored in the named element "residuals" of the list variable returned by `forecast`, `HoltWinters()` function. If the predictive model cannot be improved upon, there should be no correlations between forecast errors for successive predictions. In other words, if there are correlations between forecast errors for

successive predictions, it is likely that the simple exponential smoothing forecasts could be improved upon by another forecasting technique.

To figure out whether this is the case, we can obtain a correlogram of the in-sample forecast errors for lags 1-12. We can calculate a correlogram of the forecast errors using the `acf()` function in R. To specify the maximum lag that we want to look at, we use the “`lag.max`” parameter in `acf()`. We show a correlogram of the in-sample forecast errors for the Air Passenger data for lags 1-6 in Figure 6-16:

```
library(TSstudio)
ts_cor(ts.obj = for1$x, lag.max = 72, type = 'acf')
ts_cor(ts.obj = for1$x, lag.max = 72, type = 'pacf')
```

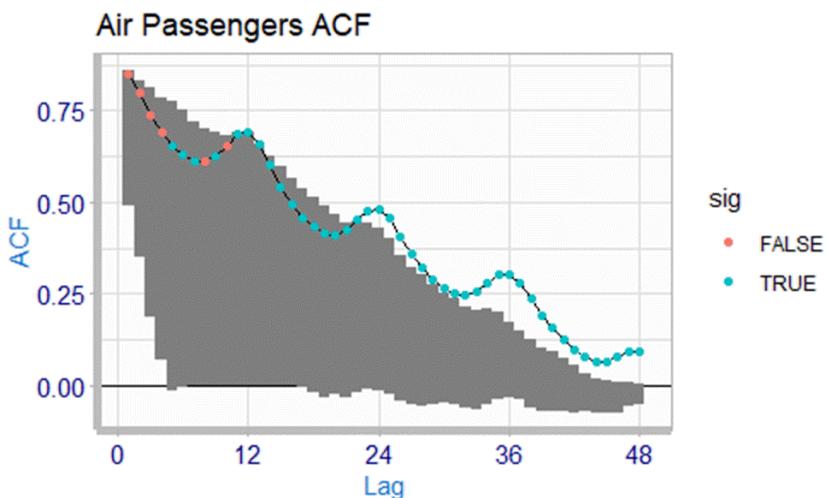


Figure 6-16. The autocorrelogram for Air Passenger time series data

We see from the correlogram (Figure 6-16) that the autocorrelations for the series is different than our previous correlograms, but we can clearly see the autocorrelations do tend to decrease and are in tolerance sometime after lag 6 (not depicted here). We would expect more than eight in 48 lags to exceed the 95% significance bounds by chance alone. The same correlation is represented in Figure 6-17, which is made using the `TSstudio` package.

`TSstudio` provides a set of tools for descriptive and predictive analysis of time series data. That includes functions for interactive visualization of

time series objects and as well utility functions for automation time series forecasting. We used `ts_cor` to produce Figure 6-16, Figure 6-17, Figure 6-18, and Figure 6-19. `ts_cor` provides an interactive visualization of the ACF and PACF in the in R-Studio's Viewer tab, rather than the Plots tab.

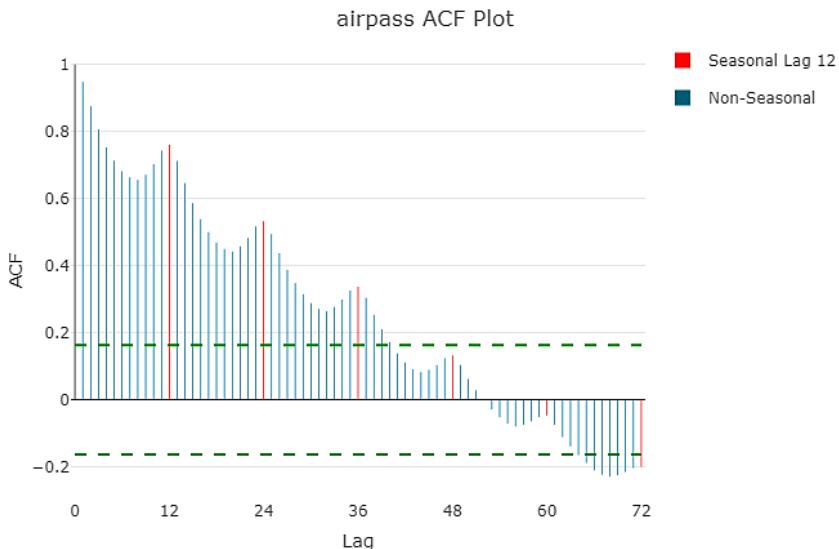


Figure 6-17. The autocorrelogram for Air Passenger time series data using TSstudio

So, now we turn to examine the PACF, which we plot using `ggtaperedpacf()` on Figure 6-18, and using `TSstudio` in Figure 6-19.

```
ggtaperedpacf(airpass, nsim=50) +
  ggtitle("Air Passengers PACF") + theme_light()
```

From the partial autocorrelogram, we see that the partial autocorrelation at beyond lag two are do not exceed the significance bounds. The partial autocorrelations approach zero as early as lag 48.

Since the partial correlogram is not out of tolerance within lag two (lag 1 is always 1.0), we can attempt to fit an Autoregressive Moving Average (ARMA) models to the time series data. The model is used to describe weakly stationary stochastic time series in terms of two polynomials. The

first of these polynomials is for autoregression (AR), the second for the moving average (MA).

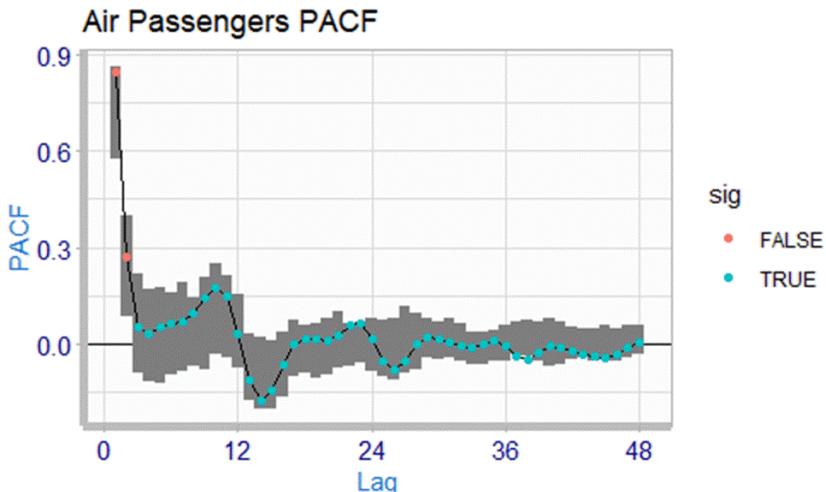


Figure 6-18. The partial autocorrelogram for Air Passenger time series data

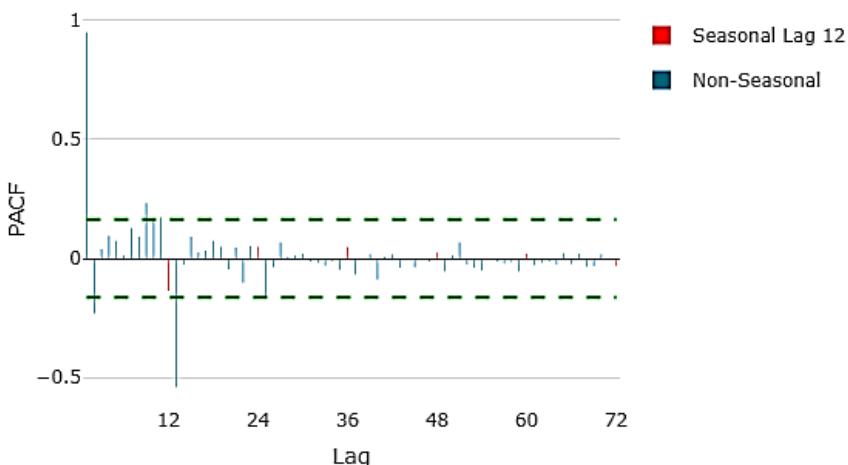


Figure 6-19. The partial autocorrelogram for Air Passenger time series data using TSstudio

You can see from the sample correlogram that the autocorrelation at lags 4, 6, 7, 11, and 12 are over the significance bounds. To test whether

there is significant evidence for non-zero correlations at lags 1-12, we can carry out a **Ljung-Box test**.

Definition 6.9. (Ljung-Box test) For a time series Y of length n , the errors are iid (independent and identically distributed), or the autocorrelations of the residuals are very small (near 0), if:

$$Q(m) = n(n + 2) \sum_{j=1}^m \frac{r_j^2}{n-j} < \chi_{1-\alpha h}^2$$

Where:

r_j = the accumulated sample autocorrelations

n = the sample size

m = the time lag

α = the level of significance

h = the degrees of freedom

The null hypothesis, H_0 , is that our model *does not* show lack of fit (the errors are iid). We reject the null hypothesis and say that the model shows lack of fit if $Q > \chi_{1-\alpha h}^2$, where χ^2 is a chi-square distribution with significance level α and h degrees of freedom. When the Ljung-Box test is applied to the residuals of an ARIMA model, the degrees of freedom h must be equal to $m-p-q$, where p and q are the number of parameters in the ARIMA(p,0,q) model.

This can be done in R using the `Box.test`, function from the `WeightedPortTest` package. The maximum lag that we want to look at is specified using the `lag` parameter in the `Box.test()` function. For example, to test whether there are non-zero autocorrelations at lags 1-12, for the in-sample forecast errors for Air Passenger data, we type:

```
Box.test(for1$fitted, lag = 20, type = 'Ljung-Box')
```

Box-Ljung test

```
data: for1$fitted  
X-squared = 1399.7, df = 20, p-value < 2.2e-16
```

```
Box.test(for1$residuals, lag = 20, type = 'Ljung-Box')
```

```
Box-Ljung test
```

```
data: for1$residuals  
X-squared = 231.7777, df = 20, p-value < 2.2e-16
```

ARMA versus ARIMA

It is not really a matter of one versus the other. It is more of a matter of how time data is categorized and consequently, we have to ask whether we deal with stationarity or not. An Autoregressive Moving Average (ARMA) model, is used to describe weakly stationary stochastic time series in terms of two polynomials. The first of these polynomials is for autoregression (AR) and the second for the moving average (MA).

Definition 6.10. The **ARMA(p,q) model** with parameters p and q , where

p is the order of the autoregressive polynomial, and
 q is the order of the moving average polynomial.

is defined by the equation is given by:

$$X_t = c + \varepsilon_t + \sum_{i=1}^p \varphi_i X_{t-i} + \sum_{i=1}^q \theta_i \varepsilon_{t-i}$$

where:

φ = the autoregressive model's parameters,

θ = the moving average model's parameters.

c = a constant,

ε = error terms (white noise).

Now, we could try one of the following ARMA models

- an ARMA(1,0) model, uses AR to deal with autocorrelation, but our shows that residuals outside the tolerance are steadily being reduced
- an ARMA(0,1) model, uses MA to deal with the autocorrelation of residuals, but that is not a problem for ours

- an ARMA(1,1) model, to deal with both autocorrelation problem, but we have neither.

Definition 6.10 implies that $\text{AR}(p)$ makes predictions using previous values of the dependent variable. Two of these, ARMA(1,0) and ARMA(1,1) uses the autoregressive term AR. An **AR (autoregressive) model** is usually used to model a time series which shows longer-term dependencies between successive observations. Intuitively, it does not make sense that an AR model to describe the time series of Volume of **airpass**, as we would not expect the number of passengers in one period, say a given year, to affect later years. Other things could have an effect, like aircraft accidents, terrorist alerts, or COVID-19, but for this analysis we are not looking at external effects. Consequently, autocorrelation is not a huge issue with our time series, so we overlook the AR parameter (i.e., $\text{AR} = 0$).

Definition 6.10 implies that $\text{MA}(q)$ makes predictions using the series mean and previous errors. Although the partial autocorrelation of the residuals is not severe, we may want to use an MA model for the **airpass** time series. With that in mind, we could use an AMRA(0,1) model.

However, stationarity did pose a considerable problem with the **airpass** time series. Recalling the one lag to achieve stationarity, along with the one moving average, we would want to use a model that included MA, but also accounted for a fix to the stationarity problem. This would give a third parameter we'll call **DIFF**. So, our model would be an ARMA(0,1,1).

Now, ARMA(0,1,1) is not an $\text{ARMA}(p, q)$ model. It is more properly an ARMA(0,1,1), but we have not introduced the third parameter, **DIFF**. In fact, this would be an Auto Regressive Integrated Moving Average (ARIMA) model, with p, q , and d , where d is the differencing component.

Definition 6.11. If the times series y_t is differenced d times, and it then follows an ARMA(p, q) process, then it is an ARIMA(p, d, q) series:

$$y'_t = c + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t,$$

where y'_t is the differenced series (it may have been differenced more than once).

This means that our model will be an ARIMA(p, q, d) with no AR term, one MA term, and one differencing (DIFF) term, or ARIMA(0,1,1).

Fitting an ARIMA Model

Now we fit an ARIMA(0,1,1) model to the `airpass` time series data. There are several ways to do this in R. One is the `arima()` function from the basic `stats`-Package. A better way is with the `Arima()` function from the `forecast` package, which is mostly a wrapper for the `arima()` function. The argument, `order=c(0,1,1)`, is the model specification for P=AR, D=DIFF, and Q=MA components. We are also assuming seasonality and differencing one time with one moving average using `seasonal=list(order=c(0,1,1), period=12)`. The term `period = 12` indicates the length of the season. The term `lambda` is the Box-Cox transformation parameter.

```
# fit an ARIMA model of order P, D, Q
# Fit model to first few years of Air Passenger data
air.model1 <- Arima(window(airpass,end = 1967+11/12),
                     order = c(0,1,1),seasonal = list(order = c(0,1,1),
                     period = 12), lambda = 0)
air.model1
```

Series: window(airpass, end = 1967 + 11/12)

ARIMA(0,1,1)(0,1,1)[12]

Box Cox transformation: lambda= 0

Coefficients:

ma1	sma1
-0.3941	-0.6129
s.e.	0.1173 0.1076

sigma^2 estimated as 0.001556: log likelihood=148.76
AIC=-291.53 AICc=-291.22 BIC=-284.27

The second line of output shows the model: ARIMA(0,1,1)(0,1,1)[12]. This indicates an ARIMA(0,1,1) base model with seasonal differencing moving average (0,1,1), and period length of 12, fits rather well. So, now we build a forecast that runs from 1968 to 1972 and plot it in Figure 6-20. The `abline()` function adds a vertical line to the plot.

```
# Forecast accuracy measures on the log scale.
# in-sample one-step forecasts.
plot(forecast::forecast(air.model11, h = 48))
abline(v = 1968, lty = 3)
```

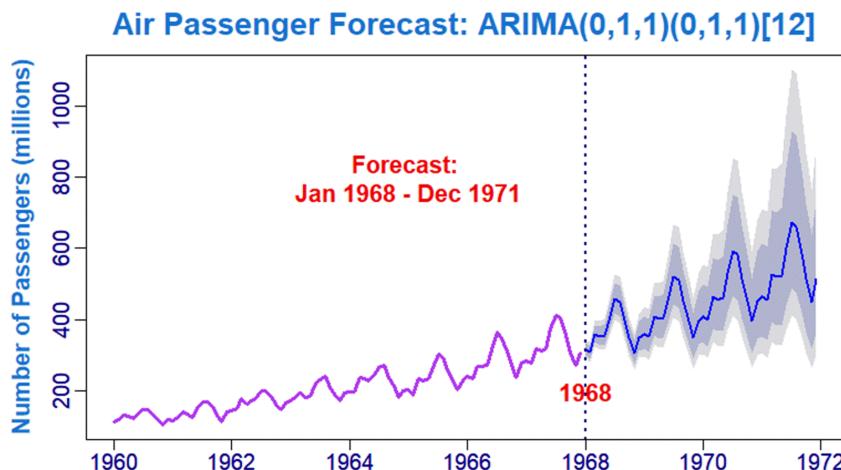


Figure 6-20. Forecast of air passengers for 1968 to 1972

In the next snippet of code, we fit `airpass` data using the ARIMA(0,1,1) model we just built, and begin the forecast where it left off in 1971. Notice that in the `Arima()` function, we specify `air.model11` as a parameter. The `window()` function introduced in Chapter 2 is useful when extracting a portion of a time series, such as we need when creating training and test sets. In the `window()` function, we specify the start and/or end of the portion of time series required using time values. For example, `window(airpass, start=1965)` extracts all data from 1965 onward. We plot the forecast in Figure 6-21.

```
air.model12 <- Arima(window(airpass,start=1965),model=air.model11)
air.model12
plot(forecast::forecast(air.model12, h = 48))
abline(v = 1972, lty = 3)
```

```

Series: window(airpass, start = 1960)
ARIMA(0,1,1)(0,1,1)[12]
Box Cox transformation: lambda= 0
Coefficients:
          ma1      sma1
        -0.3941   -0.6129
  s.e.    0.0000   0.0000
sigma^2 estimated as 0.001556:  log likelihood=244.39
AIC=-486.77   AICc=-486.74   BIC=-483.9

```

Air Passenger Forecast: ARIMA(0,1,1)(0,1,1)[12]

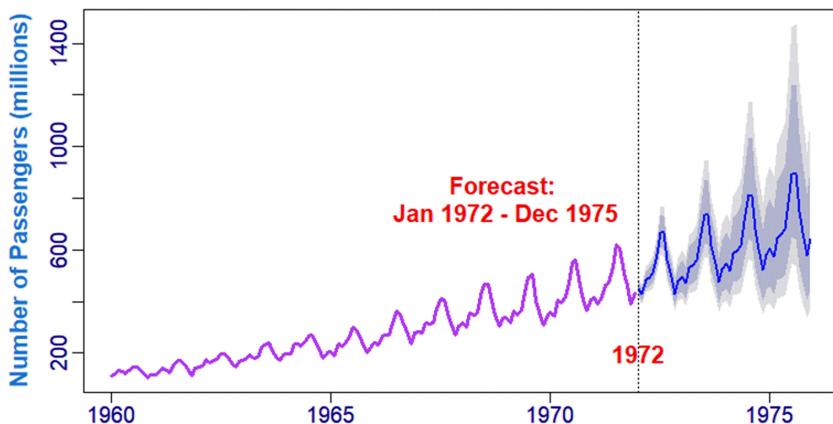


Figure 6-21. Air passenger forecast based on fitted model for later data

Next, we construct a new ARIMA(0,1,1) model without seasonal and period (we call this a structured-naïve approach since we have considered the AR, MA, and DIFF components). We also build the forecast and plot it in Figure 6-22.

```

# fit an ARIMA model of order P, D, Q
par(mar = c(4, 4, 3, .75), lwd = 3)
air.model3 <- Arima(airpass, order = c(0,1,1))
air.forecast3 <- forecast::forecast(air.model3, h = 48)
summary(air.forecast3)
plot(air.forecast3)
lines(airpass)
title(xlab = "Years", ylab = "Number of Passengers")
summary(air.forecast3)

```

Forecast method: ARIMA(0,1,1)

Model Information:

Series: airpass

ARIMA(0,1,1)

```

Coefficients:
      ma1
      0.4027
s.e.  0.0892
sigma^2 estimated as 1003:  log likelihood=-696.63
AIC=1397.26  AICc=1397.34  BIC=1403.18

```

Error measures:

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Training set	1.7195	31.4528	24.5518	0.3796	8.6849	0.7665	-0.0223

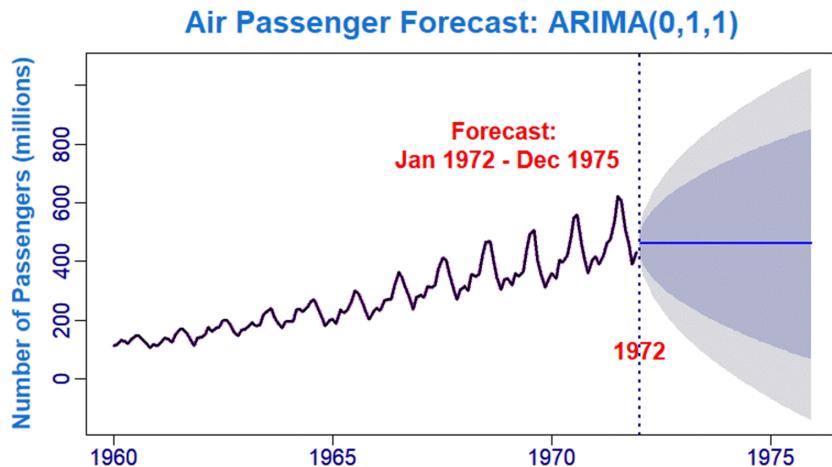


Figure 6-22. Air passenger forecast ARIMA(0,1,1) without seasonality and period assumptions

Now, we can use the `auto.arima()` function from the `forecast` package to construct a new model without making any assumptions about the time series data (we call this a naïve approach since we are allowing R to automate the process). The resulting model is an ARIMA(2,1,1)(0,1,0)[12]. The data and forecast are shown in Figure 6-23.

```

air.model4 <- auto.arima(airpass)
air.forecast4 <- forecast(air.model4, h = 48)
plot(air.forecast4)
abline(v = 1972, lty = 3)

```

```

Series: airpass
ARIMA(2,1,1)(0,1,0)[12]
Coefficients:
      ar1      ar2      ma1
      0.5960   0.2143  -0.9819

```

```

s.e. 0.0888 0.0880 0.0292
sigma^2 estimated as 132.3: log likelihood=-504.92
AIC=1017.85   AICc=1018.17   BIC=1029.35

```

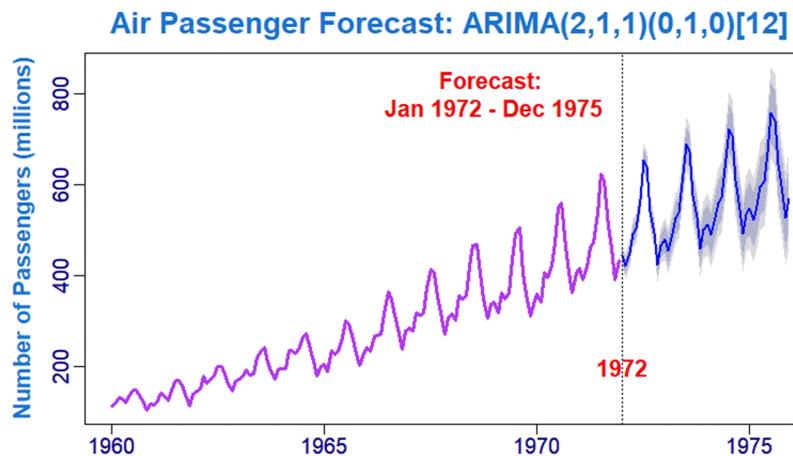


Figure 6-23. Auto ARIMA (ARIMA(2,1,1)(0,1,0)[12]) forecast

As before, the second line of output shows the model: ARIMA(2,1,1)(0,1,0)[12]. This indicates an ARIMA(2,1,1) base model with seasonal differencing (0,1,0), and period length of 12.

Next, we will use the first ARIMA model, `air.model1`, with several different forecasting schemes by iterating from 1965-1968. The code snippet iterates on `t` and then plots the four different forecasts with start years at 1965, 1966, 1967, and 1968, indicated by the abline. Notice that the data being fitted runs from January 1960 to December 1964, and subsequently to 1965, 1966, and 1967. We plot them in Figure 6-24.

```

par(mfrow = c(2,2))
for (t in 1964:1967){
  air.model1 = Arima(window(airpass, end = t + 11/12),
    order = c(0,1,1), seasonal = list(order = c(0,1,1),
    period = 12), lambda = 0)
  plot(forecast::forecast(air.model1, h = 84-12*(t - 1964)),
    xlim = c(1960, 1972),
    ylim = c(100, 1400))
  lines(airpass)
  abline (v = t+1, lty = 3)}

```

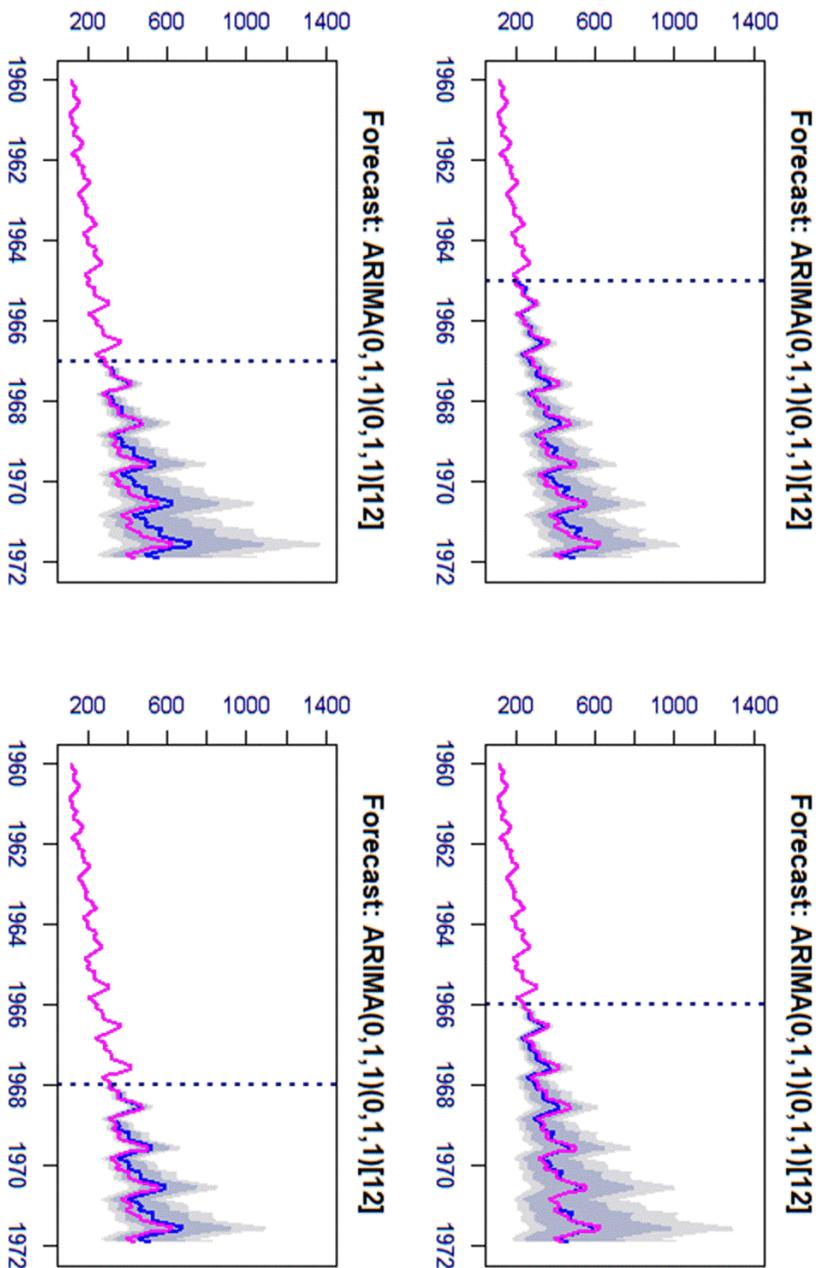


Figure 6-24. Four forecasts using `air.model1` and starting with 1965, then 1966, 1967, and 1968

Note that the forecasts are more accurate as the forecast period decreases, so that there is less variance in the 1968 – 1972 forecast.

Assessing Performance and Diagnostics

To assess the accuracy of the out-of-sample one-step forecasts, use the `accuracy()` function from the `forecast` package in R. We do this for each ARIMA model we constructed. The measures calculated are:

ME: Mean Error

RMSE: Root Mean Squared Error

MAE: Mean Absolute Error

MPE: Mean Percentage Error

MAPE: Mean Absolute Percentage Error

MASE: Mean Absolute Scaled Error

ACF1: Autocorrelation of errors at lag 1

By default, the MASE calculation is scaled using MAE of training set naive forecasts for non-seasonal time series, training set seasonal naive forecasts for seasonal time series and training set mean forecasts for non-time series data.

```
specify_decimal <- function(x, k) trimws(format(round(x, k),
nsmall=k))

acc1 <- specify_decimal(accuracy(air.forecast1), 4)
acc2 <- specify_decimal(accuracy(air.forecast2), 4)
acc3 <- specify_decimal(accuracy(air.forecast3), 4)
acc4 <- specify_decimal(accuracy(air.forecast4), 4)

row.names <- c('ME', 'RMSE', 'MAE', 'MPE', 'MAPE', 'MASE', 'ACF1')
column.names <- c('Model1', 'Model2', 'Model3', 'Model4')
result <- array(c(acc1, acc2, acc3, acc4), dim = c(7,4),
dimnames = list(row.names, column.names))
print(result, quote = FALSE)
```

	Model1	Model2	Model3	Model4
ME	0.3576	0.0471	1.7195	1.3423
RMSE	7.8973	10.2244	31.4528	10.8462
MAE	5.7883	7.4384	24.5518	7.8675

MPE	0.1458	-0.0119	0.3796	0.4207
MAPE	2.6702	2.6386	8.6849	2.8005
MASE	0.1982	0.2322	0.7665	0.2456
ACF1	0.0581	-0.0244	-0.0223	-0.0012

It would appear that we have our best model with air.model1, [Arima\(0,1,1\)\(011\)\[12\]](#), where RMSE is 7.8973 and MAPE is 2.6702. These are the smallest value when comparing model metrics.

Final Model Diagnostics

This R function, [plotForecastErrors\(\)](#), written by Avril Coghlan (Coghlan, 2016), is useful for residual diagnostics (it is a standard output from PROC UCM in SAS). Once we define the function in R-studio, the usage is [plotForecastErrors\(forecast\\$residuals\)](#) (see Figure 6-25).

```
plotForecastErrors <- function(forecasterrors)
{
  # make a histogram of the forecast errors:
  mybinsize <- IQR(forecasterrors)/4
  mysd   <- sd(forecasterrors)
  mymin  <- min(forecasterrors) - mysd*5
  mymax  <- max(forecasterrors) + mysd*3

  # generate N~(0, mysd) data
  mynorm <- rnorm(10000, mean = 0, sd = mysd)
  mymin2 <- min(mynorm)
  mymax2 <- max(mynorm)
  if (mymin2 < mymin) { mymin <- mymin2 }
  if (mymax2 > mymax) { mymax <- mymax2 }

  # make a red histogram of the forecast errors
  # with the normally distributed data overlaid:
  mybins <- seq(mymin, mymax, mybinsize)
  hist(forecasterrors, col = "red", freq = FALSE,
        breaks=mybins)

  # freq=FALSE ensures the area under the histogram = 1
  # generate N~(0, mysd) data
  myhist <- hist(mynorm, plot = FALSE, breaks = mybins)
  # plot the normal curve as a blue line on top of the histogram of forecast errors:
  points(myhist$mid, myhist$density, type = "l",
         col = "blue", lwd = 2)
}
```

```
plotForecastErrors(air.forecast2$residuals)
```

Histogram of forecasterrors

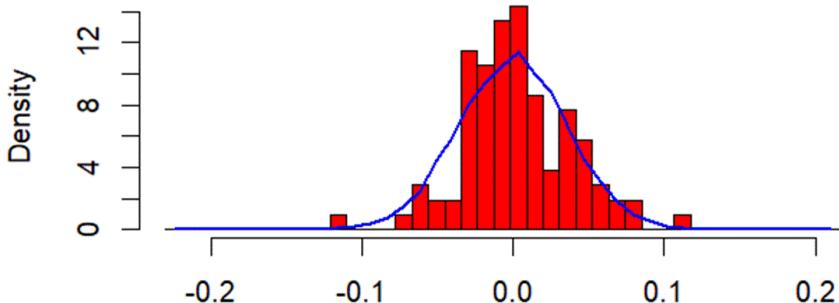


Figure 6-25. Plot to see if the forecast error are approximately normally distributed

To plot the ACF and PACF, we wrote code that provided a fixed scale from -1 to 1, and contains colors as seen in Figure 6-26.

```
##### better ACF plot #####
plot.acf <- function(ACFobj) {
  rr <- ACFobj$acf[-1]
  kk <- length(rr)
  nn <- ACFobj$n.used
  plot(seq(kk), rr, type = "h", lwd = 2, col="red", yaxs = "i",
xaxs = "i",
       ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab =
"Lag",
       ylab = "Correlation", las = 1)
  abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
  abline(h = 0)
}

##### better PACF plot #####
plot.pacf <- function(PACFobj) {
  rr <- PACFobj$acf
  kk <- length(rr)
  nn <- PACFobj$n.used
  plot(seq(kk), rr, type = "h", lwd = 2, col="red", yaxs = "i",
xaxs = "i",
       ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab =
"Lag",
       ylab = "PACF", las = 1)
  abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
}
```

```
    abline(h = 0)
}
```

Now we plot and examine the acf (Figure 6-26) and pacf (Figure 6-27).

```
airpass.acf <- acf(air.forecast1$x, lag.max = 72)
plot.acf(airpass.acf)
```

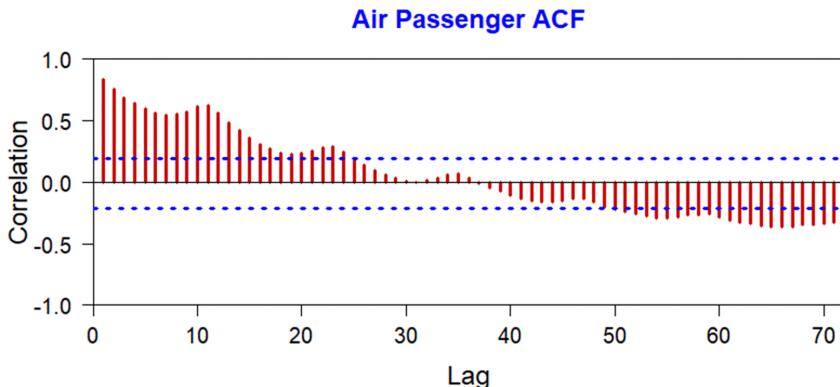


Figure 6-26. Correlogram for air.model1 forecast

```
airpass.pacf <- pacf(air.forecast1$residuals, lag.max = 72)
plot.pacf(airpass.pacf)
```

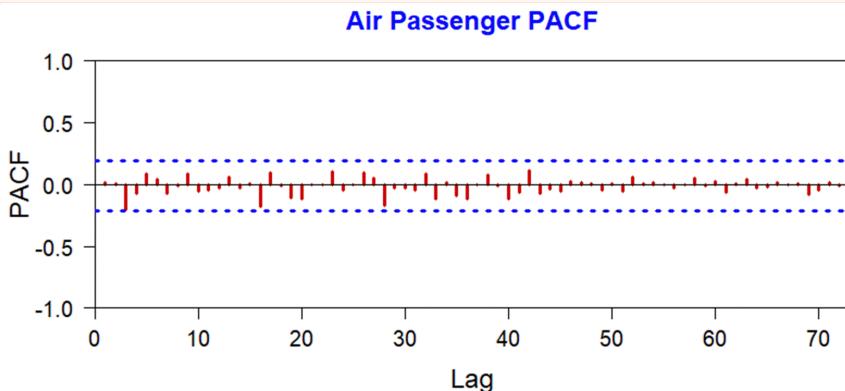


Figure 6-27. Partial correlogram for air.model1 forecast

We can conclude from Figure 6-26 and Figure 6-27 that the residuals do not show any anomalies to be concerned about.

To further examine the fit, we can also perform a weighted Box-Ljung test using the [Weighted.Box.test](#) function from the [WeightedPortTest](#)

package in R (Fisher & Gallagher, 2012). Weighted portmanteau tests are used for testing the null hypothesis of adequate ARMA fit and/or for detecting nonlinear processes (Box & Pierce, Distribution of residual correlations in autoregressive-integrated moving average time series models., 1970) (Ljung & Box, 1978). The form of the test is specified by the `type` parameter.

```
Weighted.Box.test(pred$pred, lag = 10, type = "Ljung")
```

```
Weighted Ljung-Box test (Gamma Approximation)

data: pred$pred
Weighted X-squared on Residuals for fitted ARMA process =
Inf, Shape = 3.9286, Scale = 1.4000, p-value < 2.2e-16
```

An alternative for fit diagnostics is to fit an ARIMA model with the `estimate()` function or use the `ts.diag()` function, both from the *aTSA* package (Qiu, 2015). Part of the output is the ACF plot, PACF plot, p.value of white noise checking plot, and Q-Q plot for residuals shown in Figure 6-28.

```
estimate(airpass, p = 0, d = 1, q = 1, PDQ = c(0, 1, 1))
```

```
SARIMA(0,1,1)(0,1,1)(12) model is estimated for variable:airpass

Conditional-Sum-of-Squares & Maximum Likelihood Estimation
      Estimate    S.E t.value p.value Lag
MA 1   -0.309 0.0890   -3.47 0.000689   1
SMA 1   -0.107 0.0828   -1.30 0.196595   1
-----
n = 144; 'sigma' = 11.63717; AIC = 1021.003; SBC = 1026.943
-----
Correlation of Parameter Estimates
      MA 1    SMA 1
MA 1   1.000 -0.127
SMA 1 -0.127  1.000
-----
Autocorrelation Check of Residuals
      lag    LB p.value
[1,]   4  5.26  0.2617
[2,]   8  7.10  0.5254
[3,]  12 11.45  0.4912
[4,]  16 14.85  0.5353
[5,]  20 22.54  0.3121
[6,]  24 38.56  0.0303
```

```

Model for variable: airpass
Period(s) of Differencing: airpass(1,1)

MA factors: 1 - 0.3087 B**(1)
SMA factors: 1 - 0.1074 B***(12)

```

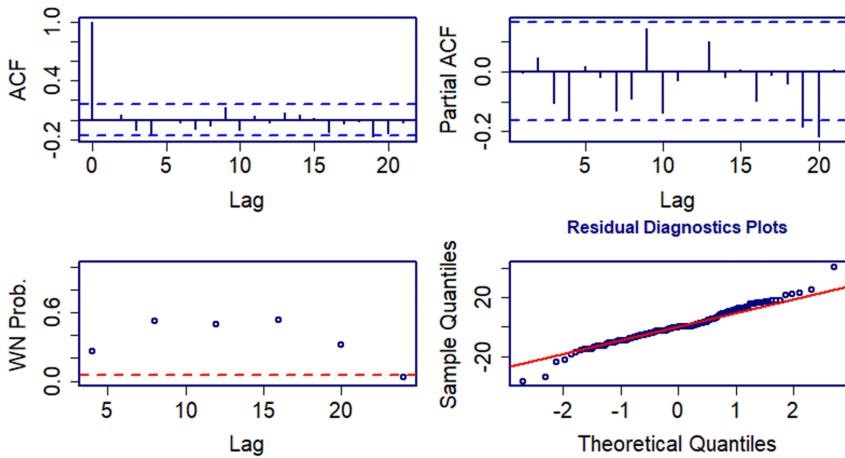


Figure 6-28. Diagnostics plots generated by the `estimate()` function

Another alternative that produces the desired diagnostic charts is to use the `Descriptives()` function from the `AnalyzeTS` package, as follows (Han, Nghi, Diem, My, & Minh, 2015). The function produces a time series plot, error histogram, ACF and PACF as shown in Figure 6-29.

```

par(mfrow = c(2,2), mar = c(5,5,4,.5), col = 'dodgerblue3',
    cex.lab = 1.45, cex.axis = 1.45, cex.main = 1.5, font = 2,
    col.main = 'dodgerblue3', col.lab = 'dodgerblue3')

plot(air.forecast1$residuals) +
  title(main = "Air Passenger Residuals")

plotForecastErrors(air.forecast1$residuals) +
  title(main = "")

plot.acf(airpass.acf) +
  title(main = "Air Passenger ACF")

plot.pacf(airpass.pacf) +
  title(main = "Air Passenger PACF")

```

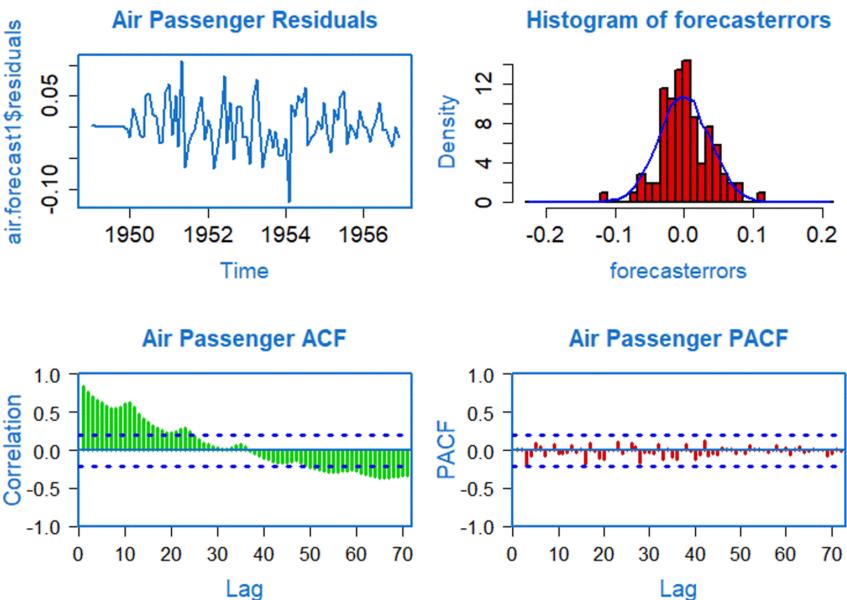


Figure 6-29. Diagnostic plots procied by `Descriptives()`

Next, using the Ljung-Box test, we get an insignificant p-value and conclude that the serieid is not autocorrelated.

```
Box.test(air.forecast1$residuals, lag = 20, type = 'Ljung-Box')
```

```
Box-Ljung test
data: air.forecast2$residuals
X-squared = 35.306, df = 48, p-value = 0.9134
```

Forecast Evaluation Metrics

After fitting several different regression or time series forecasting models to a given data set, you have many criteria by which they can be compared. We briefly discussed some of these in Chapter 3.

- **Error measures in the estimation period:** root mean squared error, mean absolute error, mean absolute percentage error, mean absolute scaled error, mean error, mean percentage error
- **Error measures in the validation period (if you have done out-of-sample testing):** Ditto

- **Residual diagnostics and goodness-of-fit tests:** plots of actual and predicted values; plots of residuals versus time, versus predicted values, and versus other variables; residual autocorrelation plots, cross-correlation plots, and tests for normally distributed errors; measures of extreme or influential observations; tests for excessive runs, changes in mean, or changes in variance (lots of things that can be "OK" or "not OK")

Qualitative considerations: intuitive reasonableness of the model, the simplicity of the model, and above all, *usefulness for decision making!*

With so many plots and statistics and considerations to worry about, it's sometimes hard to know which comparisons are most important. What's the real bottom line?

If there is any one statistic that normally takes precedence over the others, it is the **root mean squared error (RMSE)**, which is the square root of the mean squared error. When it is adjusted for the degrees of freedom for error (sample size minus number of model coefficients), it is known as **the standard error of the time series model** or **standard error of the estimate** in time series analysis or as the **estimated white noise standard deviation** in ARIMA analysis. This is the statistic whose value is minimized during the parameter estimation process, and it is the statistic that determines the width of the confidence intervals for predictions. It is a lower bound on the standard deviation of the forecast error (a tight lower bound if the sample is large and values of the independent variables are not extreme), so a 95% confidence interval for a forecast is approximately equal to the point forecast "plus or minus 2 standard errors", i.e., plus or minus 2 times the standard error of the time series model.

However, there are a number of other error measures by which to compare the performance of models in absolute or relative terms:

- **The mean absolute error (MAE)** is also measured in the same units as the data and is usually similar in magnitude to, but slightly smaller than, the root mean squared error. It is less sensitive to the occasional very large error because it does not square the errors in the calculation. The mathematically challenged usually find this an

easier statistic to understand than the RMSE. MAE and MAPE (below) are not a part of the standard time series model output, however. They are more commonly found in the output of time series forecasting procedures, such as the one in `forecast` package. It is relatively easy to compute them in Jupyter, as I will demonstrate later.

- The **mean absolute percentage error (MAPE)** is also often useful for purposes of reporting because it is expressed in generic percentage terms which will make some sense even to someone who has no idea what constitutes a "big" error in terms of dollars spent or widgets sold. The MAPE can only be computed on data that are guaranteed to be strictly positive, so if this statistic is missing from your output where you would normally expect to see it, it's possible that it has been suppressed due to negative data values. The **mean absolute scaled error (MASE)** is another relative measure of error that is applicable only to time series data. It is defined as the mean absolute error of the model divided by the mean absolute error of a naïve random-walk-without-drift model (i.e., the mean absolute value of the first difference of the series). Thus, it measures the relative reduction in error compared to a naive model. Ideally, its value will be significantly less than 1. This statistic, which was proposed by Rob Hyndman in 2006 (Hyndman, Another look at measures of forecast accuracy, 2006), is very good to look at when fitting time series models to nonseasonal time series data. It is possible for a time series model to have an impressive R-squared and yet be inferior to a naïve model, as was demonstrated in the notes. If the series has a strong seasonal pattern, the corresponding statistic to look at would be the mean absolute error divided by the mean absolute value of the seasonal difference (i.e., the mean absolute error of a naïve seasonal model that predicts that the value in a given period will equal the value observed one season ago).
- The **mean error (ME)** and **mean percentage error (MPE)** that are reported in some statistical procedures are *signed* measures of error which indicate whether the forecasts are *biased*, i.e., whether they tend to be disproportionately positive or negative. Bias is normally considered a bad thing, but it is not the bottom line. Bias is one component of the mean squared error (MSE)—in fact, mean

squared error equals the variance of the errors plus the square of the mean error. That is: $MSE = VAR(E) + (ME)^2$. Hence, if you try to minimize mean squared error, you are implicitly minimizing the bias as well as the variance of the errors.

In a model that includes a *constant* term, the mean squared error will be minimized when the mean error is *exactly zero*, so you should expect the mean error to always be zero within the estimation period in a model that includes a constant term. (Note: as reported in the `forecast()` function, the mean error in the estimation period may be slightly different from zero if the model included a log transformation as an option, because the forecasts and errors are automatically unlogged before the statistics are computed.)

The RMSE is more sensitive, than other measures, to the occasional large error: the squaring process gives disproportionate weight to very large errors. If an occasional large error is not a problem in your decision situation (e.g., if the true cost of an error is roughly proportional to the size of the error, not the square of the error), then the MAE or MAPE may be a more relevant criterion. In many cases these statistics will vary in unison—the model that is best on one of them will also be better on the others—but this may not be the case when the error distribution has outliers. If one model is best on one measure and another is best on another measure, they are probably pretty similar in terms of their average errors. In such cases, you probably should give more weight to some of the other criteria for comparing models, e.g., simplicity, intuitive reasonableness, etc.

The root mean squared error and mean absolute error can only be compared between models whose errors are measured in the same units (e.g., dollars, or constant dollars, or cases of beer sold, or whatever). If one model's errors are adjusted for inflation while those of another or not, or if one model's errors are in absolute units while another's are in logged units, their error measures cannot be directly compared. In such cases, you have to convert the errors of both models into comparable units before computing the various measures. This means converting the forecasts of one model to the same units as those of the other by *unlogging* or *undeflating* (or whatever), then subtracting

those forecasts from actual values to obtain errors in comparable units, then computing statistics of those errors. You *cannot* get the same effect by merely unlogging or undeflating the error statistics themselves! In the *forecast package*, the user-specified forecasting procedure will take care of the latter sort of calculations for you: the forecasts and their errors are automatically converted back into the original units of the input variable (i.e., all transformations performed as model options within the forecasting procedure are reversed) before computing the statistics.

There is no absolute criterion for a "good" value of RMSE or MAE: it depends on the units in which the variable is measured and on the degree of forecasting accuracy, as measured in those units, which is sought in a particular application. Depending on the choice of units, the RMSE or MAE of your best model could be measured in zillions or one-zillionths. It makes no sense to say “the model is good (bad) because the root mean squared error is less (greater) than x ”, unless you are referring to a specific degree of accuracy that is relevant to your forecasting application.

There is no absolute standard for a "good" value of adjusted R-squared. Again, it depends on the situation, in particular, on the “signal-to-noise ratio” in the dependent variable. (Sometimes much of the signal can be explained away by an appropriate data transformation, before fitting a time series model.) When comparing time series models that use the *same* dependent variable and the *same* estimation period, the *standard error of the time series model goes down as adjusted R-squared goes up*. Hence, the model with the highest adjusted R-squared will have the lowest standard error of the time series model, and you can just as well use adjusted R-squared as a criterion for ranking them. However, when comparing time series models in which the dependent variables were transformed in different ways (e.g., differenced in one case and undifferenced in another, or logged in one case and unlogged in another), or which used different sets of observations as the estimation period, R-squared is not a reliable guide to model quality.

Don't split hairs: a model with an RMSE of 3.25 is not significantly better than one with an RMSE of 3.32. Remember that the width of the

confidence intervals is proportional to the RMSE, and ask yourself how much of a relative decrease in the width of the confidence intervals would be noticeable on a plot. It may be useful to think of this in percentage terms: if one model's RMSE is 30% lower than another's, that is probably very significant. If it is 10% lower, that is probably somewhat significant. If it is only 2% better, that is probably not significant. These distinctions are especially important when you are trading off model complexity against the error measures: it is probably not worth adding another independent variable to a time series model to decrease the RMSE by only a few more percent.

The RMSE and adjusted R-squared statistics already include a minor adjustment for the number of coefficients estimated to make them “unbiased estimators”, but a heavier penalty on model complexity really ought to be imposed for purposes of selecting among models. Sophisticated software for automatic model selection seeks to minimize error measures which impose such a heavier penalty, such as the Mallows C_p statistic (Mallows, 1973), the *Akaike Information Criterion* (AIC) or *Schwarz' Bayesian Information Criterion* (BIC). How these are computed is beyond the scope of the current discussion, but suffice it to say that when you—rather than the computer—are selecting among models, you should show some preference for the model with fewer parameters, other things being approximately equal.

The RMSE is a valid indicator of relative model quality only if it can be trusted. If there is evidence that the model is badly misspecified (i.e., if it *grossly* fails the diagnostic tests of its underlying assumptions) or that the data in the estimation period has been *over-fitted* (i.e., if the model has a relatively large number of parameters for the number of observations fitted and its comparative performance deteriorates badly in the validation period), then the root mean squared error *and all other error measures* in the estimation period may need to be heavily discounted.

If there is evidence only of *minor* misspecification of the model—e.g., modest amounts of autocorrelation in the residuals—this does not completely invalidate the model or its error statistics. Rather, it only suggests that some fine-tuning of the model is still possible. For example,

it may indicate that another lagged variable could be profitably added to a time series model.

Forecasting with R

We make our first forecast here based on our decomposition with the `stl()` function, and then plot it as shown in *Figure 6-30*. We discussed in the previous section, and this is where they come into play. They are standard output of the `forecast()` function, when ARIMA models are employed. A MAPE of 5.3183 is not bad, but we can improve or models if we can lower the value.

```
for.fit <- forecast::forecast(fit, h = 12)
accuracy(for.fit)
```

	ME	RMSE	MAE	MPE	MAPE	MASE	ACF1
Train	2.2663	16.9668	12.7536	0.9115	5.3183	0.3982	0.1821

```
plot(for.fit ,lwd = 2, col = "purple")
```

Out-of-Sample Forecasting

The next snippet of R code builds a model with a training set and forecasts on a test set. Though there is not enough data to do analysis for its greatest effect, we will work with what we have. The test set is the *airpass* dataset starting at 1960 and continuing through 1971. The training set is the *airpass* dataset ending after 1965. This is done with `window(airpass, end=1965)`. The accuracy is then measured for the training set and the test set.

Forecasts from STL + ETS(A,N,N)

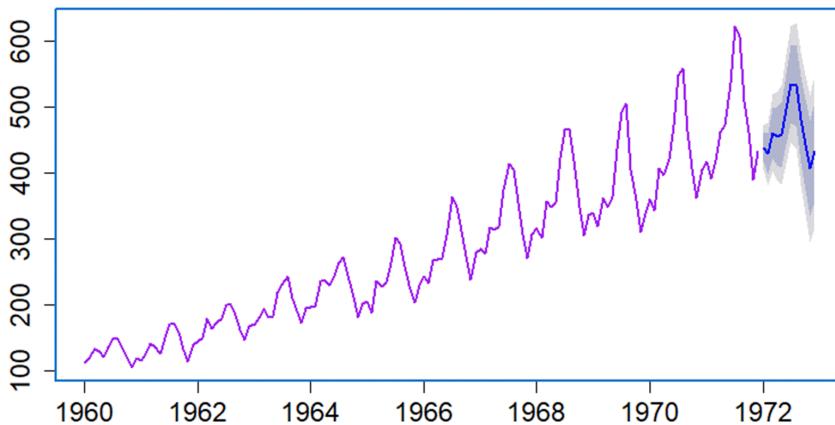


Figure 6-30. 12-month forecast for the Air Passenger data based on fit

```
air.models5 <- Arima(window(airpass, end = 1965),
                      order = c(0,1,1),
                      seasonal = list(order = c(0,1,1),
                                      period = 12),
                      lambda = 0)
air.forecast5 <- forecast(air.models5, h = 48, lambda = NULL)
accuracy(air.forecast5, log(window(airpass, start = 1960)))
```

	ME	RMSE	MAE	MPE
Training set	"0.3478"	"7.7272"	"5.6433"	"0.1754"
Test set	"0.0069"	"0.0576"	"0.0468"	"0.1143"
	MAPE	MASE	ACF1	Theil's U
Training set	"2.8706"	"0.2426"	"0.0432"	"NA"
Test set	"0.7983"	"0.0020"	"0.7967"	"0.5223"

Dynamic Plotting

A dynamic plot (see [Figure 6-31](#)) of the Air Passenger time series forecast can be implemented using the `plot()` function with `ZRA()` from the `ZRA`-package, as follows (Beiner, 2015):

```
plot(ZRA(airpass), zero = TRUE)
```

```
[1] "Significance levels: 0.8 (red), 0.95 (green)"
[1] "Prediction interval: 10 Periods"
```

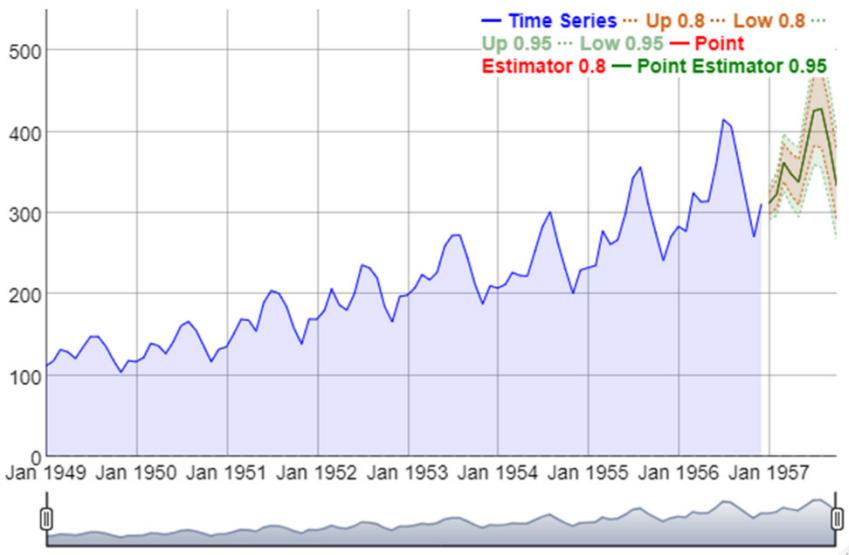


Figure 6-31. Dynamic plot of air passenger time series data

You can move the cursor to any point on the graph to see the data and value as well as zoom in and out.

Chapter Review

In this chapter, we discussed seasonal and nonseasonal ARIMA models. We learned to use R to analyze time series data to determine appropriate specification for ARIMA models, i.e., the number of autoregression terms, the number of moving averages, and the number of differences to apply. We investigated the series for seasonality and trend. We evaluated the series for stationarity and learned to apply differencing to non-stationary series. Using good modeling strategies, we also learned to develop forecasts. Finally, we learned to perform model diagnostics, access the accuracy of the model (and its forecast), and to compare models. We defined the following:

Definition Number	Definition Citation
Definition 6.1: Augmented Dickey-Fuller Test	<i>Given a time series sample, an augmented Dickey-Fuller test (ADF) tests the null hypothesis that a unit root is present by estimating the regression model</i> $Y_t = \hat{\alpha}y_{t-1} + \hat{\beta}_1y_{t-1}^* + \hat{\beta}_2y_{t-2}^* + \dots + \hat{\beta}_ky_{t-k}^*$

	where y_t' denotes the first-differenced series, $y_t' = y_t - y_{t-1}$, where α is a constant, β the coefficient on a time trend, and k is the number of lags to include in the regression
Definition 6.2: Phillips-Perron Test	The Phillips-Perron tests assess the null hypothesis of a unit root in a univariate time series y . The test regression for the PP tests is: $Y_t = c + \delta_t + \alpha y_{t-1} + e_t$. The null hypothesis restricts $\alpha = 1$.
Definition 6.3: Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test	Given a times series sample, the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test partitions the series into three parts: a deterministic trend (βt), a random walk (r_t), and a stationary error (ε_t), with the regression equation: $y_t = r_t + \beta t + \varepsilon_t$.
Definition 6.4: Stationary Times Series	A stationary time series is one whose statistical properties such as mean, variance, autocorrelation, etc. are all constant over time has statistical stationarity.
Definition 6.5: Trend Stationary	A series is said to be trend-stationary if it has a stable long-run trend and has a tendency to revert to the trend line following a disturbance.
Definition 6.6: Differenced-Stationary	A series is said to be difference-stationary if the mean, variance, and autocorrelations of the original series are not constant in time, even after detrending.
Definition 6.7: First Difference	If Y_t denotes the value of the time series Y at period t , then the first difference of Y at period t is equal to $Y_t - Y_{t-1}$. We define the difference operator as $y_t' = y_t - y_{t-1}$
Definition 6.8: Second Difference	If Y_t denotes the value of the time series Y at period t , then the second difference of Y at period t is equal to $Y_t' - Y_{t-1}'$. We define the difference operator as $y_t'' = y_t' - y_{t-1}' = (y_t - y_{t-1}) - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}$
Definition 6.9: Ljung_Box Test	(Ljung-Box test) For a time series Y of length n , the errors are iid (independent and identically distributed), or the autocorrelations of the residuals are very small (near 0), if: $Q(m) = n(n+2) \sum_{j=1}^m \frac{r_j^2}{n-j} < \chi^2_{1-\alpha h}$ Where: r_j = the accumulated sample autocorrelations n = the sample size m = the time lag α = the level of significance h = the degrees of freedom
Definition 6.10:	The ARMA(p,q) model with parameters p and q , where

ARMA(p,q) model *p is the order of the autoregressive polynomial, and q is the order of the moving average polynomial. is defined by the equation is given by:*

$$X_t = c + \varepsilon_t + \sum_{i=1}^p \varphi_i X_{t-i} + \sum_{i=1}^q \theta_i \varepsilon_{t-i}$$

where:

φ = the autoregressive model's parameters,

θ = the moving average model's parameters.

c = a constant,

ε = error terms (white noise)

Definition 6.11 *If the times series y_t is differenced d times, and it then follows an ARMA(p, q) process, then it is an ARIMA(p, d, q) series: $y'_t = c + \phi_1 y'_{t-1} + \dots + \phi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \dots + \theta_q \varepsilon_{t-q} + \varepsilon_t$, where y'_t is the differenced series (it may have been differenced more than once).*

Review Exercises

1. Using the *copper* time series, examine the trend and perform a difference operation. What is the effect?
2. Using the *chicken* dataset from the [fma](#) library in R Studio:
 - a. Plot the time series data
 - b. Decompose the time series data
 - c. Analyze the data for seasonality and trend
 - d. Analyze the data for stationarity, and if it is not stationary, take steps to make it so.
 - e. Properly specify a nonseasonal ARIMA model
 - f. Build the model from part (d)
 - g. Properly specify a seasonal ARIMA model
 - h. Build the model from part (f)
 - i. Compare all models (c), (d), and (f)
 - j. For the best model plot the residuals, ACF and PACF and perform model diagnostics
3. Using the *copper* dataset from the [fma](#) library
 - a. Plot the time series data
 - b. Decompose the time series data
 - c. Perform a unit root test to determine if the *copper* time series is stationary

- d. Perform the steps to specify an ARIMA model
 - e. Build the model from part (c)
 - f. Build an automated ARIMA model
 - g. Compare the models from (c) and (e).
4. Using the *elec* dataset from the `fma` library:
- a. Plot the time series data
 - b. Create a seasonplot
 - c. Create a monthplot
 - d. Create residual plots using the `tsdisplay()` function
 - e. Plot the time series data
 - f. Analyze the data for seasonality and trend
 - g. Analyze the data for stationarity, and if it is not stationary, take steps to make it so.
 - h. Properly specify a nonseasonal ARIMA model
 - i. Build the model from part (h)
 - j. Properly specify a seasonal ARIMA model
 - k. Build the model from part (j)
 - l. Compare all your models
 - m. For the best model plot the residuals, ACF and PACF and perform model diagnostics
 - n. Perform model diagnostics
 - o. Create a 24-month forecast
5. Using the *advsales* dataset from the `fma` library:
- a. Set up the variable “sales” as a time series
 - b. Specify an ARIMA model and explain your reasoning
 - c. Build the ARIMA model from (e)
 - d. Perform model diagnostics
 - e. Create a 24-month forecast

Chapter 7 – ARIMA Modeling

We have already mentioned that we have to properly format data, or verify its format, before we can perform time series analysis. We have not mentioned a native format of time series data, for that is not one. But as common forms and we dealt with some in chapters 4 and 6. Now, we will discuss how to get unformatted data in the format of the beer data we studied in Chapter 4.

Acquiring and Analyzing Formats

The data we will work with was downloaded from the National Aeronautics and Space Administration (NASA) Goddard Institute for Space Studies, GISS Surface Temperature Analysis (GISTEMP v4) is an estimate of global surface temperature change study. Available in TXT and CSV formats, the Global-mean monthly, seasonal, and annual means, 1880-present, is current and appropriate for our situation. As few rows of this data appears in Table.

Table 7-1. *Global-mean monthly, percent change of seasonal and annual means, 1880-present*

GLOBAL Land-Ocean Temperature Index in 0.01 degrees Celsius base period: 1951-1980												
sources: GHCN-v4 1880-08/2020 + SST: ERSST v5 1880-08/2020												
using elimination of outliers and homogeneity adjustment												
AnnMean												
Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1880	-17	-23	-8	-15	-9	-20	-17	-9	-14	-23	-21	-17
1881	-19	-13	4	6	7	-17	1	-2	-14	-20	-17	-6
1882	18	15	5	-16	-14	-22	-15	-6	-14	-23	-16	-35
1883	-28	-36	-12	-17	-16	-7	-5	-13	-21	-11	-22	-11
1884	-12	-8	-36	-39	-33	-34	-32	-27	-27	-25	-33	-31
1885	-58	-33	-26	-41	-44	-43	-33	-30	-28	-23	-23	-9
1886	-43	-50	-42	-27	-23	-34	-17	-30	-23	-27	-26	-25
1887	-71	-56	-35	-34	-29	-24	-25	-34	-25	-35	-25	-32
1888	-33	-36	-41	-20	-22	-17	-10	-15	-12	1	3	-4
1889	-8	17	6	10	-1	-10	-8	-20	-24	-26	-33	-29

The data present in Table 7-1 is the text file (TXT) version of the data as seen in the browser window. We want point out the following features.

1. The base data was acquired from historical data from 1951-1980, while the data from 1880 through 1950 and 1981 to present was provided by a model written in Fortran.
2. The data in the table for percent change is not formatted as percent change. Rather, we need to multiple the data by 0.01 Celsius to get those values. For example, $-17 \times 0.01 = 0.17$, is a percent change value. However, recording the values as a percent, like -17 percent, may be more intuitive.
3. A percent change is calculated as

$$\% \text{ Temp Change} = \frac{\text{Current month temp} - \text{Prior month temp}}{\text{Current month temp}}$$

Table 1-1

A negative present change means that the prior month was warmer than the current month and a positive percent change means the current month is warmer. Let's say that the average temperature for the current month is 10°C, then the prior month's average temperature was

$$(0.17)(10^\circ\text{C}) = (10^\circ\text{C} - T) \text{ or } T = 10^\circ\text{C} - (0.17)(10^\circ\text{C}) = 8.3^\circ\text{C}$$

So, last month was cooler than this month.

4. The years are listed as a column with the dates in each row of the column, while the months are column headings with the %Change in each row. So, in March 1882, the percent change in temperature was 5%, while in April 1883 there was a -12% change in average monthly temperature.

Reading Time Series Data

The first thing that we want to do to analyze our time series data as we import it into R, and determine if its type is “ts” for time series. We might think we can read data into R using the `scan()` function, which assumes that your data for successive time points is in a simple text file with one column, but that is not the case. In our case, it is best to import the CSV format using `read_csv()` from the `readr` package. We also want to check its class.

```
dta <- read_csv("D:/Documents/Data/glb_ts.csv")
class(dta)
```

```
[1] "spec_tbl_df"      "tbl_df"        "tbl"
```

Since the data is neither a time series or a dataframe, we want to put it into a dataframe format—it will be easier to convert to a time series format.

```
dta <- as.data.frame(read_csv("D:/Documents/Data/glb_ts.csv"))
head(dta)
class(dta)
```

```
[1] "data.frame"
```

To ensure that every entry is numeric—we cannot deal with character data in a time series—we extract just the %change values and check their class, and check a couple of rows to see that we only have %change values (i.e., no months or years).

```
myvector=dta[,2]
class(myvector)
```

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

```
head(myvector)
```

```
[1] -0.17 -0.23 -0.08 -0.15 -0.09 -0.20 -0.17 -0.09 -0.14
[10] -0.23 -0.21 -0.17 -0.19 -0.13  0.04  0.06  0.07 -0.17
```

So, the data looks good. Now, we just need to convert the dataframe to a time series format, with a `start` and `end` date. For instance(2019,12) represented December 2019. We use the `ts()` to format the dates to the time series from January 1880 (1880,1) to December 2019 (2019,12) and a `frequency` of 12 to indicate that it has monthly data. Then we check the class and the output.

```
dta.ts <- ts(myvector, start=c(1880, 1), end=c(2019, 12))
class(dta.ts)
head(dta.ts)
```

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

```
> head(dta.ts)
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep
1880	-0.17	-0.23	-0.08	-0.15	-0.09	-0.20	-0.17	-0.09	-0.14

```
1881 -0.19 -0.13  0.04  0.06  0.07 -0.17  0.01 -0.02 -0.14
```

The output is good, since it tells us that we have a time series, `ts`, that starts in 1880 and is divided into months (this is partial output and Oct, Nov, and Dec are in the data).

Now that we have read a time series in R, the next step is to make a plot of the time series data, which we do with the `plot.ts()` function in Figure 7-1. We also added labels, colors, and font size using `title()`. Then, we renumbered the x-axis using `labels()`.

```
plot.ts(dta.ts, col='dodgerblue3', lwd = 2, xlab = '', ylab = '')
labels=axis(1, seq(1880,2030,10))
title(main = "Percent Change in Temperature",
      col.main="dodgerblue2", cex.main = 1.5,
      xlab = "Year", ylab = "Percent Change",
      col.lab = "dodgerblue3", font.lab = 2, cex.lab = 1.25)
```

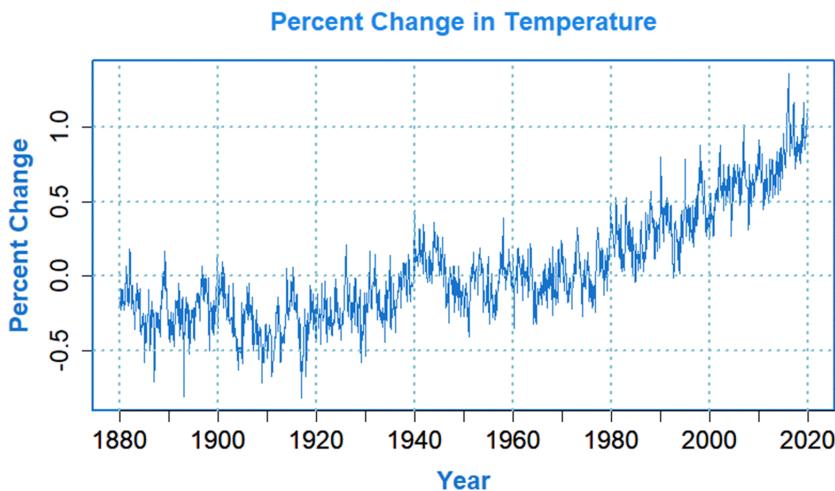


Figure 7-1. Time series plot of global percent change in average monthly temperatures

We can see from Figure 7-1 that this time series that there seems to be an upward trend in temperature change (a positive slope) indicating an increase in monthly average temperatures since about 1950, and a slight upward trend in the years preceding 1950. Of course, we do not have an explanation for the slight drop in 1950. Obviously, we would have had

seasonal changes, since temperatures fluctuate on average during the different calendar seasons.

Seasonal Decomposition

A time series with additive trend, seasonal, and irregular components can be decomposed using the `stl()` function. As we saw earlier in the book, decomposition can help us detect time series issues, like non-stationarity. Once these issues are discovered, we have a toolkit for solving them.

Decomposing Seasonal Data

As we have seen, a seasonal time series consists of a trend component, a seasonal component, and an irregular (random) component. Decomposing the time series means separating the time series into these three components. The function `stl()` decomposes a time series into its components using loess, acronym STL. We do this for the entire series. However, for the use on our limited space on a page, we only looked at the part of the series after December 1959.

```
fit <- stl(window(dta.ts, start = 1960), s.window = "period")
summary(fit)
```

```
Call:
stl(x = window(dta.ts, , start = 1960), s.window = "period")
Time series components:
    seasonal          trend      remainder
Min.   :-0.02527472   Min.   :-0.1964938   Min.   :-0.3193868
1st Qu.:-0.01726327   1st Qu.: 0.0630778   1st Qu.:-0.0576108
Median :-0.00997371   Median : 0.3121570   Median : 0.0052199
Mean   : 0.00000000   Mean   : 0.3397365   Mean   : 0.0000552
3rd Qu.: 0.01243368   3rd Qu.: 0.6107526   3rd Qu.: 0.0535157
Max.   : 0.06045759   Max.   : 1.0701854   Max.   : 0.3090715
IQR:
    STL.seasonal STL.trend STL.remainder data
    0.0297      0.5477     0.1111      0.5400
    % 5.5       101.4      20.6       100.0
Weights: all == 1
Other components: List of 5
$ win : Named num [1:3] 7201 19 13
$ deg : Named int [1:3] 0 1 1
$ jump : Named num [1:3] 721 2 2
$ inner: int 2
$ outer: int 0
```

We plot this data in Figure 7-2 but only for the series (x), seasonal, trend, and random (remainder), using the fit and only the part of the series from 1960 to 2020 we fitted.

```
plot(fit, col = 'deeppink3', lwd = 1.5)
title(main = "Percent Change in Temperature Decomposition",
      col.main = "deeppink3", cex.main = 1.25,
      col.lab = "deeppink3", font.lab = 2, cex.lab = 1.25)
```

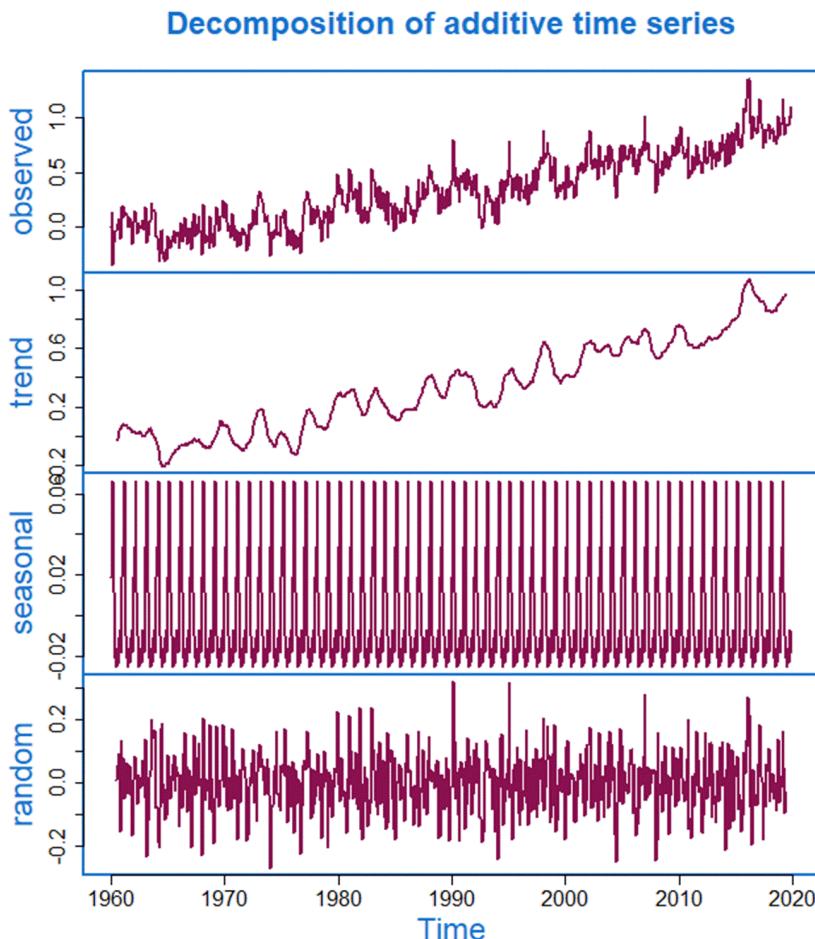


Figure 7-2. Seasonal decomposition of the global percent temperature change series

Figure 7-2 shows that there is seasonality, as well an upper trend .

Alternative Seasonal Decomposition

A time series with additive trend, seasonal, and irregular components can be decomposed using the `decompose()` function. Figure 7-3 shows the same seasonality, trend random parts. Note that a series with multiplicative effects can often be transformed into series with additive effects through a log transformation (i.e., `newts <- log(myts)`).

```
Tempdecomposition <- decompose(window(dta.ts, start = 1960))
plot(Tempdecomposition, col = 'deeppink4', lwd = 1.5)
```

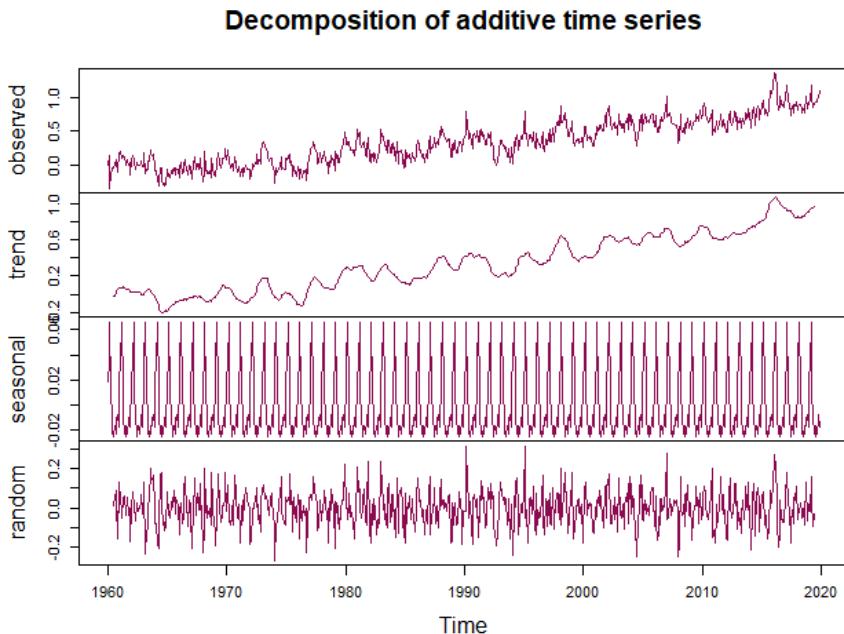


Figure 7-3. Decomposition Plot for the global percent temperature change series

Seasonal plot

`Seasonplot()` plots a seasonal plot as described in (Hyndman, et al., 2015). 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. A seasonal plot allows the underlying seasonal pattern to be seen more clearly and is especially useful in identifying years in which the pattern changes.

In the case of global temperature changes, there is an obvious seasonal pattern in the time series data (see Figure 7-3 and Figure 7-4).

```
library("RColorBrewer")
ggseasonplot(window(dta.ts, start = 2000),
             col=brewer.pal(n = 12, name = "Paired"),
             year.labels = TRUE) +
  geom_line(lwd = 1.1) +
  theme_gray()
```

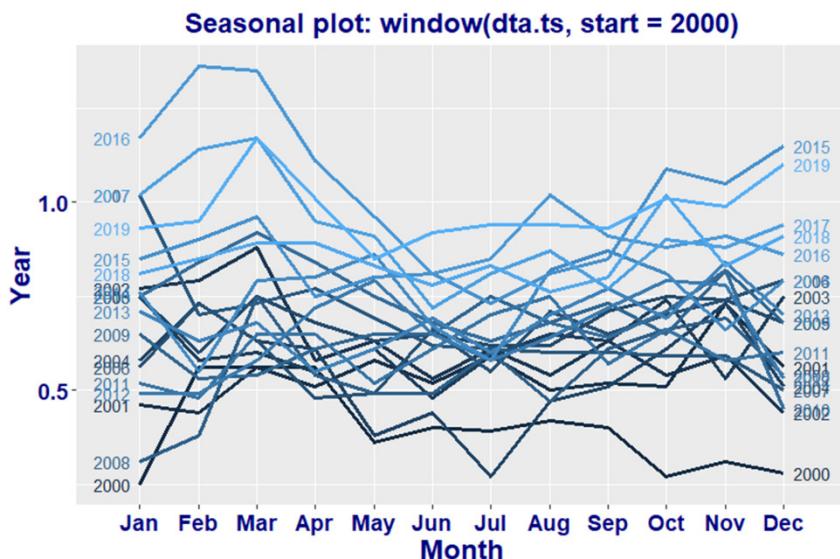


Figure 7-4. Seasonplot the global percent temperature change series

These functions plot seasonal (or other) subseries of a time series. For each season (or another category), a time series is plotted.

These functions extract subseries from a time series and plot them all in one frame. The `ts`, `stl`, and `StructTS` methods use the internally recorded frequency and start and finish times to set the scale and the seasons. The default method assumes observations come in groups of 12 (though this can be changed).

If the `labels` are not given but the `phase` is given, then the `labels` default to the unique values of the `phase`. If both are given, then the

`phase` values are assumed to be indices into the `labels` array, i.e., they should be in the range from 1 to `length(labels)`.

An alternative plot that emphasizes the seasonal patterns is where the data for each season are collected together in separate mini time plots.

Month Plot

`Monthplot()` plots the average percent change in temperatures for each month over the duration of the time series, as well as the variance of each mean, as we show in Figure 7-5.

```
monthplot(window(dta.ts, start = 1950),
         main = 'Monthplot of Percent Change in Temperature',
         xlab = 'Months', ylab = 'Percent Change',
         lwd = 1.75, col = 'deepskyblue3')
```

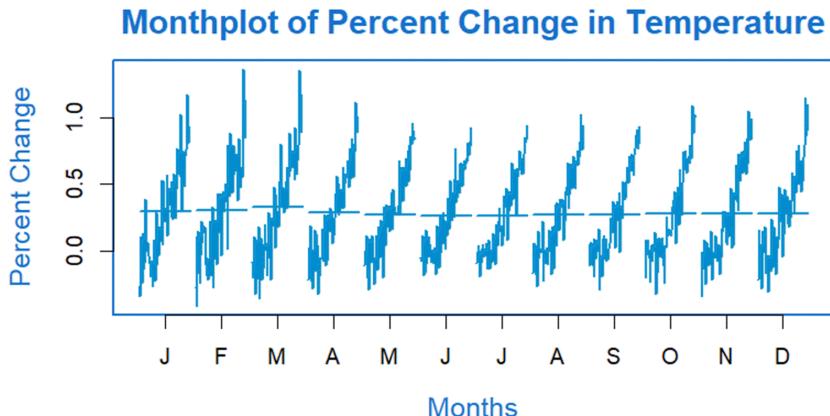


Figure 7-5. *Monthplot* showing the mean monthly temperature changes

The horizontal lines in Figure 7-5 indicate the means for each month. 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. Since the means are not equal across months, the data is not stationary.

```
# Reject nonstationarity if p <= 0.05
stationary.test(ts(dta.ts), method = "kpss")
```

```

KPSS Unit Root Test
alternative: nonstationary

Type 1: no drift no trend
lag stat p.value
 9 3.68    0.01
-----
Type 2: with drift no trend
lag stat p.value
 9 6.97    0.01
-----
Type 1: with drift and trend
lag stat p.value
 9 1.58    0.01
-----
Note: p.value = 0.01 means p.value <= 0.01
      : p.value = 0.10 means p.value >= 0.10

```

Testing for Stationarity

Recall from Definition 6.1 that we use the **Augmented Dickey-Fuller test** to test the null hypothesis that a unit root is present in an autoregressive model of a given time series and that the process is thus not stationary. If the null hypothesis is true, then we cannot use an ARIMA model to fit and forecast the series.

The fact that the p-value is less than 0.05 gives use cause to reject the null hypothesis (the series has a unit root) and treat the series as stationary. We can also compare the calculated DF_T statistic with a tabulated critical value. If the DF_T statistic is more negative than the table value, reject the null hypothesis of a unit root. Note: The more negative the DF test statistic, the stronger the evidence for rejecting the null hypothesis of a unit root. Existence of a unit root implies nonstationarity. Two values of the test statistic are given, one for unit root with trend, and the other for a unit root with trend and deterministic time trend.

```
adf.test(dta.ts)
```

```

Augmented Dickey-Fuller Test

data: dta.ts
Dickey-Fuller = -4.2592, Lag order = 11, p-value = 0.01
alternative hypothesis: stationary

```

In R, the default value of k is set to $\left\lfloor (T - 1)^{\frac{1}{3}} \right\rfloor$ where T is the length of the time series and $\lfloor x \rfloor$ means the largest integer not greater than x . The half-square brackets $\lfloor \quad \rfloor$ indicate the floor function.

Since the null-hypothesis for an ADF test is that the data are non-stationary, large p-values are indicative of non-stationarity, and small p-values suggest stationarity. Using the usual 5% threshold, differencing is required if the p-value is greater than $\alpha = 0.05$.

Another version of the ADF test comes from the *urca* package or the *Unit Root and Cointegration Tests for Time Series Data* package., which provides a little more information regarding test results. We can set the test to run with trend (drift) or without trend.

```
df=ur.df(dta.ts, type = "drift", lags = 1)
summary(df)
```

```
#####
# Augmented Dickey-Fuller Test Unit Root Test #
#####

Test regression drift

Call:
lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.46750 -0.06836  0.00199  0.06864  0.40861 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.002452   0.002761   0.888   0.375    
z.lag.1     -0.033539   0.007646  -4.386 1.22e-05 ***  
z.diff.lag  -0.334089   0.023091 -14.468 < 2e-16 ***  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1 

Residual standard error: 0.1123 on 1675 degrees of freedom
Multiple R-squared:  0.1334,          Adjusted R-squared:  0.1324 
F-statistic: 129 on 2 and 1675 DF,  p-value: < 2.2e-16

Value of test-statistic is: -4.3865 9.6919

Critical values for test statistics:
 1pct  5pct 10pct
```

```
tau2 -3.43 -2.86 -2.57  
phi1  6.43  4.59  3.78
```

Compared with the Augmented Dickey-Fuller test, **Phillips-Perron test** makes correction to the test statistics and is robust to the unspecified autocorrelation and heteroscedasticity in the errors.

Variants of the test, appropriate for series with different growth characteristics, restrict the drift and deterministic trend coefficients, c and δ , respectively, to be 0. The tests use modified Dickey-Fuller statistics (see `adftest`) to account for serial correlations in the innovations process e_t .

```
pp.test(dta.ts)
```

```
Phillips-Perron Unit Root Test  
  
data: dta.ts  
Dickey-Fuller Z(alpha) = -258.52, Truncation lag parameter = 8,  
p-value = 0.01  
alternative hypothesis: stationary
```

Since the series appears to be stationary, we can turn to modeling it using ARIMA.

A third test for stationarity is the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test.

If the data is stationary, it will have a fixed element for an intercept or the series will be stationary around a fixed level (Wang, 2006, p. 33). The test uses OLS find the equation, which differs slightly depending on whether you want to test for level stationarity or trend stationarity (Kocenda & Cerný, 2007, p. 21). A simplified version, without the time trend component, is used to test level stationarity. We normally log-transform the data before running the KPSS test, to turn any exponential trends into linear ones.

To run the test, we use the `kpss.test()` function from the *tseries* package in R (Trapletti, Hornik, & LeBaron, 2015). This reverses the hypotheses, so the null-hypothesis is that the data are stationary. In this case, small p-values (e.g., less than $\alpha = 0.05$) suggest that differencing is required, i.e., the time series is not stationary.

```
kpss.test(diff(dta.ts, 1), null = "Trend")
```

```
KPSS Test for Trend Stationarity
```

```
data: diff(dta.ts, 1)
KPSS Trend = 0.005404, Truncation lag parameter = 8, p-value=0.1
```

We prefer the `stationary.test()` from the `aTSA` package. This function combines the existing functions `adf.test()`, `pp.test ()` and `kpss.test ()` for testing the stationarity of a univariate time series x

```
stationary.test(ts(dta.ts),method="kpss")
```

```
KPSS Unit Root Test
```

```
alternative: nonstationary
```

```
Type 1: no drift no trend
```

```
lag stat p.value
 9 3.68    0.01
```

```
----
```

```
Type 2: with drift no trend
```

```
lag stat p.value
 9 6.97    0.01
```

```
----
```

```
Type 1: with drift and trend
```

```
lag stat p.value
 9 1.58    0.01
```

```
-----
```

```
Note: p.value = 0.01 means p.value <= 0.01
```

```
: p.value = 0.10 means p.value >= 0.10
```

Building the Models

Now that we are familiar with ARIMA models, we will apply them to our temperature data. We start with four models: (1) ARIMA(1,0,0), (2) ARIMA(1,1,0), (3) ARIMA(0,1,1), and (4) ARIMA(1,1,2).

ARIMA(1,0,0)

This is a first-order autoregressive model: if the series is stationary and autocorrelated, perhaps it can be predicted as a multiple of its own previous value, plus a constant. The forecasting equation in this case is

$$\hat{Y}_t = \mu + \varphi_1 Y_{t-1},$$

which is Y regressed on itself lagged by one period. This is an “ARIMA(1,0,0) + constant” model. If the mean of Y is zero, then the constant term would not be included.

If the slope coefficient φ_1 is positive and less than 1 in magnitude (it must be less than 1 in magnitude if Y is stationary), the model describes mean-reverting behavior in which next period’s value should be predicted to be φ_1 times as far away from the mean as this period’s value. If φ_1 is negative, it predicts mean-reverting behavior with alternation of signs, i.e., it also predicts that Y will be below the mean next period if it is above the mean this period.

In a second-order autoregressive model (ARIMA(2,0,0)), there would be a Y_{t-2} term on the right as well, and so on. Depending on the signs and magnitudes of the coefficients, an ARIMA(2,0,0) model could describe a system whose mean reversion takes place in a sinusoidally oscillating fashion, like the motion of a mass on a spring that is subjected to random shocks.

ARIMA(1,1,0)

This is a differenced first-order autoregressive model: If the errors of a random walk model are autocorrelated, perhaps the problem can be fixed by adding one lag of the dependent variable to the prediction equation--i.e., by regressing the first difference of Y on itself lagged by one period. This would yield the following prediction equation:

$$\hat{Y}_t - Y_{t-1} = \mu + \varphi_1(Y_{t-1} - Y_{t-2})$$

$$\hat{Y}_t - Y_{t-1} = \mu$$

which can be rearranged to:

$$\hat{Y}_t = \mu + Y_{t-1} + \varphi_1(Y_{t-1} - Y_{t-2})$$

This is a first-order autoregressive model with one order of nonseasonal differencing and a constant term—i.e., an ARIMA(1,1,0) model.

ARIMA(0,1,1) without constant

This is a simple exponential smoothing: Another strategy for correcting autocorrelated errors in a random walk model is suggested by the simple

exponential smoothing model. Recall that for some nonstationary time series (e.g., ones that exhibit noisy fluctuations around a slowly-varying mean), the random walk model does not perform as well as a moving average of past values. In other words, rather than taking the most recent observation as the forecast of the next observation, it is better to use an average of the last few observations in order to filter out the noise and more accurately estimate the local mean. The simple exponential smoothing model uses an exponentially weighted moving average of past values to achieve this effect. The prediction equation for the simple exponential smoothing model can be written in a number of mathematically equivalent forms, one of which is the so-called “error correction” form, in which the previous forecast is adjusted in the direction of the error it made:

$$\hat{Y}_t = \hat{Y}_{t-1} + \alpha e_{t-1}$$

Because $e_{t-1} = Y_{t-1} - \hat{Y}_{t-1}$ by definition, this can be rewritten as:

$$\hat{Y}_t = Y_{t-1} - (1 - \alpha)e_{t-1} = Y_{t-1} - \theta_1 e_{t-1}$$

which is an ARIMA(0,1,1)-without-constant forecasting equation with $\theta_1 = 1 - \alpha$. This means that you can fit a simple exponential smoothing by specifying it as an ARIMA(0,1,1) model without constant, and the estimated MA(1) coefficient corresponds to 1-minus-alpha in the SES formula. Recall that in the SES model, the average age of the data in the 1-period-ahead forecasts is $1/\alpha$, meaning that they will tend to lag behind trends or turning points by about $1/\alpha$ periods. It follows that the average age of the data in the 1-period-ahead forecasts of an ARIMA(0,1,1)-without-constant model is $1/(1 - \theta_1)$. So, for example, if $\theta_1 = 0.8$, the average age is 5. As θ_1 approaches 1, the ARIMA(0,1,1)-without-constant model becomes a very-long-term moving average, and as θ_1 approaches 0 it becomes a random-walk-without-drift model.

ARIMA(1,1,2) without constant

This is a damped-trend linear exponential smoothing:

$$\hat{Y}_t = Y_{t-1} + \phi_1 Y_{t-1} - \theta_1 e_{t-1}$$

A linear, exponential smoothing models, ARIMA(1,1,2), which use two nonseasonal differences in conjunction with MA terms is.

$$\hat{Y}_t = Y_{t-1} + \varphi_1(Y_{t-1} - Y_{t-2}) - \theta_1 e_{t-1} - \theta_1 e_{t-1}$$

This model extrapolates the local trend at the end of the series but flattens it out at longer forecast horizons to introduce a note of conservatism, a practice that has empirical support (Gardner & McKenzie, 2011) (Armstrong, Green, & Graefe, 2015).

It is generally advisable to stick to models in which at least one of p and q is no larger than 1, i.e., do not try to fit a model such as ARIMA(2,1,2), as this is likely to lead to overfitting and "common-factor" issues that are discussed in more detail in the notes on the mathematical structure of ARIMA models. Figure 7-6 shows the fitted models.

```
temp.model1 <- Arima(window(dta.ts, start=1950), order=c(1,0,0))
temp.model2 <- Arima(window(dta.ts, start=1950), order=c(1,1,0))
temp.model3 <- Arima(window(dta.ts, start=1950), order=c(0,1,1))
temp.model4 <- Arima(window(dta.ts, start=1950), order=c(1,1,2))
plot(window(dta.ts, start=2000, end=t+1))
  lines(temp.model1$fitted, col = 'green', lwd = 2)
  lines(temp.model2$fitted, col = 'blue2', lwd = 2)
  lines(temp.model3$fitted, col = 'red3', lwd = 2)
  lines(temp.model4$fitted, col = 'orange', lwd = 2)
  title('Fitted ARIMA Models for Global Temperature Change')
  xlab('Year')
  ylab('Percent Change (%)')
  legend(2011, 0.5,
         c( 'ARIMA(1,0,0)', 'ARIMA(1,1,0)', 'ARIMA(0,1,1)',
            'ARIMA(1.1.2)'), 
         Col = c('green', 'blue2', "red3", "orange"),
         lwd = 2, lty = 1, cex = 0.8, box.lty = 0)
```

Fitted ARIMA Models for Global Temperature Change

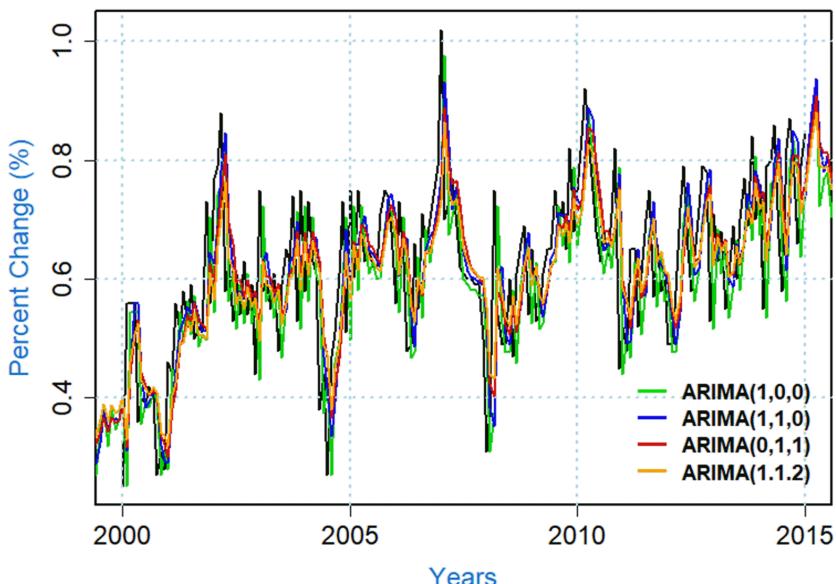


Figure 7-6. Fitted ARIMA models for global temperature changes

Developing the Forecast

We developed forecast for each of our four ARIMA models, using `forecast()` from the `forecast` package. We then plotted the forecast the basic plot function, `plot()`. These plots are shown in Figure 7-7.

```
# CODE FOR ARIMA MODELS FORECASTS

temp.forecast1<-forecast::forecast(temp.model1, h = 48)
plot(temp.forecast1)
lines(dta.ts, col = 'red', lwd = 1.5)

temp.forecast2<-forecast::forecast(temp.model2, h = 48)
plot(temp.forecast2)
lines(dta.ts, col = 'red', lwd = 1.5)

temp.forecast3<-forecast::forecast(temp.model3, h = 48)
plot(temp.forecast3)
lines(dta.ts, col = 'red', lwd = 1.5)

temp.forecast4<-forecast::forecast(temp.model4, h = 48)
plot(temp.forecast4)
lines(dta.ts, col = 'red', lwd = 1.5)
```

Next, we plotted the global temperature percent changes forecast PACFs in Figure 7-8, using our customized `plot.acf()` from Chapter 6. From inspecting the graphs alone, it appears that models 3 and 4 provide the best forecasts, but we will have to further investigate with additional diagnostic measures.

Inspection of the plots led us to the following observations:

1. The forecasted values have a wide range of possibilities, except for the ARIMA(1,1,2) model.
2. The forecasts for ARIMA(0,1,1) shows a less drastic decrease than the ARIMA(1,0,0)
3. The forecast period in the ARIMA(1,1,2) has the least amount of variability
4. The second MA seems to have improved the model significantly.
5. The ARIMA(0,1,1) and ARIMA(1,1,2) models fit the series a little better than the other two.
6. We need to explore additional metrics for goodness-of-fit to validate our tentative conclusion that ARIMA(0,1,1) and ARIMA(1,1,2) models may be the best of the four models.

The goodness-of-fit (GOF) of a statistical model describes how well it fits into a set of observations. GOF indices summarize the discrepancy between the observed values and the values expected under a statistical model.

```
# CODE FOR PACF

temp.acf1 <- pacf(temp.model1$residuals, lag.max = 36)
temp.acf2 <- pacf(temp.model2$residuals, lag.max = 36)
temp.acf3 <- pacf(temp.model3$residuals, lag.max = 36)
temp.acf4 <- pacf(temp.model4$residuals, lag.max = 36)

#CODE FOR PACF PLOTS

plot.acf(temp.acf1); title("Percent Change in Temperature ACF")
plot.acf(temp.acf2); title("Percent Change in Temperature ACF")
plot.acf(temp.acf3); title("Percent Change in Temperature ACF")
plot.acf(temp.acf4); title("Percent Change in Temperature ACF")
```

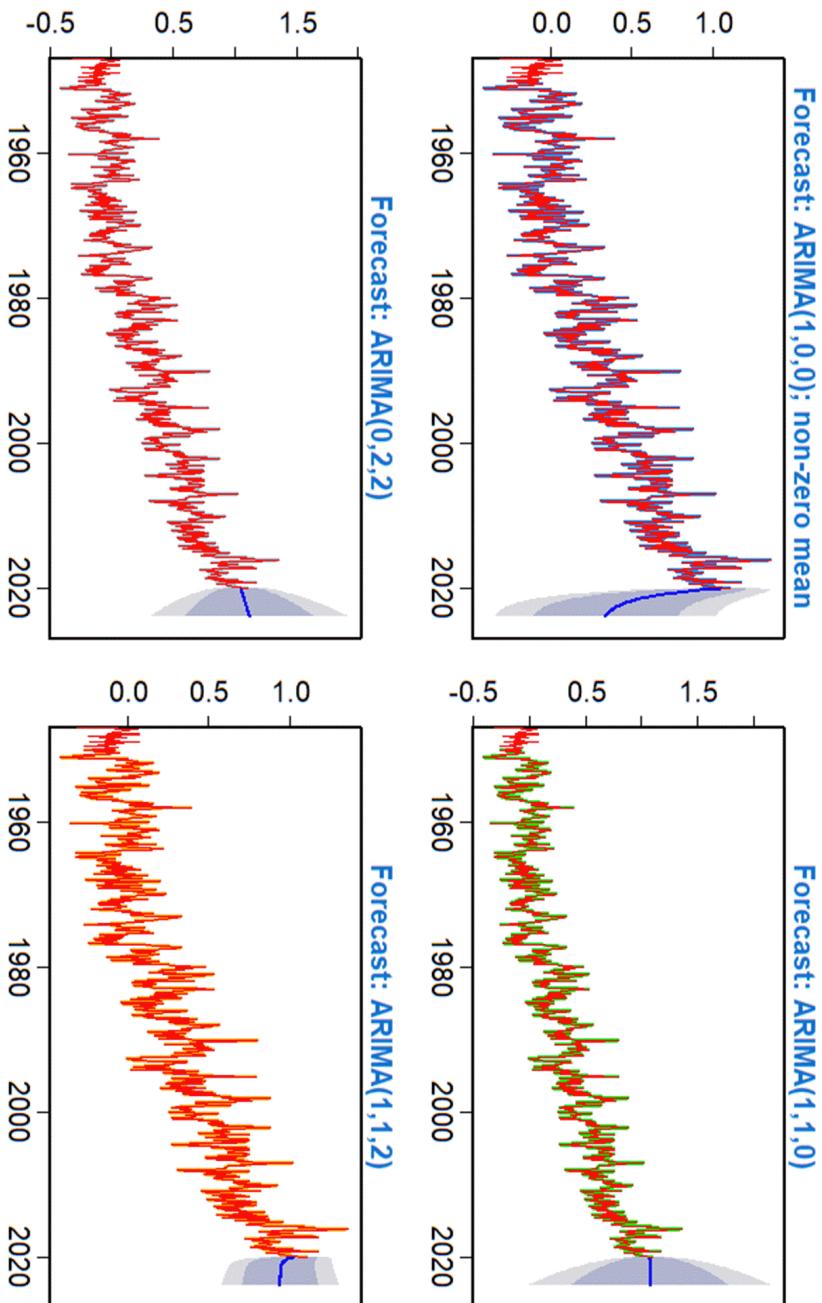


Figure 7-7. Four ARIMA models applied to mean monthly temperature changes

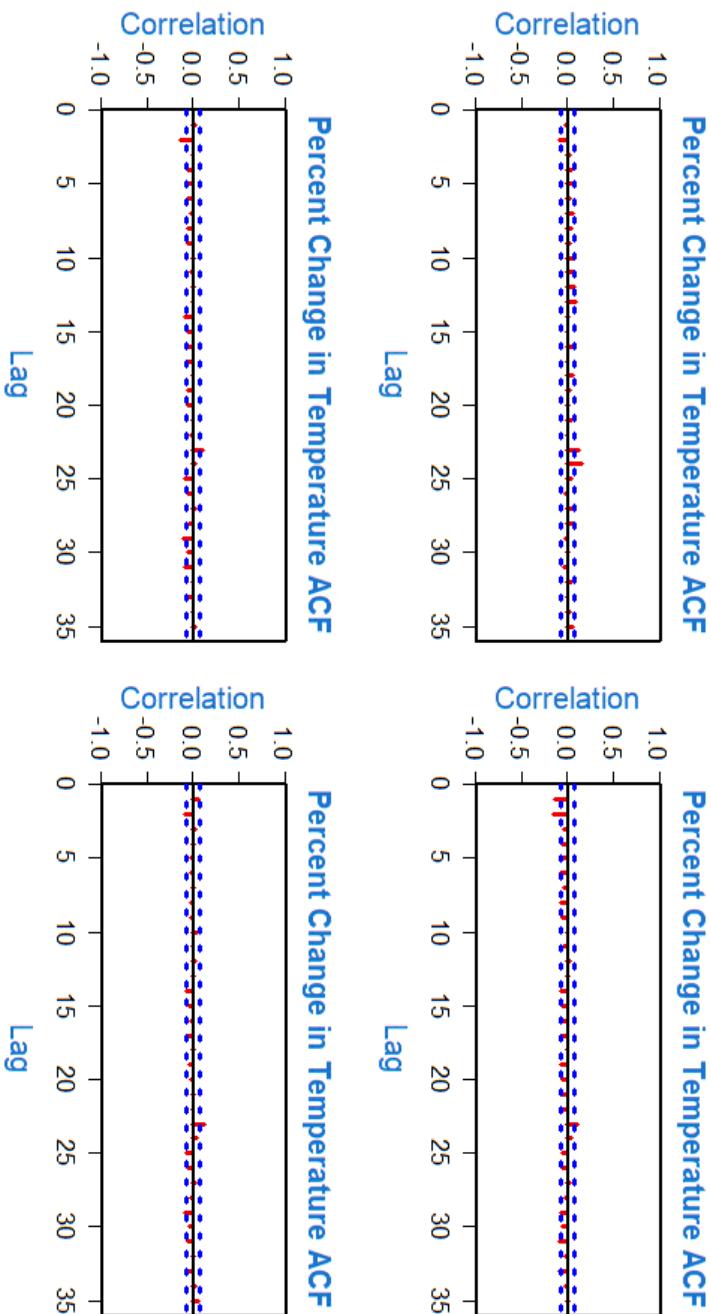


Figure 7-8. ACFs for four ARIMA temperature change models

Model Forecast Diagnostics

Having made an educated guess on model goodness-of-fit, we look closer at models 3 and 4 with their corresponding forecasts. First, we plot the PACFs in Figure 7-9.

```
ggtaperedacf(temp.forecast2$fitted, nsim = 50) +  
  ggtitle("Percent Change in Temperature ACF") +  
  theme_light() + theme(legend.position = "bottom")  
ggtaperedpacf(temp.forecast2$fitted, nsim = 50) +  
  ggtitle("Percent Change in Temperature PACF") +  
  theme_light() + theme(legend.position = "bottom")
```

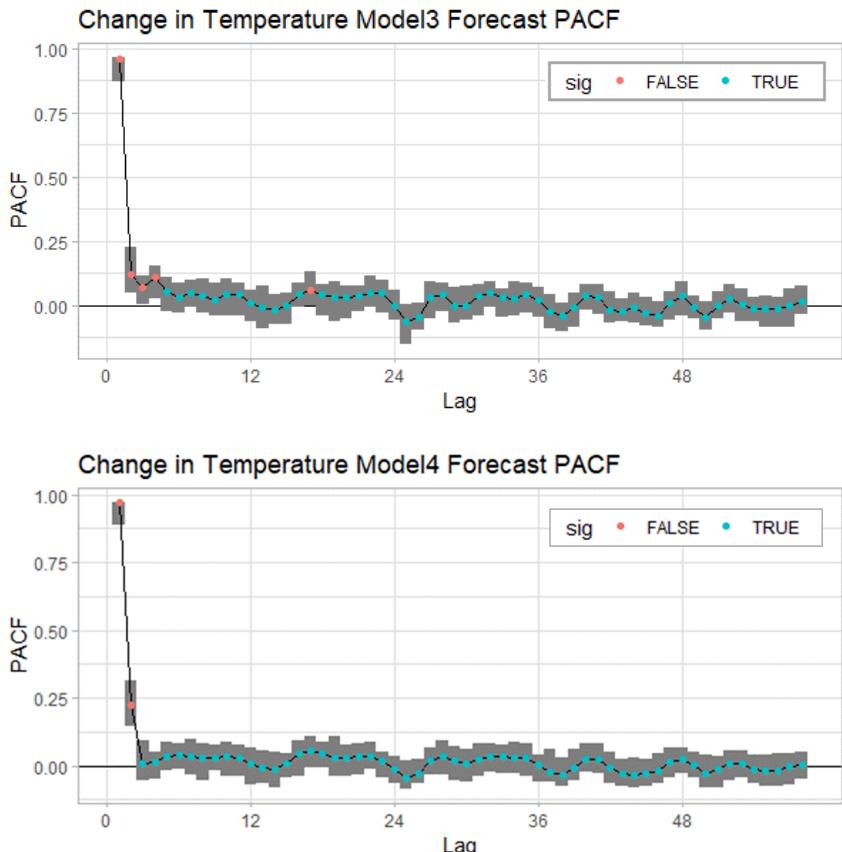


Figure 7-9. PACFs for model 3 (top) and Model 4 (bottom)

The PACFs look very similar, but there is one fewer false reading in Model 4.

Analyzing the Goodness-of-Fit

Since we do not have definitive graphic support for the model, we get a summary of the models, which contain goodness-of-fit metrics, such as Root Mean Square Error (RMSE). We see that the RMSEs are very close between the two models. The other metrics are also close.

```
summary(temp.model3)
```

```
Series: window(dta.ts, start = 1950)
ARIMA(0,2,2)

Coefficients:
      ma1     ma2
-1.4862  0.4862
s.e.  0.0353  0.0351

sigma^2 estimated as 0.0119:  log likelihood=664.5
AIC=-1323  AICc=-1322.97  BIC=-1308.8

Training set error measures:
      ME    RMSE     MAE     MPE     MAPE     MASE     ACF1
Training set -0.0005 0.1088 0.0864  NaN     Inf  0.5772  0.0238
```

```
summary(temp.model4)
```

```
Series: window(dta.ts, start = 1950)
ARIMA(1,1,2)

Coefficients:
      ar1     ma1     ma2
  0.8350 -1.3434  0.3629
s.e.  0.0285  0.0424  0.0387

sigma^2 estimated as 0.01156:  log likelihood=1314.09
AIC=-2620.19  AICc=-2620.16  BIC=-2598.63

Training set error measures:
      ME    RMSE     MAE     MPE     MAPE     MASE     ACF1
Training set 0.0045 0.1074 0.0831  NaN     Inf  0.5674  0.0050
```

Given our dilemma, we will keep performing diagnostics. Here we look at the Box-Ljung test and the null hypothesis is a lack of fit. Since the p -value is less than 0.05, we can reject the null hypothesis.

```
Box.test(temp.model3$fitted, lag = 24, type = 'Ljung-Box')
Box.test(temp.model4$fitted, lag = 24, type = 'Ljung-Box')
```

```

Box-Ljung test
data: temp.model3$fitted
X-squared = 14918, df = 24, p-value < 2.2e-16

Box-Ljung test
data: temp.model4$fitted
X-squared = 15674, df = 24, p-value < 2.2e-16

```

The *p*-value for the Box-Ljung Test is less than 0.0001. This information still does not answer which model is better. So, we'll keep up our investigation.

Model Accuracy

Now we use the `accuracy()` function to gather more model fit information.

```

specify_decimal <- function(x, k)
  trimws(format(round(x, k), nsmall = k))
acc1 <- specify_decimal(accuracy(temp.forecast1), 4)
acc2 <- specify_decimal(accuracy(temp.forecast2), 4)
acc3 <- specify_decimal(accuracy(temp.forecast3), 4)
acc4 <- specify_decimal(accuracy(temp.forecast4), 4)
row.names <- c('ME', 'RMSE', 'MAE', 'MPE', 'MAPE', 'MASE', 'ACF1')
column.names <- c("Model1", "Model2", "Model3", "Model4")
result <- array(c(acc1, acc2, acc3, acc4), dim = c(7,4),
                 dimnames = list(row.names, column.names))
print(result, quote = FALSE)

```

	Model1	Model2	Model3	Model4
ME	0.0010	0.0022	-0.0005	0.0095
RMSE	0.1183	0.1105	0.1088	0.1067
MAE	0.0940	0.0873	0.0864	0.0844
MPE	NaN	NaN	NaN	NaN
MAPE	Inf	Inf	Inf	Inf
MASE	0.6276	0.5828	0.5772	0.5637
ACF1	-0.3450	-0.0360	0.0238	-0.0207

Based on these results, all the metrics are still very close across all the models. So, we will now examine the forecast error metrics graphically as seen in Figure 7-10.

```

par(mfrow = c(2,2))
plotForecastErrors(temp.forecast1$residuals)
plotForecastErrors(temp.forecast2$residuals)
plotForecastErrors(temp.forecast3$residuals)
plotForecastErrors(temp.forecast4$residuals)

```

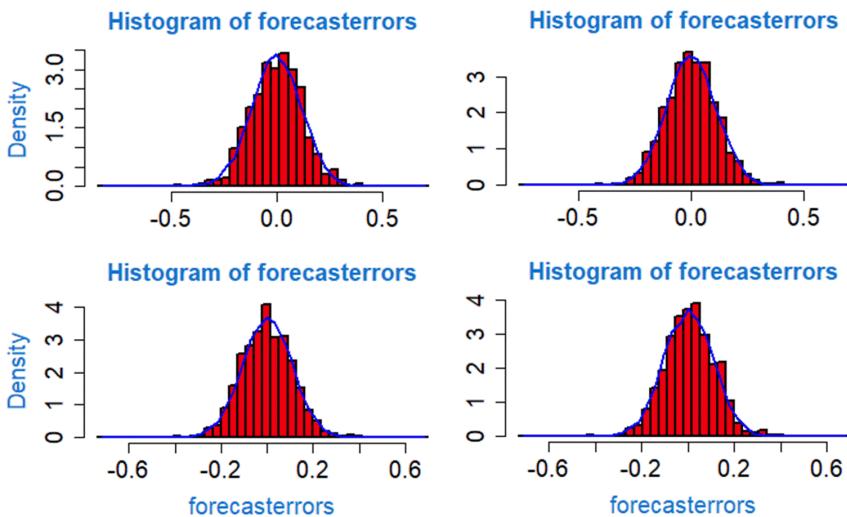


Figure 7-10. Plots of forecast errors between the four models

The graphs seem to have the same shape (normal), but the distributions are different when we look at the spread (x-axis). This implies that model 3 and 4 are still the best fit using this diagnostic measure.

```
library(WeightedPortTest)
Weighted.Box.test(temp.forecast3, lag = 10, type = 'Ljung')
```

```
Weighted Ljung-Box test (Gamma Approximation)

data: temp.forecast3
Weighted X-squared on Residuals for fitted ARMA process = Inf, S
hape = 3.9286,
Scale = 1.4000, p-value < 2.2e-16

> library(WeightedPortTest)
```

```
Weighted.Box.test(temp.forecast4, lag = 10, type = 'Ljung')
```

```
Weighted Ljung-Box test (Gamma Approximation)

data: temp.forecast4
Weighted X-squared on Residuals for fitted ARMA process = Inf, S
hape = 3.9286,
Scale = 1.4000, p-value < 2.2e-16
```

Here, we print the descriptive statistics of the ARIMA(0,1,1) and ARIMA(1,1,2) models. We also perform a Box-Ljung and Box-Pierce test. Recall that the Box-Ljung test is not as powerful as the Weighted Box-Ljung test, but it also answers the null hypothesis that there is a lack of fit. For our models, the p-values are so small ($p \ll 0.05$), we can reject the null hypothesis. Therefore, we still believe we have a good fit.

```
library(AnalyzeTS)
Descriptives(temp.model3$residuals, plot = TRUE)
```

N:	840.00	NaN:	0.00
Min:	-0.41	1sq QU:	-0.07
Median:	0.00	Mean:	0.00
3rd QU:	0.07	Max:	0.39
VAR:	0.01	SD:	0.10
SE:	0.00		

Notice here that the two "Box tests" yield different results. However, we noted that the `Box.test()` function in R can call either the Ljung-Box test or the Box-Pierce test, which we see below, in that order.

```
Box.test(temp.forecast3$residuals, lag = 48, type= "Ljung-Box")
```

```
Box-Ljung test
data: temp.forecast3$residuals
X-squared = 101.77, df = 48, p-value = 9.775e-06
```

```
Box.test(temp.forecast3$residuals, lag = 48, type= "Box-Pierce")
```

```
Box-Pierce test
data: temp.forecast3$residuals
X-squared = 98.776, df = 48, p-value = 2.247e-05
```

Now we run the same tests for Model 4's forecast.

```
Descriptives(temp.model4$residuals, plot = TRUE)
```

N:	1620.00	NaN:	0.00
Min:	-0.47	1sq QU:	-0.06
Median:	0.00	Mean:	0.00
3rd QU:	0.07	Max:	0.43
VAR:	0.01	SD:	0.10
SE:	0.00		

```
Box.test(temp.forecast4$residuals, lag = 48, type= "Ljung-Box")
```

```
Box-Ljung test
```

```
data: temp.forecast4$residuals  
X-squared = 78.312, df = 48, p-value = 0.003727
```

```
Box.test(temp.forecast4$residuals, lag = 48, type= "Box-Pierce")
```

```
Box-Pierce test  
data: temp.forecast4$residuals  
X-squared = 75.808, df = 48, p-value = 0.006415
```

With cycle treated deterministically, the external variables are not significant, something that is intuitively unrealistic. Let's look at the forecast for this version of the model (see Figure 7-11).

```
par(mfrow = c(2,2))  
for (t in 2011:2014){  
  temp.model4 = Arima(window(dta.ts, end = t+11/12),  
    order = c(1,1,2))  
  plot(forecast::forecast(temp.model4, h = 84-12*(t-2011)),  
    xlim = c(1980,2020), ylim = c(-1,2))  
  lines(window(dta.ts, start=1980, end=t+1), col="blue", lwd=2)  
  labels = axis(1, seq(1980,2020,5))  
  abline(v = t+1, lty = 3, col = "red")  
}
```

It may not seem clear from the plots that the first model, ARIMA(1,1,2) with a forecast starting in 2012, has a slight increase in slope before becoming horizontal (a slope of zero). The second model, ARIMA(1,1,2) with a forecast starting in 2013, has a lesser increase in slope before tapering off to horizontal. The third model, ARIMA(1,1,2) with a forecast starting in 2014, has a decreasing slope before tapering off to horizontal. Finally, the fourth model , ARIMA(1,1,2) with a forecast starting in 2015 behaves nearly identical to the third model.

Intuitively, models three and four perform better because they are forecasting a shorter period (the end point for all of them is 2020). However, if we take the in-sample data up to 2020 and use it to fit the model, the forecasting "power" of each model would be better, even beyond 2020. It is like modeling in the future to predict the future, even though we are starting our forecast in the past.

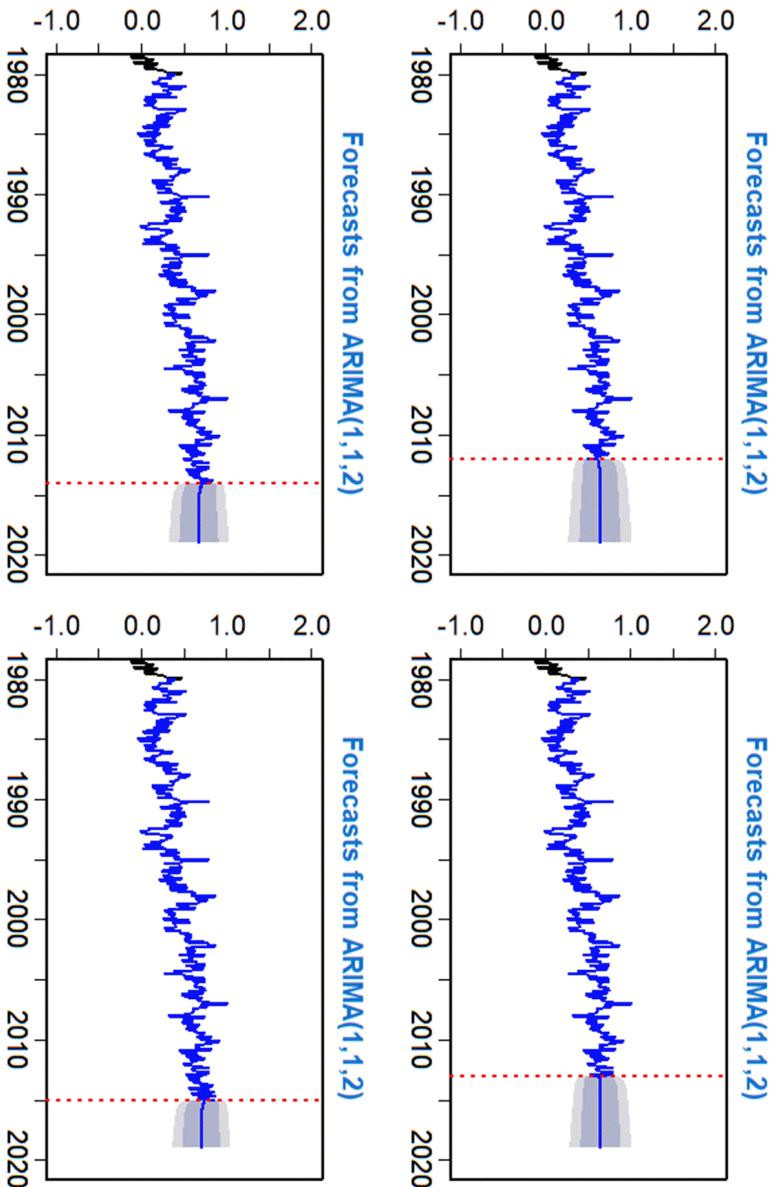


Figure 7-11. ARIMA(1,1,2) with four different forecast windows

Seasonal ARIMA models

When there is seasonality in the time series, as our series has, we should use seasonal ARIMA models or SARIMA models. We did not do so here in order to illustrate some other ARIMA features. However, we can estimate the effect of SARIMA models based on the work we have done already. We use the `estimate()` function for this purpose.

The model output shows us information on the components we used such as the estimate, test statistic, and lag for each moving average and each simple smoothing model. It also provides the correlation estimates and autocorrelation checks for several lag values.

```
estimate(dta.ts, p = 0, d = 1, q = 1, PDQ = c(0,1,1))
```

```
SARIMA(0,1,1)(0,1,1)(12) model is estimated for variable: dta.ts

Conditional-Sum-of-Squares & Maximum Likelihood Estimation
      Estimate      S.E t.value   p.value Lag
MA 1     -0.519  0.0249    -20.9 3.55e-86   1
SMA 1    -0.953  0.0098    -97.2 0.00e+00   1
-----
n = 1680; 'sigma' = 0.1092768; AIC = -2615.437; SBC = -2604.584
-----
Correlation of Parameter Estimates
      MA 1      SMA 1
MA 1  1.0000 -0.0406
SMA 1 -0.0406  1.0000
-----
Autocorrelation Check of Residuals
      lag    LB  p.value
[1,]  4 21.9 2.08e-04
[2,]  8 28.1 4.61e-04
[3,] 12 30.7 2.22e-03
[4,] 16 44.2 1.85e-04
[5,] 20 49.0 3.10e-04
[6,] 24 61.3 4.23e-05
-----
Model for variable: dta.ts
Period(s) of Differencing: dta.ts(1,1)

MA factors: 1 - 0.5191 B**(1)
SMA factors: 1 - 0.9526 B**(12)
```

```
estimate(dta.ts, p = 1, d = 1, q = 2, PDQ = c(1,1,2))
```

```

SARIMA(1,1,2)(1,1,2)(12) model is estimated for variable: dta.ts

Conditional-Sum-of-Squares & Maximum Likelihood Estimation
   Estimate   S.E t.value p.value Lag
AR 1    0.8658 0.0233  37.149  0.00e+00  1
MA 1   -1.3836 0.0384 -36.060  3.33e-211  1
MA 2    0.3945 0.0359  10.974  0.00e+00  2
SAR 1   -0.9319 0.0501 -18.595  2.62e-70  1
SMA 1   -0.0437 0.0581  -0.752  4.52e-01  1
SMA 2   -0.8582 0.0564 -15.217  4.48e-49  2
-----
n = 1680; 'sigma' = 0.1067915; AIC = -2680.971; SBC = -2648.412
-----
Correlation of Parameter Estimates
      AR 1     MA 1     MA 2     SAR 1     SMA 1     SMA 2
AR 1  1.00000 -0.7937  0.7601  0.0173  0.00713  0.0566
MA 1  -0.79366  1.0000 -0.9951  0.0154 -0.03946 -0.0210
MA 2   0.76006 -0.9951  1.0000 -0.0172  0.03668  0.0148
SAR 1   0.01731  0.0154 -0.0172  1.0000 -0.96852  0.9656
SMA 1   0.00713 -0.0395  0.0367 -0.9685  1.00000 -0.9337
SMA 2   0.05657 -0.0210  0.0148  0.9656 -0.93365  1.0000
-----
Autocorrelation Check of Residuals
   lag    LB p.value
[1,]  4  6.09  0.193
[2,]  8  7.79  0.454
[3,] 12 10.04  0.612
[4,] 16 19.96  0.222
[5,] 20 22.46  0.316
[6,] 24 26.77  0.315
-----
Model for variable: dta.ts
Period(s) of Differencing: dta.ts(1,1)

AR factors: 1 + 0.8658 B**(1)
MA factors: 1 - 1.3836 B**(1) + 0.3945 B**(2)
SAR factors: 1 - 0.9319 B**(12)
SMA factors: 1 - 0.0437 B**(12) - 0.8582 B**(24)

```

When we fit a full (as opposed to estimate) SARIMA model in R, the heuristic algorithm iterates until it converges to a local (as opposed to global) optimum solution, by iterating through the non-seasonal and seasonal parameters to find the optimum coefficients with minimal diagnostic metric values, like RMSE. Using what we have discovered with our experiments, we will try an ARIMA(1,1,2)(1,1,2)12 model and look at its performance.

```
sarima(dta.ts, 1, 1, 2, P = 1, D = 1, Q = 2, S = 12)
```

```
initial value -1.788148
iter  2 value -2.021854
iter  3 value -2.173523
iter  4 value -2.178366
iter  5 value -2.179381
...
iter  54 value -2.205732
iter  55 value -2.205770
iter  56 value -2.205783
iter  57 value -2.205784
iter  57 value -2.205784
final value -2.205784
converged
initial value -2.218097
iter  2 value -2.218105
iter  3 value -2.223418
iter  4 value -2.224290
iter  5 value -2.224645
...
iter  90 value -2.227266
iter  91 value -2.227267
iter  92 value -2.227268
iter  92 value -2.227268
final value -2.227268
converged
$fit
Call:
stats::arima(x = xdata, order = c(p, d, q), seasonal = list(orde
r = c(P, D,
      Q), period = S), include.mean = !no.constant, transform.pars
= trans, fixed = fixed,
      optim.control = list(trace = trc, REPORT = 1, reltol = tol))
Coefficients:
            ar1      ma1      ma2      sar1      sma1      sma2
       0.8658  -1.3836  0.3945  -0.9319  -0.0437  -0.8582
  s.e.  0.0233   0.0384  0.0359   0.0501   0.0581   0.0564

sigma^2 estimated as 0.0114:  log likelihood = 1347.49,  aic = -
2680.97
$degrees_of_freedom
[1] 1661
$ttable
    Estimate      SE  t.value p.value
ar1    0.8658  0.0233  37.1489  0.000
ma1   -1.3836  0.0384 -36.0605  0.000
ma2    0.3945  0.0359  10.9738  0.000
sar1   -0.9319  0.0501 -18.5947  0.000
```

```

sma1 -0.0437 0.0581 -0.7522 0.452
sma2 -0.8582 0.0564 -15.2170 0.000

$AIC
[1] -1.597718
$AICc
[1] -1.597688
$BIC
[1] -1.575113

```

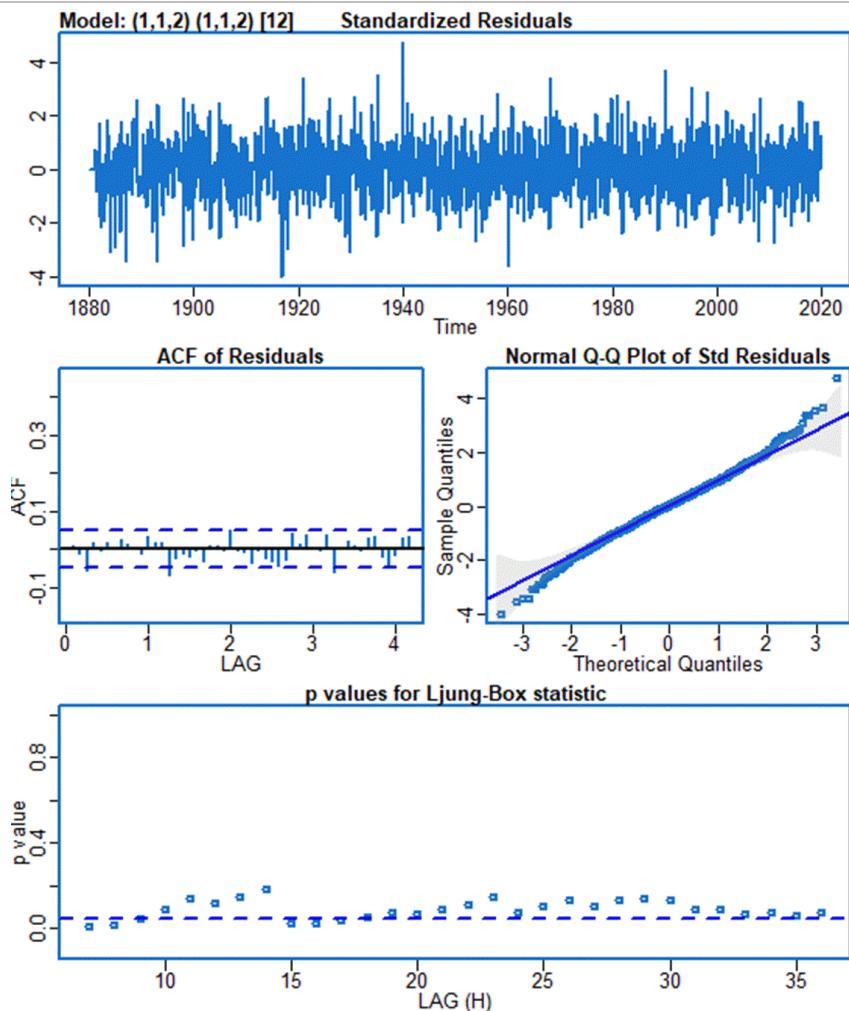


Figure 7-12. ARIMA(1,1,2)(1,1,2)12 plots for temperature change

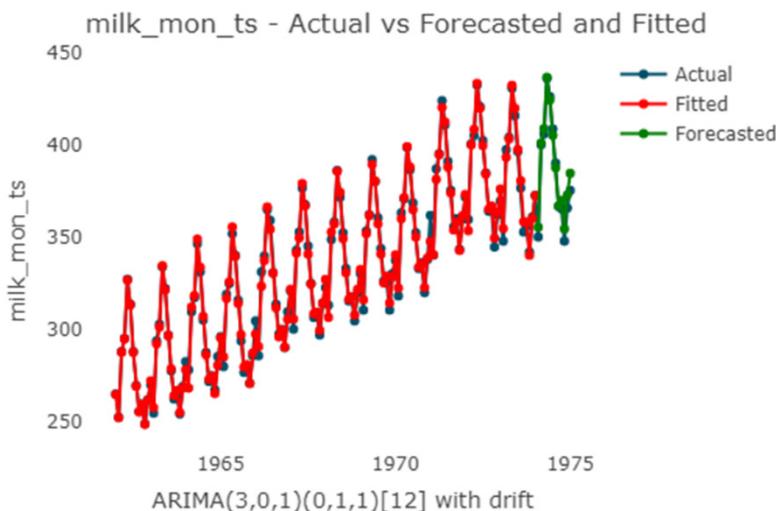
Got Milk? (Train-Test Split)

It has been a while since we visited the Milk problem. Here, we'll investigate the concept of training a model with `auto.arima`, while withholding a test set for validating or model when we forecast. We use the `TSstudio` and `forecast` packages for this purpose.

```
library(TSstudio)
# Setting training and testing partitions
milk_s <- ts_split(ts.obj = milk_mon_ts, sample.out = 12)
train <- milk_s$train
test <- milk_s$test

# Forecasting with auto.arima
library(forecast)
md <- auto.arima(train)
fc <- forecast::forecast(md, h = 12)

# Plotting actual vs. fitted and forecasted
test_forecast(actual = milk_mon_ts, forecast.obj = fc, test = te
st)
```



```
accuracy(fc)
```

	ME	RMSE	MAE	MPE
Training set	0.02808121	3.058498	2.214152	-0.000479586
	MAPE	MASE	ACF1	
Training set	0.6701402	0.2167621	0.007400993	

Chapter Review

In this chapter we revisited the ARIMA models using historic global percent change in temperature data. We first learned to get the data into a time series format suitable for analysis with R. Next, we determined if we could use ARIMA to model the series by decomposing the series and examining it for seasonality, trend, and stationarity.

As we studied stationarity, we learned the details of using the Dickey-Fuller tests, standard and augmented, as well as using the Phillips-Perron test.

Next, we investigated how ARIMA models work and covered four specific ones along with their features. Then we used these models to fit the temperature data and compared the outcomes.

Finally, we performed several diagnostic tests, visual and numeric, to ascertain which model would be most suitable for final analysis. Here, we discovered that the best choice is not always obvious, and that the best choice might not be much better than others. This is where judgment comes into play.

Review Exercises

1. Using the *lynx* dataset from the *fma* (Forecasting Methods and Applications) package in R:
 - a. Read the documentation and describe the dataset.
 - b. Download, save and import the time series data.
 - c. Inspect the data to determine if it is in a format R can read or otherwise transform it.
 - d. Determine if the data can be fitted with ARIMA or otherwise fix the problems, like non-stationarity.
 - e. Analyze the time series using R, and specify an appropriate ARIMA model.
 - f. Perform diagnostics and evaluate the ARIMA model
 - g. Build at least one additional ARIMA models and compare it (them) with the original.
2. Using the *m.urate* dataset from the *FinTS* package in R:
 - a. Read the documentation and describe the dataset.

- b. Download, save and import the Inflation Rate for the same monthly periods as the m.urate time series data.
 - c. Analyze the time series using R, and specify an appropriate ARIMA model.
 - d. Perform diagnostics and evaluate the ARIMA model.
 - e. Build a 12-quarter forecast using R.
 - f. Build the ARIMA model with the Inflation Rate as an external predictor.
 - g. Compare the two models.
- 3. Using the *q.gnp4791* dataset from the *FinTS* package in R:
 - a. Read the documentation and describe the dataset.
 - b. Analyze the time series using R, and specify an appropriate Structural model.
 - c. Build the Structural model in R.
 - d. Perform diagnostics and evaluate the Structure model.
 - e. Build a 12-quarter forecast using R.
 - f. Build a challenger model and repeat (e) and (f).
 - g. Compare the two models.
- 4. There are several "Most Popular Historic Data Pages" at the NASDAQ.
 - a. Download data for Apple, Inc. and Microsoft Inc, for the most recent 5-year period.
 - b. Make each set of data a times series and plot them on the same graph
 - c. State anything you notice about the two stocks.
 - d. Using proper techniques, fit ARIMA model to the data.
 - e. Evaluate the fit of your models.
 - f. Forecast out to the next 3-months, 4-months, and 6-months and discuss your results.

Chapter 8 – ARIMA Models using Python

In this chapter, we aim to cover the use state space models and advanced visualization of time series data in Python. Like R, Python has multiple libraries or practices that support time series. Our coverage of state space model is merely to introduce the concept and language. Although we will not dwell on it, state space is the underlying theory.

State Space and ARIMA

In Chapter 3, we made reference to state space models when discussing Error-Trend-Seasonality (ETS) models. ETS models are a type of Exponential Smoothing State Space model. The ETS model arose in the space tracking setting, where the state equation defines the motion equations for the position or state of a spacecraft with location x_t and the data y_t reflect information that can be observed from a tracking device such as velocity and azimuth.

Definition 8.1. A general state space model is of the form

$$y_t = \varphi y_{t-1} + d_t + \varepsilon_t$$

$$x_{t+1} = \theta x_t + b_t + \eta_t$$

where y_t refers to the observation vector at time t , x_t refers to the (unobserved) state vector at time t , φ and θ are deterministic, and where the irregular components are defined as

$$\varepsilon_t \sim N(\mathbf{0}, \varphi_t)$$

$$\eta_t \sim N(\mathbf{0}, \theta_t)$$

One class of state space models are the so-called Seasonal Autoregressive Integrated Moving Average Exogenous models (SARIMA). Like the seasonal ARIMA models we discussed in Chapter 6, SARIMA models also can be described under the ETS taxonomy. For instance, a SARIMAX(2 1, 0) \times (1, 1, 0, 12) is one with ARIMA (2,1,0) \times (1,1,0)12

Recall that a nonseasonal ARIMA model (Definition 6.11) is classified as an ARIMA(p, d, q) model, where:

p = autoregressive terms (AR) order

d = differences (DIFF) order needed for stationarity

q = moving average (MA) order

In this model, we will add a difference to achieve stationarity. We will also make it a seasonal model. The first three numeric values (p, d, q) in $\text{sarima}(x, p, d, q, P, D, Q, S)$ are the usual AR, DIFF, and MA components. We used the second set of values (P, D, Q) in Chapter 6 but without much explanation. These are the seasonal AR, DIFF, and MA components, and S represents the seasonal period. In other words, we have a seasonal autoregressive order denoted by upper-case P , an order of seasonal integration denoted by upper-case D , and a seasonal moving average order signified by upper-case Q . In summary,

x = univariate time series

p = AR order (must be specified)

q = DIFF order (must be specified)

q = MA order (must be specified)

P = SAR order; use only for seasonal models

D = seasonal difference; use only for seasonal models

Q = SMA order; use only for seasonal models

S = seasonal period; use only for seasonal models

If your time series is in x and we want to fit an ARIMA(p, d, q) model to the data, the basic call is $\text{sarima}(x, p, d, q)$. The values p, d, q , must be specified as there is no default. The results are the parameter estimates, standard errors, AIC, AICc, BIC (as defined in Chapter 2) and diagnostics. To fit a seasonal ARIMA model, we use $\text{sarima}(x, p, d, q, P, D, Q, S)$. For example, $\text{sarima}(x, 2, 1, 0)$ will fit an ARIMA(2,1,0) model to the series in x , and

`sarima(x, 2, 1, 0, 0, 1, 1, 12)`

will fit a seasonal ARIMA(2,1,0) * (0,1,1)12 model to the series in x . The difference between the information criteria given by `sarima()` and `arima()` is that they differ by a scaling factor of the effective sample size.

Now, for putting an ARIMA model in state space form, we have

$$Y_t = \varphi_1 Y_{t-1} + \varphi_2 Y_{t-2} + \varepsilon_t, \quad \varepsilon_t \sim NID(0, \sigma^2)$$

Taxonomy

As we described a taxonomy for exponential smoothing models, we will now define one for the models at present. Each model consists of a measurement equation that describes the observed data, and some state equations that describe how the unobserved components or states (level, trend, seasonal) change over time. Hence, we refer to them as state space models.

While we classified the exponential smoothing models as additive and nonadditive *trend* and *seasonal* requirements., we add *level* to the taxonomy. Thus, ETS(A, N, N) is **simple exponential smoothing with additive errors**. Out[2] shows some of the additive ETS framework models

Working with ARIMA(p,q,d) with Python

Whether it is during personal projects or our day-to-day work as a Data Scientist, it is likely that we will encounter situations that require the analysis and visualization of multiple time series at the same time. Assuming that the data for each time series is stored in distinct columns of a file, the `pandas` library makes it easy to work with multiple time series. We will work with a new time series dataset that contains the amount of different types of meat produced in the USA between 1948 and 2018.

The *Livestock & Meat Domestic Data* contains current and historical data on pork, beef, veal, and poultry, including production, supply, utilization, and farm prices. The data can be found at the USDA, ERS Livestock and Meat Domestic Data page:

<https://www.ers.usda.gov/data-products/livestock-meat-domestic-data/>

```
In [1]
import numpy as np
from scipy import stats
import pandas as pd
import matplotlib
from matplotlib import *
import statsmodels.api as sm
from statsmodels.graphics.api import qqplot
from pandas import Series, DataFrame, Panel, Timestamp
import plotnine
from ggplot import *
from plotnine import *
from plotly.tools import mpl_to_plotly as ggplotly
```

The `Panel` class will be removed from pandas. Accessing it from the top-level namespace will also be removed in the next version. We downloaded the data (slightly different) from the .gov page. we show the monthly meat production from January 1948 – December 1948 in Out[2].

```
In [2]
meat= pd.read_csv("D:/Documents/Data/meat2.csv")
meat_df=meat[['date','cattle','hogs','sheep','poultry']]
meat_df.head(12)
```

	date	cattle	hogs	sheep	poultry
0	1948-01-01	51.3	144.6	1464.0	10.4
1	1948-02-01	37.7	665.6	1306.7	87.6
2	1948-03-01	35.7	1242.0	1297.2	79.6
3	1948-04-01	45.1	377.5	1178.1	14.7
4	1948-05-01	52.9	833.5	1095.3	67.2
5	1948-06-01	81.0	1377.2	1405.0	84.9
6	1948-07-01	75.4	1398.8	1333.1	16.3
7	1948-08-01	83.6	857.7	1406.2	31.2
8	1948-09-01	68.9	1025.4	1615.7	83.3
9	1948-10-01	54.1	1653.6	1768.7	15.7
10	1948-11-01	49.7	1522.9	1578.5	4.4
11	1948-12-01	45.7	1706.2	1448.4	10.2

Using feature of `pandas` dataframes, we call out information, `info()` and retrieve the following output, which tells us that we have a data frame with 871 entries with five columns, the meat data in the columns are `float64` (numeric) and the date field is an object that we will need to convert the data to a times series.

```
In [3]
meat_df.info()
```

```
Out [3]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 871 entries, 0 to 870
Data columns (total 5 columns):
 #   Column   Non-Null Count  Dtype  
---  -- 
 0   date      871 non-null    object  
 1   cattle    871 non-null    float64 
 2   hogs     871 non-null    float64 
 3   sheep    871 non-null    float64 
 4   poultry   871 non-null    float64 
dtypes: float64(4), object(1)
memory usage: 34.1+ KB
```

To convert the data to a times series, we need to determine the date format, which we see is `datetime64` or date and time. If a datetime fielded was missing, we would have to add one to it to get a times series.

```
In [4]
meat_df['date'] = pd.to_datetime(meat_df['date'])
meat_df['date']
```

```
Out [4]
0    1948-01-01
1    1948-02-01
2    1948-03-01
3    1948-04-01
4    1948-05-01
...
866   2020-03-01
867   2020-04-01
868   2020-05-01
869   2020-06-01
870   2020-07-01
Name: date, Length: 871, dtype: datetime64[ns]
```

Then, we need to change the index to date, which we can do with resampling for years months, using `set_index()`. The output of Out[5] shows the meat production annual totals time series from 1948 – 1957.

```
In [5]
meat_yr = meat_df.set_index ('date').resample('1Y').sum()
meat_yr.head(10)
```

Out [5]

	cattle	hogs	sheep	poultry
date				
1948-12-31	681.1	12805.0	16896.9	505.5
1949-12-31	630.7	17095.0	13376.6	388.5
1950-12-31	1465.0	15056.0	12852.3	596.3
1951-12-31	3392.1	16528.5	11074.8	425.0
1952-12-31	2765.7	16198.9	13962.4	559.6
1953-12-31	2761.1	12407.1	15967.4	547.5
1954-12-31	2530.5	11374.8	15919.6	666.8
1955-12-31	23762.3	14033.7	16215.1	596.0
1956-12-31	24996.1	13713.3	15993.4	555.7
1957-12-31	24956.8	12122.2	14957.4	651.0

Now, we set the time series frequency as months, again using `set_index()`. Out [6] shows the monthly meat production time series from 1948-2019

In [6]

```
meat_ts = meat_df.set_index(['date'])
meat_ts.head(10)
```

Out [6]

	cattle	hogs	sheep	poultry
date				
1948-01-01	51.3	144.6	1464.0	10.4
1948-02-01	37.7	665.6	1306.7	87.6
1948-03-01	35.7	1242.0	1297.2	79.6
1948-04-01	45.1	377.5	1178.1	14.7
1948-05-01	52.9	833.5	1095.3	67.2
1948-06-01	81.0	1377.2	1405.0	84.9
1948-07-01	75.4	1398.8	1333.1	16.3
1948-08-01	83.6	857.7	1406.2	31.2
1948-09-01	68.9	1025.4	1615.7	83.3
1948-10-01	54.1	1653.6	1768.7	15.7

Our next step is to plot the time series, as shown in Figure 8-1. Notice the aesthetics we applied, combining a light grid background and multicolored time series plots, with titles, labels, and a legend.

In [7]

```
import matplotlib.dates as mdates
from matplotlib.dates import YearLocator

fig, ax = plt.subplots(figsize = (6.5, 4), dpi = 150)
plt.xticks(np.arange(1, 767, 12*6))
ax = df.plot(figsize = (6.5,4), linewidth = 1, fontsize = 10,
```

```

        ax = ax)
plt.xticks(rotation = 45, ha = 'right')
plt.xlabel('Year', fontsize = 14, color = 'steelblue',
           fontdict = dict(weight = 'bold'))
plt.ylabel('Value', fontsize=14, color = 'steelblue',
           fontdict = dict(weight = 'bold'))
plt.title('Annual Meat Production', fontsize=18,
           color = 'steelblue', fontdict = dict(weight = 'bold'))
plt.legend(fontsize = 11)
plt.grid(True, linestyle = ':')
ax.spines['bottom'].set_color('darkgray')
ax.spines['left'].set_color('darkgray')
ax.spines['right'].set_color('gray')
ax.spines['top'].set_color('gray')
ax.tick_params(axis = 'x', colors = 'gray')
ax.tick_params(axis = 'y', colors = 'gray')
plt.show()

```

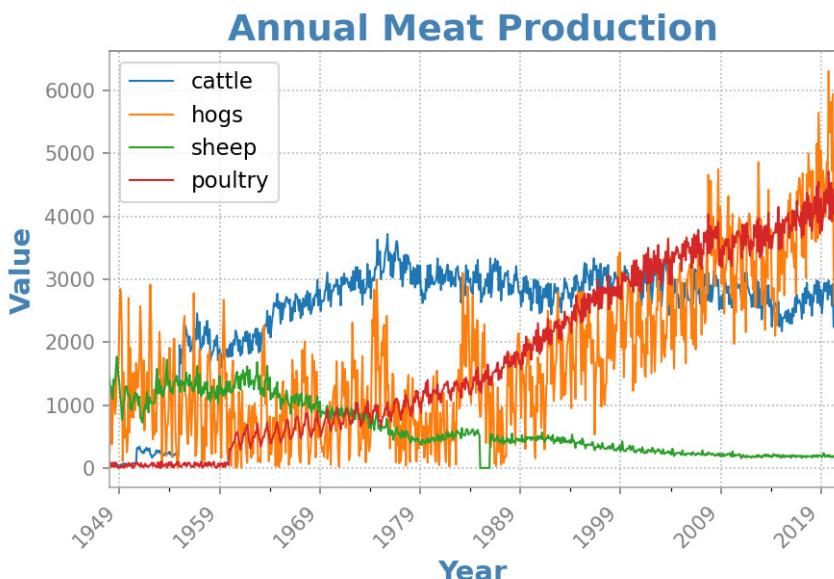


Figure 8-1. Meat production time series from 1948 to 2019

In **Figure 8-2**, we add more aesthetics by changing the grid background, and border, but leaving the times series colors the same.

```

In [8]
fig, ax1 = plt.subplots(figsize = (7, 4), dpi = 150)
plt.xticks(np.arange(1, 767, 12*6))

```

```
In [9]
ax = df.plot(figsize = (7,4), linewidth = 1, fontsize = 14,
             ax = ax1)
plt.rcParams.update({'figure.figsize' : (7, 4),
                     'figure.facecolor': 'navy',
                     'axes.facecolor' : 'azure',
                     'axes.edgecolor': 'white',
                     'text.color': 'slateblue',
                     'font.weight': 'bold',
                     'axes.labelweight': 'bold'}))
plt.tick_params(axis = 'both', direction = 'out', length = 6,
                width = 2, labelcolor = 'w', colors = 'r',
                grid_color = 'white', grid_alpha = 0.5)
ax.set_xlabel('Date', color = "white", fontsize = 14)
ax.set_ylabel('Year', color = "white", fontsize = 14)
plt.title('Meat production', fontsize = 18, color = 'white',
          fontdict = dict(weight = 'bold'))
ax.legend(fontsize = 12);

plt.grid(True, color = 'royalblue', linestyle = ':')
pyplot.show()
```

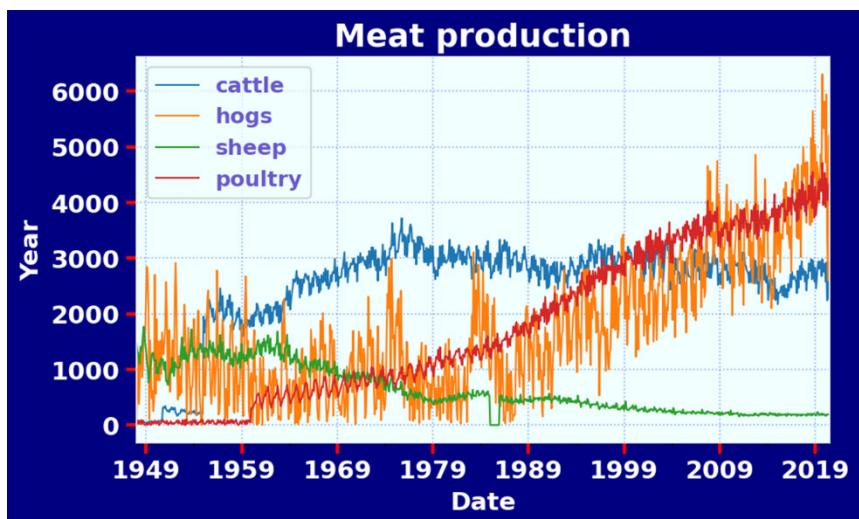


Figure 8-2. Meat production time series from 1948 to 2019 (enhanced)

The next code snippet just reprints the meat time series in table without the `head()` parameter so we see the beginning and end of the data. Notice that it is relatively current (it's September 2020 right now.) We

show the output of monthly meat production time series data in Out[10].

```
In [10]
meat_ts
```

```
Out [10]
```

	cattle	hogs	sheep	poultry
date				
1948-01-01	51.3	144.6	1464.0	10.4
1948-02-01	37.7	665.6	1306.7	87.6
1948-03-01	35.7	1242.0	1297.2	79.6
1948-04-01	45.1	377.5	1178.1	14.7
1948-05-01	52.9	833.5	1095.3	67.2
...
2020-03-01	2921.7	5942.0	187.5	4427.8
2020-04-01	2239.1	3412.0	180.8	4098.9
2020-05-01	2277.5	2595.5	195.3	4039.1
2020-06-01	2876.3	5176.8	193.4	4334.7
2020-07-01	2918.4	5225.4	195.1	4303.0

871 rows × 4 columns

Visualizing multiple time series

If there are multiple time series in a single dataframe, you can still use the `.plot()` method to plot a line chart of all the time series. Another interesting way to plot these is to use area charts. Area charts are commonly used when dealing with multiple time series, and can be used to display cumulated totals.

With the `pandas` library, you can simply leverage the `plot.area()` method to produce area charts in Figure 8-3, of the smoothed time series data from our dataframe.

```
In [11]
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose
fig, ax = plt.subplots(linewidth=2, figsize=(6,3.5), dpi = 300)
plt.plot(meat_yr)
plt.xlabel('Year', fontsize=12, color = 'navy', fontdict =
           dict(weight = 'bold'))
plt.ylabel('Value', fontsize=12, color = 'navy', fontdict =
           dict(weight = 'bold'))
plt.title('Annual Meat Production', fontsize=16, color = 'navy',
           fontdict = dict(weight = 'bold'))
lines = [Line2D([0], [0], color=c, linewidth=2, linestyle='-' )
```

```

        for c in colors]
plt.grid(True, color = 'gray', linestyle = ':')
plt.legend(['Cattle', 'Hogs', 'Sheep', 'Poultry'],
           loc = 'upper left', fontsize=12);
plt.show()

```

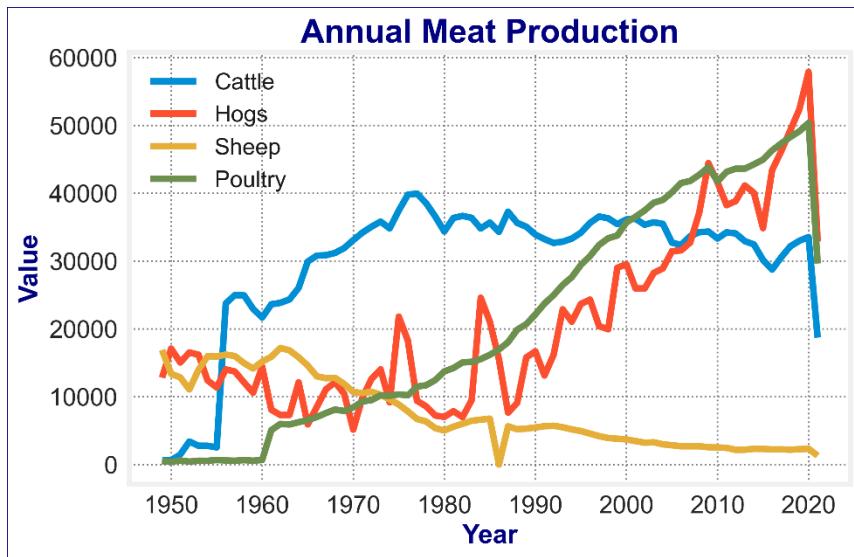


Figure 8-3. Smoothed meat production data for 1948 to 2019

Now, using the `describe()` function, we can view the descriptive statistics of the time series data.

```

In [12]
# Print the summary statistics of the DataFrame
print(meat_ts.describe())

```

	cattle	hogs	sheep	poultry
count	871.000000	871.000000	871.000000	871.000000
mean	2493.762457	1757.632262	634.203444	1781.018255
std	842.522071	1245.662321	427.655423	1385.624937
min	29.800000	2.500000	0.000000	0.700000
25%	2421.950000	796.700000	247.750000	609.000000
50%	2773.800000	1520.300000	477.500000	1360.400000
75%	2966.350000	2551.200000	1013.000000	3180.900000
max	3716.200000	6302.500000	1768.700000	4711.400000

Defining color palettes

When visualizing multiple time series, it can be difficult to differentiate between various colors in the default color scheme.

To remedy this, we can define each color manually, but this may be time-consuming. Fortunately, it is possible to leverage the colormap argument to `plot()` to automatically assign specific color palettes with varying contrasts. We can either provide a [Matplotlib colormap](#) as an input to this parameter, or provide one of the default strings that is available in the `colormap()` function available from [matplotlib](#). For example, we can specify the '`'viridis'`' colormap using the code snippet below, combined with the `plot()` function, producing the plot in Figure 8-4:

```
In [13]
meat_ts.plot(figsize = (10,5), linewidth = 2, fontsize = 16,
             colormap = 'viridis')
```

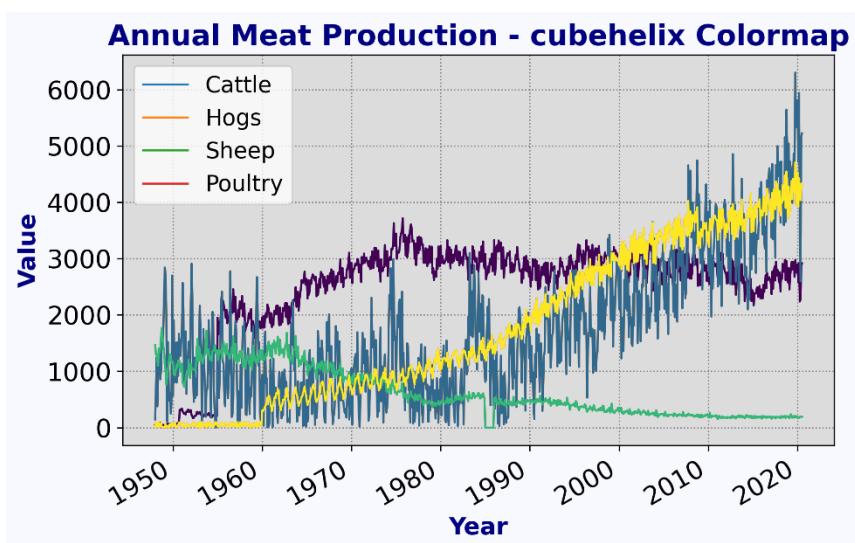


Figure 8-4. Meat production time series data using a colormap

Next, we repeat the plot in Figure 8-4, with a different colormap as seen in Figure 8-5.

```
In [14]
ax = meat_ts.plot(figsize = (10,5), linewidth = 2,
```

```

    fontsize = 16, colormap = 'cubeHelix');
# Additional customizations
ax.set_xlabel('Date');
ax.legend(fontsize = 16);

```

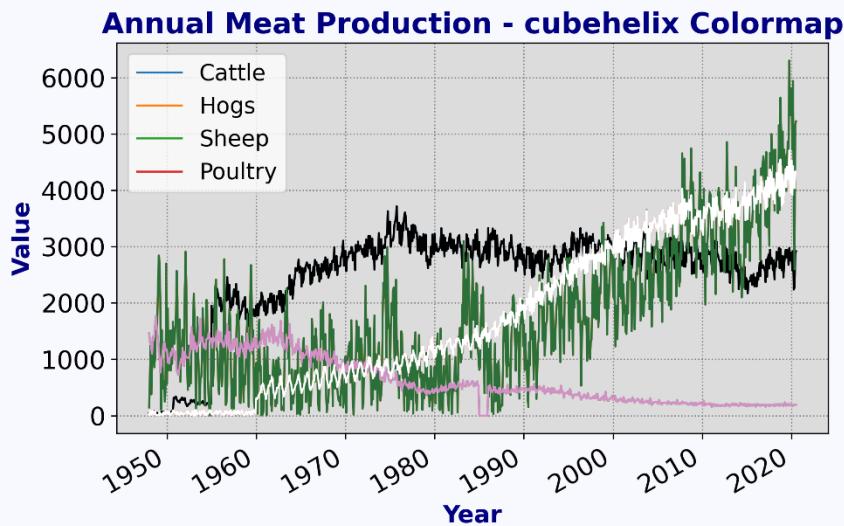


Figure 8-5. Same plots as Figure 8-4 with different colormap

It is possible to visualize time series plots and numerical summaries on one single graph by using the *pandas* API to *Matplotlib* along with the *table* method:

```
In [15]
meat_sum = meat_ts.sum().to_frame().T
meat_mean = meat_ts.mean().to_frame().T
meat_sum.rename({0:'sum'}, inplace = True)
meat_mean.rename({0:'mean'}, inplace = True)
meat_sum, meat_mean
```

	cattle	hogs	sheep	poultry
sum	2172067.1	1530897.7	552391.2	1551266.9,
	cattle	hogs	sheep	poultry
mean	2493.762457	1757.632262	634.203444	1781.018255)

Now, we take the tabular data for means and combine them with the times series plot as seen in Figure 8-6.

```

In [16]
from matplotlib.lines import Line2D
fig, ax = plt.subplots(figsize = (5.5, 4), dpi = 300)
ax.plot(df, linewidth = 0.75)

plt.rcParams.update({'figure.facecolor': 'linen',
                     'axes.facecolor' : 'linen',
                     'text.color': 'slateblue',}),
# Add x-axis labels
ax.set_xlabel('Date', fontsize = 12)

# Add summary table information to the plot
sax = ax.table(cellText = meat_mean.values.round(2),
               colWidths = [0.25] * len(meat_mean.columns),
               rowLabels = meat_mean.index,
               colLabels = meat_mean.columns, loc = 'top')
sax.auto_set_font_size(False)
sax.set_fontsize(10)
sax.scale(1, 1.2)
ax.grid(b = None, which = 'major', axis = 'both',
color = 'gray', linestyle = ':')
# Specify the fontsize and location of your legend
Lines = [Line2D([0], [0], color = c, linewidth = 2, linestyle = '-')
          for c in colors]
labels = ['Cattle', 'Hogs', 'Sheep', 'Poultry']
plt.legend(loc = 'upper left', labels = labels);

```

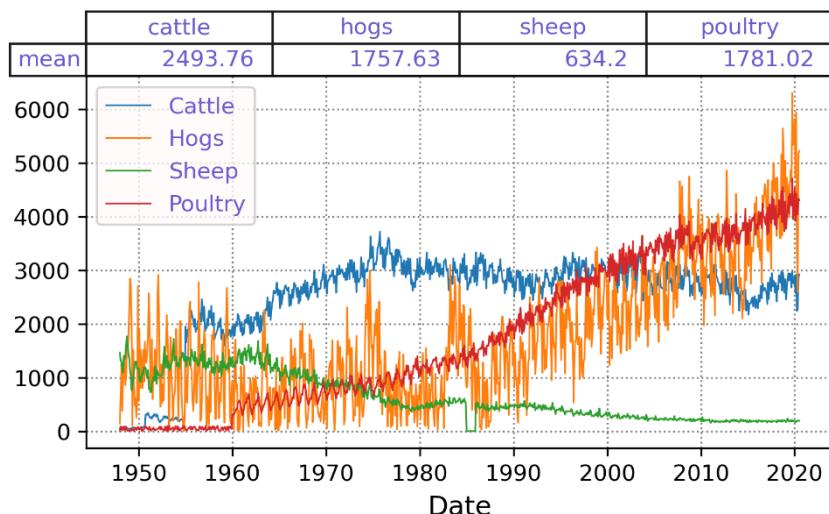


Figure 8-6. Meat production times series with tabular mean data

It can be beneficial to plot individual time series on separate graphs as this may improve clarity and provide more context around each time series in your dataframe.

It is possible to create a "grid" of individual graphs, "faceting" each time series by setting the `subplots()` argument to `True`, as seen in Figure 8-7. In addition, the arguments that can be added are:

- `Layout()`: specifies the number of rows x columns to use.
- `Sharex()` and `sharey()`: specifies whether the x-axis and y-axis values should be shared between your plots.

```
In [17]
fig, ax = plt.subplots(figsize = (5.5, 10), dpi = 300)
plt.rcParams.update({'grid.color': 'lightgray',
                     'grid.linestyle': ':'})
ax.grid(b = None, which = 'major', axis = 'both',
        color = 'lightgray', linestyle = ':')
df.plot(subplots = True,
         layout = (4, 1),
         grid = True,
         sharex = True,
         sharey = True,
         colormap = 'Dark2',
         fontsize = 12,
         rot = 90,
         legend = True,
         linewidth = 0.5,
         ax = ax);
plt.savefig('03_meat_ind_ts.png', figsize = (6.5,4), dpi = 150,
bbox_inches = 'tight')
plt.tight_layout();
```

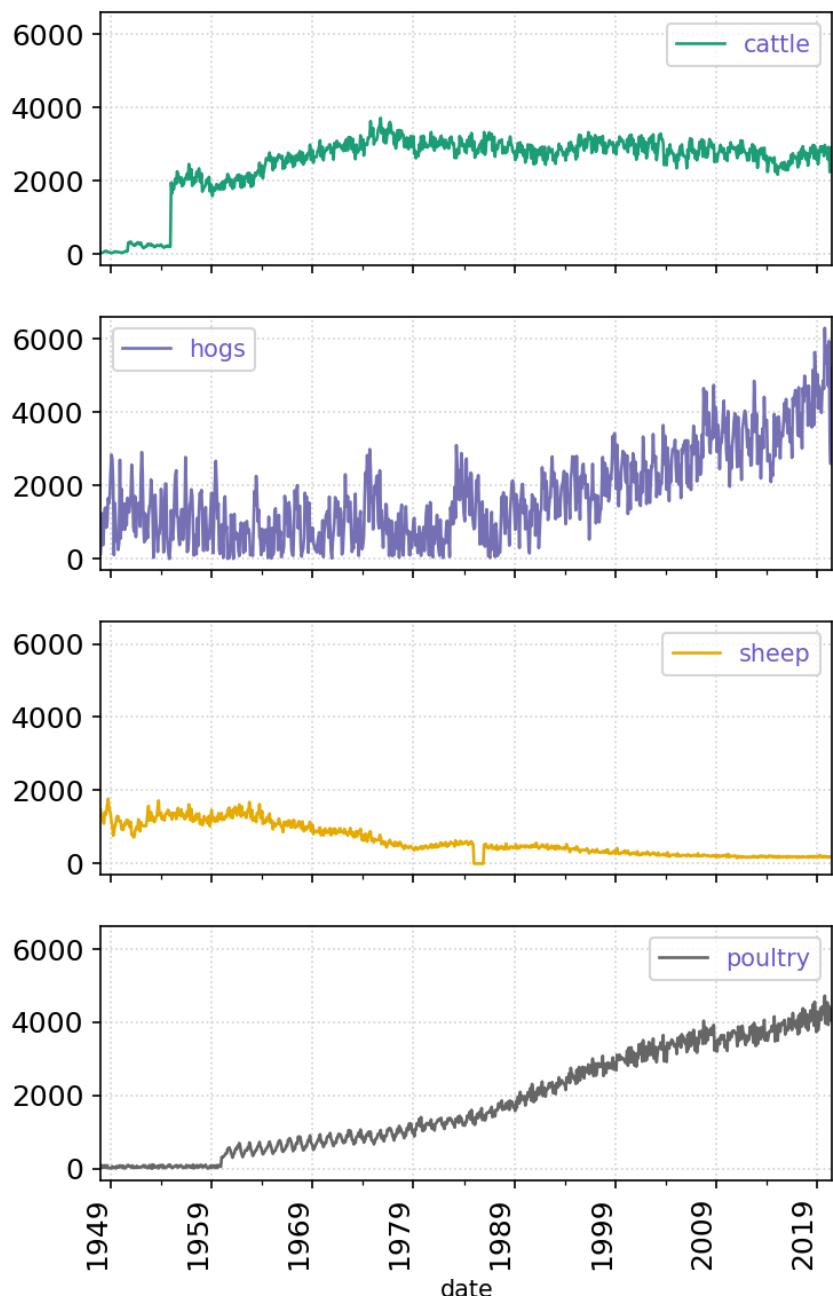


Figure 8-7. Grid plot of each meat product in the dataframe

Here, we introduce the construction of a bar plot using *Matplotlib*. We believe it is much more flexible than ggplot. The bar plot we make uses an process in which we gather the data by product (variable) and stack the sum of the values of the products by decades. So we have a little work to do before we plot.

First, we define a floor function to group the products by decade and ensure the date format is conducive for plotting.

```
In [18]
def floor_decade(date_value):
    "Takes a date. Returns the decade."
    return (date_value.year // 10) * 10
pd.to_datetime('2013-10-9')
```

Next, we execute the function we defined and check the date format. If the date format is correct, then we can easily pull out the years (the first four characters of the date string). Then, grouping can be performed by year.

```
In [19]
dd = df.groupby(floor_decade).sum()
dd.head(5)
```

```
Out [19]
      cattle   hogs   sheep  poultry
1940  1311.8 29900.0     30273.5      894.0
1950  131274.6        136436.7     146286.3      5833.0
1960  285752.8        88021.7     141454.2     68782.4
1970  366708.6        117982.6     81315.7     109069.6
1980  356206.1        134759.4     53007.9     173902.4
```

The readout looks correct and usable for our bar plot. Now, we need to breach the data down and gather it is a format that can be easily plotted by a bar graph. For the year field, we will just make a list of the ten decades, starting with 1949 and ending with 2020, Then we will make arrays for each of the four products.

```
In [20]
# Bar plot data wrangling
years =
['1940', '1950', '1960', '1970', '1980', '1990', '2000', '2010', '2020']
cattle = dd['cattle']
hogs  = dd['hogs']
```

```
sheep = dd['sheep']
poultry = dd['poultry']
ind = [x for x, _ in enumerate(years)]
```

Now we are ready to construct the plot, which we do by making each product a bar of the plot. Since we have already indexed the data by date, the floor function also "compresses" the dates into decades and these are mapped to the values of each product.

```
In [21]
# Set up figure of bars
fig, ax = plt.subplots(figsize = (8, 5), dpi = 300)
plt.bar(ind, cattle, width = 0.8, label='cattle',
        color = 'dodgerblue', bottom = hogs + sheep + poultry) +\
plt.bar(ind, hogs, width = 0.8, label = 'hogs',
        color = 'salmon',
        bottom = sheep + poultry) +\
plt.bar(ind, sheep, width = 0.8, label = 'sheep',
        color = 'blueviolet', bottom = 'poultry') +\
plt.bar(ind, poultry, width = 0.8, label = 'poultry',
        color = 'lightgreen')
```

And, now we are ready to plot. First, we set up the axis. Recall that we are trying to stack the values of each product by year (it might be helpful to sneak a peek at Figure 8-8). Next, we set up any aesthetics we want, including labels and a legend. We can use either a color map or assign colors to the product bars manually, which fairly easy with just four products. One step that is missing from the process is saving the image. We do this for all our plots (they appear in this book). This is pretty straight forward for *Matplotlib* graphs and cumbersome for *ggplot* graphs. Here is an example for saving a Matplotlib plot:

```
In [22]
plt.savefig('03_meat_box.png', figsize = (6,3), dpi = 300,
            bbox_inches = 'tight')
```

We simply, call on `plt.savefig`, give the plot a name and suffix (jpg, png, tiff, bmp, or emf, to name a few), and give it a figure size, which can be different for our save work vs our plot in Jupyter. Finally, we set the resolution by dots-per-inch or dpi, which we set a fairly high resolution of 300 dpi. Our plot appears as Figure 8-8.

```
In [23]
# Set up axes
plt.xticks(ind, years)
plt.xticks(fontsize = 12)
plt.yticks(fontsize = 12)

# Set up labels & legend
plt.ylabel('Head Count', fontsize = 16, color = 'teal',
           fontdict = dict(weight = 'bold'))
plt.xlabel('Years', fontsize = 16, color = 'teal',
           fontdict = dict(weight = 'bold'))
plt.title('Meat production by Decade', fontsize = 20,
           color = 'teal', fontdict = dict(weight = 'bold'))
plt.legend(loc = 'upper left', fontsize = 14)
```

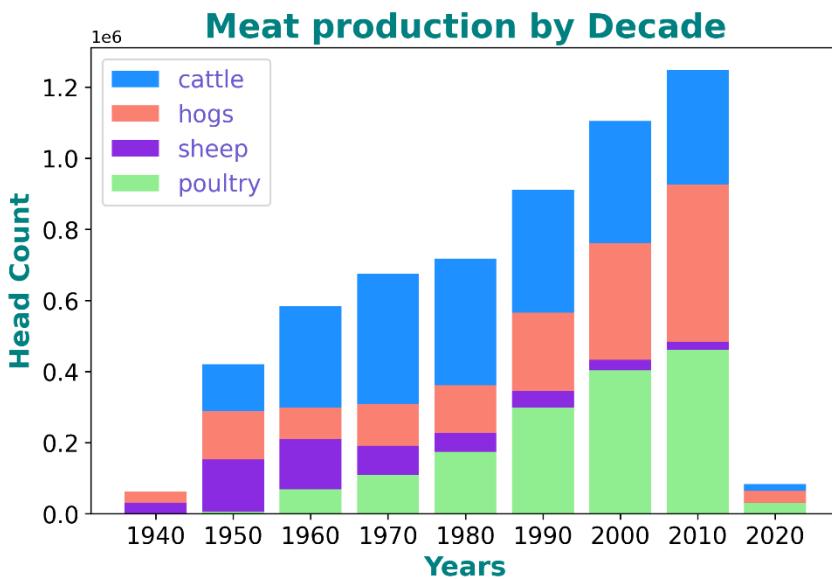


Figure 8-8. Stacked bar chart for each product head count by year

Additional descriptive plots, like the box plots in Figure 8-9, can be produced. Referring to the box plots, do you think having equal means and variances might be a valid hypothesis? This answer can be easily represented by a multi-variable box plot, which we show in Figure 8-9. It's a pretty safe bet to conclude that cattle (beef) is statistically different than sheep at a high level of confidence. We can confirm our belief using the bar plot in Figure 8-8.

```
In [24]
fig, ax = plt.subplots(linewidth = 2, figsize = (8, 5),
                      dpi = 300)
plt.rcParams.update({'figure.facecolor': 'white',
                     'lines.linewidth': 2,
                     'axes.facecolor' : 'white'}),
ax = df.boxplot(figsize = (6, 3), rot = 90, fontsize = 14,
                 boxprops = dict(color = 'blue'),
                 capprops = dict(color = 'green'),
                 whiskerprops = dict(color = 'indigo'),
                 medianprops = dict(color = 'darkmagenta'),
                 flierprops = dict(color = 'yellow',
                                   markeredgecolor = 'crimson'),
                 )
plt.title('Box Plot of Meat Production by Type', fontsize = 16,
          fontdict = dict(weight = 'bold'))
plt.savefig('03_meat_box.png', figsize = (6, 3), dpi = 300,
            bbox_inches = 'tight')
pyplot.show()
```

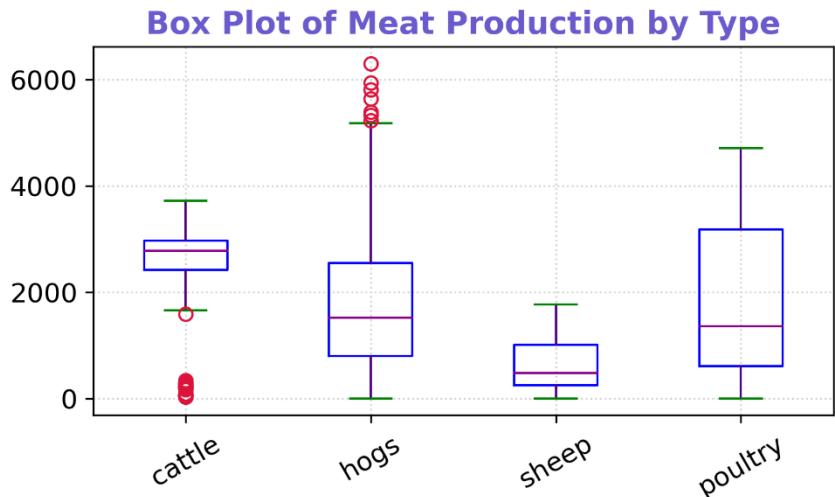


Figure 8-9. Box plots of the meat production data

Correlations between two Variables

Recall, the correlation coefficient is a measure used to determine the strength or lack of relationship between two variables. **Pearson's coefficient** can be used to compute the correlation coefficient between

variables for which the relationship is thought to be linear. **Kendall Tau** or **Spearman rank** can be used to compute the correlation coefficient between variables for which the relationship is thought to be non-linear

Correlation and Correlation Matrices

When computing the correlation coefficient between more than two variables, you obtain a correlation matrix

- Range: [-1, 1]
- 0: No relationship
- 1: Strong positive relationship
- -1: Strong negative relationship
- A correlation matrix is always "symmetric"
- The diagonal values will always be equal to 1

The correlation coefficient can be used to determine how multiple variables (or a group of time series) are associated with one another. The result is a correlation matrix that describes the correlation between time series. Note that the diagonal values in a correlation matrix will always be 1, since a time series will always be perfectly correlated with itself.

Correlation coefficients can be computed with the `pearson`, `kendall` and `spearman` methods. A full discussion of these different methods is outside the scope of this course, but the `pearson` method should be used when relationships between your variables are thought to be linear, while the `kendall` and `spearman` methods should be used when relationships between your variables are thought to be non-linear.

```
In [25]
print(meat[['cattle','hogs','sheep','poultry']].corr(
    method = 'pearson'))
```

```
Out [25]
      cattle      hogs      sheep     poultry
cattle  1.000000  0.175676 -0.600732  0.494452
hogs   0.175676  1.000000 -0.579480  0.785756
sheep  -0.600732 -0.579480  1.000000 -0.858030
poultry 0.494452  0.785756 -0.858030  1.000000
```

Visualize Correlation Matrices

The correlation matrix generated in the previous exercise can be plotted using a heatmap. To do so, you can leverage the `heatmap()` function from the `seaborn` library which contains several arguments to tailor the look of our heatmap, as seen in Figure 8-10. A heatmap is an excellent visual for detecting correlation between variables. In Python, these plots are each to manipulate. We can use the `xticks()` and `yticks()` methods to rotate the axis labels so they don't overlap. To the right of the heatmap is a thermometer that is graduated from -0.75 , or hot in a negative direction, and 1 , which is hot in a positive direction. Zero is cold.

```
In [26]
import seaborn as sns
corr_meat = df.corr(method = 'spearman')
fig, ax = plt.subplots(linewidth = 2, figsize = (6, 4),
                      dpi = 300)
# Customize the heatmap of the corr_meat correlation matrix
sns.heatmap(corr_meat,
            annot = True,
            linewidths = 0.4,
            annot_kws = {'size': 12, 'weight' : 'bold'});
plt.title('Heat Map of Meat Correlations', fontsize = 14,
          color = 'navy', fontdict = dict(weight = 'bold'))
plt.xticks(rotation = 90);
plt.yticks(rotation = 0);
```

Clustered heatmaps

Heatmaps are extremely useful to visualize a correlation matrix, but **clustermaps** can be better. A `clustermap()`, from the `seaborn` library, allows to uncover structure in a correlation matrix by producing a hierarchically-clustered heatmap as we show in Figure 8-11.

```
In [27]
df_corr = df.corr()
fig = sns.clustermap(df_corr)
plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(),
         rotation = 90)
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
```

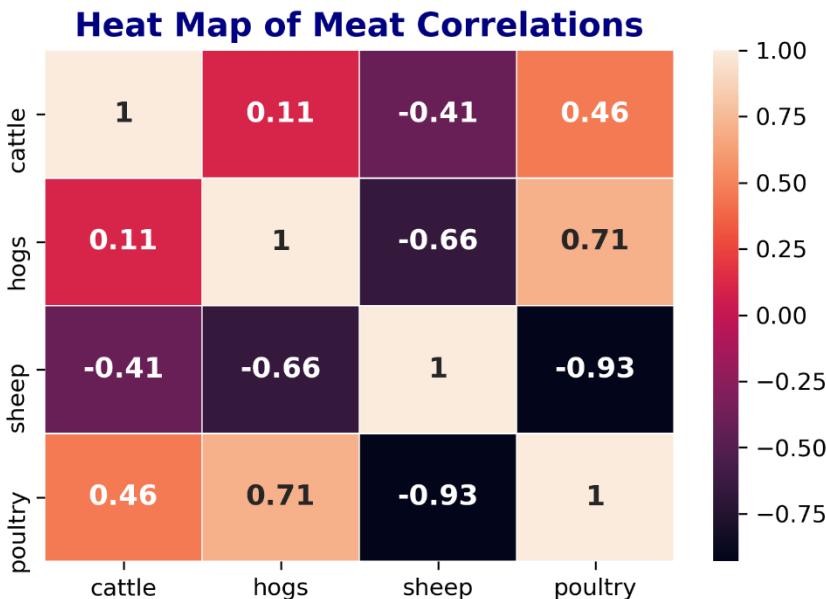


Figure 8-10. Heatmap of the meat production times series

To prevent overlapping of axis labels, you can reference the axes from the underlying fig object and specify the rotation. You can learn about the arguments to the `clustermap()` function here.

```
In [28]
corr_meat = meat_df.corr(method = 'pearson')
corr_meat = df.corr(method = 'pearson')

# Customize the heatmap of the corr_meat correlation matrix
fig = sns.clustermap(corr_meat,
                      row_cluster = True,
                      col_cluster = True,
                      figsize = (6, 6),
                      cmap = 'mako',
                      annot = True,
                      annot_kws={'size': 14, 'weight' : 'bold'});
plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(),
         rotation = 90, fontsize = 14);
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(),
         rotation = 0, fontsize = 14);
plt.title('Heat Map of Meat Correlations', fontsize = 16,
          color = 'navy', fontdict = dict(weight = 'bold'))
```

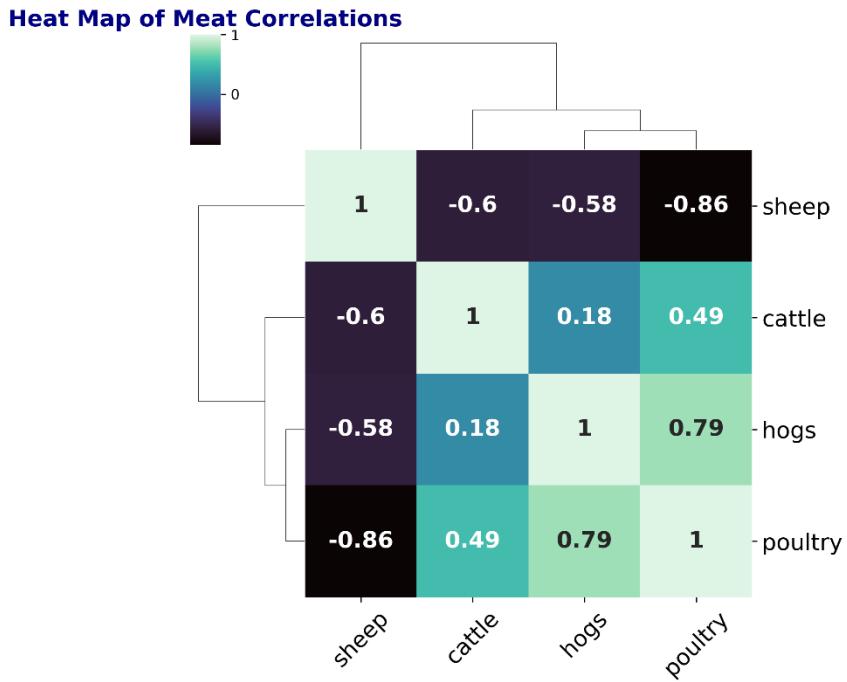


Figure 8-11. Clustered heatmap of the meat times series

Visual Decomposition

Now, we part from the full meat times series and examine the beef series in detail, starting with decomposing to the component level, as seen in

```
In [29]
df1 = meat_df[['date','cattle']]
df1 = df1.set_index ('date')
```

Some distinguishable patterns appear when we plot the data. The time-series has seasonality pattern, such as sales are always low at the beginning of the year and high at the end of the year. There is always an upward trend within any single year with a couple of low months in the mid of the year.

We can also visualize our data using our now familiar time-series **decomposition** that allows us to decompose our time series into three distinct components: trend, seasonality, and noise. Note the autocorrelation of residuals and the seasonality in Figure 8-12.

```
In [30]
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df1, freq = 12)
plt.rcParams.update({'figure.figsize': (10,6)})
decomposition.plot()
```

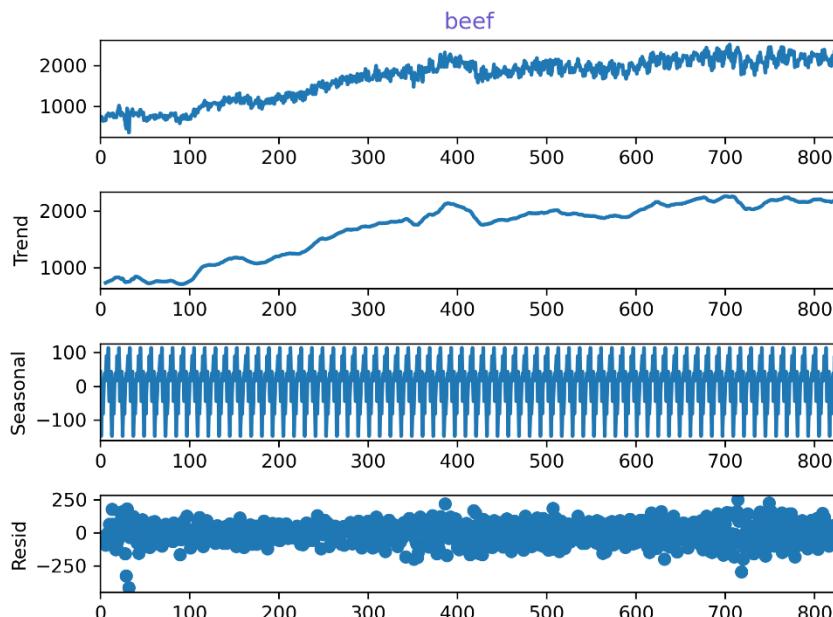


Figure 8-12. Decomposition of the beef times series

The plot above clearly shows that the production of beef is unstable, along with its obvious seasonality. So, we take this opportunity to remedy the beef series, starting averaging the months for a yearly output, and applying two moving averages, one at 10 and one at 20. The moving averages (MA) as we saw in Chapter 6, correct for autocorrelated errors or residuals, which are present in our data.

```
In [31]
df_beef = df[['cattle']]
df_beef.index.name = 'date'
df_beef['avg_beef'] = df_beef.mean(axis = 1)
df_beef = df_beef[['avg_beef']]
# the simple moving average over a period of 10 years
df_beef['SMA_10'] = df_beef.avg_beef.rolling(10,
```

```

    min_periods = 1).mean()
# the simple moving average over a period of 20 year
df_beef['SMA_20'] = df_beef.avg_beef.rolling(20,
    min_periods = 1).mean()

```

Now, we plot the two MAs in Figure 8-13, and note the smoothing effect.

```

In [32]
fig, ax = plt.subplots(linewidth=2, figsize = (7,4), dpi = 300)

# colors for the line plot
colors = ['lightblue', 'red', 'purple']

# line plot - the yearly average air temperature in Barcelona
df_beef.plot(color = colors, linewidth = 1.25, ax = ax)

# modify ticks size
plt.xticks(fontsize = 12)
plt.yticks(fontsize = 12)
plt.legend(labels = ['Beef Production', '10-years SMA',
    '20-years SMA'], fontsize = 14)

# title and labels
plt.title('The yearly average beef production in the US',
    fontsize = 16, color = 'dodgerblue')
plt.xlabel('Years', fontsize = 14, color = 'dodgerblue')
plt.ylabel('Cattle Head Count', fontsize = 14,
    color = 'dodgerblue')

plt.grid(True, color = 'gray', linestyle = ':')
plt.savefig('03_beef_line.png', figsize = (7.4), dpi = 300,
    bbox_inches = 'tight');
plt.show()

```

For the remainder of our analysis, we will need the following libraries.

```

In [33]
from scipy import stats
import statsmodels.api as sm
from statsmodels.graphics.api import qqplot
from pandas import Series, DataFrame, Panel
import warnings
import itertools
from matplotlib import *
warnings.filterwarnings("ignore")

```

```
plt.style.use('fivethirtyeight')
```

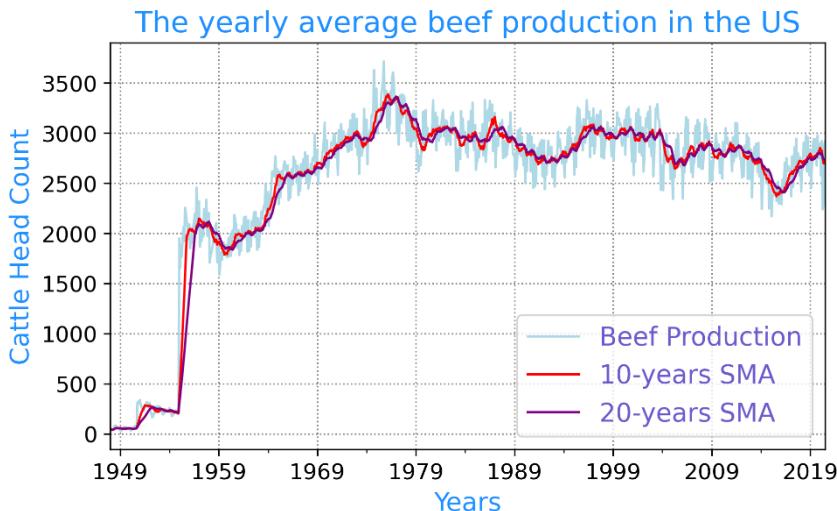


Figure 8-13. Time series plot with two moving average models

We will use monthly averages, that we established above, for this part of the analysis.

```
In [34]
beef_ts = beef.set_index(['date'])
y = beef_ts['beef']

Out [34]
date
1944-01-01    751.0
1944-02-01    713.0
1944-03-01    741.0
1944-04-01    650.0
1944-05-01    681.0
...
2012-07-01    2200.8
2012-08-01    2367.5
2012-09-01    2016.0
2012-10-01    2343.7
2012-11-01    2206.6
Name: beef, Length: 827, dtype: float64
```

Time series forecasting with ARIMA

We are going to apply one of the most commonly used method for time-series forecasting, ARIMA, as we did in Chapter 6. Recall that ARIMA stands for Autoregressive Integrated Moving Average. ARIMA models are denoted with the notation ARIMA(p, d, q). These three parameters account for seasonality, trend, and noise in data; they are the order. We are making our model a seasonal one, due to what we saw back in Figure 8-12. And we are adding a period of 12 months (1 year).

```
In [35]
mod = sm.tsa.statespace.SARIMAX(y,
        order = (1, 1, 1),
        seasonal_order = (1, 1, 0, 12),
        enforce_stationarity = False,
        enforce_invertibility = False)
results = mod.fit()
print(results.summary().tables[1])
```

Out [35]

	coef	std err	z	P> z	[0.025	0.975]
ar.L1	-0.1754	0.043	-4.091	0.000	-0.259	-0.091
ma.L1	-0.6502	0.033	-19.777	0.000	-0.715	-0.586
ar.S.L12	-0.2392	0.029	-8.239	0.000	-0.296	-0.182
sigma2	9325.1986	359.347	25.950	0.000	8620.891	1e+04

No frequency information was provided, so we use the inferred frequency month-start (MS). We approach the analysis by formulating several seasonal ARIMA (SARIMA) models.

```
In [36]
p = d = q = range(0, 2)
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in
                 list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal
ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

Out [36]

```
Examples of parameter combinations for Seasonal ARIMA...
SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
SARIMAX: (0, 0, 1) x (0, 1, 0, 12)
SARIMAX: (0, 1, 0) x (0, 1, 1, 12)
SARIMAX: (0, 1, 0) x (1, 0, 0, 12)
```

This next step is parameter selection for our beef production SARIMA time series model. Our goal here is to use a “grid search” to find the optimal set of parameters that yields the best performance for our beef model.

In [37]

```
for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(y,
                order = param,
                seasonal_order = param_seasonal,
                enforce_stationarity = False,
                enforce_invertibility = False)
            results = mod.fit()
            print('ARIMA{}x{}12 - AIC:{}'.format(param,
                param_seasonal, results.aic))
```

Out [37]

```
except:
    continue
ARIMA(0, 0, 0)x(0, 0, 0, 12)12 - AIC:14689.166432367274
ARIMA(0, 0, 0)x(0, 0, 1, 12)12 - AIC:13465.176599348248
ARIMA(0, 0, 0)x(0, 1, 0, 12)12 - AIC:10051.608773940949
ARIMA(0, 0, 0)x(0, 1, 1, 12)12 - AIC:9906.38668856096
ARIMA(0, 0, 0)x(1, 0, 0, 12)12 - AIC:10053.833767516373
ARIMA(0, 0, 0)x(1, 0, 1, 12)12 - AIC:10039.669508190076
ARIMA(0, 0, 0)x(1, 1, 0, 12)12 - AIC:9918.1844737085
ARIMA(1, 0, 1)x(1, 0, 1, 12)12 - AIC:9554.586195600772
ARIMA(1, 0, 1)x(1, 1, 0, 12)12 - AIC:9606.29322846887
ARIMA(1, 0, 1)x(1, 1, 1, 12)12 - AIC:9374.902205761002
ARIMA(1, 1, 0)x(0, 0, 0, 12)12 - AIC:10241.239691320767
ARIMA(1, 1, 0)x(0, 0, 1, 12)12 - AIC:9813.589416938194
:
ARIMA(1, 1, 0)x(0, 1, 1, 12)12 - AIC:9583.386846076639
ARIMA(1, 1, 0)x(1, 0, 0, 12)12 - AIC:9765.759341807641
ARIMA(1, 1, 0)x(1, 0, 1, 12)12 - AIC:9762.665163880922
ARIMA(1, 1, 0)x(1, 1, 0, 12)12 - AIC:9792.177377399905
ARIMA(1, 1, 0)x(1, 1, 1, 12)12 - AIC:9532.056646787012
ARIMA(1, 1, 1)x(0, 0, 0, 12)12 - AIC:10166.449943373576
```

```

ARIMA(1, 1, 1)x(0, 0, 1, 12)12 - AIC:9727.347376650938
ARIMA(1, 1, 1)x(0, 1, 0, 12)12 - AIC:9770.118057797648
ARIMA(1, 1, 1)x(0, 1, 1, 12)12 - AIC:9393.904613113136
ARIMA(1, 1, 1)x(1, 0, 0, 12)12 - AIC:9626.71424740387
ARIMA(1, 1, 1)x(1, 0, 1, 12)12 - AIC:9534.893296215272
ARIMA(1, 1, 1)x(1, 1, 0, 12)12 - AIC:9603.123626401208
ARIMA(1, 1, 1)x(1, 1, 1, 12)12 - AIC:9359.009975398578

```

The above output suggests that `SARIMAX(1,1,1)x(1,1,0,12)` yields the lowest AIC value of 9359.0099. Therefore, we should consider this to be optimal option. Of course, we are not guaranteed a global optimum from this search, but a local one will do.

```

In [38]
mod = sm.tsa.statespace.SARIMAX(y,
                                  order = (1, 1, 1),
                                  seasonal_order = (1, 1, 1, 12),
                                  enforce_stationarity = False,
                                  enforce_invertibility = False)
results = mod.fit()
print(results.summary().tables[1])

```

Out [38]

	coef	std err	z	P> z	[0.025	0.975]
ar.L1	-0.1312	0.047	-2.811	0.005	-0.223	-0.040
ma.L1	-0.6459	0.034	19.248	0.000	-0.712	-0.580
ar.S.L12	0.2437	0.031	7.769	0.000	0.182	0.305
ma.S.L12	-0.9067	0.019	-47.033	0.000	-0.944	-0.869
sigma2	6822.231	274.164	24.884	0.000	6284.88	7359.58

Model Diagnostics

Now that we have the results for our SARIMA model, we should run (always) model diagnostics to investigate any unusual behavior, which we have done in Figure 8-14. The `plot_diagnostics()` function will get us started.

```

In [39]
results.plot_diagnostics(figsize = (12, 8))
mpl.rcParams['font.size'] = 16
plt.grid(True, color = 'gray', linestyle = ':')
plt.tight_layout()
plt.savefig('03_beef_diag.png', figsize = (12,8), dpi = 300);
plt.show()

```

It is not perfect, but our model diagnostics suggests that the model residuals are near normally distributed.

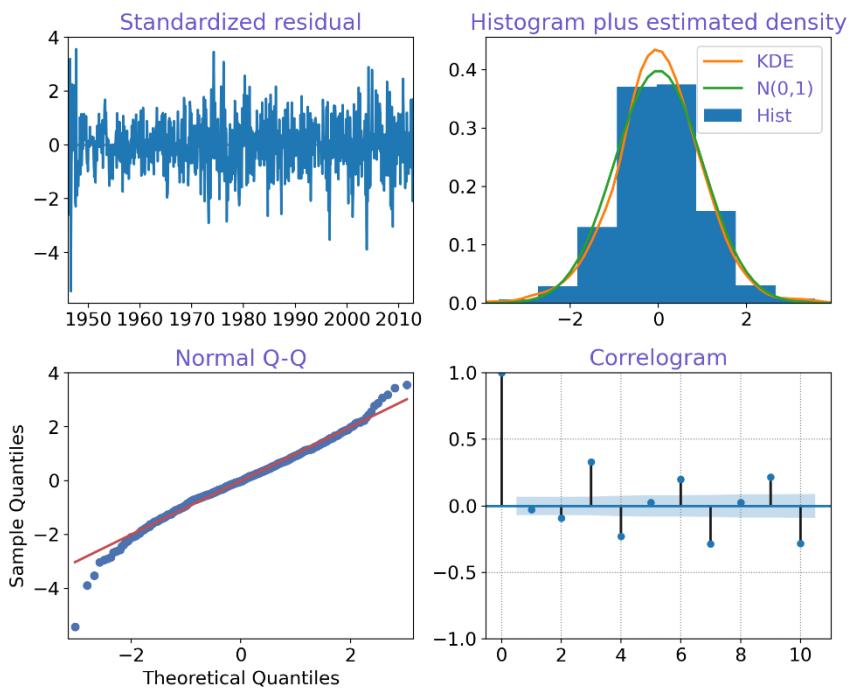


Figure 8-14. Diagnostic plots for the beef production series (cattle)

Validating forecasts

To help us understand the accuracy of our forecasts, we compare predicted sales to real sales of the time series, and we set forecasts to start at 2005-01-01 to the end of the data, as shown in Figure 8-15.

```
In [40]
pred = results.get_prediction(start = pd.to_datetime(
    '2005-01-01'), dynamic = False)
pred_ci = pred.conf_int()
ax = y['1990':].plot(label = 'Observed Values', color = 'gray')
pred.predicted_mean.plot(ax = ax, label = 'One-step ahead
Forecast', alpha = 0.7, figsize = (8, 4), color = 'purple')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
```

```

pred_ci.iloc[:, 1], color = 'k', alpha = 0.2)
ax.set_xlabel('Date', color = 'navy')
ax.set_ylabel('Beef Production', color = 'navy')
ax.set_title('Annual Beef Production in the U.S.',
             fontsize = 18, color = 'navy')
plt.grid(True, color = 'gray', linestyle = ':')
plt.tight_layout()
ax.legend()
plt.show()

```

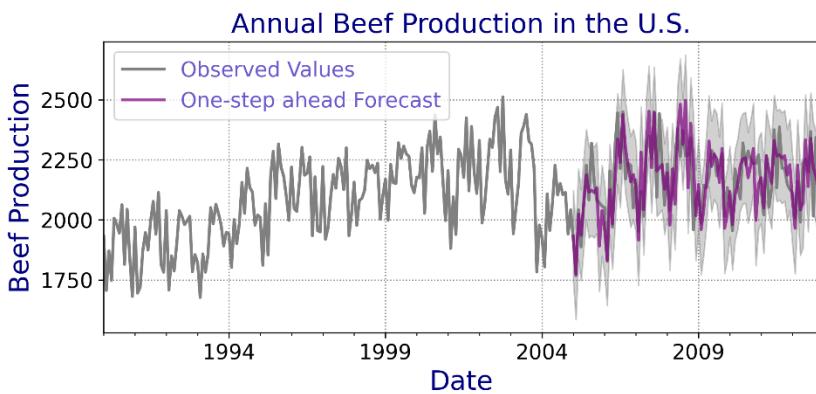


Figure 8-15. Beef production forecast beginning in 2005

The line plot is showing the observed values compared to the rolling forecast predictions. Overall, our forecasts align with the true values very well, showing an upward trend starts from the beginning of the year and captured the seasonality toward the end of the year.

Now, we compute the Mean Squared Error (MSE).

```

In [41]
forecasted = pred.predicted_mean
truth = y['2005-01-01':]
mse = ((forecasted - truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is
{}'.format(round(mse, 2)))

```

The MSE of our forecasts is 7468.64. we can have this printed out sa a statement using the next code snippet.

```

In [42]

```

```
print('The Root Mean Squared Error of our forecasts is  
{}.'.format(round(np.sqrt(mse), 2)))
```

The Root Mean Squared Error of our forecasts is 86.42

We have seen, the MSE of an estimator measures the average of the squares of the errors — that is, the average squared difference between the estimated values and what is estimated. The MSE is a measure of the quality of an estimator — it is always non-negative, and the smaller the MSE, the closer we are to finding the line of best fit.

Recall, Root Mean Square Error (RMSE) tells us that our model was able to forecast the average beef production in the test set within 86.42 of the real production. Our beef production ranges from around 1800 to over 2400. In our opinion, this is a pretty good model so far.

Now we are set to forecast from 2012 to 2020, shown in Figure 8-16.

```
In [43]  
pred_uc = results.get_forecast(steps = 100)  
pred_ci = pred_uc.conf_int()  
ax = y['1990:'].plot(label = 'observed', figsize = (14, 7))  
pred_uc.predicted_mean.plot(ax = ax, label = 'Forecast')  
ax.fill_between(pred_ci.index,  
                 pred_ci.iloc[:, 0],  
                 pred_ci.iloc[:, 1], color = 'k', alpha = 0.25)  
ax.set_xlabel('Date')  
ax.set_ylabel('Beef Production')  
plt.legend()  
plt.show()
```

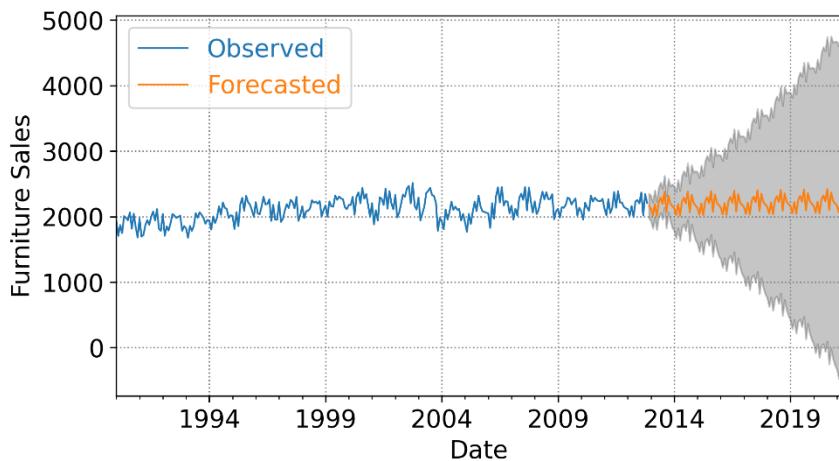


Figure 8-16. Beef production forecast from 2012 to 2020

Our model had to deal with nonstationarity using a differencing parameter, and Figure 8-16 shows a stationary series, compared with the original series shown in Figure 8-17. As we forecast further out into the future, it is natural for us to become less confident in our values. This is reflected by the confidence intervals generated by our model, which grow larger as we move further out into the future.

```
In [44]
# The palette with grey:
cbp1 = ('#999999', '#E69F00', '#56B4E9', '#009E73',
        '#F0E442', '#0072B2', '#D55E00', '#CC79A7')
p = ggplot(aes(x = 'date', y = 'beef'), data = meat)
p + geom_point(color = cbp1[2]) +\
    theme(text = element_text(size=10), axis_text_x =
element_text(angle = 90, hjust = 1)) +\
    gtitle("Beef Production")
```

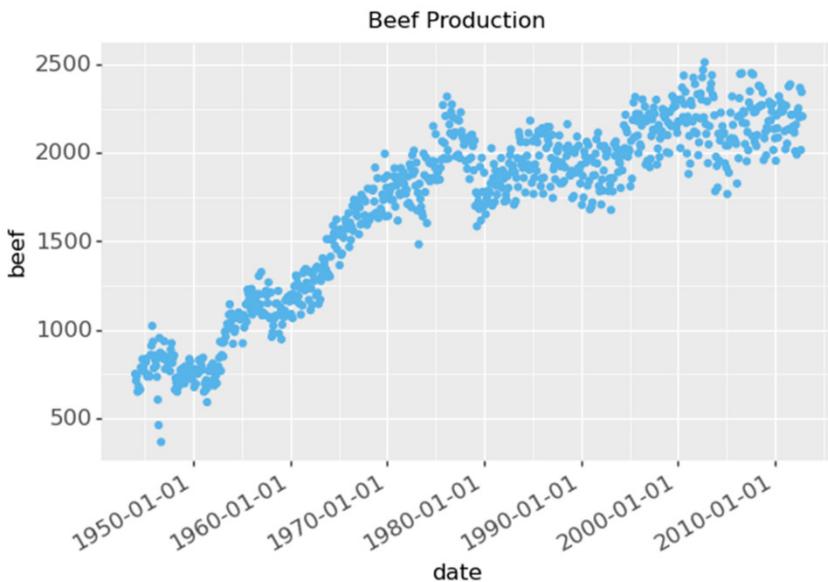


Figure 8-17. Beef production scatterplot

Finally, we look at the model fit summary. We can see validation for stationarity, as the Ljung-Box statistic allows us to reject stationarity. Also, the components of our ARIMA model are all significant

```
In [45]
mod.fit().summary()
```

```
Out [45]
SARIMAX Results
Dep. Variable: beef      No. Observations:      827
Model: SARIMAX(1, 1, 1)x(1, 1, 1, 12)      Log Likelihood:   -4674.505
Date: Thu, 24 Sep 2020      AIC:      9359.010
Time: 18:35:51      BIC:      9382.433
```

```

Sample: 01-01-1944      HQIC     9368.008
        - 11-01-2012
Covariance Type: opg
            coef    std errz      P>|z|   [0.025  0.975]
ar.L1   -0.1312 0.047   -2.811  0.005   -0.223  -0.040
ma.L1   -0.6459 0.034   -19.248 0.000   -0.712  -0.580
ar.S.L12      0.2437 0.031    7.769  0.000    0.182   0.305
ma.S.L12     -0.9067 0.019   -47.033 0.000   -0.944  -0.869
sigma2 6822.2311      274.164 24.884  0.000   6284.880
                7359.582

Ljung-Box (Q):      1207.72 Jarque-Bera (JB):      84.21
Prob(Q):      0.00  Prob(JB):      0.00
Heteroskedasticity (H):      1.45  Skew:      -0.33
Prob(H) (two-sided): 0.00  Kurtosis:      4.44

```

Chapter Review

In this chapter we focus more on ARIMA models for time series analysis. We investigated the ARIMA(p,d,r) and SARIMA(x,p,d,q)(P,D,Q,S) taxonomies given:

x = univariate time series

p = AR order (must be specified)

q = DIFF order (must be specified)

q = MA order (must be specified)

P = SAR order; use only for seasonal models

D = seasonal difference; use only for seasonal models

Q = SMA order; use only for seasonal models

S = seasonal period; use only for seasonal models

We used the Meat dataset to explore various time series visualizations that can help with our modeling, analysis, diagnostics, and predictions. We also discussed the following definitions:

Definition Number	Definition Citation
Definition 8.1: General State Space Model	A general state space model is of the form $\begin{aligned} \mathbf{y}_t &= \boldsymbol{\varphi} \mathbf{y}_{t-1} + \mathbf{d}_t + \boldsymbol{\varepsilon}_t \\ \mathbf{x}_{t+1} &= \boldsymbol{\theta} \mathbf{x}_t + \mathbf{b}_t + \boldsymbol{\eta}_t \end{aligned}$ where \mathbf{y}_t refers to the observation vector at time t , \mathbf{x}_t refers to the (unobserved) state vector at time t , $\boldsymbol{\varphi}$ and $\boldsymbol{\theta}$ are deterministic, and where the irregular components are defined as $\boldsymbol{\varepsilon}_t \sim N(\mathbf{0}, \boldsymbol{\varphi}_t)$, $\boldsymbol{\eta}_t \sim N(\mathbf{0}, \boldsymbol{\theta}_t)$

Review Exercises

1. **Seattle Burke Gilman Trail.** (Note: this is not a trivial problem.) Using sensors, data is collected continuously for this Seattle trail. These sensors count both people riding bikes and pedestrians. Separate volumes are tallied for each travel mode. Wires in a diamond formation in the concrete detect bikes and an infrared sensor mounted on a wooden post detects pedestrians. The counters also capture the direction of travel for both bikes and pedestrians. Data from January 1, 2014 to December 31, 2019 is posted in an Excel workbook: burke-gilman-trail-north-of-ne-70th-st-bike-and-ped-counter.csv. We have taken the data and simplified it for this exercise and it can be downloaded from our GitHub data directory (<https://github.com/stricje1/Data>, file name is "bg_trail2.csv"). The data contains five measures: Ped South, Ped North, Bike North, Bike South, and a total count. You are to choose one of these variables and perform a complete time series analysis as follows:
 - a. Load and rename the columns of the dataframe using pandas' functions so that there are no white spaces, e.g., from "Ped North" to "Ped_North".
 - b. Create a times series using "date" as the index, with the date format changed from "%m,%d,%Y" to dates with the format "%Y,%m,%d" (this should give you a times series with 52584 rows \times 7 columns).
 - c. Using pandas' functions, delete the "old" date column and the BGT_Total column.
 - d. Transform the data to a form where the month totals (sums) are tallied. (Hint: this would render 72 rows \times 4 columns instead of 52584 rows \times 6 columns. Try `ts.groupby(ts.Month).sum()` or something similar.)
 - e. Plot one time series (it should look like one of the measures from Figure 8-18).
 - f. Perform decomposition and plot the outcome
 - g. Explain any issues, like seasonality, trend, stationarity, etc.
 - h. Determine the best parameter p, d, q, P, D, Q, S to use for your time series x .
 - i. Fit the selected ARIMA (SARIMA) model for your time series x .

- j. Diagnose your model's goodness-of-fit.
- k. Develop a 3-month forecast using your fitted ARIMA (ARIMA) model.
- l. Repeat e, f, and g, using an auto ARIMA model using code similar to In[37] in this chapter.
- m. Perform the analysis in a Jupyter Notebook and annotate your code. Use markdown when providing explanations.

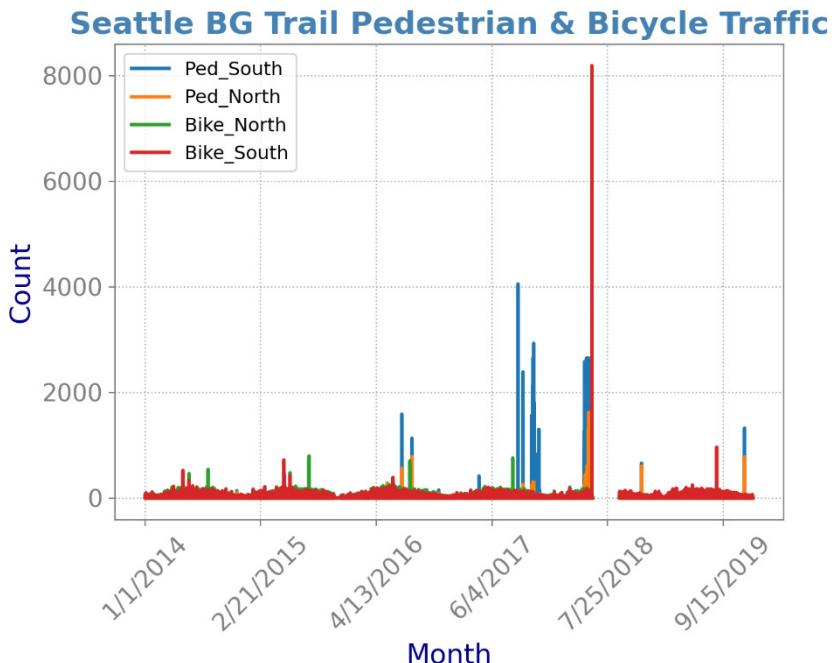


Figure 8-18. Seattle Burke Gilman Trail pedestrian and bicycle data

2. Plot a boxplot of the series (Ped_South, Ped_North, Bike_North, and Bike_South) in Exercise 1. (Hint: try using a log scale on the y-axis).
 - a. Based on the plot, could you hypothesize equal means?
 - b. Based on the plot, could you hypothesize equal variance?
3. Construct a heat map of the correlation matrix with Ped_South, Ped_North, Bike_North, and Bike_South.
 - a. What conclusions can you draw about correlation between variables?
 - b. Do your conclusions seem intuitive, why or why not?

Chapter 9 – Structural Models in R

Introduction

In Chapter 2, we discussed the trend, seasonal and cyclical components of time series data. Here, we will discuss an effective method for performing analysis of this data.

The **Unobserved Components Model (UCM)** is special cases of more general and powerful tool in time series called State Space Models having an observation equation, which relates the dependent series to an unobserved state vector, and a state equation describing the evolution of the state vector over time (Petris & Petrone, 2011). UCMs are also called structural models in time series. For a detailed discussion of State Space Models refer (Harvey, 1989) or (Helske, 2015). It is fair to say that the UCMs capture the versatility of the ARIMA models while possessing the interpretability of the smoothing models.

Definition 9.1. *Unobserved Components Model (UCM) performs a time series decomposition into components such as trend, seasonal, cycle, and the regression effects due to predictor series. It can be represented as follows:*

$$y_t = \mu_t + \gamma_t + \varphi_t + AR_t + \sum_{j=1}^m (\beta_j x_{jt} + \epsilon_t)$$
$$\epsilon_t \sim i.i.d. N(0, \sigma_\epsilon^2)$$

The **components** μ_t , γ_t , and φ_t represent the trend, seasonal, and cyclical components, respectively; AR_t represents an unobserved autoregressive component; $\sum_{j=1}^m (\beta_j x_{jt} + e_t)$ gives the contribution of regression variables with fixed or time varying regression coefficients. The irregular or error terms ϵ_t are assumed to be independently and identically distributed as **Standard Normal distributions**.

Trend

For simplicity, **trend** is the natural tendency of the series to increase or decrease or remain constant over a period, in the absence of any other influencing variable.

UCM can model trend in two ways. The first is the **random walk model** implying that trend remains roughly constant over the period of the series. The second relies on locally linear trend having an upward or downward slope.

Cycle

Cycle in a time series data exists when the data exhibit rises and falls that are not of fixed period. The duration of these fluctuations is usually, at least, two years.

UCM provides two basic ways of modeling the unobserved cyclical component, φ_t : a deterministic (non-stochastic) trigonometric cycle (or cycles) and a stochastic trigonometric cycle.

Seasonality

A **seasonal** pattern exists when there is a consistent pattern of variation influenced by seasonal factors (e.g., the quarter of the year, or day of the week, etc.).

UCM provides two ways to deal with the unobserved seasonal component: a Stochastic Dummy Variable Seasonal model and a Deterministic Dummy Variable Seasonal model

Autoregressive Component

Rather than modeling the cyclical nature of a time series by either the deterministic cyclical model or the stochastic cyclical model, one can use the rather straight-forward autoregressive component.

Definition 9.2. The autoregressive component is specified by:

$$AR_t = \rho AR_{t-1} + \nu_t, \nu_t \sim i.i.id. N(0, \sigma_\varepsilon^2),$$

where the **unobserved autoregressive component** AR_t follows a first-order autoregression with $-1 < \rho < 1$.

This autoregression, despite its simplicity, can capture many of the movements in time series data that represent business cycle inertia and that are present in many business and economic time series. Definition 9.2 reflects the same meaning as Definition 2.6.

For a detailed discussion of all the above three factors see Chapter 2, where we defined **trend** (Definition 2.2), **seasonality** (Definition 2.3), **cycle** (Definition 2.4), and the **irregular component** (Definition 2.6).

rucm package

The *rucm* package is written keeping in mind the easier specification of UCM in SAS using PROC UCM. *rucm* provides a wrapper function called *ucm* containing arguments specifying the formula for predictor variables, and other decomposition components such as level, slope, season, cycle, etc., to run UCM. *Ucm()* can also handle cases where we want to fix the variance of any of the above decomposition components (Chowdhury, 2015). To install *rucm*:

```
install.packages("rucm")
library(rucm)
```

For help and the list of values that are returned see the `help(package = rucm)` or `?rucm`.

Modeling the Flow of the Nile River

Here we work with the *Nile* dataset which comes along with the *datasets* package, measures the annual flow of the river Nile taken at the Dongola measurement station just upstream from Ashwan, in south Egypt, between 1871 and 1984.

In this example, this series is modeled using an unobserved component model called the basic structural model (BSM). The BSM models a time series as a sum of three stochastic components: a trend component μ_t (comprise of the `level` and `slope` in the `ucm()` function), a seasonal component γ_t , and random error ϵ_t (SAS-Institute, 2010). Formally, a BSM for a response series y_t can be described as $y_t = \mu_t + \gamma_t + \epsilon_t$. Each of the stochastic components in the model is modeled separately. The random error ϵ_t , also called the **irregular component**, is modeled simply as a sequence of independent, identically distributed (i.i.d.) zero-mean Gaussian random variables. The trend and the seasonal components can be modeled in a few different ways. The model for trend used here is called a locally linear time trend. This trend model can be written as follows:

$$\mu_t = \mu_{t-1} + \beta_{t-1} + \eta_t, \quad \eta_t \sim i.i.d. \mathcal{N}(0, \sigma_\eta^2)$$

$$\beta_t = \beta_{t-1} + \xi_t, \quad \xi_t \sim i.i.d. \mathcal{N}(0, \sigma_\xi^2)$$

These equations specify a trend where the level μ_t as well as the slope β_t is allowed to vary over time. This variation in slope and level is governed by the variances of the disturbance terms η_t and ξ_t in their respective equations. Some interesting special cases of this model arise when you manipulate these disturbance variances. For example, if the variance of ξ_t is zero, the slope will be constant (equal to β_0); if the variance of η_t is also zero, μ_t will be a deterministic trend given by the line $\mu_0 + \beta_0 t$.

To model the annual flow of *Nile River*, we use the following code, where the trend is represented by the level and slope, and using a naïve estimate of 12 months for seasonality.

```
modelNile <- ucm(formula = Nile ~ 0, data = Nile, level = TRUE,
                  slope = TRUE, season = TRUE, season.length = 12)
modelNile #Printing method for class ucm
```

```
Call:
ucm(formula = Nile ~ 0, data = Nile, level = TRUE, slope = TRUE,
     season = TRUE, season.length = 12)
Parameter estimates:
NULL
Estimated variance:
Irregular_Variance      Level_Variance      Slope_Variance
        14112.1510          1986.6852          0.0006
Season_Variance
        0.0233
```

We now plot the observed and modeled flows in Figure 9-1.

```
par(col.lab = 'dodgerblue3', font.lab = 2, cex.axis = .85)
plot(Nile, col = 'navy', xlab = "Months", ylab = "Monthly Flow",
     lwd = 2, col.lab = 'dodgerblue3', cex.lab = 1,
     panel.first = grid(10, lty = 3, lwd = 1,
     col = 'lightblue'))
title(main = "Nile overflow data 1871-1984",
      cex.main = 1.5, font.main= 4, col.main = 'dodgerblue4',
      cex.sub = 0.75, font.sub = 3, col.sub = 'dodgerblue2')
lines(modelNile$s.level, col = 'purple1', lwd = 2)
legend("bottomleft", legend = c("Observed flow","S_level"),
       col = c('navy','purple1'), lwd = 3, lty = 1,
       x.intersp = 2, y.intersp = 2, bty = 'n')
```

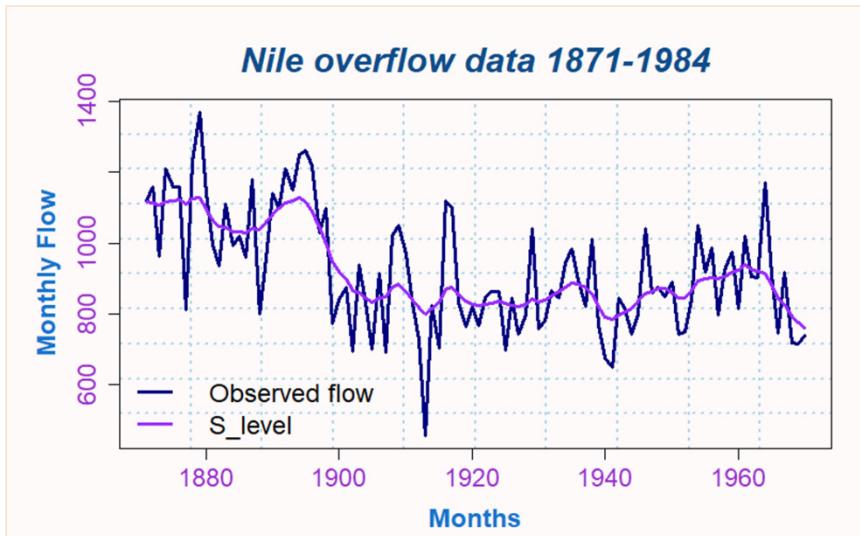


Figure 9-1. Observed and fitted flows of the Nile

In general, the formula argument in the `ucm()` function takes an argument of the form `as.formula` in R. For multivariate UCMs, the right-hand-side (rhs) of the formula should contain the independent variables. If the model is univariate, we write a 0 in the rhs of the formula specification. Slope, seasonality, and cyclicity can be included by using `slope = TRUE`, `season = TRUE`, `cycle = TRUE`, specifying the seasonal and cyclical lengths of the series in the arguments `season.length` and `cycle.period`, respectively.

The function `ucm()` returns an object of class `ucm` having the estimate of predictors, estimated variances, time series of unobserved components (level, slope, whatever is included), and time series of the variances of these components.

Forecasting with UCM

To forecast the time series, we use the `predict()` function supplying the model name and number of periods to forecast in `n.ahead`. Unlike other time series methods, we have shown, `ucm()` requires a formula for modeling. When we consider no external effects on the series, the formula is: `Time Series ~ 0`, where the '`~`' represents equality.

```
modelNile <- ucm(formula = Nile ~ 0, data = Nile, level = TRUE,  
slope = TRUE)  
predict(modelNile$model, n.ahead = 12) # Forecasting
```

```
Time Series:  
Start = 1971  
End = 1982  
Frequency = 1  
[1] 779.5699 776.1597 772.7496 769.3395 765.9293 762.5192  
[7] 759.1091 755.6989 752.2888 748.8786 745.4685 742.0584
```

Alternative Implementation

Since, UCMs are structural models, the `StructTS()` function is an alternative for implementing this particular UCM. `StructTS` is a part of base R. The code follows (recalled that we implemented our previous model with `ucm()` as a BSM:

```
fit <- StructTS(Nile, type = c("level", "trend", "BSM"))  
fit
```

```
Call:  
StructTS(x = Nile, type = c("level", "trend", "BSM"))  
Variances:  
  level  epsilon  
 1469    15099
```

Plotting with the `tsdiag()` function provides residuals scaled by the estimate of their (individual) variance, and use the Ljung-Box version of the portmanteau test. A **portmanteau test** is a type of statistical hypothesis test in which the null hypothesis is well specified, but the alternative hypothesis is more loosely specified. Tests constructed in this context can have the property of being at least moderately powerful against a wide range of departures from the null hypothesis. In time series analysis, a portmanteau test is available for testing for autocorrelation in the residuals of a model: it tests whether any of a group of autocorrelations of the residual time series are different from zero. Figure 9-2 shows the fitted model. We created the plot using plotting parameters, `par()`, including a scaling factor (`cex`), label and axes colors, as well as background color. We also added tick marks using `labels=axis()` and a sequence of years from 1870 to 1970 with 20-year intervals. Figure 9-3 shows three diagnostic plots for our time

series, and they look pretty good for this fit. Once again, we plotted them with aesthetic plot parameters.

We plot the time series and fitted model again to demonstrate the use of plotting parameters, `par()`. Parameters can be set by specifying them as arguments to `par` in tag = value form, or by passing them as a list of tagged values, such as `cex = 1.25` (<tag> = <value>). Recall that `cex` is a numerical value giving the amount by which plotting text and symbols should be magnified relative to the default. Using `cex.<parameter>`, like `cex.lab`, only changes the size of the parameter, lab or label. Note that a value in <tag> = <value>, value can be a string, like color. In the code below, `col.axis` is the color of the axes, such as '`brown4`'. Also, `bg` represents background color and `fg` is foreground color. Also note that `par()` can be a list

```
# Set plotting parameters
par(cex = 1.25)
par(col.lab = 'chocolate3')
par(col.axis = 'brown4')
par(bg = 'cornsilk', fg = 'orange')

plot(Nile, col = 'deepskyblue', lwd=3)
lines(fitted(fit), col = 'blueviolet', lwd=3)
  title(main="Nile Time Series", cex.main=1.25,
        font.main=2, col.main='goldenrod4', col.lab ='darkblue')
  labels=axis(1, seq(1870,1970,20))
```

When using `par()`, use cautiously as it set global plotting parameters. Always define your starting parameters, with `opar <- par()`, which allows us to reset the initial parameters using, `par(opar)`. Here, we plot diagnostics for our model in Figure 9-3. `tsdiag()` is a generic function. It will generally plot the residuals, often standardized, the autocorrelation function of the residuals, and the p-values of a *portmanteau test* for all lags up to `gof.lag`.

```
par(cex = 2, col = 'blue', pch = 19, lwd = 1.5,
    col.lab = 'darkred', col.axis = 'darkred',
    bg = 'white', fg = 'red')
tsdiag(fit)
```

Nile overflow data 1871-1984

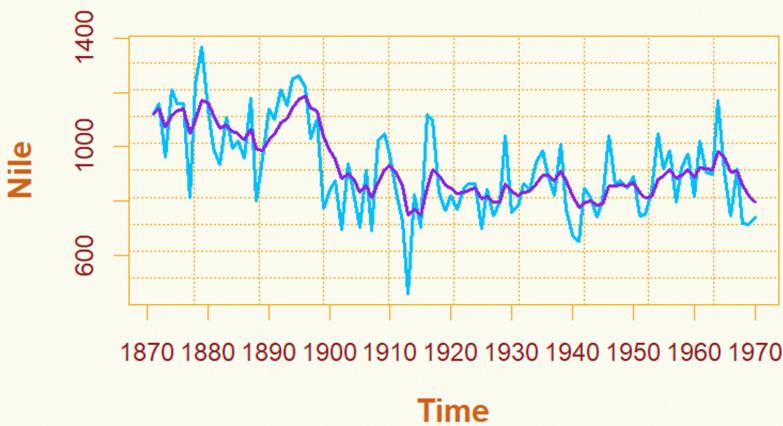


Figure 9-2. Fitted time series for the Nile flow data

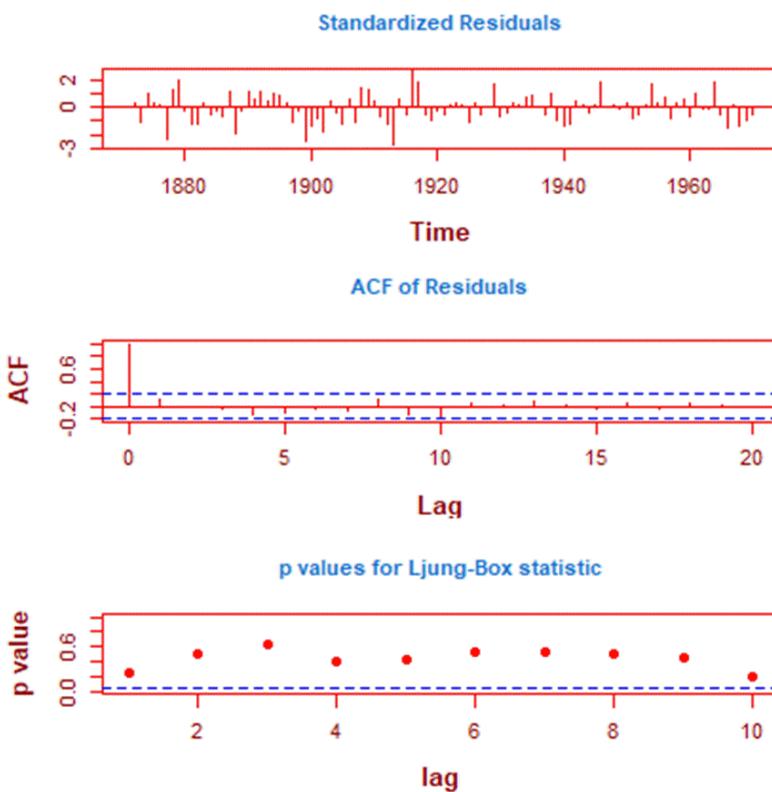


Figure 9-3. Nile fitted time series diagnostic plots

Figure 9-3 Shows the p-values for the test in the bottom plot, as dots at each lag for goodness-of-fit (GOF) analysis. We have used the top two plots in previous chapters, for model diagnostics.

In the next code snippet, we generate a forecast using our fitted model and plot the series, fitted model, and forecast in Figure 9-4.

```
plot(forecast(fit, level = c(50, 90), h = 10, fan = TRUE,
              robust=TRUE),
      xlim = c(1950, 1980), lwd = 2, col = 'red2', main = "",
      xlab = "Year", ylab = "Annual Flow")
lines(fit$fitted, lwd = 2, col = 'blue2')
title(main = "Flow of the Nile River", cex.main = 1.25,
      font.main = 2, col.main = 'darkblue')
legend("bottomleft", legend = c("Observed flow", "fitted flow"),
       col = c('red2','blue'), lty = 1, box.lwd = 2, lwd = 2,
       x.intersp = 2, y.intersp = 2, box.col = 'purple',
       bty = 'y', inset = .02)
```

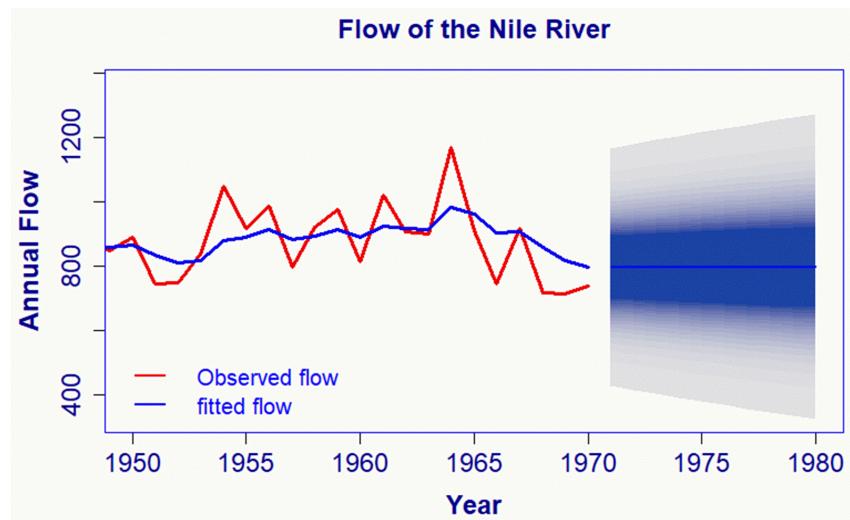


Figure 9-4. Forecast of Nile River flow from 1970 to 1980

Now, we get a summary of the fitted model using `summary()`.

```
summary(Nile.for)
```

```
Forecast method: Local level structural model
```

Model Information:

Call:

```
StructTS(x = Nile, type = c("level", "trend", "BSM"))
```

Variances:

level	epsilon
1469	15099

Error measures:

	ME	RMSE	MAE	MPE	MAPE
Train	-0.0832397	0.9949873	0.7814857	-0.02491327	0.0901223
	MASE	ACF1			
Train	0.005864697	0.1153388			

Forecasts:

	Point Forecast	Lo 80	Hi 80	Lo 95	Hi 95
1971	798.3682	614.4315	982.3048	517.0613	1079.675
1972	798.3682	607.9854	988.7509	507.2029	1089.533
1973	798.3682	601.7506	994.9858	497.6675	1099.069
1974	798.3682	595.7074	1001.0289	488.4253	1108.311
1975	798.3682	589.8394	1006.8969	479.4509	1117.285
1976	798.3682	584.1320	1012.6043	470.7222	1126.014
1977	798.3682	578.5727	1018.1636	462.2200	1134.516
1978	798.3682	573.1507	1023.5857	453.9277	1142.809
1979	798.3682	567.8561	1028.8802	445.8304	1150.906
1980	798.3682	562.6804	1034.0559	437.9149	1158.821
1981	798.3682	557.6160	1039.1203	430.1695	1166.567
1982	798.3682	552.6560	1044.0803	422.5838	1174.153

Next, we plot the filtered and smoothed estimates in Figure 9-5. Since we set our plotting parameters using `par()`, we reset them here.

```
# Reset initial plot parameters
par(opar)
# Plot Parameters
par(col = 'blue', font.lab = 2, col.lab = 'darkblue',
    col.axis = 'darkblue'

# Plot series, fitted, and tsSmooth
plot(Nile, type = "o", col = 'purple', xlab = "Year",
      ylab = "Annual Flow")
lines(fitted(fit), lty = "dashed", col = 'red2', lwd = 2)
lines(tsSmooth(fit), lty = "dotted", col = 'blue', lwd = 2)
title(main="Flow of the Nile River", cex.main=1.25,
      font.main = 2, col.main = 'darkblue')
legend("bottomleft",
       legend = c("Fitted flow", "tsSmooth", "Series"),
       col = c('red2','blue','purple'), lty = c(2,3,1),
```

```
box.lwd = 2, lwd = 2, x.intersp = 1, y.intersp = 1,  
box.col = 'purple', bty = 'y', inset = .02)
```

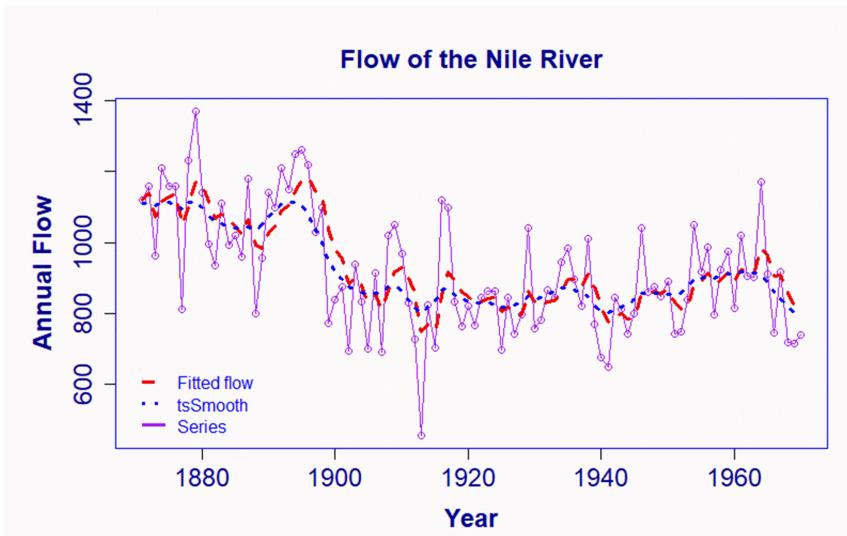


Figure 9-5. Filtered and smoothed estimate of the Nile River flow

Model Comparison

For comparison, we look at the forecast error measures. Note, however, that the `ucm()` function in R does not produce the desired output, unlike other time series function in R. So, we have to perform some calculations.

Mean absolute error: The mean absolute error (MAE) value is computed as the average absolute error value. If this value is 0 (zero), the fit (forecast) is perfect. As compared to the mean squared error value, this measure of fit will "de-emphasize" outliers, that is, unique or rare large error values will affect the MAE less than the MSE value.

Mean Forecast Error (Bias). The mean forecast error (MFE) is the average error in the observations. A large positive MFE means that the forecast is undershooting the actual observations, and a large negative MFE means the forecast is overshooting the actual observations. A value near zero is ideal.

Mean Absolute Percent Error (Bias). The mean absolute percent error (MAPE) is:

$$MAPE = \frac{1}{n} \sum_{t=1}^n \left| \frac{Actual_t - Fitted_t}{Actuals_t} \right|$$

MAPE is the best measure of forecast error among these three, so we will compare MAPE.

```
> mymod <- ucm(formula = Nile ~ 0, data = myts, level = TRUE,
+ slope = TRUE)
> mypred <- predict(mymod$model, n.ahead = 12)
> summary(myts)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	813.0	994.5	1120.0	1092.0	1160.0	1370.0

```
mape<-(1/15)*(sum((myts-mypred)/myts))
mape
```

```
[1] 0.2387985
```

The `ucm()` MAPE is 0.2387985 versus the `StructTS()` MAPE of 0.0901223, so based on MAPE alone it would seem that the `StructTS()` model is better in this instance.

Just to ensure that this is a correct assessment, let's look at the Root Mean Square Error (RMSE), which is the square root of the MSE and the Mean Absolute Error.

```
mae <- (1/15)*abs((sum((mypred-myts))))
mae
```

```
[1] 275.082
```

```
rmse<-sqrt((1/15)*(sum((myts-mypred)^2)))
rmse
```

```
[1] 333.5453
```

Summarizing the results, we have:

Measure	<code>ucm()</code> model	<code>StructTS()</code> model
MAPE	0.2387985	0.0901223
MAE	275.082	0.7814857
RMSE	333.5453	0.9949873

Chapter Review

In this chapter, we examined structural models using R, including the unobserved components model (UCM). This provided us with a review of trend, cycle, and seasonality, as well as autoregressive and moving average components. We used UCM to perform time series analysis on annual flood water volume of the Nile River. We also looked at an alternative for UCM in R. We also discussed these two definitions.

Definition Number	Definition Citation
Definition 9.1: Unobserved Components Model (UCM)	<p><i>Unobserved Components Model (UCM) performs a time series decomposition into components such as trend, seasonal, cycle, and the regression effects due to predictor series. It can be represented as follows:</i></p> $y_t = \mu_t + \gamma_t + \varphi_t + AR_t + \sum_{j=1}^m (\beta_j x_{jt} + \epsilon_t)$ $\epsilon_t \sim i.i.d. N(0, \sigma_\epsilon^2)$ <p><i>The components μ_t, γ_t, and φ_t represent the trend, seasonal, and cyclical components, respectively; AR_t represents an unobserved autoregressive component; $\sum_{j=1}^m (\beta_j x_{jt} + \epsilon_t)$ gives the contribution of regression variables with fixed or time varying regression coefficients. The irregular or error terms ϵ_t are assumed to be independently and identically distributed as Standard Normal distributions.</i></p>
Definition 9.2: UCM Autoregressive component	<p><i>The autoregressive component is specified by:</i></p> $AR_t = \rho AR_{t-1} + v_t, v_t \sim i.i.d. N(0, \sigma_v^2),$ <p><i>where the unobserved autoregressive component AR_t follows a first-order autoregression with $-1 < \rho < 1$.</i></p>

Review Exercises

1. Use the “elec” data set from the fma-Package to:
 - a. Decompose the times series and analyze the components
 - b. Fit a UCM model to the data (you may have to manipulate the data to get it in the proper format)
 - c. Analyze the model results and perform diagnostics
 - d. Generate forecast from the UCM model
2. Repeat Exercise 1 but use the function `StructTS()` to build a structural model.
3. Repeat Exercise 1 but perform a log transformation on the data before modeling
4. Use the “advsales” data set from the fma-Package to:
 - a. Set up the variable “sales” as a time series
 - b. Decompose the times series and analyze the components
 - c. Fit a UCM model to the data (you may have to manipulate the data to get it in the proper format)
 - d. Analyze the model results and perform diagnostics
 - e. Generate forecast from the UCM model

Chapter 10 – Advanced Time Series Topics

Time Series Intervention

This example illustrates an analysis of the President George W. Bush's job approval from January 2001 through Sep 2004 with disposable income excluded from the statistical model, depicted in Figure 10-1. Presidents with a job approval rating of less than 50 percent are unlikely to be re-elected. During June 2001, Bush's job approval rating averaged 47 percent in five major polls (Eichenberg & Stoll, 2008).

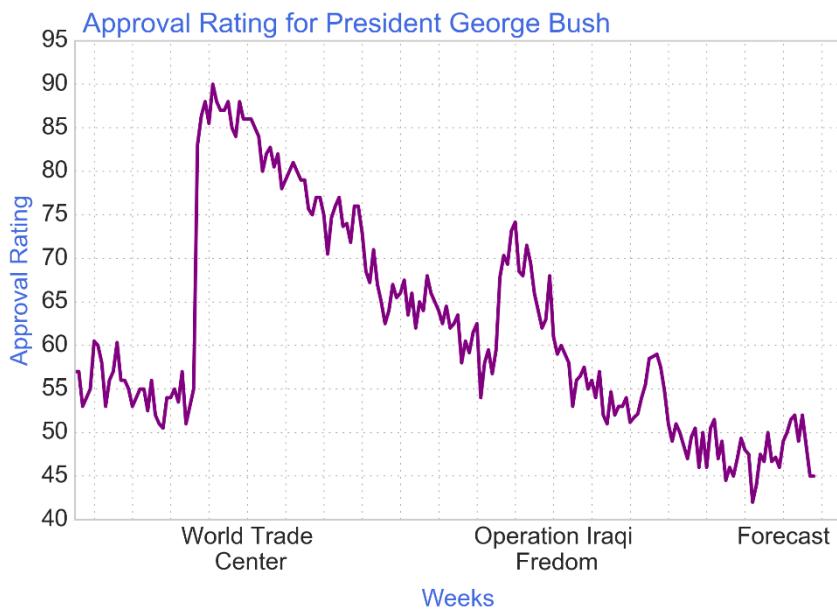


Figure 10-1. Presidential approval rating time series plot

Observations

Viewing Figure 10-2, we note immediately the unusual, roller-coaster characteristic of the Bush Presidency, divided into several distinct phases. A first phase, lasting through September 10, 2001, resembles the course of many other presidencies: approval begins to decline soon after the inauguration, although it may respond to short-term events and the presidents' policy successes and failures, especially on economic issues. The second phase began on September 11, 2001. In the aftermath of the

attacks in New York and Washington, Bush's approval surged to the highest level ever recorded, and well into 2002, it remained above the average for his entire term. This was the largest and longest "rally" in presidential approval that has ever been recorded. A third phase of the Bush Presidency began with the war against Iraq when his slowly declining approval from September 11th surged once again after the war began (although not to the levels of fall 2001). After the end of major hostilities on May 1, 2003, approval began a long decline to the 47 percent level cited immediately above.

Since we are only looking at job approval rating, we will exclude the third phase, for we do not know how much of the decline in the post-Iraq war phase was due to the unpopularity of the war and occupation, and in particular to the casualties that were been suffered. Invading a country is quite different than an emergency response to an attack on our country on 9/11.

The President's Job Approval Rating takes account of citizen policy evaluations beyond the single issue of the War with Iraq. We consider here the external factors:

- (1) responsive attack on terrorist in Afghanistan
- (2) World Trade Center (WTC) terrorist attack
- (3) Operation Iraqi Freedom (Iraq war)
- (4) the battles with Al-Qaida in Kabul, Afghanistan

We will also experiment with a dummy variable reflecting a change on September 9, 2001.

Time Series Analysis

During this analysis, we will use the Applied Statistical Time Series Analysis ([astsa](#)) package. [Statsmodels](#) library provides the `SARIMX()` function. The implementation is called SARIMAX instead of SARIMA because the "X" addition to the method name means that the implementation also supports exogenous variables. In R, we call on `sarima(x,p,d,q,P,D,Q,S)` to implement a model. In Python, we use an order parameter with SARIMAX, like `SARIMAX(data, order = (p,d,q),`

seasonal order = (P,D,Q), exog = expg) [period]. The results provided are the parameter estimates, standard errors, AIC, BIC, and residual diagnostics, etc.

The parameters for `SARIMAX()` that we use the most are defined as:

`endog` : (array_like) the observed time-series process y

`exog` : (array_like, optional) array of exogenous regressors

`order`: (iterable, optional) the (p,d,q) order of the model; default is an AR(1) model: (1,0,0)

`p` = the number of auto-regressors (AR) parameters

`d` = the number of differences

`q` = the number of moving average (MA) parameters.

`seasonal_order`: (iterable, optional) the (P,D,Q,s) order of the seasonal component of the model

`P` = the number of AR parameters

`D` = the number of differences

`Q` = the number of MA parameters

`s` = the periodicity (number of periods in season), often it is 4 for quarterly data or 12 for monthly data. Default is no seasonal effect.

`Trend` : (str{'n','c','t','ct'} or iterable, optional) parameter controlling the deterministic trend polynomial $A(t)$. Can be specified as a string where

'`c`' = indicates a constant (i.e. a degree zero component of the trend polynomial)

'`t`' = indicates a linear trend with time

'`ct`' = is both; can also be specified as an iterable defining the non-zero polynomial exponents to include, in increasing order. For example, [1,1,0,1] denotes $a + bt + ct^3$; default no trend component.

External Predictor Variables

As we create a model to fit the Presidential approval ratings, we include external regressors or predictors: disposable personal income growth per capita (RDPI) and World Trade Center (WTC) response. Also, we build the model using the `sarima()` function. It fits ARIMA models (including improved diagnostics) in a short command and offers additional diagnostic plots (see Figure 12-5). In this model, we will add a difference to achieve stationarity. We will also make it a seasonal model. The first three numeric values (p, d, q) in `sarima(x, p, d, q, P, D, Q, S, xreg)` are the usual AR, DIFF, and MA components. The second set of values (P, D, Q) are seasonal AR, DIFF, and MA components. S represents the seasonal period.

Intervention

When we analyzed the Unsecured Consumer Loan time series data in Chapter 9, we saw the effects of a policy change. Here we see another set of data where a “jump” occurs. In this case, it is due to the approval ratings based on the immediate response to events on September 11, 2001, the WTC attack.

In general, by intervention, we mean a change to a procedure, or law, or policy, etc. that is intended to change the values of the series x_t . We can look at President Bush’s actions on 9/11 (and thereafter) as a policy change. Here we introduce a dummy variable into the data (I added it to the .CSV file), where the value is zero before 9/11 and one thereafter. The effect is essentially that anything in Bush’s administration affecting job approval status is irrelevant—a policy change has so effected present and future ratings that old ones are dominated.

Predictor Contribution

The predictors we use to build the models on this chapter include the WTC attack, as we have mentioned, and a dummy variable for the event/date September 11, 2001. In general, an intervention is a change to a procedure, or law, or policy, etc. that is intended to change the values of the series, or an event that was not intended to have the desired effect. The **intervention** in this time series is the Executive Response to the terrorist attacks on the WTC.

Definition 10.1. Suppose that at time $= T$ (where T may or may not be known), there has been an **intervention** to a time series. Suppose that the ARIMA model for (the observed series) with no intervention is

$$y_t - \mu = \frac{\theta(B)}{\varphi(B)} e_t$$

where $\theta(B)$ is the usual moving average (MA) component, $\varphi(B)$ is the usual autoregressive (AR) term, e_t is error term.

Let z = the amount of change at time t that is attributable to the intervention. By definition, $z_t = 0$ before time T (time of the intervention). The value of z_t may or may not be 0 after time T . Then the overall model, including the **intervention effect**, may be written as

$$y_t - \mu = z_t + \frac{\theta(B)}{\varphi(B)} e_t$$

To help understand the actual value of the external predictor variables or exogenous regressors, WTC, we construct a contribution chart. The basic idea of a contribution chart is to simply plot the model's conditional forecasts where certain independent variables take on their true value while others are held equal to their sample means. This allows the modeler to understand the substantive impact that collections of variables "contribute" to the explanatory power of the model:

$$y_t = \beta x_t + \phi y_{t-1} + e \varepsilon_t$$

where y_t is the dependent variable, y_{t-1} is the dependent lagged variable, is ϕ the coefficient of the dependent lag, x_t is the independent variable with coefficient β , and ε_t is the error term.

The simplest example of a contribution chart involves one explanatory variable with one the lagged dependent variable (LDV). In this instance, the lagged dependent variable functions as a feedback loop, and the coefficient on the LDV defines the magnitude of this amplification. So, a naïve way of constructing the contribution chart is to simply plot the contemporaneous contribution of the independent variable, magnified by the geometric factor implied by the intervention effect.

We want to estimate how much the intervention has changed the series (if at all). For example, suppose that a region has instituted a new maximum speed limit on its highways and wants to learn how much the new limit has affected accident rates. This is the most straightforward approach but assumes all amplification is instantaneous, which is not the case.

Contribution Chart

The contribution of WTC toward the Presidential Job Approval Ratings is shown in Figure 10-2. This intervention event clearly had an impact on President Bush's job approval ratings. The contribution chart depicts the effect of WTC on the same scale as the approval rating, by implementing the *intervention effect equation*.

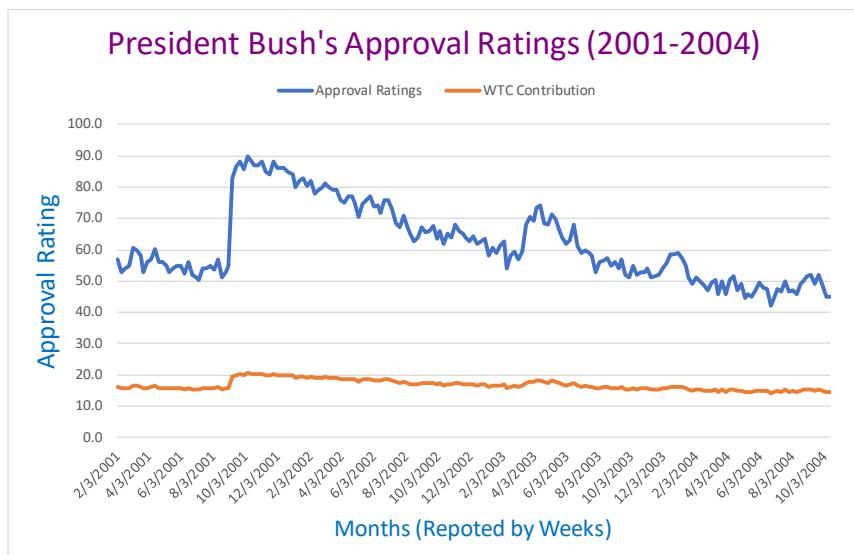


Figure 10-2. Contribution of WTC toward the Presidential Job Approval Ratings

ARIMA Model Parameters

The model parameters for the independent variable (see Table 10-1) and the dependent lag coefficients (see Table 10-1), and the Level for the intercept component) are required for implementing intervention effect equation.

Table 10-1. Model parameters for the independent variables

Final Estimates of the Free Parameters					
Component	Parameter	Estimate	Approx Std Error	t Value	Approx Pr > t
WTC	Coefficient	0.00023	0.000033	6.00	<.0001
DepLag	Phi_1	0.63303	0.04772	13.26	<.0001
Level		5.38622175	1.0803414		
wtc		0.0005	2.56E-05		
iraqwar		0.0004	1.37E-05		

The Excel formulas representing LDV are shown I

Table 10-2. Excel formulas for the intervention effect equation

date	WTC Contribution	RDPI Contribution	11SEP Contribution
Feb-01	=(\$B\$16+(\$C\$8*G2))/(1-\$C\$11)	=(\$B\$16+(\$C\$9*H2))/(1-\$C\$11)	=(\$B\$16+(\$C\$10*I2))/(1-\$C\$11)
=J2+7	=(\$B\$16+(\$C\$8*G3))/(1-\$C\$11)	=(\$B\$16+(\$C\$9*H3))/(1-\$C\$11)	=(\$B\$16+(\$C\$10*I3))/(1-\$C\$11)

The Art of Modeling

The art of modeling in part is choosing exogenous regressors based partly on our data and partly on our intuition. Even though a regressor may be significant, there is a choice to make in terms of overfitting the model or watering-down the effect of a predictive regressor by adding too many with weak predictive power. I have seen modelers rejoice due to their model with 40 independent variables, yet not be able to explain them intuitively. They can speak to significance and p-values, but wonder why management is not using their masterpiece. The contribution chart in Figure 10-3 includes two of the exogenous regressors. When we apply the intervention effect equation, it may be clear why we do not need 40 exogenous explanatory variables.

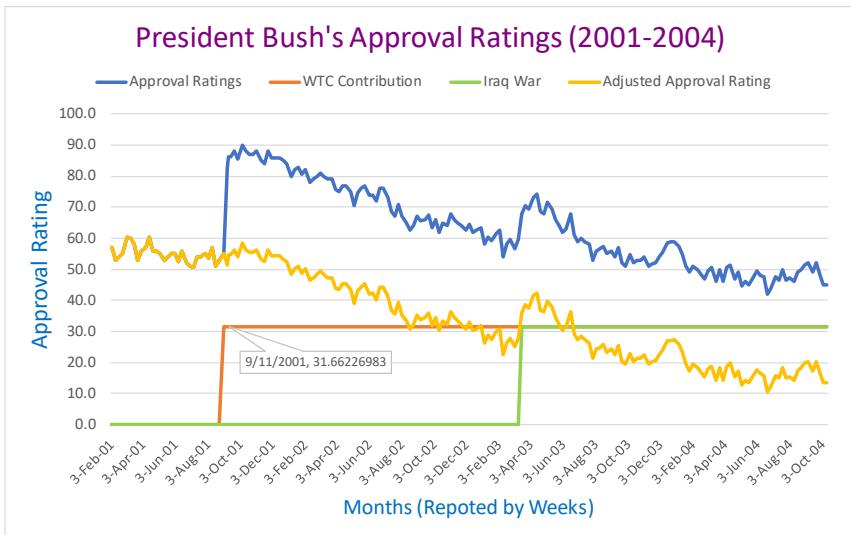


Figure 10-3. Contribution chart with all three independent variables

We could take the adjusted approval rating (yellow curve) and fit it with an ARIMA model, assuming we resolve any stationarity issues, i.e., using differencing. We did fit the data with an ARIMA(1,1,1) and the result is plotted in Figure 10-4. However, we want to continue our analysis to grasp some deeper concepts.

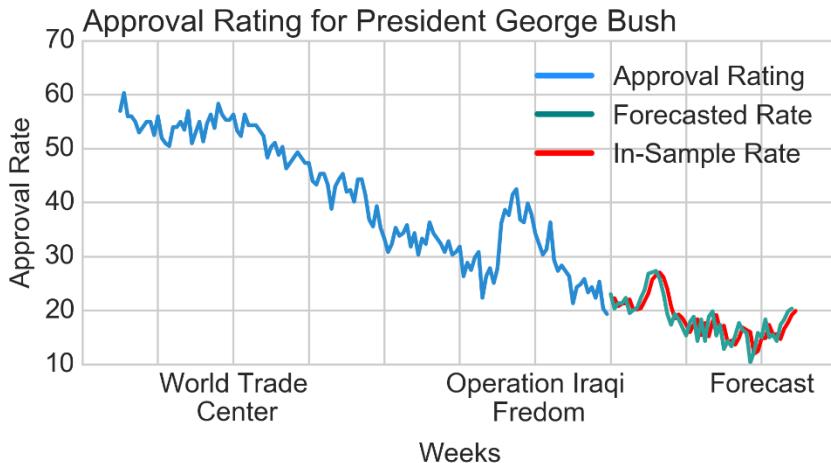


Figure 10-4. Adjusted approval ratings ARIMA(1,1,1) model

Conclusion

Contribution graphs in time series analysis is an excellent way to understand how significant predictor variables impact the dependent variable as demonstrated by WTC for the Presidential Job Approval Ratings.

Exogenous Regressors in Python

Now, we will work the same example, Presidential approval ratings, except we will use Python programming as our analysis tool.

We will examine ARIMA and SARIMA models with Python using the *Statsmodels* package. Auto-Regressive Integrated Moving Average (ARIMA) and Seasonal Auto-Regressive Integrated Moving Average (SARIMA) can be called from the *tsa* (Time Series) module from the *Statsmodels* package using *Jupyter*.

The big difference between an ARIMA model and a SARIMA model is the addition of seasonal error components to the model. Remember that the purpose of an ARIMA model is to make the time-series that we are working with act like a stationary series. This is important because if it is not stationary, we can get biased estimates of the coefficients.

There is no difference with a SARIMA model. We are still trying to get the series to behave in a stationary way, so that our model gets estimated correctly. I want to emphasize that we could get away with a regular old ARIMA model for this if we satisfy a couple of conditions.

1. We have enough data to estimate a large number of coefficients
2. We are willing to assume a really complicated error structure

Generally, we are not willing to entertain either of those assumptions, and that is why we have a SARIMA model. Seasonality can come in two basic varieties, multiplicative and additive. By default, *statsmodels* works with a multiplicative, seasonal components. For our model it really does not matter.

We can view the ARIMA model as a special case of the SARIMA model, i.e. $\text{ARIMA}(1,1,1) = \text{SARIMA}(1,1,1)(0,0,0,X)$ where X can be any whole number.

Data

The data can be downloaded from <https://github.com/stricie1/Data>. The file name is *bushapprove.csv*.

Import Packages

In addition to *Statsmodels*, we will need to import additional packages, including *Numpy*, *Scipy*, *Pandas*, and *Matplotlib*. Also, from *Statsmodels* we will need to import *qqplot*.

```
In [1]:  
import numpy as np  
from scipy import stats  
import pandas as pd  
from pandas import Series, DataFrame, Panel  
import matplotlib.pyplot as plt  
import statsmodels.api as sm  
from statsmodels.graphics.api import qqplot
```

Loading the Dataset

We will use pandas to load the data. There are a couple of ways of loading the data. One way is to split the data into two comma separated value files, one containing the index (date) and the dependent variable, '*approve*', with the other containing the index (date) and the independent variables '*wtc*', '*iraqwar*', etc. We can then name them "endog" and "exog", respectively. Another way is to load the complete data set in Python and manipulate it there, which is the method we decided to use.

```
In [2]:  
df = pd.read_csv("D:\\Documents\\DATA\\bushjob.csv")
```

Data Prep

Before we can perform our time series analysis (like every time before) we have to prepare our data. There are more than a dozen columns in the csv file and we only need a few of them. (By experimentation we discovered we need at least five.) We can view the column names as a list as follows.

```
In [3]:  
df.columns
```

```
Out [3]:  
Index(['week', 'early', 'bd', 'nn', 'approve', 'iraq', 'rdpi',  
'pop', 'rdpipcly', 'rdpipclh', 'rdpipclq', 'totbd', 'l10tbd',  
'date', 'wtc', 'afghan', 'sfafghan', 'kabul', 'bushun',  
'congress', 'powell', 'iraqwar', 'baghdad', 'mission',  
'unoccupy', 'igc', 'unhq', 'italian', 'saddam', 'prisoner',  
'uniraq', 'karzai', 'obs', 'rdpipc'],  
dtype='object')
```

Next, we want to pull out the data we are interested in, which includes the dependent variable of the time series, `approve`, that represents the approval ratings. We also want the exogenous regressors:

- `iraqwar` = Operation Iraqi Freedom
- `afgan` = US forces deploy to Afghanistan to war on terror
- `wtc` = World Trade Center (Sept 11)
- `Kabul` = a city in Afghanistan, which was in the News a lot

We will do this by drawing out a subset from the dataframe. However, let's first index the dataframe for building our time series. In this data set, if we look at the names, birth the `date` field and `week` field are candidates for our index. In this instance, we want `week`, as the approval ratings is a weekly poll.

```
In [4]:  
df.set_index ('week', inplace = True)
```

We can check the results with:

```
In [5]:  
df.index
```

```
In [5]:  
Int64Index([ 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,  
...  
189, 190, 191, 192, 193, 194, 195, 196, 197, 198],  
dtype='int64', name='week', length=194)
```

Now, we pull our variables from the indexed dataframe. We will name the dependent variable `y` and the exogenous variable `X`, but we want to remember that we have already index the dataframe, giving it a times series format:

```
In [6]:
# Generate a subset of the series
df = df[['approve','iraqwar','wtc','afghan','kabul']]

# Create the train and validation sets
train = df[:int(0.75*(len(df)))]
valid = df[int(0.75*(len(df))):]

X_train = np.asarray(train[['iraqwar','wtc','afghan','kabul']])
X_valid = np.asarray(valid[['iraqwar','wtc','afghan','kabul']])
y_train = np.asarray(train[['approve']])
y_valid = np.asarray(valid[['approve']])

df.head(5)
```

	approve	iraqwar	wtc	afghan	kabul
week					
5	57.0	0	0	0	0
6	57.0	0	0	0	0
7	53.0	0	0	0	0
8	54.0	0	0	0	0
9	55.0	0	0	0	0

Plot the Time Series Data

Now, we plot the Presidential approval ratings in Figure 10-5, with the exogenous regressors. To see the approval ratings and exogenous regressors on one plot, we have to have dual axes, and two legends. We put the approval axis on the left and adjust the exogenous regressors axis so that the plots are visible.

```
In [7]:
# Plot parameters
fig, ax1 = plt.subplots(figsize = (8, 5), dpi = 300)

# Setup figure graphics
plt.xticks(np.arange(0, 200, 20))
plt.grid(b = None, which = 'major', axis = 'both',
         color = 'lightgray', linestyle = ':')
ax1.set_facecolor('azure')

# Approval ratings plot
ax = ytrain.plot(ax = ax1)
# Change of axes
```

```

ax = ax1.twinx()
align_yaxis(ax1, 0, ax2, -30000)
# Exogenous regressors plots
ax = X.plot(ax = ax,
            color = ['red', 'orange', 'green', 'purple'])

# Plot aesthetics
fig.suptitle('Approval Ratings for President George Bush
(2001-2004)', fontsize = 14, color = 'olive')
plt.ylabel('Approval Ratings', fontsize = 12, color = 'olive')
plt.ylabel('Exogenous Regressors', fontsize = 12,
           color = 'olive')
plt.xlabel('Weeks', labelpad = 8, fontsize = 14,
           color = 'indigo')

# Major interventions
labels = [item.get_text() for item in ax2.get_xticklabels()]
labels[2] = 'World Trade\n Center'
labels[6] = 'Operation\n Iraqi Freedom'

# Axes modifications
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
ax.set_xticklabels(labels, fontsize = 14)

# Generate legend and save image
ax1 = plt.legend(loc = 'upper center')
plt.savefig('01_bush_all_ts.png', figsize = (8,5), dpi = 300,
            bbox_inches = 'tight')

plt.show()

```

One thing to notice in Figure 10-5**Error! Reference source not found.** is the exogenous regressors, `wtc`, `afghan`, and `kabul`. From the plot alone, it appears that `wtc` trumps the other two factors. However, the effect of the large increase in approval ratings are obvious. Does this give us any issues with time series analysis? Yes, it does. Recall that this phenomenon is termed “intervention,” with the implication that some event occurred that intervened with the trend of the series. We would see the phenomenon occur during a stock market crash. So, with this data we will see two advanced concepts: exogenous regressors and adjustments for interventions.

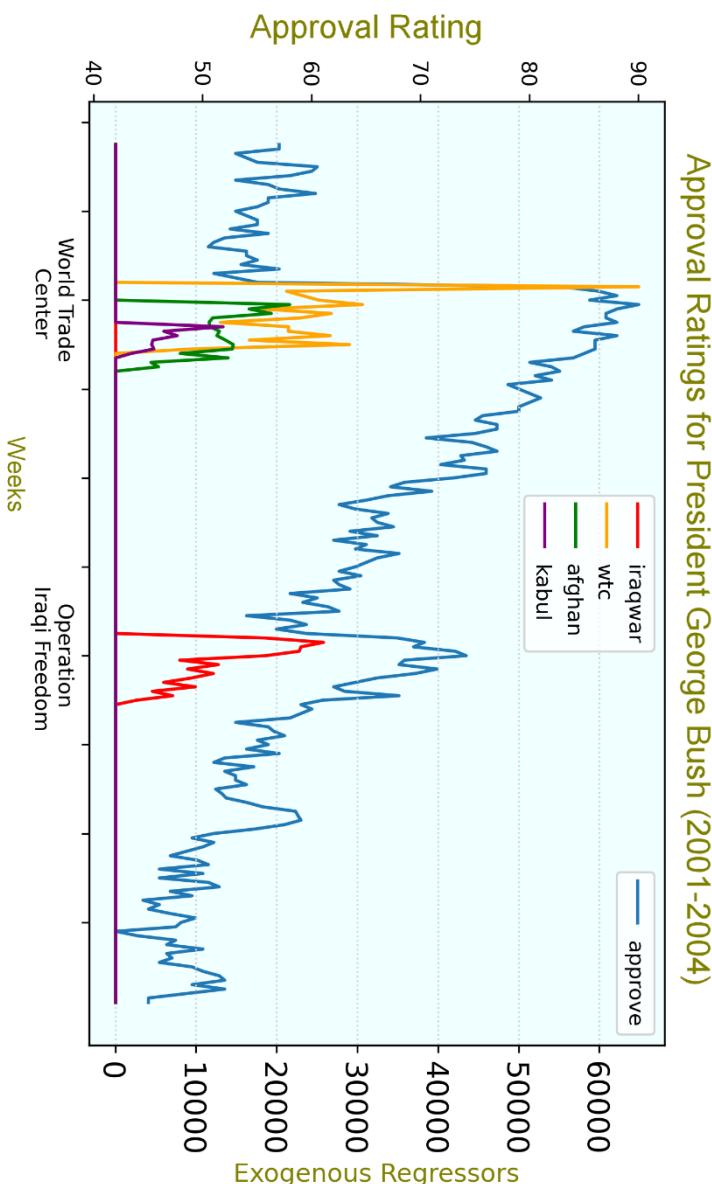


Figure 10-5. Time Series plot of the weekly Presidential approval ratings of George Bush from 2001 to 2004

Stationarity

Before we begin fitting models, we need to determine if the series is stationary or not. To do this, we will run a Ljung-Box test.

```
In [8]
```

```
sm.stats.acorr_ljungbox(y, lags = [10], return_df = True)
```

```
Out [8]
```

lb_stat	lb_pvalue
10	1335.482751
	8.402968e-281

The p-value is so low, that we may reject the null hypothesis of nonstationarity and consider the time series as stationary. However, we have not yet determined the effects of intervention, which we will do later.

Seasonality

One final check is for seasonality. For this we will perform a decomposition of the series and plot them in Figure 10-6.

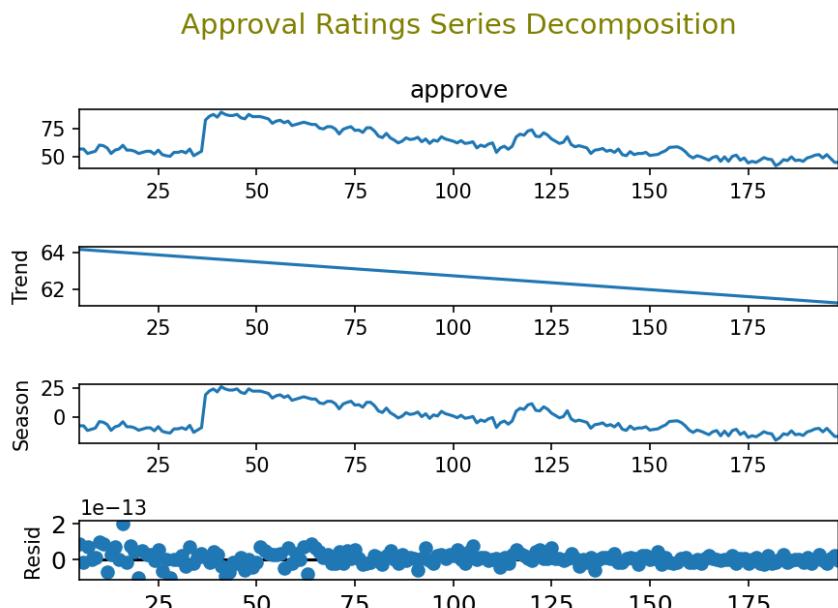


Figure 10-6. Presidential approval rating series decomposition

Based on our plots in *Figure 10-6*, there does not appear to be any seasonality in the season plot. However, note the negative trim, which matches our observations. We will make one more set of plots to diagnose seasonality. The monthly seasonal series plot is Figure 10-7 and the monthly seasonal box plot is Figure 10-8.

```
In [9]
import seaborn as sns
variable = 'approve'
fig, ax = plt.subplots(figsize = (14, 6))
plt.subplots_adjust(top = 0.9)
palette = sns.color_palette("ch:2.5,-.2,dark=.3", 10)
sns.lineplot(df['month'], df[variable])
plt.title('Approval Rating for President George Bush',
          fontsize = 16)
plt.savefig('bush_seas_bx.png', figsize = (4,3), dpi = 300,
            bbox_inches = 'tight')
        fontdict = dict(weight = 'bold'))
ax.set_xlabel('month', fontsize = 16,
              fontdict = dict(weight = 'bold'))
ax.set_ylabel('Approval Ratings', fontsize = 16,
              fontdict = dict(weight = 'bold'))
plt.xticks(rotation = 45, ha = 'right')

fig, ax = plt.subplots(figsize = (16, 7))
sns.boxplot(df['month'], df[variable], ax = ax)
plt.subplots_adjust(top = 1)
ax.set_title('Month-wise Box Plot\n(The Seasonality)',
              fontsize = 20, loc = 'center',
              fontdict = dict(weight = 'bold'))
ax.set_xlabel('Month', fontsize = 16,
              fontdict = dict(weight = 'bold'))
ax.set_ylabel('Approval Ratings', fontsize = 16,
              fontdict = dict(weight = 'bold'))
plt.xticks(rotation = 45, ha = 'right')
```

We also plot the trend as a box plot in Figure 10-9. To do this, we had to append a column in the data for 'year'.

```
In [10]
fig, ax = plt.subplots(figsize = (16, 7))

sns.boxplot(df['year'], df[variable], ax = ax)
```

```

ax.set_title('Year-wise Box Plot\n(The Trend)', fontsize = 20,
             loc = 'center', fontdict = dict(weight = 'bold'))
ax.set_xlabel('Year', fontsize = 16,
              fontdict = dict(weight = 'bold'))
ax.set_ylabel('Approval Ratings', fontsize = 16,
              fontdict = dict(weight = 'bold'))

```

In Figure 10-9, we can see the intervention in 2001, by the range of the box plot (in blue), and then the downward trend through 2004. The plot does not show any effect on approvals from Operation Iraqi Freedom in 2003, unlike the observed data seemed to appear.

```

import seaborn as sns
fig, ax = plt.subplots(figsize = (14, 6))
plt.subplots_adjust(top = 0.9)
palette = sns.color_palette("ch:2.5,-.2,dark=.3", 10)
sns.lineplot(df.index, df['approve'])
plt.title('Approval Rating for President George Bush',
           fontsize = 16)
ax.set_title('Seasonal plot of Presidential Approval Ratings',
             fontsize = 20, loc='center',
             fontdict=dict(weight='bold'))
ax.set_xlabel('month', fontsize = 16,
              fontdict = dict(weight = 'bold'))
ax.set_ylabel('Approval Ratings', fontsize = 16,
              fontdict = dict(weight = 'bold'))
plt.xticks(rotation = 45, ha = 'right')
plt.show()

```

Figure 10-10 shows a seasonal plot grouped by year, comprised of months. The groupings in and Figure 10-10 allows use to generate a confidence interval that we can see in those plots.

```

import seaborn as sns
fig, ax = plt.subplots(figsize = (12, 5))
plt.subplots_adjust(top = 0.9)
palette = sns.color_palette("ch:2.5,-.2,dark=.3", 10)
sns.lineplot(df.index, df['approve'])
plt.title('Approval Rating for President George Bush',
           fontsize = 16)
ax.set_xlabel('month', fontsize = 16,
              fontdict = dict(weight = 'bold'))
ax.set_ylabel('Approval Ratings', fontsize = 16,
              fontdict = dict(weight = 'bold'))

```

```
plt.xticks(rotation = 45, ha = 'right')
plt.show()
```

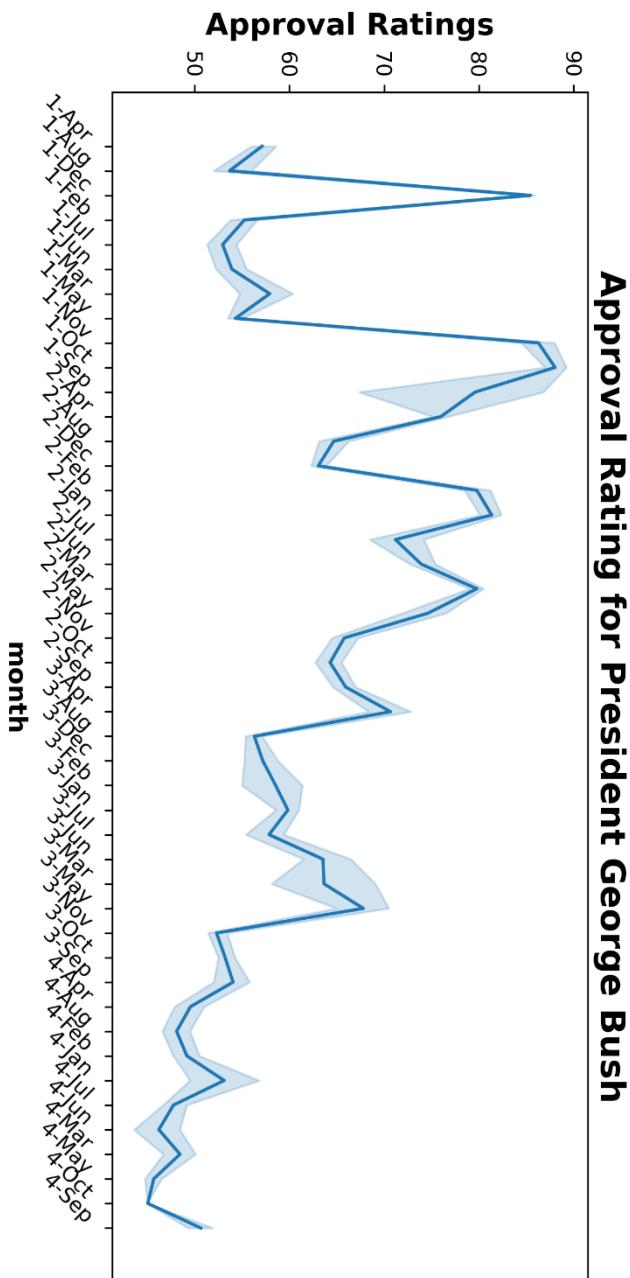


Figure 10-7. Season plot for Presidential approval ratings

Month-wise Box Plot (The Seasonality)

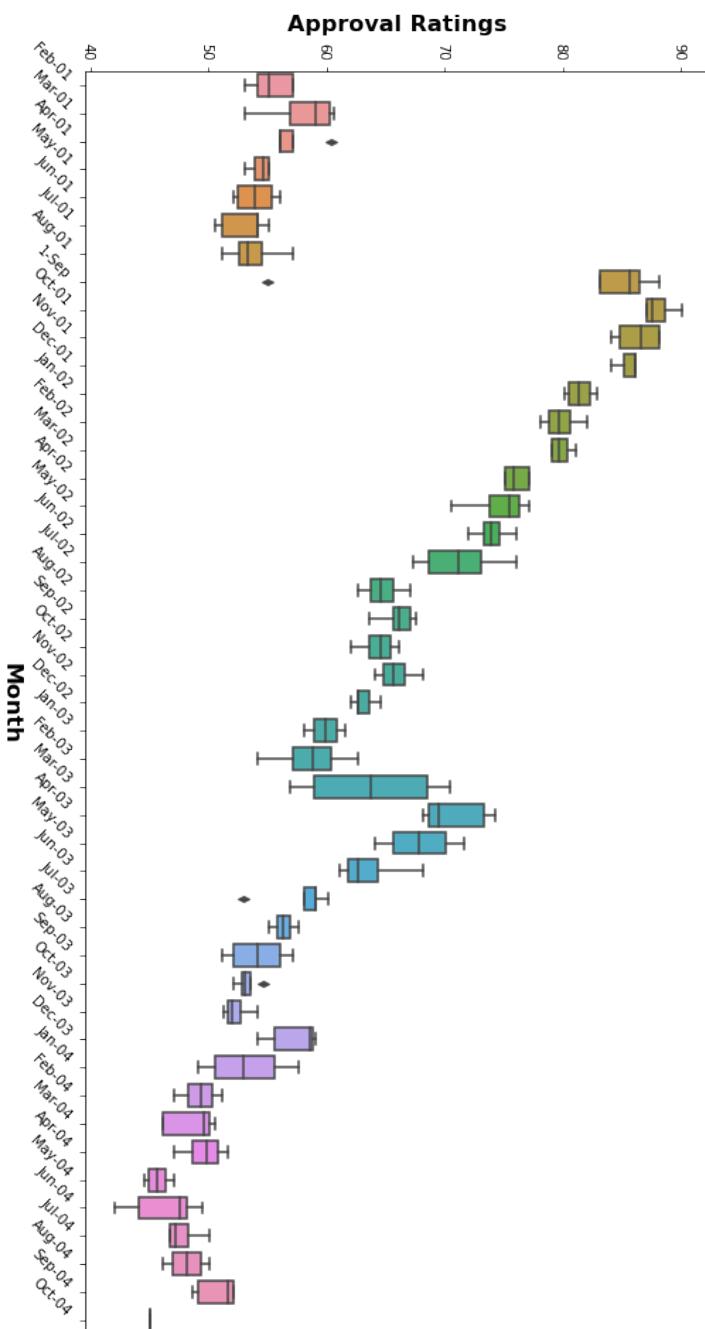


Figure 10-8. Monthly seasonal box plot for Presidential approval ratings

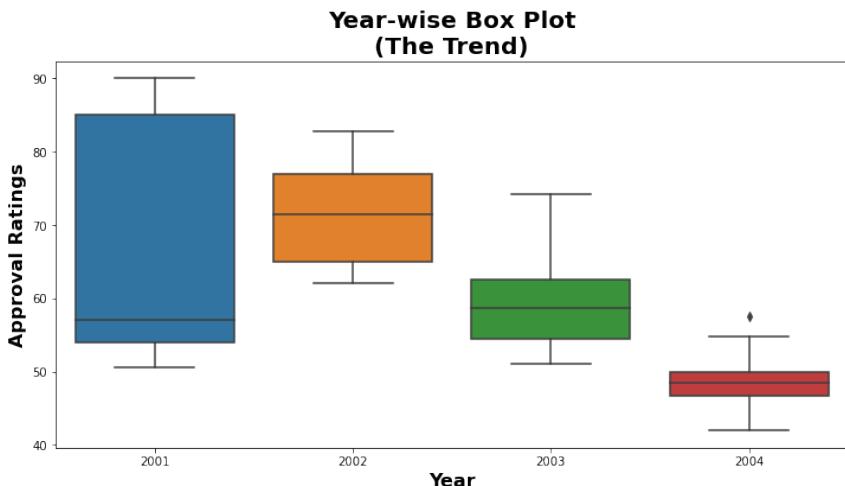


Figure 10-9. Annual trend box plot for Presidential approval ratings

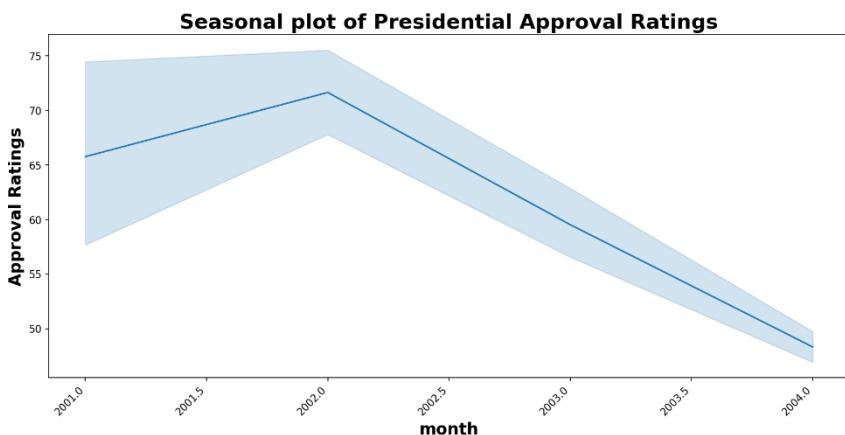


Figure 10-10. Seasonal plot by years of months

Auto-correlations

Before we decide which model to use, we also need to look at auto-correlations. A popular test for serial correlation is the Durbin-Watson (DW) statistic.

Definition 10.2. If e_t is the residual given by $e_t = \rho e_{t-1} + v_t$, then the Durbin-Watson statistic states that null hypothesis, $\rho = 0$, with the alternative hypothesis $\rho \neq 0$, then the test statistic is where T is the number of observations.

$$DW = \frac{\sum_{t=2}^T (e_t - e_{t-1})^2}{\sum_{t=1}^T e_t^2}$$

The DW statistic will lie in the 0-4 range, with a value near two indicating no first-order serial correlation. Positive serial correlation is associated with DW values below 2, and negative serial correlation with DW values above 2.

The value of Durbin-Watson statistic is close to 2 if the errors are uncorrelated. In our example, it is 0.0029. That means that there is strong evidence that the approval ratings are not highly autocorrelation. This effect is shown in the first plot in Figure 10-11.

```
In [11]:  
sm.stats.durbin_watson(dta1)
```

```
Out[11]:  
array([ 0.00288214])
```

Auto Selecting an ARIMA Model

Recall that a nonseasonal ARIMA model is classified as an ARIMA(p,d,q) model, where:

p is the number of autoregressive terms

d is the number of nonseasonal differences needed for stationarity

q is the number of lagged forecast errors in the prediction equation

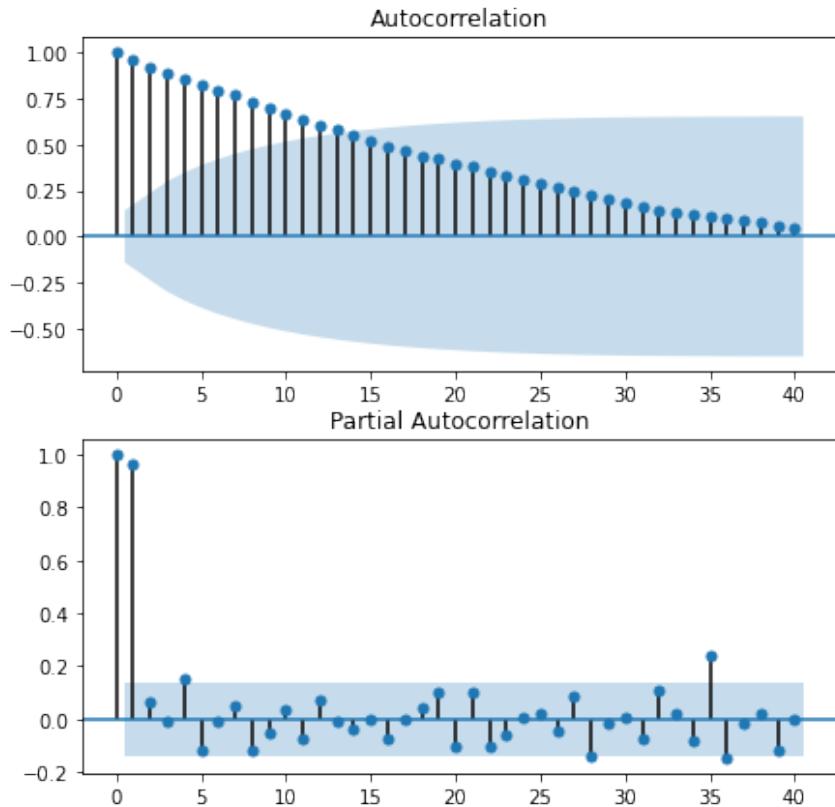


Figure 10-11. ACF (top) and PACF (bottom)

The forecasting equation is constructed as follows. First, let y denote the d th-difference of Y , which means:

$$\text{If } d = 0: y_t = Y_t$$

$$\text{If } d = 1: y_t = Y_t - Y_{t-1}$$

$$\text{If } d = 2: y_t = (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) = Y_t - 2Y_{t-1} + Y_{t-2}$$

Note that the second difference of Y (the $d=2$ case) is not the difference from two periods ago. Rather, it is the first-difference-of-the-first difference, which is the discrete analog of a second derivative, i.e., the local acceleration of the series rather than its local trend.

ARIMA assumes stationarity in the time series data, and we have already dealt with the issue of nonstationarity with our Durbin-Watson (DW)

test. It is noteworthy that the DW test does not work well when seasonality is present, but there appears to be none in our time series.

Identifying the numbers of AR or MA terms in an ARIMA model

Since stationarity is present in our series, we will start with no difference, $d = 0$. Also, we have included four exogenous regressors, and now at least two of them may not be significant, so $p = 4$. Since we know we have trend in the series, we'll add a moving average term, so $q = 1$.

Before we proceed, however, let's see what model would we get with auto-ARIMA. We use the *pmdarima* library, formerly the *pyramid arima* library, and its `auto_arima()` function. We also do not include the exogenous regressors, which we will do later, so that we can evaluate the approval series without external evaluating external effects.

```
In [12]:  
from pmdarima.arima import auto_arima  
auto_model = auto_arima(yt, trace = True,  
                        error_action = 'ignore', suppress_warnings = True)  
auto_model.fit(yt)
```

```
Out [12]  
Performing stepwise search to minimize aic  
ARIMA(2,1,2)(0,0,0)[0] intercept : AIC=789.031, Time=0.94 sec  
ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=784.554, Time=0.02 sec  
ARIMA(1,1,0)(0,0,0)[0] intercept : AIC=786.011, Time=0.03 sec  
ARIMA(0,1,1)(0,0,0)[0] intercept : AIC=785.992, Time=0.02 sec  
ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=782.558, Time=0.01 sec  
ARIMA(1,1,1)(0,0,0)[0] intercept : AIC=787.756, Time=0.06 sec  
  
Best model: ARIMA(0,1,0)(0,0,0)[0]  
Total fit time: 1.345 seconds
```

We see from the output that the algorithm stops after 11 runs and gives us an ARIMA(1,0,0)(0,0,0)[0], which should not be surprising, since there was negligible seasonality and stationarity in the series data. Also, we did not include the exogenous regressors, so there is only one auto-regressor. This is reflected in the results below.

```
In [13]  
print(auto_model.summary())
```

```
Out [13]
```

SARIMAX Results

```
=====
Dep. Variable:          y      No. Observations:    155
Model: SARIMAX(0, 1, 0)   Log Likelihood: -413.581
Date: Mon, 09 Nov 2020   AIC:                 829.082
Time: 13:51:44           BIC:                 830.315
                           - 155
Covariance Type: opg
=====
            coef  std err      z   P>|z|   [0.025  0.975]
-----
sigma2    12.5885  0.403   31.215   0.000   11.798  13.379
-----
Ljung-Box (Q):        31.83   Jarque-Bera (JB): 3740.29
Prob(Q):             0.82   Prob(JB):       0.00
Heteroskedasticity (H): 0.44   Skew:          3.05
Prob(H) (two-sided):  0.00   Kurtosis:     26.36
-----
```

Given that this is a good fit (with exogenous regressors), we can use it to forecast or make a prediction.

Prediction

For forecasting the near-term approval ratings, we use the `predict()` function from `pmdarima`. We call the prediction, 'forecast' to explicitly keep the nomenclature consistent.

```
In [14]
```

```
forecast = auto_model.predict(n_periods = len(yv))
forecast = pd.DataFrame(forecast, index = valid.index,
                        columns = ['Prediction'])
```

To evaluate the fit, we use MAPE as we have before. The outcome is not indicative of a great fit, so to use the forecast, we would need to keep to short-term predictions.

```
In [15]
```

```
MAPE(ytrue, forecast)
```

```
Out [15]
```

```
9.81647858616668
```

Now, we plot our time series, the out-of-sampling forecast, and our true approval ratings (short-term). Considering Figure 10-12, it would appear

that the forecast is far from accurate. As such, we will do some diagnostics.

```
In [16]:  
fig, ax = plt.subplots(figsize=(8, 6), dpi = 150)  
plt.xticks(np.arange(0, 195, 12))  
  
# Plot the series, the test set, and predictions  
plt.plot(y[155:].index, y_test, label = 'Test Set')  
plt.plot(y[:155].index, y_train, label = 'Training Set')  
plt.plot(y[155:].index, forecast, label = 'Prediction')  
  
# Predictions confidence interval  
ci = np.mean(test) * np.std(test)/np.mean(test)  
x = np.ravel(y[155:].index)  
pred = np.ravel(forecast)  
pred_low = (pred - ci)  
pred_hi = (pred + ci)  
plt.plot(x, pred_low, color = 'red', linestyle = ':')  
plt.plot(x, pred_hi, color = 'red', linestyle = ':')  
plt.fill_between(x, pred_low, pred_hi, color = 'violet',  
alpha = 0.1)  
  
# Plot aesthetics  
fig.tight_layout()  
plt.title('Approval Rating for President George Bush  
          (2001-2004)', fontsize = 16, color = 'indigo')  
plt.xlabel('Weeks', labelpad = 8, fontsize = 14,  
          color = 'indigo')  
plt.ylabel('Approval Rating', labelpad = 8, fontsize = 14,  
          color = 'indigo')  
plt.xticks(np.arange(0, 200, 12))  
labels = [item.get_text() for item in ax2.get_xticklabels()]  
labels[3] = 'World Trade\n Center'  
labels[10] = 'Operation Iraqi\n Freedom'  
labels[15] = 'Forecast'  
plt.grid(b = None, which = 'major', axis = 'both',  
         color = 'lightgray', linestyle = ':')  
plt.xticks(fontsize = 14)  
plt.yticks(fontsize = 14)  
ax.set_xticklabels(labels, fontsize = 14)  
plt.legend()
```

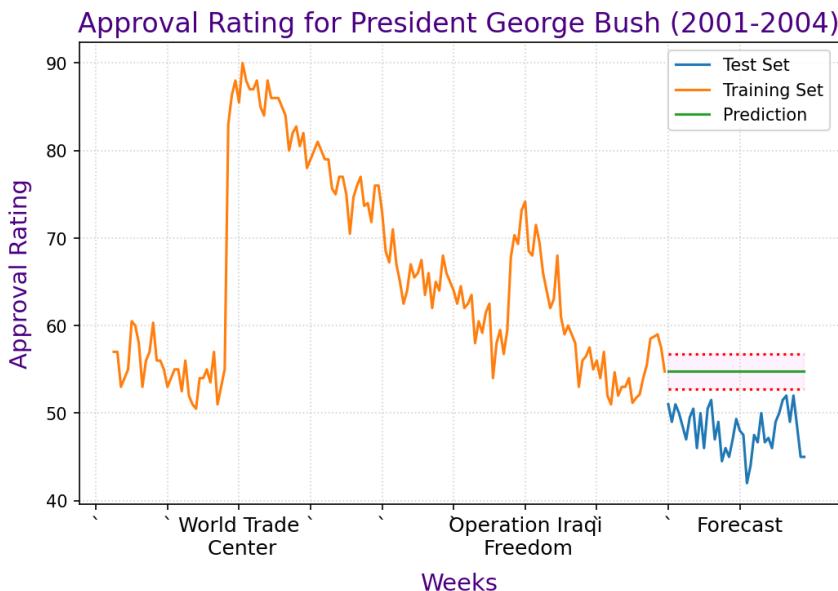


Figure 10-12. Auto-ARIMA model (SARIMAX(1, 0, 0)) of Presidential approval ratings

Residual Diagnostics

Next, we plot and study the residual data of our ARIMA(1,0,0) model, as seen in Figure 10-13. The ACF and PACF do not throw up alarms

The ACF has 3 significant spikes and the PACF has 6 significant spikes. If we therefore set the order of the AR term to 6--i.e., fit an ARIMA(6,1,0) model--we obtain the following ACF and PACF plots for the residuals shown in Figure 10-13. Also, since the trend constant is not significant, we drop it.

```
In [14]:
arima_mod_2 = sm.tsa.ARIMA(dta1, order = (5,0,1))
arima_res_2 = arima_mod_2.fit(trend = 'nc', disp = 1)
resid_opt_2 = arima_res_2.resid
%matplotlib inline
fig = plt.figure(figsize = (12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(resid_opt_2.values.squeeze(),
    lags=40, ax = ax1)
ax2 = fig.add_subplot(212)
```

```
fig = sm.graphics.tsa.plot_pacf(resid_opt_2, lags= 40, ax = ax2)
```

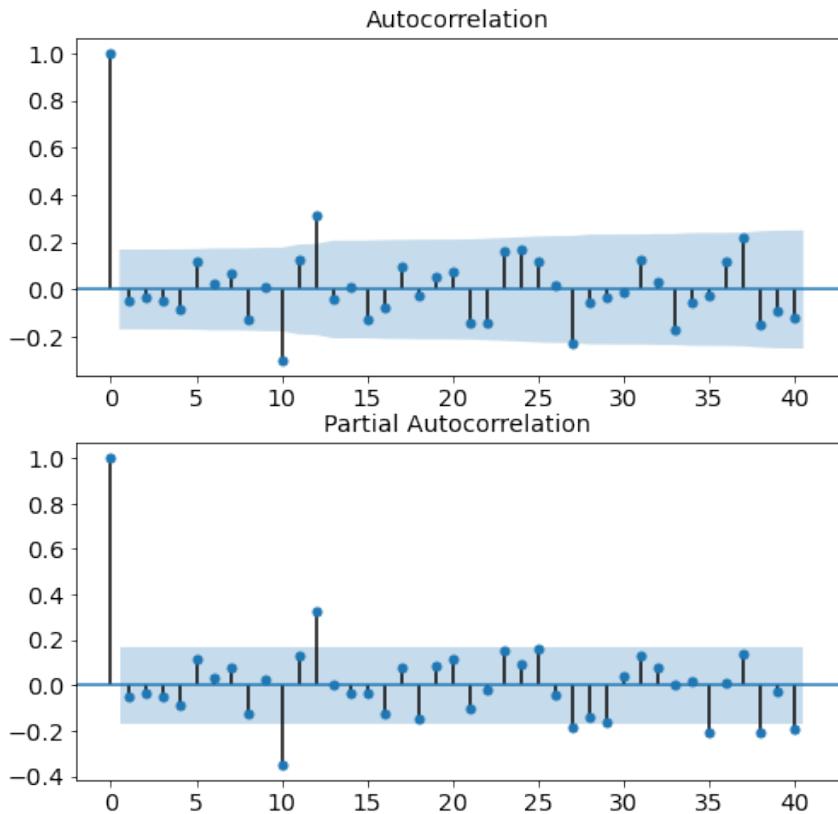


Figure 10-13. ARIMA (0,1,0) model ACF (top) and PACF (bottom)

The autocorrelation at the crucial lags—namely lags 1 and 2—has been eliminated, and there is no discernible pattern in higher-order lags. The time series plot of the residuals shows a slightly worrisome tendency to wander away from the mean (see Figure 10-14):

```
In [15]:  
# show plots in the notebook  
%matplotlib inline  
fig = plt.figure(figsize = (12,8))  
ax = fig.add_subplot(111)  
ax = arma_res_2.resid.plot(ax = ax);
```

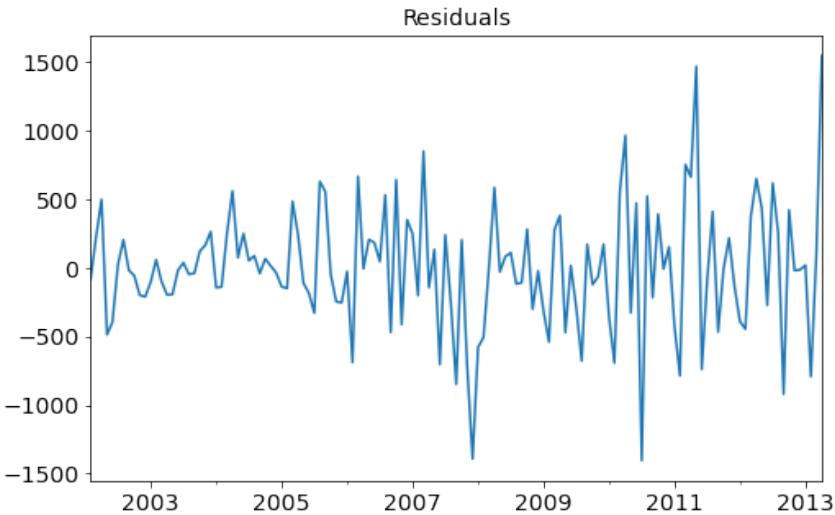


Figure 10-14. ARIMA(6,0,1) model residuals

However, the analysis summary report shows that the model nonetheless performs quite well in the validation period, both AR coefficients are significantly different from zero, and the standard deviation of the residuals has been reduced from 512.636 to 467.41 (nearly 10%) by the addition of the AR terms. Furthermore, there is no sign of a "unit root" because the sum of the AR coefficients is not close to 1, which we can confirm with a Ljung-Box test. On the whole, this appears to be a good model.

In Figure 10-15, we plot the graphical metrics associated with diagnostic testing for our model fit.

```
In [16]
auto_model.plot_diagnostics(figsize=(14, 8))
plt.show()
```

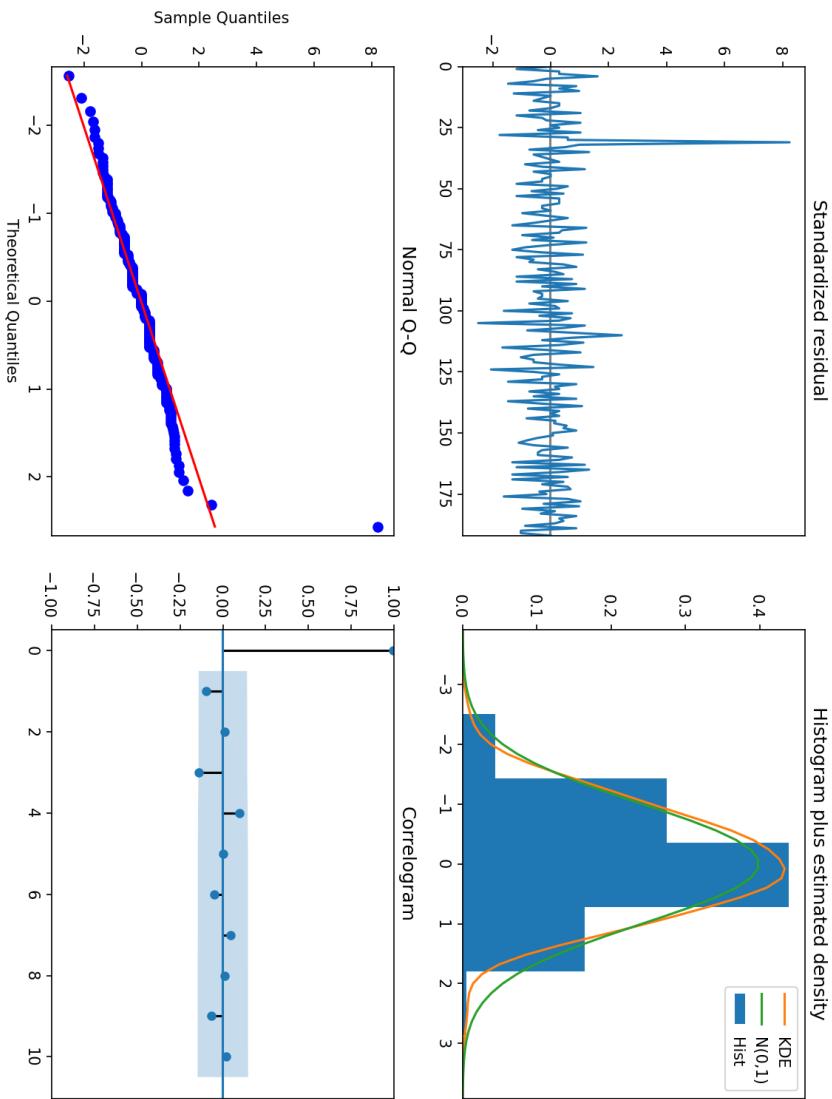


Figure 10-15. Diagnostic plots for the Presidential Approving Ratings

In [17]:

```
exogx = np.array(df[['wtc', 'iraqwar', 'afghan', 'kabul']])
fit2 = sm.tsa.ARIMA(df.ts, (4,0,1), exog = exogx).fit()
print(fit2_2.summary())
```

Out [17]:

ARMA Model Results

Dep. Variable:	approve	No. Observations:	194			
Model:	ARMA(4, 1)	Log Likelihood	-484.440			
Method:	css-mle	S.D. of innovations	2.919			
Date:	Sun, 01 Nov 2020	AIC	990.880			
Time:	21:18:15	BIC	1026.826			
Sample:	0	HQIC	1005.435			
=====						
	coef	std err	z	P> z	[0.025	0.975]

const	52.2077	6.306	9.389	0.000	46.848	71.568
'wtc'	0.0002	0.00004	6.169	0.000	0.000	0.000
'iraqwar'	0.0004	0.000	3.624	0.000	0.000	0.001
'afghan'	0.0001	0.000	1.144	0.252	-0.0001	0.000
x4	-0.0002	0.000	-1.161	0.246	-0.001	0.000
ar.L1.	1.1059	0.437	2.532	0.011	0.250	1.962
approve						
ar.L2.	-0.1605	0.389	-0.412	0.680	-0.923	1.962
approve						
ar.L3.	-0.0540	0.114	-0.472	0.637	-0.278	0.602
approve						
ar.L4.	0.0879	0.086	1.023	0.306	-0.081	0.256
approve						
ma.L1.	-0.2425	0.437	-0.555	0.570	-1.098	0.613
approve					Roots	
=====						
	Real	Imaginary		Modulus	Frequency	

AR.1	1.0212	-0.0000j		1.0212		-0.0000
AR.2	0.0960	-1.7564j		2.2703		-0.1612
AR.3	0.0960	+2.7564j		2.2703		0.1612
AR.4	-2.5986	-0.0000j		2.5986		-0.5000
MA.1	4.1232	+0.0000j		4.1232		0.0000

In [18]

```
sm.stats.diagnostic.acorr_ljungbox(fit2.resid, lags = [36],  
    return df = True, boxpierce = False)
```

Out [18]

```
[1] 36      lb_stat      lb_pvalue
```

Forecasting

The linear predictor is the optimal h -step ahead forecast in terms of mean-squared error:

$$y_t(h) = v + A_1 y_t(h-1) + \cdots + A_p y_t(h-p)$$

We can use the predict function to produce this forecast by adding the argument `exog` (exogenous predictors) that we defined in In[13] (`exogx`)

```
In [19]:  
yPred = fit2.predict(start = 131, end = 189, exog = val)  
print(pred)
```

```
Out [19]  
139    1.127267  
140    0.157766  
141    0.754541  
142    0.443914  
143    0.381727  
      ...  
195   -0.320879  
196   -0.297329  
197   -0.279804  
198   -0.266714  
199   -0.256900  
Length: 61, dtype: float64
```

The (untransformed) forecasts for the model show a linear upward trend projected into the future, as seen in Figure 10-16. Note that we have had some trouble with the dates format '`2013m5`', '`2017m5`', so we provide the number of months format in the comment.

```
In [20]:  
import matplotlib.pyplot as plt  
from matplotlib.lines import Line2D  
fig, ax = plt.subplots(figsize = (9, 5))  
ax = df.ts[10:].plot(ax=ax)  
fig = fit2.plot_predict(10, 200, exog = exogx, dynamic = True,  
                       ax = ax, plot_insample = False)  
plt.title('Approval Rating for President George Bush',  
          fontsize = 16)  
plt.xlabel('Weeks', labelpad = 8, fontsize = 14)  
plt.ylabel('Approval Rate', fontsize = 14)  
labels = [item.get_text() for item in ax.get_xticklabels()]  
labels[2] = 'World Trade\n Center'  
labels[6] = 'Operation Iraqi\n Freedom'  
labels[9] = 'Forecast'  
plt.xticks(fontsize = 14)  
plt.yticks(fontsize = 14)  
ax.set_xticklabels(labels)  
colors = ['blue', 'orange', 'gray']
```

```

lines = [Line2D([0], [0], color=c, linewidth = 3,
              linestyle ='-')
          for c in colors]
labels = ['Approval Rating', 'Forecasted Rate',
          '95% Confidence Interval']
plt.legend(lines, labels, prop = {'size': 14})
plt.show()

```

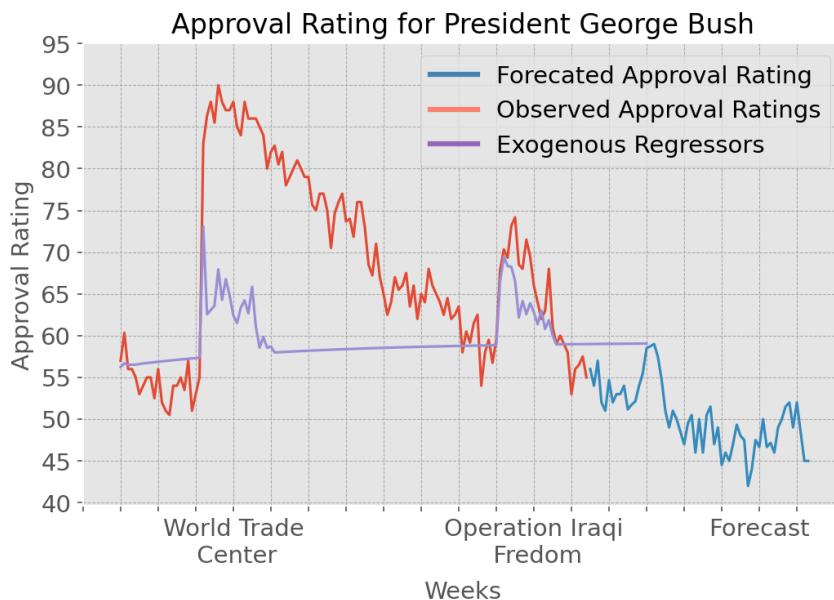


Figure 10-16. Approval rating ARIMA(1,1,1) model with external regressors: World Trade Center and Operation Iraqi Freedom

We show the fitted model with predictions and confidence intervals (upper and lower) in Figure 10-17.

```

In [21]
fig, ax = plt.subplots(figsize = (8, 5), dpi = 150)
grid(b = None, which = 'major', axis = 'both',
      color = 'lightgray', linestyle = ':')
plt.grid(b=None, which='major', axis='both',
      color = 'lightgray', linestyle = ':')
plt.xticks(np.arange(0, 200, 20))
# Plot aesthetics
# Plot the series, the test set, and predictions
ax.plot(y[155:].index, y_test, label = 'Test Set')

```

```

ax.plot(y[:155].index, y_train, label = 'Training Set')
ax.plot(y[155:].index, forecast, label = 'Prediction')

ax.set_xticklabels(labels, fontsize = 12)
labels = [item.get_text() for item in ax2.get_xticklabels()]
labels[2] = 'World Trade\n Center'
labels[6] = 'Operation Iraqi\n Freedom'
labels[9] = 'Forecast'

plt.title('Approval Rating for President George Bush  
(2001-2004)', fontsize = 16, color = 'indigo')
plt.xlabel('Weeks', labelpad = 8, fontsize = 14,
color = 'indigo')
plt.ylabel('Approval Rating', labelpad = 8, fontsize = 14,
color = 'indigo')

# Predictions confidence interval
ci = t + np.std(test)/np.mean(test)
x = np.ravel(y[155:].index)
pred = np.ravel(forecast)
pred_low = (pred - ci)
pred_hi = (pred + ci)
plt.plot(x, pred_low, color = 'red', linestyle = ':')
plt.plot(x, pred_hi, color = 'red', linestyle = ':')
plt.fill_between(x, pred_low, pred_hi, color = 'violet',
alpha = 0.1)

plt.legend()
plt.show()

```

The plot in Figure 10-17 shows a very "wide" confidence interval (approximately -2000 to +2000) for the forecast.

Detailed Analysis of the Residuals

In the following steps, we calculate the residuals, tests the null hypothesis that the residuals come from a normal distribution, and construct a qq-plot (see Figure 10-19).

Next, autocorrelation (AC), we calculate the lag, Q statistic, and $Prob > Q$. The Ljung–Box Q test—named for Greta M. Ljung and George E. P. Box (Ljung & Box, 1978) is a type of statistical test of whether any of a group of autocorrelations of a time series is different from zero. The null

hypothesis is, H_0 : The data are independently distributed (i.e. the correlations in the population from which the sample is taken are zero so that any observed correlations in the data result from the randomness of the sampling process).

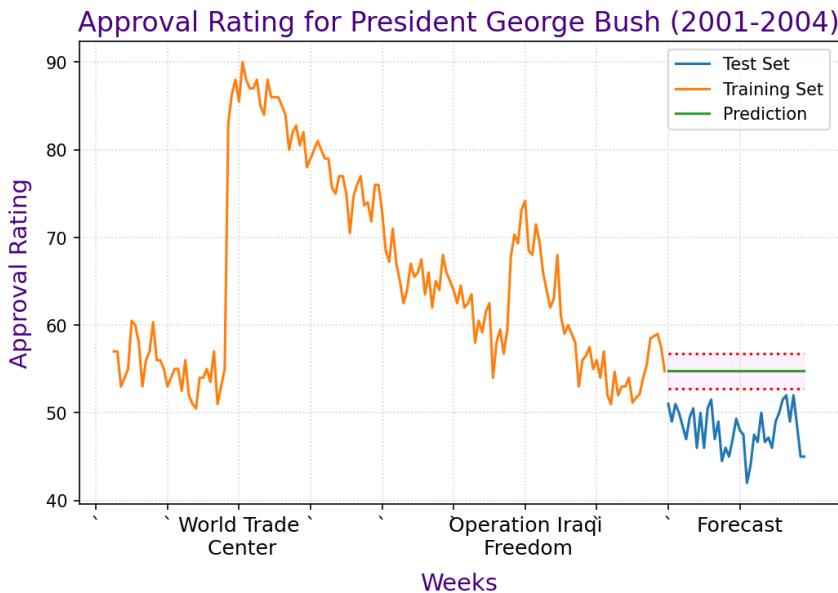


Figure 10-17. Presidential approval ratings with a confidence interval

Does our model obey the theory? We will use the Durbin-Watson test for autocorrelation. The Durbin-Watson statistic ranges in value from zero to 4. A value near 2 indicates non-autocorrelation; a value toward zero indicates positive autocorrelation; a value toward 4 indicates negative autocorrelation. Figure 10-19 shows the Q-Q plot for the model.

```
In [21]:  
sm.stats.durbin_watson(resid_3.values)
```

```
Out[21]:  
1.9966595580653848  
The Durbin-Watson test shows no autocorrelation.
```

```
In [22]:  
stats.normaltest(resid_3)
```

```
Out[22]:  
NormaltestResult(statistic=10.739349282735674,  
pvalue=0.0046556458184460851)
```

```
In [22]:  
# show plots in the notebook  
%matplotlib inline  
fig = plt.figure(figsize=(12,8))  
ax = fig.add_subplot(111)  
fig = qqplot(resid_3, line='q', ax=ax, fit=True)
```

Now, we calculate the MAPE.

```
In [23] MAPE(yt,yp)
```

```
Out[23]:  
18.76506441538958
```

In Figure 10-18, we show the residuals in a scatter plot.

```
In [24]  
plt.scatter(df[["approve"]],residual)  
plt.xlabel("approve - a predictor")  
plt.ylabel("residual")  
plt.title('Approval Rating Model Residuals', fontsize = 16)  
matplotlib.rcParams['axes.labelsize'] = 14  
plt.savefig('bush_scatter.png', figsize=(8,4), dpi=300,  
bbox_inches='tight')  
plt.show()
```

Since we are concerned with the distribution of the error (residuals), we generate a Q-Q plot to provide a visual. We are looking to see the residuals grouped along the diagonal line in Figure 10-19, and we see that they are except at the extremes. In spite of the curvature, we would say that the residuals appear to be approximately normally distributed.

```
In [25]  
fig, ax = plt.subplots(figsize = (12, 8), dpi = 300)  
fig = plt.subplot(222)  
ax = stats.probplot(yp, plot = plt)  
matplotlib.rcParams['axes.labelsize'] = 12  
plt.savefig('bush_prob.png', figsize = (12,8), dpi = 300,  
bbox_inches = 'tight')  
plt.show()
```

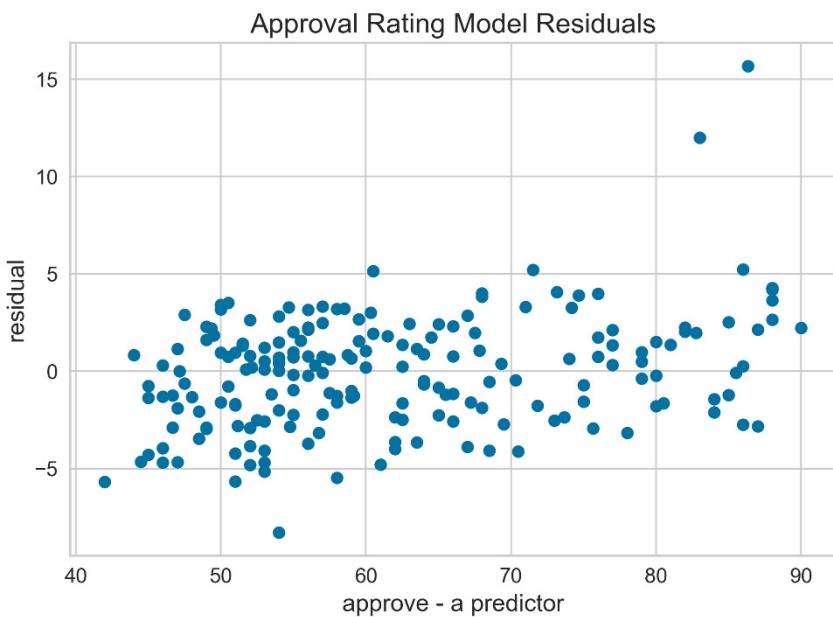


Figure 10-18. Scatterplot of approval rating model residuals

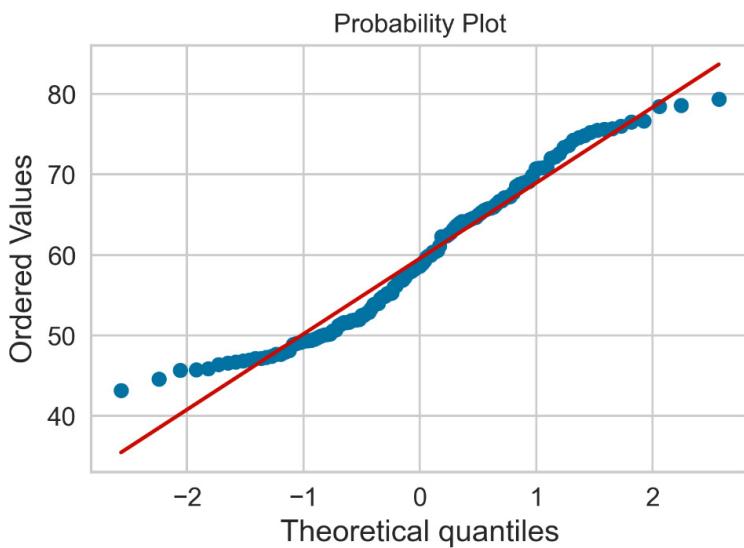


Figure 10-19. ARIMA(6,1,0) Q-Q Plot for Unsecure Consumer Loans

We can test to see if the residual are normal as follow:

```
In [26]
stats.normaltest(bush_resid)
```

```
Out[26]:
NormaltestResult(statistic=46.604873577, pvalue=7.5836850e-11)
```

Recall that the Durbin–Watson statistic is a test statistic used to detect the presence of autocorrelation at lag 1 in the residuals (prediction errors). A value close to 2 means there is no autocorrelation.

```
In [27]
sm.stats.durbin_watson(bush_resid.values)
```

```
Out[25]:
1.990514441938174
```

In Figure 10-20, we use the `plot_diagnostics()` function to generate a pot of the standardized residuals, Q-Q plot, correlogram, and histogram.

```
In [27]
results.plot_diagnostics(figsize=(16, 8))
```

We now calculate the Akaike Information Criterion (AIC), Schwarz Bayesian Information Criterion (BIC), and Hannan-Quinn Information Criterion (HQIC). Our goal is to choose a model that minimizes AIC, BIC, HQIC. Sometimes, there is not a model that has all three metrics that a minimum. In this case, we have to choose the "best minimum" based on knowledge about the problem and a little judgment, which comes with experience. In the meantime, it is not poor practice to pick a metric for comparison and sticking with it, until that point where analysis methods have matured.

```
In [28]:
r,q,p = sm.tsa.acf(resid_3.values.squeeze(), qstat = True)
data = np.c_[range(1,41), r[1:], q, p]
table = pd.DataFrame(data, columns = ['lag', 'AC', 'Q',
"Prob(>Q)"])
print table.set_index ('lag')
```

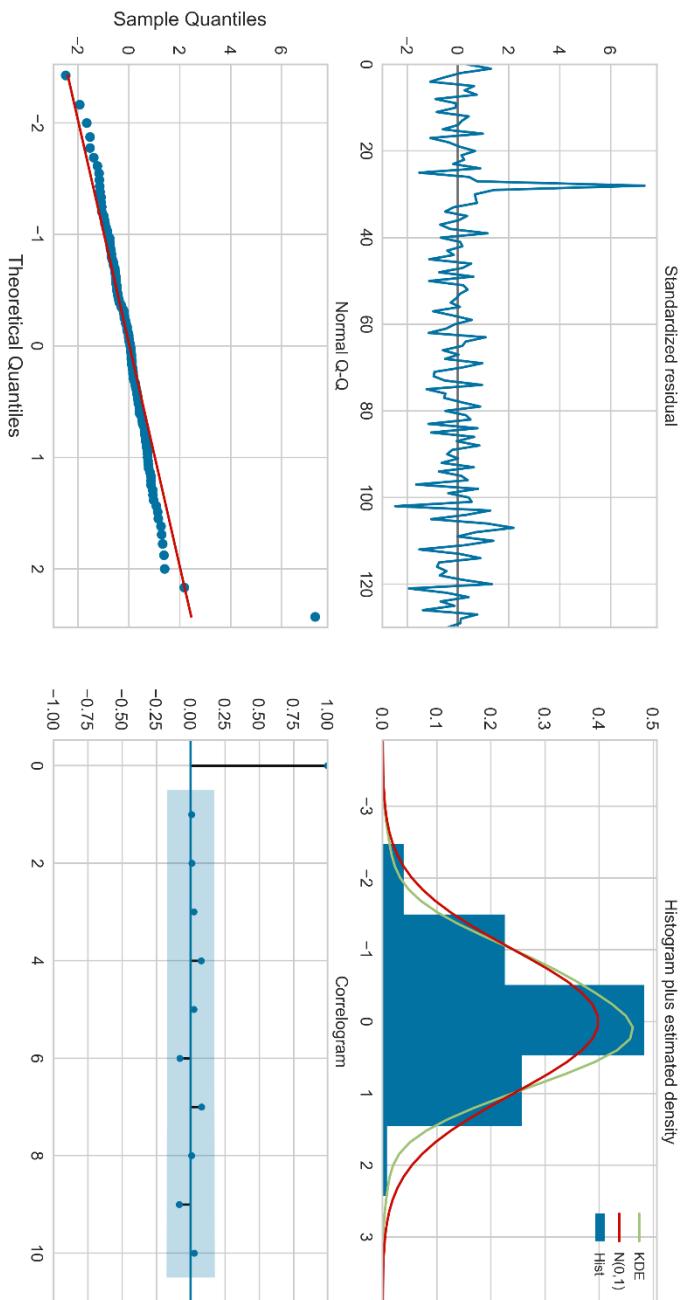


Figure 10-20. ARIMA model diagnostic plots

Out [28]

lag	AC	Q	Prob(>Q)
1	-0.052502	0.380459	5.373578e-01
2	-0.089685	1.498981	4.726072e-01
3	-0.031343	1.636628	6.511134e-01
4	-0.046713	1.944707	7.459281e-01
5	0.060760	2.469930	7.810167e-01
6	-0.012620	2.492765	8.692765e-01
7	-0.059041	2.996436	8.853317e-01
8	-0.054887	3.435159	9.041633e-01
9	0.031595	3.581684	9.367303e-01
10	-0.332166	19.906788	3.014674e-02
11	0.176572	24.557061	1.057854e-02
12	0.329212	40.853754	5.183810e-05
13	-0.022673	40.931685	9.774781e-05
14	-0.005088	40.935642	1.820674e-04
15	-0.087550	42.117018	2.156204e-04
16	-0.079394	43.096683	2.702335e-04
17	0.086414	44.267104	3.127767e-04
18	-0.090614	45.565054	3.432357e-04
19	0.003502	45.567010	5.705713e-04
20	0.132708	48.399397	3.738050e-04
21	-0.148798	51.991475	1.915056e-04
22	-0.149559	55.652507	9.588070e-05
23	0.177426	60.850894	2.875519e-05
24	0.164556	65.362792	1.076684e-05
25	0.120195	67.791849	8.144506e-06
26	0.027712	67.922159	1.324907e-05
27	-0.228150	76.836130	1.128476e-06
28	-0.019364	76.900944	1.914405e-06
29	-0.045064	77.255281	2.899787e-06
30	-0.074288	78.227360	3.541494e-06
31	0.121550	80.854778	2.506880e-06
32	0.044941	81.217443	3.698692e-06
33	-0.162566	86.009435	1.299424e-06
34	-0.031320	86.189067	2.030011e-06
35	-0.001899	86.189734	3.312200e-06
36	0.121705	88.956895	2.244679e-06
37	0.253170	101.053229	7.426355e-08
38	-0.157138	105.761315	2.642634e-08
39	-0.099640	107.674021	2.387589e-08
40	-0.063603	108.461592	3.123677e-08

Notice that the p-values for the Ljung–Box Q test all are well above 0.05 for lags 1 through 9 indicating “significance.” This is not a desirable result. However, the p-values for the remaining lags through 40 data values as less than 0.05. So, there is much data not contributing to correlations at high lags.

```
In [28]:  
print arma_mod_3.aic, arma_mod_3.bic, arma_mod_3.hqic
```

```
Out [28]  
2056.68132177 2079.92352 2066.12629956
```

Calculate Forecast Errors

We saw in Chapter 6 that the **Mean Absolute Percent Error (Bias)** is:

$$MAPE = \frac{1}{n} \sum_{t=1}^n \left| \frac{Actual_t - Fitted_t}{Actuals_t} \right|$$

We will use MAPE as our measure of goodness-of-fit here. We define the test at In[29]. To execute the test, our actuals and fitted must have the same dimensions, so we use the `shape()` function to evaluate this requirement.

```
In [29]:  
import numpy as np  
def MAPE(actual, pred):  
    actual, pred = np.array(actual), np.array(pred)  
    return np.mean(np.abs((actual - pred) / actual)) * 100
```

Now we determine the shape of our objects.

```
In [30]:  
test = np.array(predict_loans_3)  
ytrue.shape
```

```
Out[30]:  
(59,)
```

```
In [31]:  
ypred.shape
```

```
Out[31]:  
(59,)
```

```
In [32]:  
MAPE(ytrue, ypred)
```

```
Out[32]:  
9.184058754841384
```

The metric , MAPE = 9.4930%, shows a good fit for the ARIMA(6,1,0) model.

SARIMA Model with Exogenous Regressors

First, we generate subsample datasets, train and valid with the independent exogenous variables we chose. We use train (`trn`) for training the model and valid (`val`) for testing the predictions against the model. We do the same for the dependent variable, approve, where we train using `yt` and calculate predictions with `yv`.

```
In [33]
#creating the train and validation set
train = df[:int(0.7*(len(df)))]
valid = df[int(0.7*(len(df))):]
trn = np.asarray(train[['iraqwar', 'wtc', 'afghan', 'kabul']])
val = np.asarray(valid[['iraqwar', 'wtc', 'afghan', 'kabul']])
yt = train[['approve']]
yv = valid[['approve']]
```

Approval Forecasting

Intuitively, as time progresses, the effects of the WTC attack diminish and without other interventions, the approval ratings decline. We see a similar effect with Operation Iraqi Freedom, but to a lesser magnitude. Our prediction should show a continued decline at about the same rate.

```
In [34]
Ytrue = yt[131:190]
ypred = model_fit.predict(start = 131, end = 189, exog = val)
```

So, our plot in Figure 10-21 should show the declining ratings, and it does. The figure shows the observed values (blue), the out-of-sample predicted approval ratings (orange), along with the true (in-sample)approval ratings (green).

```
In [35]
# Plot object: time series, prediction, in-sample truth
fig, ax = plt.subplots(figsize=(8, 5))
fig = ypred[5:].plot(ax = ax, color = 'coral')
fig = ytrue[0:].plot(ax = ax, color = 'lightseagreen')
ax = yt[10:189].plot(ax = ax, color = 'dodgerblue')
```

```

# Plot title and axis labels
ax.set_title('Approval Rating for President George Bush
(2001-2004)',
             loc = 'center', fontsize = 16, color = 'slateblue',
             fontweight = 'bold')
plt.xlabel('Weeks', labelpad = 8, fontsize = 14,
           color = 'slateblue', fontweight = 'bold')
plt.ylabel('Approval Rate', fontsize = 14,
           color = 'slateblue', fontweight = 'bold')

# Plot tick marks a tick labels:
plt.xticks(np.arange(0, 200, 10))
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
ax.set_xticklabels(labels)

# Plot tick mark substitute labels
labels = [item.get_text() for item in ax.get_xticklabels()]
labels[4] = 'World Trade\n Center'
labels[12] = 'Operation Iraqi\n Freedom'
labels[18] = 'Forecast'

# Legend line colors and labels
colors = ['dodgerblue', 'coral', 'lightseagreen']
lines = [Line2D([0], [0], color = c, linewidth = 3,
               linestyle = '-') for c in colors]
labels = ['Approval Rating', 'Forecasted Rate',
          'Out of Sample Approval']
plt.legend(lines, labels, prop = {'size': 14})

# Save as PNG file and show plot
plt.savefig('bush_forecast.png', figsize = (5,3), dpi = 300,
            bbox_inches = 'tight')
plt.show()

```

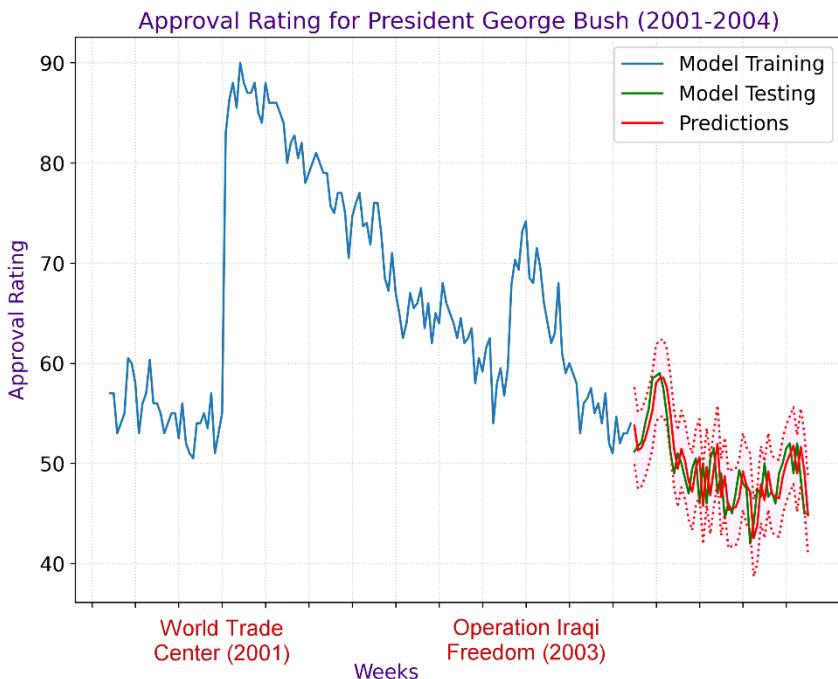


Figure 10-21. Forecast of the Presidential approval ratings with the impact of exogenous regressors

As one final measure, we look at the MSE between the true approval values and the predicted ones. We are looking for a low MSE value and we get one, 9.8029. So, we think we have a pretty good fit.

```
In [37]
from sklearn.metrics import mean_squared_error
mean_squared_error(y_true, ypred)
```

```
Out[37]:
9.802944943759462
```

Vector Autoregression (VAR) processes

Vector Autoregression (VAR) is a forecasting algorithm we can use when two or more time series influence each other. That means, the basic requirements in order to use VAR(p) are:

1. We need at least two time series (variables)
2. The time series should influence each other

In other words, the relationship between the time series involved is bi-directional.

In this section, we will see the concepts, intuition behind VAR models and see a comprehensive and correct method to train and forecast VAR models in python using [Statsmodels](#).

There is a primary difference between Vector Auto Regression (VAR) models and the ARIMA models we have been using, which are unidirectional, where, the predictors influence the Y and not vice-versa. Var is bi-directional, where the variables influence each other.

We are interested in modeling a $T \times K$ multivariate time series Y , where T denotes the number of observations and K the number of variables. One way of estimating relationships between the time series and their lagged values is the vector autoregression process:

$$Y_t = \nu + \beta_1 Y_{t-1} + \cdots + \beta_p Y_{t-k} + \varepsilon_t$$

Where $Y_{t-1} \dots Y_{t-k}$ are the lagged variables, β_i is a $K \times K$ matrix of coefficient lags, $\varepsilon_t \sim \text{Normal}(0, \Sigma_u)$ and errors.

In the VAR model, each variable is modeled as a linear combination of past values of itself and the past values of other variables in the system. Since we have multiple time series that influence each other, we model it as a system of equations with one equation per time series.

Suppose we have two time series, Y_1 and Y_2 , and you need to forecast the values of these variables at time (t). To calculate $Y_1(t)$, VAR will use the past values of both Y_1 as well as Y_2 . Likewise, to compute $Y_2(t)$, the past values of both Y_1 and Y_2 be used.

For example, the system of equations for a VAR(1) model with two time series (variables 'Y1' and 'Y2') is as follows:

$$Y_{1,t} = \nu_1 + \beta_{11,1} Y_{1,t-1} + \beta_{12,1} Y_{2,t-1} + \varepsilon_{1,t}$$

$$Y_{2,t} = \nu_2 + \beta_{21,1} Y_{1,t-1} + \beta_{22,1} Y_{2,t-1} + \varepsilon_{2,t}$$

Where, $Y_{1,t-1}$ and $Y_{2,t-1}$ are the first lag of time series Y_1 and Y_2 respectively.

The above equation is referred to as a VAR(1) model, because, each equation is of order $k = 1$, that is, it contains up to one lag of each of the predictors (Y_1 and Y_2). Since the Y terms in the equations are interrelated, the Y 's are considered as endogenous variables, rather than as exogenous predictors.

Likewise, the second order VAR(2) model for two variables would include up to two lags for each variable (Y_1 and Y_2).

$$Y_{1,t} = \nu_1 + \beta_{11,1}Y_{1,t-1} + \beta_{12,1}Y_{2,t-1} + \beta_{11,2}Y_{1,t-2} + \beta_{12,2}Y_{2,t-2} + \varepsilon_{1,t}$$

$$Y_{2,t} = \nu_2 + \beta_{21,1}Y_{1,t-1} + \beta_{22,1}Y_{2,t-1} + \beta_{21,2}Y_{1,t-2} + \beta_{22,2}Y_{2,t-2} + \varepsilon_{2,t}$$

Definition 10.3. *The Vector Autoregression (VAR) statistical model is given by a p -dimensional time Y_t of length $K + T$ satisfying a K^{th} order vector autoregressive equation*

$$Y_t = \sum_{l=1}^K \beta_l X_{t-l} + \mu D_t + \varepsilon_t$$

conditional on the initial values Y_0, \dots, Y_{1-K} , with a vector of deterministic terms D_t , and, $\varepsilon_t \sim \text{iid Normal}(0, \Sigma_\varepsilon)$ and errors.

The effective sample will remain Y_1, \dots, Y_T when discussing autoregressions with $k < K$ to allow comparison of likelihood values. The component D_t is a vector of deterministic terms such as a constant, a linear trend, or seasonal dummies. For the sake of defining a likelihood function it is initially assumed that the innovations, (ε_t) , are independently, identically distributed (IID) normal, $N_p(0, \Omega)$ errors.

The procedure to build a VAR model involves the following steps:

1. Analyze the time series characteristics
2. Test for causation amongst the time series
3. Test for stationarity
4. Transform the series to make it stationary, if needed
5. Find optimal order (p)
6. Prepare training and test datasets

7. Train the model
8. Roll back the transformations, if any.
9. Evaluate the model using test set
10. Forecast to future

Analyze the time series characteristics

We are using the *S&P Composite* dataset. The data set consists of monthly stock price, dividends, and earnings data and the consumer price index (to allow conversion to real values), all starting January 1871. We are only using the years 1992 through 2019, and have added the *Nominal Home Price Index* from *Historical Housing Market Data*.
<https://www.quandl.com/data/YALE-Yale-Department-of-Economics?page=2>.

The columns of the dataset consist of the dates, with a monthly frequency, and ten times series:

1. Month
2. S&P Composite
3. Dividend
4. Earnings
5. CPI
6. Long Interest Rate
7. Real Price
8. Real Dividend
9. Real Earnings
10. Cyclically Adjusted PE Ratio
11. Nominal Home Price Index

First, we assemble the tools we'll need:

```
In [38]
# Basic imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Import Statsmodels
from statsmodels.tsa.api import VAR
```

```
from statsmodels.tsa.stattools import adfuller
from statsmodels.tools.eval_measures import rmse, aic
from statsmodels.tsa.stattools import grangercausalitytests
```

Next, we load the dataset and take a look at what columns are available.

```
In [39]
df = pd.read_csv("D:\\Documents\\DATA\\nat_price_idx.csv")
list(df.columns)
```

```
Out [39]
['month', 'year', 'date', 'sp_composite', 'dividend',
'earnings', 'cpi', 'long_int_rate', 'real_price', 'real_div',
'real_earn', 'cyclic_adj', 'nom_home']
```

So, we will create a dataframe comprised of month, *real price*, *real dividend*, *long-term interest rate*, and *nominal home price index*. We will also index the month column for the time series.

```
In [40]
df =
df[['month','nom_home','real_earn','real_div','ln_int_rate']]
df_index = df.set_index('month', inplace = True)
df.head()
```

```
Out [40]
      nom_home    real_earn    real_div    cyclic_adj
month
1      75.69    30.26    23.09    19.77
2      75.65    30.30    23.08    19.58
3      75.81    30.27    23.04    19.28
4      76.08    30.78    23.00    19.30
5      76.40    31.26    22.97    19.66
```

Now, we want to see what the data looks like and we plot the four series in **Error! Reference source not found..**

```
In [41]
fig, axes = plt.subplots(nrows = 2, ncols = 2, dpi = 300,
 figsize = (8,6))
for i, ax in enumerate(axes.flatten()):
    data = df[df.columns[i]]
    ax.plot(data, color = 'red', linewidth = 1)
    ax.set_title(df.columns[i])
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
```

```

ax.spines['top'].set_alpha(0)
ax.tick_params(labelsize = 10)
plt.gcf().text(0.05, 1, 'Selected time series from the S&P
Composite', fontsize = 16, color = 'purple')
bbox_inches = 'tight')

```

All of the series have a fairly similar trend patterns over the years.

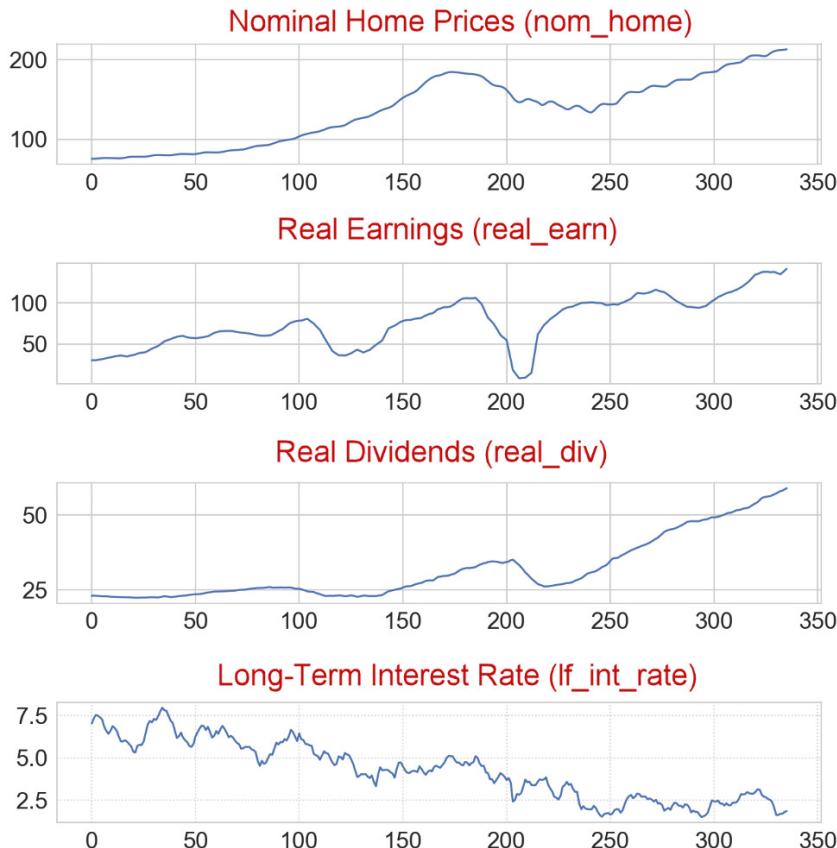


Figure 10-22. Times series plots for nominal home price, real earnings, real dividends, and the cyclic adjusted PE ratio

Fitting an Approval model

We are now ready to fit the time series, Presidential approval ratings, using a SARIMA model. Here, we make the same assumption as made with the ARIMA(4,0,1) model. We use SARIMA so that we can include

the exogenous regressors in the model for fitting their effect on approval.

```
In [42]
# fit model
model = SARIMAX(df.ts, exog=exogx, order=(4, 0, 1),
seasonal_order=(0, 0, 0, 0))
model_fit = model.fit(disp = False)
model_fit.summary()
```

Out[42]:

SARIMAX Results

Dep. Variable:	approve	No. Observations:	194			
Model:	SARIMAX(4, 0, 1)	Log Likelihood	-551.261			
Date:	Sun, 01 Nov 2020	AIC	1122.523			
Time:	21:14:00	BIC	1155.201			
Sample:	0 - 194	HQIC	1135.755			
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
iraqwar	0.0004	0.000	2.261	0.024	5.74e-05	0.001
wtc	0.0004	4.41e-05	9.575	0.000	0.000	0.001
afghan	3.548e-06	0.000	0.014	0.989	-0.001	0.001
kabul	-0.0003	0.000	-1.200	0.230	-0.001	0.000
ar.L1	-0.7852	0.381	-2.060	0.039	-1.532	-0.038
ar.L2	0.1826	0.096	1.895	0.058	-0.006	0.371
ar.L3	0.9065	0.048	18.820	0.000	0.812	1.001
ar.L4	0.6952	0.368	1.889	0.059	-0.026	1.417
ma.L1	0.9113	0.373	2.444	0.015	0.181	1.642
sigma2	19.0564	2.226	8.562	0.000	14.694	23.419
Ljung-Box (Q):	84.45	Jarque-Bera (JB):	710.22			
Prob(Q):	0.00	Prob(JB):	0.00			
Heteroskedasticity (H):	0.38	Skew:	1.85			
Prob(H) (two-sided):	0.00	Kurtosis:	11.61			

In the results table (Out [34]), the MA coefficients represent a linear combination of the exogenous impacts on the endogenous variable at the corresponding lag. Note that there is no significant constant in the model.

Our explanatory variables, $x_1 = \text{'iraqwar'}$, $x_2 = \text{'wtc'}$, $x_3 = \text{'afghan'}$, and $x_4 = \text{'kabul'}$ are assumed to be exogenous (or predetermined), that is

we assume they are not to be correlated with the approval ratings, or not with the approval rating in the current period, which is a reason to use the lag. The impact of these exogenous variables on the expected value (conditional mean) of the dependent variable is given by the corresponding four coefficients.

The MA term refers to the external shocks to the system, and reflects the impact of all other factors that are not included in the model and are part of the residuals. In our case, there is only one shock, which is '[ma.L1](#)', of the WTC attack on approval ratings.

In general, AR terms introduce autocorrelation that can last for many periods until they become negligible. Pure MA terms, which we do not have, would have a short run impact (up to the given lag). When both AR and MA terms are present, then it gets a bit more complicated and the best way to see the dynamic impact is to look at the impulse response function or similar diagnostic.

Lagged impact (effects) of explanatory [exog](#) variables are included, namely '[ar.L1](#)', and '[ar.L3](#)'. The best way to understand this version of ARMAX with X is to think of a linear model (like in ordinary least squares) $y = xb$, where the deviations from this linear relationship follow an ARMA process.

So, we could write an equation for our model as

$$Y_i = \beta_0 + \beta_1 X_{1i} + \cdots + \beta_k X_{ki} + \beta_{k+1} W_{1i} + \cdots + \beta_{k+r} W_{ri} + u_i$$

$$Y_t = 0.0004x_1 + 0.004x_2 - 0.7852w_1 + 0.9065w_3 + 0.9113\mu_1$$

[*Test for Causation Among the Series*](#)

Our next step is to analyze the series for causality among them. We use the *Granger's Causality Test* and the *Cointegration Test* to help us with that.

[Granger's Causality Test](#)

The basis behind VAR(k) is that each of the time series in the system influences each other. In other words, we can predict the series with past values of itself along with other series in the system. Using *Granger's Causality Test*, we can test these relationships before building

the model. Granger's causality test evaluates the null hypothesis that the coefficients of past values in the regression equation are zero.

In simpler terms, the past values of time series (Y_1) do not cause the other series (Y_2, \dots, Y_k). So, if the p -value obtained from the test is less than the significance level of 0.05, then, we can safely reject the null hypothesis. So, we define a test that implements the Granger's Causality Test for all possible combinations of the time series in a given dataframe and stores the k -values of each combination in the output matrix.

```
In [43]
maxlag = 12
test = 'ssr_chi2test'
def grangers_causation_matrix(data, variables,
                               test = 'ssr_chi2test', verbose = False):
    """Check Granger Causality of all possible combinations of
    the Time series. The rows are the response variable, columns are
    predictors. The values in the table are the P-Values. P-Values
    lesser than the significance level (0.05), implies the Null
    Hypothesis that the coefficients of the corresponding past
    values are zero, that is, the X does not cause Y can be
    rejected.
    data: pandas dataframe containing the time series variables
    variables: list containing names of the time series variables
    """
    df = pd.DataFrame(np.zeros((len(variables),
                                len(variables))), columns = variables, index = variables)
    for c in df.columns:
        for r in df.index:
            test_result = grangercausalitytests(data[[r, c]],
                                                 maxlag = maxlag, verbose = False)
            p_values = [round(test_result[i+1][0][test][1],4)
                        for i in range(maxlag)]
            if verbose: print(f'Y = {r}, X = {c},
                               P Values = {p_values}')
            min_p_value = np.min(p_values)
            df.loc[r, c] = min_p_value
    df.columns = [var + '_x' for var in variables]
    df.index = [var + '_y' for var in variables]
    return df
```

Now that we have the definition, we implement it for our data.

```
In [44]
grangers_causation_matrix(df, variables = df.columns)
```

```
Out [44]
      nom_home_x    real_earn_x    real_div_x    cyclic_adj_x
nom_home_y    1.0000  0.0000  0.0000  0.0000
real_earn_y    0.0085  1.0000  0.0000  0.0003
real_div_y    0.0000  0.0000  1.0000  0.0260
lg_int_rate_y 0.1221  0.0004  0.0100  1.0000
```

The rows are the responses (Y s) and the columns are the predictor series (X s). For example, if you take the value 0.0000 in (row 1, column 2), it refers to the p-value of real earning causing nominal home prices. Since the corresponding p -value of 0.0000 is less than the significance level of 0.05, we reject the null hypothesis and conclude that real earning causing nominal home prices . Also, the 0.0085 in (row 2, column 1), refers to the p -value of nominal home prices causing real earnings.

So, if a given p-value is less than ($<$) significance level (0.05), then, the corresponding X series (column) causes the Y (row). Then, we can reject the null hypothesis and conclude X causes Y .

Looking at the p -values in the above table, we can pretty much observe that all the variables (time series) in the system are interchangeably causing each other, even though the p-value of 0.1221 is greater than the level of significance (this a modeler's call). This makes this system of multi time series a good candidate for using VAR models to forecast.

Next, we will perform the Cointegration test.

Cointegration Test

Cointegration test helps us establish the presence of a statistically significant connection between two or more time series. But, what does cointegration mean?

To understand that, we first need to know what is 'order of integration' (d). **Order of integration** (k) is the number of differencing required to make a non-stationary time series stationary.

Now, when we have two or more time series, and there exists a linear combination of them that has an order of integration (k) that is less than

that of the individual series, then the collection of series is said to be cointegrated. When two or more time series are **cointegrated**, it means they have a long run, statistically significant relationship.

This is the basic premise on which Vector Autoregression (VAR) models is based on. So, it's fairly common to implement the cointegration test before starting to build VAR models. We will do it now.

Johanssen (1991) devised a procedure to implement the cointegration test. It is fairly straightforward to implement with Python's *statsmodels*, as we can see below.

```
[In [45]
from statsmodels.tsa.vector_ar.vecm import coint_johansen

def cointegration_test(df, alpha):
    """Perform Johanson's Cointegration Test and Report
    Summary"""
    out = coint_johansen(df,-1,5)
    d = {'0.9':0, '0.95':1, '0.99':2}
    traces = out.lrt
    cvts = out.cvt[:, d[str(1-alpha)]]
    def adjust(val, length = 6): return str(val).ljust(length)

    # Summary
    print('Name :: Test Stat >
          C((1-alpha)*100),% => Signif \n', '--'*20)
    for col, trace, cvt in zip(df.columns, traces, cvts):
        print(adjust(col), ':: ', adjust(round(trace,2), 9),
              ">, adjust(cvt, 8), '=> ', trace > cvt)Name :: 
        Test Stat > C(95%) => Signif
```

Now that we have defined the test, we will run it at the 0.1 level of significance.

```
In [46]
cointegration_test(df, 0.1)
```

```
Out [46]
Name :: Test Stat > C( 0.9      %)    => Signif
-----
nom_home ::  40.03      > 37.0339    =>   True
real_earn ::  16.02      > 21.7781    =>   False
```

```
real_div :: 7.54      > 10.4741    =>  False
lg_int_rate :: 0.12     > 2.9762    =>  False
```

Test for Stationarity

As we have previously discussed, the VAR model requires the time series we want to forecast to be stationary, it is customary to check all the time series in the system for stationarity. Again, a stationary time series is one whose characteristics like mean and variance does not change over time. To test for this in Python, we will use the **Augmented Dickey-Fuller Test** (ADF Test).

We have also discussed that if a series is found to be non-stationary, we can make it stationary by differencing the series once and repeat the test again until it becomes stationary. If we choose difference any of the series in a system, we have to apply the same differencing to all the series in the system, since differencing reduces the length of the series by one (for one difference).

User Defined ADF Test In Python

First, we implement a nice function (`adfuller_test()`) that writes out the results of the ADF test for any, given time series, and implement this function on each series one-by-one. The code at In[56] defines an algorithm that executes ADF Test.

```
In [47]
from statsmodels.tsa.stattools import adfuller
def adfuller_test(series, signif = 0.05, name = '',
                  verbose = False):
    """Perform ADFuller to test for Stationarity of given series
    and print report"""
    r = adfuller(series, autolag='AIC')
    output = {'test_statistic':round(r[0], 4),
              'pvalue':round(r[1], 4), 'n_lags':round(r[2], 4),
              'n_obs':r[3]}
    p_value = output['pvalue']
    def adjust(val, length= 6): return str(val).ljust(length)

    # Print Summary
    print(f'    Augmented Dickey-Fuller Test on "{name}"',
          f'\n    ', '-'*47)
    print(f' Null Hypothesis: Data has unit root.
```

```

        Non-Stationary.')
print(f' Significance Level      = {signif}')
print(f' Test Statistic          = {output["test_statistic"]}')
print(f' No. Lags Chosen         = {output["n_lags"]}')

for key, val in r[4].items():
    print(f' Critical value {adjust(key)} =
          {round(val, 3)})')
if p_value <= signif:
    print(f' => P-Value = {p_value}.
          Rejecting Null Hypothesis.')
    print(f' => Series is Stationary.')
else:
    print(f' => P-Value = {p_value}.
          Weak evidence to reject the Null Hypothesis.')
print(f' => Series is Non-Stationary.')

```

After we run this code, we use it on each of the series, which occurs automatically with the code at In[48].

```

In [48]
# ADF Test on each column
for name, column in df_train.iteritems():
    adfuller_test(column, name = column.name)
    print('\n')

```

```

Out [48]
Augmented Dickey-Fuller Test on "nom_home"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -0.9701
No. Lags Chosen         = 13
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.764. Weak evidence to reject the Null
Hypothesis.
=> Series is Non-Stationary.

Augmented Dickey-Fuller Test on "real_earn"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -2.5201
No. Lags Chosen         = 13
Critical value 1%       = -3.452

```

```
Critical value 5%      = -2.871
Critical value 10%     = -2.572
=> P-Value = 0.1107. Weak evidence to reject the Null
Hypothesis.
=> Series is Non-Stationary.
```

```
Augmented Dickey-Fuller Test on "real_div"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic         = 0.3983
No. Lags Chosen       = 4
Critical value 1%     = -3.452
Critical value 5%      = -2.871
Critical value 10%     = -2.572
=> P-Value = 0.9814. Weak evidence to reject the Null
Hypothesis.
=> Series is Non-Stationary.
```

```
Augmented Dickey-Fuller Test on "lg_int_rate"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic         = -1.558
No. Lags Chosen       = 5
Critical value 1%     = -3.452
Critical value 5%      = -2.871
Critical value 10%     = -2.572
=> P-Value = 0.5047. Weak evidence to reject the Null
Hypothesis.
=> Series is Non-Stationary.
```

The null hypothesis for the ADF Test is that the series is stationary, so a p-value less than our level of significance, then we reject the non-stationary hypothesis. The output at Out[48] tells us that all of our series are not stationary. We should always check the other series, because if we apply a difference one, we have to apply the same difference to all the series.

First-Order Differencing

So, we will difference all of the series once and check for stationarity.

```
In [49]
df_differenced = df_train.diff().dropna()
# ADF Test on each column of 1st Differences Dataframe
for name, column in df_differenced.iteritems():
    adfuller_test(column, name = column.name)
```

```
print('\n')

Out [49]
Augmented Dickey-Fuller Test on "nom_home"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -2.4997
No. Lags Chosen         = 12
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.1156. Weak evidence to reject the Null Hypothesis.
=> Series is Non-Stationary.

Augmented Dickey-Fuller Test on "real_earn"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -3.8183
No. Lags Chosen         = 12
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0027. Rejecting Null Hypothesis.
=> Series is Stationary.

Augmented Dickey-Fuller Test on "real_div"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -3.4653
No. Lags Chosen         = 3
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0089. Rejecting Null Hypothesis.
=> Series is Stationary.

Augmented Dickey-Fuller Test on "lg_int_rate"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -8.591
No. Lags Chosen         = 4
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0. Rejecting Null Hypothesis.
=> Series is Stationary.
```

The results show us that differencing once fixed the all series, except *nominal home price index*, which is not stationary. So, let's apply a second difference and recheck.

Since at least one series is not stationary, we are left with one of two choices: (1) either proceed with 1st differenced series or (2) difference all the series one more time.

Second-Order Differencing

We have decided to apply a second difference and re-run ADF test again on each second differenced series.

```
In [50]
df_differenced = df_differenced.diff().dropna()
# ADF Test on each column of 2nd Differences Dataframe
for name, column in df_differenced.iteritems():
    adfuller_test(column, name = column.name)
    print('\n')
```

```
Out [50]
Augmented Dickey-Fuller Test on "nom_home"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -4.4912
No. Lags Chosen         = 11
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0002. Rejecting Null Hypothesis.
=> Series is Stationary.

Augmented Dickey-Fuller Test on "real_earn"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -7.5352
No. Lags Chosen         = 11
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0. Rejecting Null Hypothesis.
=> Series is Stationary.

Augmented Dickey-Fuller Test on "real_div"
```

```

Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -13.875
No. Lags Chosen         = 2
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0. Rejecting Null Hypothesis.
=> Series is Stationary.

Augmented Dickey-Fuller Test on "lg_int_rate"
-----
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level      = 0.05
Test Statistic          = -9.8119
No. Lags Chosen         = 9
Critical value 1%       = -3.452
Critical value 5%       = -2.871
Critical value 10%      = -2.572
=> P-Value = 0.0. Rejecting Null Hypothesis.
=> Series is Stationary.

```

So, we solved our stationarity problem.

Find the Optimal Order (K)

To select the right order of the VAR model, we iteratively fit increasing orders of the VAR model and pick the order that gives a model with least AIC. Though the usual practice is to look at the AIC, we can also check other best fit comparison estimates of BIC, FPE and HQIC.

```

In [51]
model = VAR(df_differenced)
for i in [1,2,3,4,5,6]:
    result = model.fit(i)
    print('Lag Order =', i)
    print('AIC : ', result.aic)
    print('BIC : ', result.bic)
    print('FPE : ', result.fpe)
    print('HQIC: ', result.hqic, '\n')

```

```

Out [51]
Lag Order = 0
AIC : -7.555536878502039
BIC : -7.507323042405533
FPE : 0.0005232052157190554
HQIC: -7.536263023760307

```

```
Lag Order = 1
AIC : -8.036193908453379
BIC : -7.794553696356437
FPE : 0.0003235416809337151
HQIC: -7.939585690892763

Lag Order = 2
AIC : -8.340971608558112
BIC : -7.904985919639123
FPE : 0.00023855634208495396
HQIC: -8.166644426678662

Lag Order = 3
AIC : -8.582764801823753
BIC : -7.951507202883903
FPE : 0.00018734436642101776
HQIC: -8.330330971869593

Lag Order = 4
AIC : -8.58205167491718
BIC : -7.7545883189277625
FPE : 0.00018752590184511962
HQIC: -8.25112039708097

Lag Order = 5
AIC : -8.64896868555575
BIC : -7.6243582290474805
FPE : 0.00017546064506217957
HQIC: -8.239146009714968

Lag Order = 6
AIC : -8.819404474861834
BIC : -7.596697994136102
FPE : 0.0001480566871977318
HQIC: -8.330293265839071
```

In the above output, the AIC increase from no lag to one, and continues to do so. However, BIC starts to decrease after lag 3. So, we can model with lag 3, or look at an alternative method.

An alternate method to choose the order(k) of the VAR models is to use the `model.select_order(maxlags)` method. The selected order(k) is the order that gives the lowest ‘AIC’, ‘BIC’, ‘FPE’ and ‘HQIC’ scores.

```
In [52]
x = model.select_order(maxlags = 12)
x.summary()
```

Out [52]

VAR Order Selection (* highlights the minimums)

	AIC	BIC	FPE	HQIC
0	-7.430	-7.381	0.0005930	-7.411
1	-7.927	-7.679	0.0003608	-7.828
2	-8.253	-7.807	0.0002604	-8.075
3	-8.508	-7.862*	0.0002020	-8.249
4	-8.515	-7.671	0.0002006	-8.177
5	-8.579	-7.536	0.0001883	-8.161
6	-8.761	-7.520	0.0001570	-8.264
7	-8.777	-7.338	0.0001546	-8.201
8	-8.909	-7.272	0.0001356	-8.254
9	-9.055	-7.219	0.0001174	-8.320
10	-9.080	-7.045	0.0001148	-8.265
11	-9.124	-6.891	0.0001100	-8.230
12	-9.379*	-6.947	8.554e-05*	-8.405*

According to AIC and BIC, the optimal lag is observed at a lag order of 3 based on BIC and lag 12 for HQIC. However, the minimum FPE has not been reached, and does not do so until around 17 lags, and AIC never converges. We do not have an explanation for why the observed BIC values differ so much from FPE, nor why AIC does not converge. So, we are comfortable is using a lag of 12 when we model.

Prepare Training and Testing Datasets

We have discussed one way to split data into training and test sets. Now we do it another way, a refit the VAR model using the set `df_train` and the model to forecast the next 12 (weeks) observations. These forecasts will be compared against the actuals present in test data. To do the comparisons, we will use multiple forecast accuracy metrics.

```
In [53]
nobs = 12
df_train, df_test = dfts[0:-nobs], dfts[-nobs:]
# Check size
print(df_train.shape)
print(df_test.shape)
```

Out [53]

(312, 4)
(24, 4)

Next, we check for stationarity and make the time series stationary, if necessary.

Train the VAR model

To estimate a VAR model, we must first use `NumPy` to create the time series using a `np.asarray()` of homogeneous or structured `dtype`. When using a structured or record array, the class will use the passed variable names. Otherwise they can be passed explicitly.

```
In [54]
model_fitted = model.fit(4)
model_fitted.summary()
```

Then, we run the model fit and call for a summary of the results. We have highlighted the significant results in Out [39], where $L1, \dots, L4$ refer to the 1st, 2nd, and 3rd lags.

```
Out[54]:
Summary of Regression Results
=====
Model:      VAR
Method:     OLS
Date:       Sun, 08, Nov, 2020
Time:       8:29:39
-----
No. of Equations:   4          BIC:        7.95151
Nobs:            307         HQIC:      -8.33033
Log likelihood: -373.002      FPE:        0.000187344
AIC:             -8.58276    Det(Omega_mle): 0.000158
-----
Results for equation nom_home
=====
      coefficient  std. error    t-stat  prob
-----
const      0.002057  0.018507    0.111  0.911
L1.nom_home  0.564987  0.056462   10.007  0
L1.real_earn  0.015329  0.012993    1.18   0.238
L1.real_div  -0.670527  0.147139   -4.557  0
L1.lg_int_rate -0.045211  0.082075   -0.551  0.582
L2.nom_home   0.062762  0.0615     1.021  0.307
L2.real_earn  -0.076511  0.012959   -5.904  0
L2.real_div  -0.450131  0.148252   -3.036  0.002
L2.lg_int_rate -0.128086  0.083694   -1.53   0.126
L3.nom_home   -0.323282  0.050649   -6.383  0
L3.real_earn   0.020929  0.013942    1.501  0.133
L3.real_div   -0.07224   0.154981   -0.466  0.641
L3.lg_int_rate  0.06334   0.079603    0.796  0.426
```

=====

Results for equation real_earn

=====

	coefficient	std. error	t-stat	
	prob			
const	0.005244	0.07885	0.067	0.947
L1.nom_home	-0.372979	0.240561	-1.55	0.121
L1.real_earn	-0.004504	0.05536	-0.081	0.935
L1.real_div	-0.501103	0.626901	-0.799	0.424
L1.lg_int_rate	-0.527342	0.34969	-1.508	0.132
L2.nom_home	0.106195	0.262027	0.405	0.685
L2.real_earn	-0.000587	0.055214	-0.011	0.992
L2.real_div	-1.723213	0.631644	-2.728	0.006
L2.lg_int_rate	0.181047	0.356587	0.508	0.612
L3.nom_home	0.04219	0.215797	0.196	0.845
L3.real_earn	-0.350946	0.0594	-5.908	0
L3.real_div	-0.488603	0.660311	-0.74	0.459
L3.lg_int_rate	0.206479	0.339157	0.609	0.543

=====

Results for equation real_div

=====

	coefficient	std. error	t-stat	prob
const	0.001419	0.007308	0.194	0.846
L1.nom_home	-0.002736	0.022297	-0.123	0.902
L1.real_earn	-0.009094	0.005131	-1.772	0.076
L1.real_div	-0.084372	0.058106	-1.452	0.146
L1.lg_int_rate	-0.002028	0.032412	-0.063	0.95
L2.nom_home	-0.026879	0.024287	-1.107	0.268
L2.real_earn	0.002079	0.005118	0.406	0.685
L2.real_div	-0.214345	0.058545	-3.661	0
L2.lg_int_rate	-0.006658	0.033051	-0.201	0.84
L3.nom_home	0.018054	0.020002	0.903	0.367
L3.real_earn	0.006313	0.005506	1.147	0.252
L3.real_div	-0.216752	0.061202	-3.542	0
L3.lg_int_rate	-0.004966	0.031435	-0.158	0.874

=====

Results for equation lg_int_rate

=====

	coefficient	std. error	t-stat	prob
const	0.000866	0.013098	0.066	0.947
L1.nom_home	-0.066294	0.03996	-1.659	0.097
L1.real_earn	0.008898	0.009196	0.968	0.333
L1.real_div	-0.311911	0.104137	-2.995	0.003
L1.lg_int_rate	-0.482765	0.058088	-8.311	0
L2.nom_home	0.066831	0.043526	1.535	0.125
L2.real_earn	0.024863	0.009172	2.711	0.007
L2.real_div	-0.106589	0.104925	-1.016	0.31
L2.lg_int_rate	-0.451265	0.059234	-7.618	0

L3.nom_home	-0.085865	0.035847	-2.395	0.017
L3.real_earn	-0.015403	0.009867	-1.561	0.119
L3.real_div	-0.12318	0.109686	-1.123	0.261
L3/lg_int_rate	-0.126301	0.056339	-2.242	0.025
<hr/>				
Correlation matrix of residuals				
	nom_home	real_earn	real_div	lg_int_rate
nom_home	1.000000	0.059250	-0.198828	0.19030
real_earn	0.059250	1.000000	-0.082090	-0.02392
real_div	-0.198828	-0.082090	1.000000	-0.02879
lg_int_rate	0.190303	-0.023920	-0.028797	1.000000

For the results shown in the correlation matrix of residuals, we are going to take values between -0.1 and 0.1 as uncorrelated. We have heighted them in Out[54]. Thus, the correlated series are `nom_home` with `real_div` and `lg_int_rate`. Even so, the correlation is not high.

Check for Serial Correlation of Residuals (Errors)

We use serial correlation of residuals to check if there is any leftover pattern in the residuals (errors). If there is any correlation left in the residuals, then, there is some pattern in the time series that is still remaining to be explained by the model. In that case, our typical course of action is to either increase the order of the model or induce more predictors into the system, or look for a different algorithm to model the time series. So, checking for serial correlation is how we ensure that the model is sufficiently able to explain the variances and patterns in the time series.

A common way of checking for serial correlation of errors can be measured using the Durbin-Watson statistic that we defined in Definition 10.2. The value of this statistic can vary between 0 and 4. The closer it is to the value 2, then there is no significant serial correlation. The closer to 0, there is a positive serial correlation, and the closer it is to 4 implies negative serial correlation.

```
In [55]
from statsmodels.stats.stattools import durbin_watson
out = durbin_watson(model_fitted.resid)

for col, val in zip(df.columns, out):
    print(adjust(col), ':', round(val, 2))
```

```
out [55]
nom_home :    2.1
real_earn :   2.03
real_div :   2.02
lg_int_rate : 2.04
```

The serial correlation seems okay, so we will proceed with the forecast.

In order to forecast, the VAR model expects up to the lag order number of observations from the past data, which is three for this case. This is due the terms in the VAR model are essentially the lags of the various time series in the dataset. So, we need to provide it as many of the previous values as indicated by the lag order used by the model.

```
In [ 56]
# Get the lag order
lag_order = model_fitted.k_ar
print(lag_order) #> 3

# Input data for forecasting
forecast_input = df_differenced.values[-lag_order:]
forecast_input

Out [56]
3
array([[-0.2 ,  0.57,  0.24,  0.17],
       [ 0.1 , -0.08, -0.03, -0.17],
       [ 0.03,  0.07,  0.03,  0.06]])
```

Forecast VAR model

Now we are ready for the “true” forecast.

```
In [57]
fc = model_fitted.forecast(y = forecast_input, steps=nobs)
df_forecast = pd.DataFrame(fc, index = df.index[-nobs:], 
                           columns = df.columns + '_2d')
df_forecast
```

```
Out [57]
```

	nom_home_2d	real_earn_2d	real_div_2d	lg_int_rate_2d
month				
313	0.167442	-2.014208e-01	-0.054553	0.025637
314	0.053710	-5.642327e-01	-0.008642	-0.061698
315	0.063970	1.849108e-01	0.010429	0.025558
316	0.039846	2.774986e-02	0.009343	-0.012980

317	-0.035511	3.014604e-01	0.003612	0.015998
318	-0.050330	-5.064255e-02	-0.004178	-0.011020
319	-0.066807	-1.044734e-02	0.003253	0.000671
320	-0.020538	-7.645926e-02	0.006252	-0.001731

Invert the Transformation to Get the Real Forecast

The forecasts are generated but it is on the scale of the training data used by the model. So, to bring it back up to its original scale, you need to de-difference it as many times you had differenced the original input data. In this case it is two times. Invert the transformation to get the real forecast.

```
In [58]
df_differenced

def invert_transformation(df_train, df_forecast,
    second_diff = False):
    """Revert back the differencing to get the forecast to
    original scale."""
    df_fc = df_forecast.copy()
    columns = df_train.columns
    for col in columns:
        # Roll back 2nd Diff
        if second_diff:
            df_fc[str(col)+'_1d'] = (df_train[col].iloc[-1] -
                df_train[col].iloc[-2]) +
                df_fc[str(col) + '_2d'].cumsum()
        # Roll back 1st Diff
        df_fc[str(col) + '_forecast'] = df_train[col].iloc[-1] +
            df_fc[str(col) + '_1d'].cumsum()
    return df_fc

# Perform transformation
df_results = invert_transformation(df_train, df_forecast,
    second_diff=True)
df_results.loc[:,['nom_home_forecast','real_earn_forecast','real
    _div_forecast','lg_int_rate_forecast']]
```

Plot of Forecast vs Actuals

Now, we plot the fitted values (`results`) for each regressor in Figure 10-23.

```
In [59]
fig, axes = plt.subplots(nrows = int(len(df.columns)/2),
```

```

    ncols = 2, dpi = 150, figsize= (10,10))
for i, (col,ax) in enumerate(zip(df.columns, axes.flatten())):
    df_results[col+'_forecast'].plot(legend = True,
    ax=ax).autoscale(axis = 'x',tight = False)
    df_train[col][0:][0:].plot(legend = True, ax = ax)
    df_test[col][-nobs:][0:].plot(legend = True, ax = ax)
    ax.set_title(col + " : Forecast vs Actuals")
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
    ax.spines[ 'top'].set_alpha(0)
    ax.tick_params(labelsize = 12)
plt.tight_layout();

```

S&P Composite time series and 24-month predictions

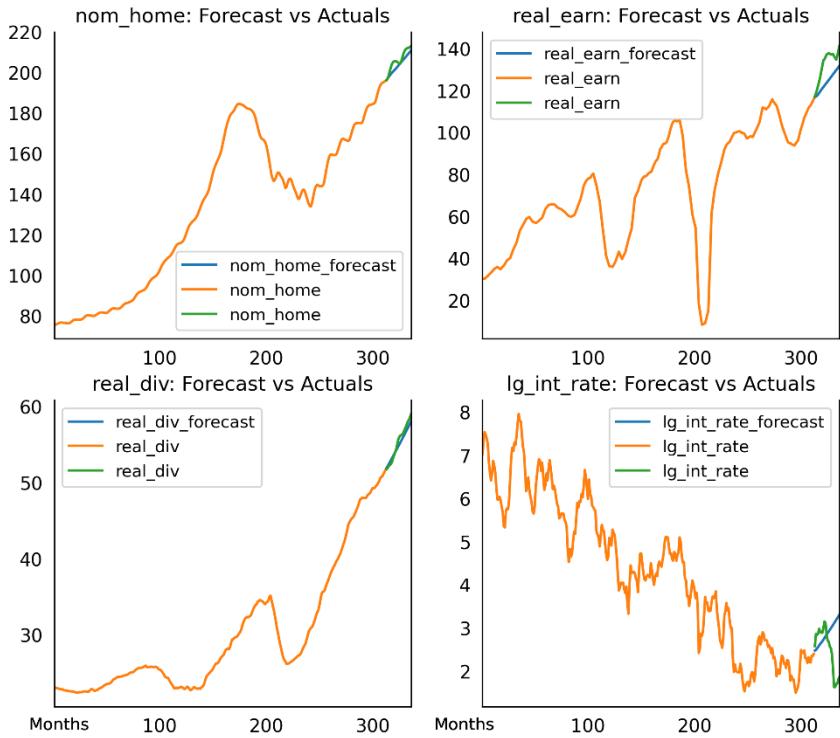


Figure 10-23. Multivariate time series VAR 24-month forecasts

Evaluate the Model using the Test Set

Now we turn to ensuring that our model adequately fits the data using the measures we defined.

```

In [60]
from statsmodels.tsa.stattools import acf
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual))
    me   = np.mean(forecast - actual)
    mae  = np.mean(np.abs(forecast - actual))
    mpe  = np.mean((forecast - actual)/actual)
    rmse = np.mean((forecast - actual)**2)**.5
    corr = np.corrcoef(forecast, actual)[0,1]
    mins = np.amin(np.hstack([forecast[:,None],
                               actual[:,None]]), axis = 1)
    maxs = np.amax(np.hstack([forecast[:,None],
                               actual[:,None]]), axis = 1)
    minmax = 1 - np.mean(mins/maxs)
    return({'mape':mape, 'me':me, 'mae': mae,
           'mpe': mpe, 'rmse':rmse, 'corr':corr,
           'minmax':minmax})

print('Forecast Accuracy of: nom_home')
accuracy_prod =
forecast_accuracy(df_results['nom_home_forecast'].values,
                  df_test['nom_home'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: real_earn')
accuracy_prod =
forecast_accuracy(df_results['real_earn_forecast'].values,
                  df_test['real_earn'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: real_div')
accuracy_prod =
forecast_accuracy(df_results['real_div_forecast'].values,
                  df_test['real_div'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: lg_int_rate')
accuracy_prod =
forecast_accuracy(df_results['lg_int_rate_forecast'].values,
                  df_test['lg_int_rate'])
for k, v in accuracy_prod.items():

```

```
print(adjust(k), ': ', round(v,4))
```

```
Out [60]
Forecast Accuracy of: nom_home
mape   :  0.0119
me     : -2.4293
mae    :  2.4649
mpe    : -0.0117
rmse   :  2.85
corr   :  0.9528
minmax :  0.0119

Forecast Accuracy of: real_earn
mape   :  0.0631
me     : -8.5105
mae    :  8.5105
mpe    : -0.0631
rmse   :  9.3142
corr   :  0.8559
minmax :  0.0631

Forecast Accuracy of: real_div
mape   :  0.0097
me     : -0.3433
mae    :  0.5458
mpe    : -0.0059
rmse   :  0.6184
corr   :  0.9931
minmax :  0.0097

Forecast Accuracy of: lg_int_rate
mape   :  0.3126
me     :  0.3614
mae    :  0.6241
mpe    :  0.2232
rmse   :  0.8183
corr   : -0.8312
minmax :  0.2009
```

Next, we plot the time series autocorrelation function in Figure 10-24. Although it may not be readily apparent, y_1 , the approval ratings, is the upper left plot. Then the other regressors follow suit along the top row in succession and left column. We added a line across the diagonal, and this gives us a graphical representation of the correlation matrix we just discussed.

```
In [61]
results.plot_acorr()
```

ACF plots for residuals with $2/\sqrt{T}$ bounds

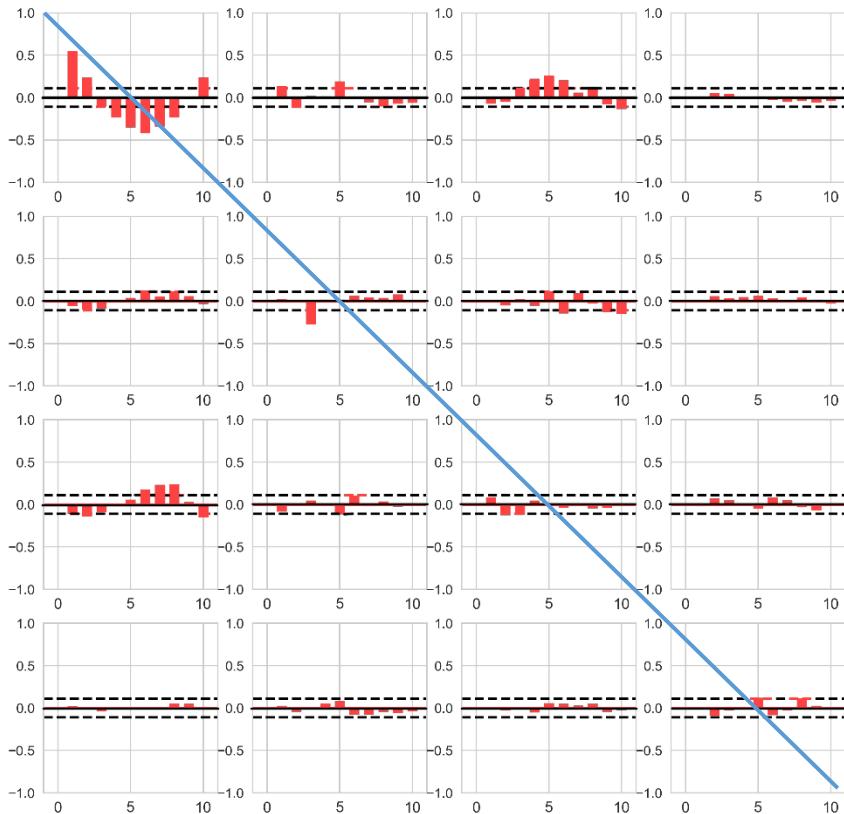


Figure 10-24. Approval rating model ACF plots for residuals with $2/\sqrt{T}$ bounds

Lag order selection

Choice of lag order can be a difficult problem. Standard analysis employs likelihood test or information criteria-based order selection. We have implemented the latter, accessible through the VAR class:

```
In [62]:  
model.select_order(62)
```

```
Out[62]:  
<statsmodels.tsa.vector_ar.var_model.LagOrderResults object.  
Selected orders are: AIC -> 12, BIC -> 9, FPE -> 12, HQIC -> 9>
```

When calling the fit function, we can pass a maximum number of lags and the order criterion to use for order selection. In this instance, we use a lag of 12, the maximum of the selected orders in Out [42].

```
In [63]:  
results = model.fit(maxlags=12, ic='aic')
```

Having run the model to fit the series, we now want to plot the new results, which we do in Figure 10-25. Notice that in the plotting code at In [44], we have passed several *Matplotlib* plotting parameters. We need to take care when doing this, since they are global parameters, and will remain until we change them elsewhere. We also used a plot style with the `plt.style.use()` function. We can see the available styles using `print(plt.style.available)`, which includes several *Seaborn* styles.

```
In [64]  
plt.style.use('seaborn')  
matplotlib.rcParams['lines.labelsize'] = 14  
matplotlib.rcParams['lines', linewidth=2, linestyle='--')  
results.plot_forecast(10)  
plt.xlabel('weeks', labelpad = 8, fontsize = 14)  
plt.show()
```

Forecasting¶

The linear predictor is the optimal h -step ahead forecast in terms of mean-squared error:

$$y_t(h) = \nu + A_1 y_t(h-1) + \cdots + A_p y_t(h-p)$$

We can use the `forecast` function to produce the forecast we plotted in Figure 10-25. Note that we have to specify the “initial value” for the forecast:

```
In [65]  
yhat = results.forecast(results.y, steps=len(valid))
```

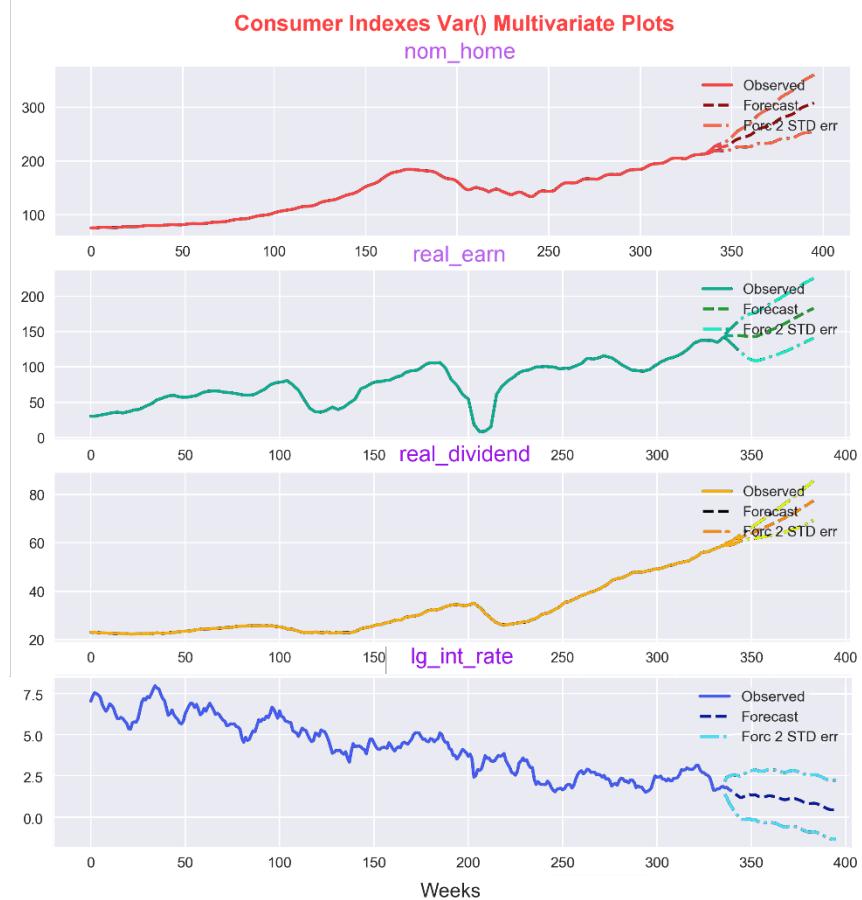


Figure 10-25. Approval rating model multivariate plots (12-lags)

Impulse Response Analysis

Impulse responses are of interest in econometric studies: they are the estimated responses to a unit impulse in one of the variables. They are computed in practice using the $MA(\infty)$ representation of the $VAR(p)$ process:

$$Y_t = \mu + \sum_{i=0}^{\infty} \phi_i u_{t-i}$$

We can perform an impulse response analysis by calling the Impulse-Response Function (IRF), `irf()` on a `VARResults` object.

Orthogonalization is done using the Cholesky decomposition of the estimated error covariance matrix $\hat{\Sigma}_u$ and hence interpretations may change depending on variable ordering. We plot the IRF function with no orthogonalization in Figure 10-26 and with orthogonalization in Figure 10-27.

```
In [66]
plt.style.use('seaborn-whitegrid')
irf = results.irf(12)
irf.plot(orth=True)
```

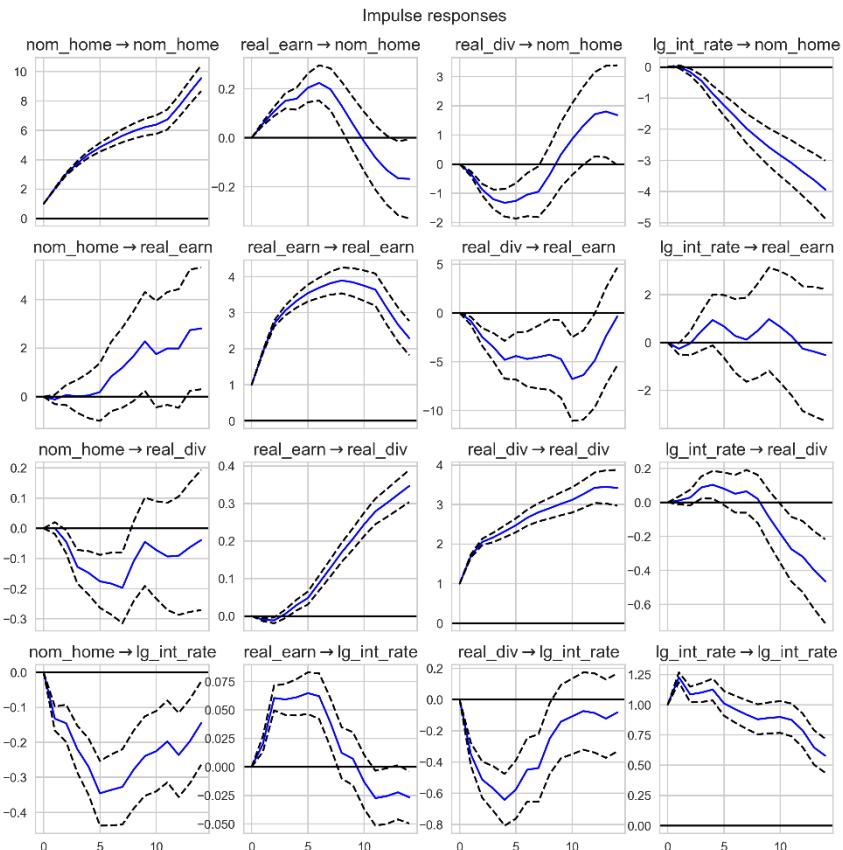


Figure 10-26. Approval ratings model impulse response plots

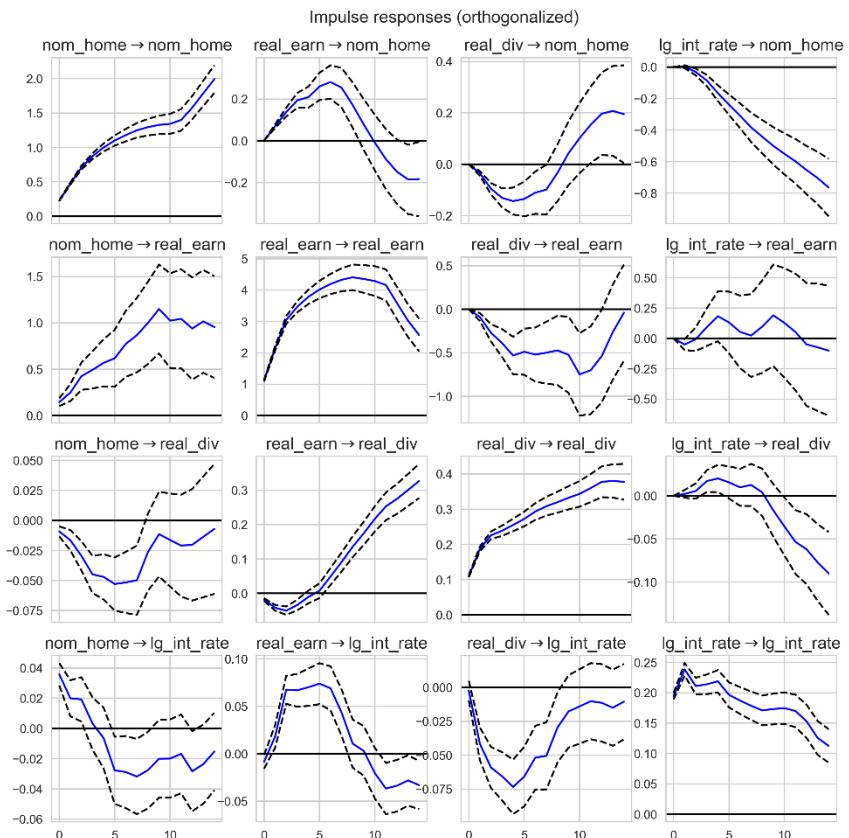


Figure 10-27. Approval ratings model impulse orthogonalized response plots

The plot function is flexible and can plot only variables of interest if we desire. Notice that there are five subplots in Figure 10-28. Each one is y_1 versus y_i , for $i = 2,3,4,5$.

```
In [67]
plt.style.use('seaborn-whitegrid')
irf.plot(impulse='y1', )
```

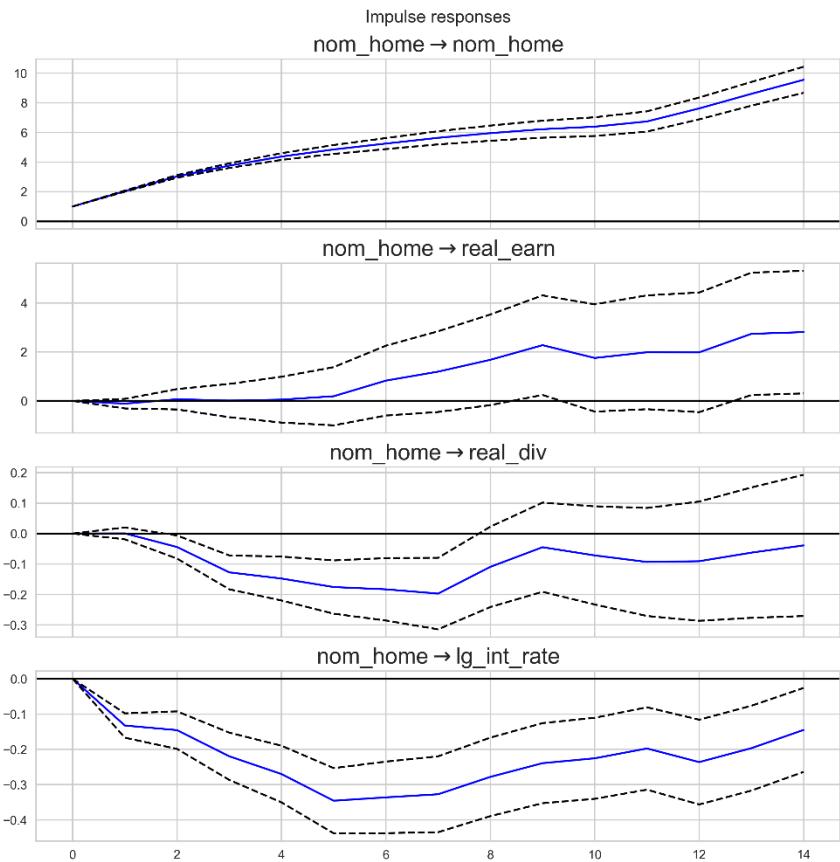


Figure 10-28. Impulse plots for y_1 only.

The cumulative effects $\Psi_n = \sum_{i=0}^n \Phi i$ can be plotted with the long run effects as we have plotted in Figure 10-29.

```
In [68]
plt.style.use('seaborn-whitegrid')
irf.plot_cum_effects(orth=False)
```

S&P Composite Indexes Cumulative Response Plots

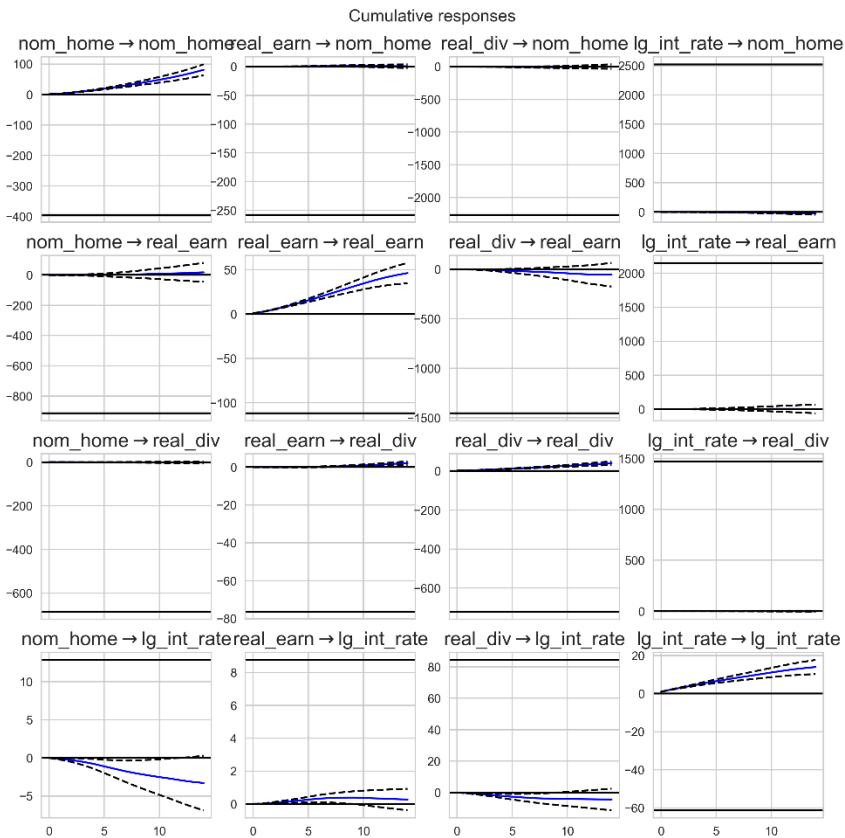


Figure 10-29. Cumulative response plots for approval rating VAR model

Forecast Error Variance Decomposition (FEVD)

Forecast errors of component j on k in an i -step ahead forecast can be decomposed using the orthogonalized impulse responses θ_i :

$$\omega_{jk,i} = \sum_{i=0}^{h-1} \frac{(e_j' \theta_i e_k)^2}{MSE_j(h)}$$

$$MSE_j(h) = \sum_{i=0}^{h-1} e_j' \Phi_i \Sigma_u \Phi_i' e_j$$

These are computed via the `fevd()` function up through a total number of steps ahead.

```
In [69]
fevd = results.fevd(12)
fevd.summary()

Out [-69]
FEVD for nom_home
    nom_home  real_earn  real_div  lg_int_rate
0   1.000000   0.000000   0.000000   0.000000
1   0.978134   0.017282   0.004576   0.000007
2   0.956798   0.029136   0.012799   0.001267
3   0.940506   0.037422   0.016887   0.005185
4   0.930580   0.038464   0.017704   0.013251

FEVD for real_earn
    nom_home  real_earn  real_div  lg_int_rate
0   0.016422   0.983578   0.000000   0.000000
1   0.013841   0.984375   0.001359   0.000425
2   0.016968   0.977619   0.005247   0.000166
3   0.018119   0.973179   0.008299   0.000404
4   0.019331   0.967632   0.011993   0.001044

FEVD for real_div
    nom_home  real_earn  real_div  lg_int_rate
0   0.006716   0.026652   0.966633   0.000000
1   0.007168   0.042448   0.950282   0.000101
2   0.011796   0.044780   0.943080   0.000344
3   0.019719   0.035313   0.942970   0.001998
4   0.023458   0.025492   0.947890   0.003160

FEVD for lg_int_rate
    nom_home  real_earn  real_div  lg_int_rate
0   0.032779   0.001857   0.000159   0.965205
1   0.017063   0.004140   0.017696   0.961101
2   0.013529   0.032692   0.034589   0.919190
3   0.009995   0.045772   0.046308   0.897925
4   0.007945   0.054306   0.056518   0.881231
```

We can also visualize FEVD through the returned `fevd` object, as seen in Figure 10-30.

```
In [70]
from statsmodels.tsa.vector_ar.var_model import FEVD
fevd = results.fevd(20)
FEVD.plot(fevd)
```

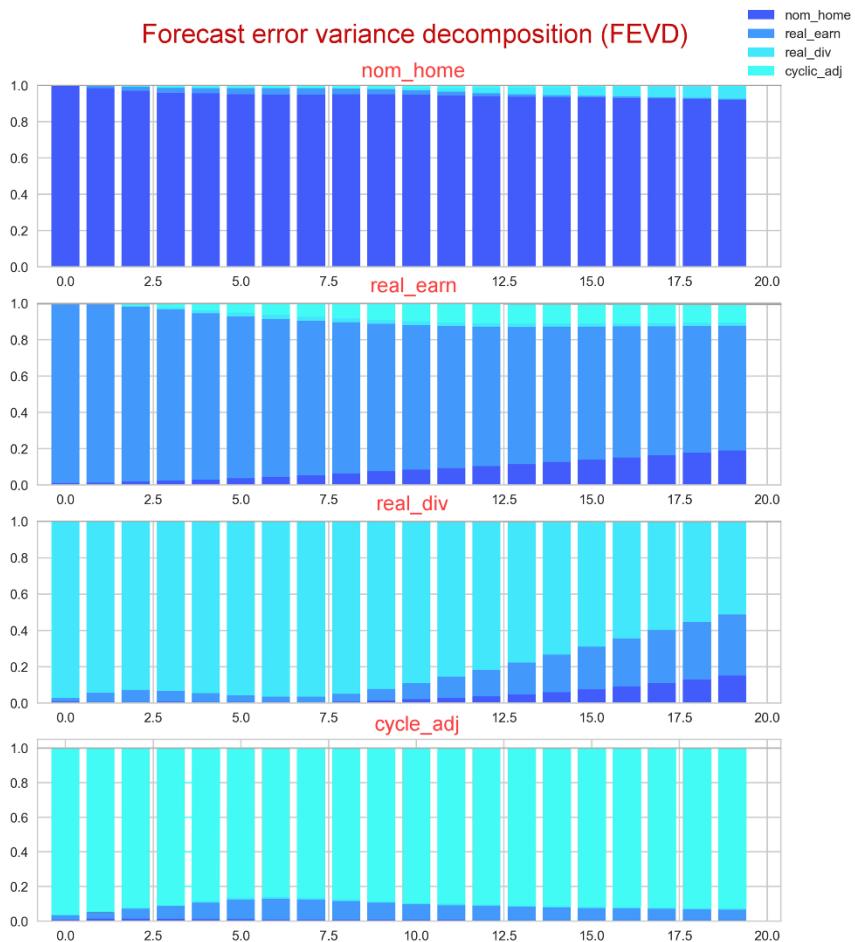


Figure 10-30. Approval rating model forecast error variance decomposition (FEVD)

We can also print out a summary of FEVD, which we have done in Out[71].

Statistical tests

A number of different methods are provided to carry out hypothesis tests about the model results and also the validity of the model assumptions (normality, whiteness / “iid-ness” of errors, etc.).

```
In [71]
from statsmodels.tsa.vector_ar.vecm import coint_johansen
```

```

def cointegration_test(df, alpha=0.05):
    """Perform Johanson's Cointegration Test and Report
    Summary"""
    out = coint_johansen(df,-1,5)
    d = {'0.90':0, '0.95':1, '0.99':2}
    traces = out.lrt1
    cvts = out.cvt[:, d[str(1-alpha)]]
    def adjust(val, length= 6): return str(val).ljust(length)

    # Summary
    print('Name :: Test Stat > C(95%) => Signif \n', '--'*20)
    for col, trace, cvt in zip(df.columns, traces, cvts):
        print(adjust(col), ':: ', adjust(round(trace,2), 9),
              ">", adjust(cvt, 8), ' => ', trace > cvt)

cointegration_test(df)

```

Name	:: Test Stat	> C(95%)	=>	Signif
<hr/>				
approve ::	367.63	> 60.0627	=>	True
iraqwar ::	158.38	> 40.1749	=>	True
wtc ::	37.47	> 24.2761	=>	True
afghan ::	9.00	> 12.3212	=>	False
kabul ::	0.29	> 4.1296	=>	False

Normality¶

As we pointed out several times, the white noise component u_t is assumed to be normally distributed. While this assumption is not required for parameter estimates to be consistent or asymptotically normal, results are generally more reliable in finite samples when residuals are Gaussian white noise. To test whether this assumption is consistent with a data set, `VARResults` offers the `test_normality` method.

```
In [72]: results.test_normality()
```

```
Out[54]:
<statsmodels.tsa.vector_ar.hypothesis_test_results.NormalityTest
Results object. H_0: data generated by normally-distributed
process: reject at 5% significance level. Test statistic:
5288.342, critical value: 15.507>, p-value: 0.000 >
```

The results indicate that we should reject the hypothesis that the data (residuals) arise from a normal distribution process.

Whiteness of residuals

To test the whiteness of the estimation residuals (this means absence of significant residual autocorrelations) one can use the `test_whiteness` or Portmanteau-test for residual autocorrelation method of `VARResults`.

```
In [73]
print(results.test_whiteness())

Out [93]
<statsmodels.tsa.vector_ar.hypothesis_test_results.WhitenessTest
Results object. H_0: residual autocorrelation up to lag 10 is
zero: reject at 5% significance level. Test statistic: 740.856,
critical value: 260.992>, p-value: 0.000>
```

So, we reject the hypothesis that residual auto correlation up to lag 10 is zero.

Chapter Review

In this chapter, we looked at the application of time series analysis using more advanced concepts, including time series intervention, exogenous regressors, and multivariable time series analysis.

We investigated the effects of interventions and how to analyze their effects and account for them in our modeling.

We also discovered the use of Vector Autoregression (VAR), cointegration and causality testing, and more rigorous methods for testing for stationarity.

We learned how to perform transformations on exogenous variables to counter the effects of seasonality, non-stationarity, and so on, to get good fitting models; and then undo those transforms to arrive at more accurate predictions.

Finally, we learned new methods for diagnosing the goodness of fit for our models, such as impulse response charts, Forecast Error Variance Decomposition (FEVD), and lag order selection. We also discussed the definitions:

Definition Number	Definition Citation
Definition 10.1: Time Series Intervention	<p>Suppose that at time $t = T$ (where T may or may not be known), there has been an intervention to a time series. Suppose that the ARIMA model for (the observed series) with no intervention is</p> $y_t - \mu = \frac{\theta(B)}{\varphi(B)} e_t$ <p>where $\theta(B)$ is the usual moving average (MA) component, $\varphi(B)$ is the usual autoregressive (AR) term, e_t is error term.</p> <p>Let z_t = the amount of change at time t that is attributable to the intervention. By definition, $z_t = 0$ before time T (time of the intervention). The value of z_t may or may not be 0 after time T. Then the overall model, including the intervention effect, may be written as</p> $y_t - \mu = z_t + \frac{\theta(B)}{\varphi(B)} e_t$
Definition 10.2: Durbin-Watson Statistic	<p>If e_t is the residual given by $e_t = \rho e_{t-1} + v_t$, then the Durbin-Watson statistic states that null hypothesis, $\rho = 0$, with the alternative hypothesis $\rho \neq 0$, then the test statistic is where T is the number of observations.</p> $DW = \frac{\sum_{t=2}^T (e_t - e_{t-1})^2}{\sum_{t=1}^T e_t^2}$
Definition 10.3: Vector Autoregression (VAR)	<p>The Vector Autoregression (VAR) statistical model is given by a p-dimensional time Y_t of length $K+T$ satisfying a K^{th} order vector autoregressive equation</p> $Y_t = \sum_{l=1}^K \beta_l X_{t-l} + \mu D_t + \varepsilon_t$ <p>conditional on the initial values Y_0, \dots, Y_{1-K}, with a vector of deterministic terms D_t, and, $\varepsilon_t \sim iid \text{Normal}(0, \Sigma_\varepsilon)$ and errors.</p>

Review Exercises

1. Download **Polity IV Project** spreadsheet (Excel format) at www.systemicpeace.org/inscr/. Polity IV Project, provides Political Regime Characteristics and Transitions, from 1800-2014, with annual, cross-national, time-series and polity-case formats coding democratic and autocratic "patterns of authority" and regime changes in all independent countries with total population greater than 500,000 in 2018 (167 countries in 2018)
 - Excel data: <http://www.systemicpeace.org/inscr/p4v2018.xls>;
 - Codebook:
<http://www.systemicpeace.org/inscr/p4manualv2018.pdf>).
 - a. Construct a frequency table of countries over time with a parity level of 8 or higher and save the frequencies as a .CSV file.
 - b. Load the .CSV file into R and set it up as time series data
 - c. Analyze the time series data
 - d. Construct a moving average model and evaluate its fit
 - e. Construct an ARIMA model and evaluate its fit
 - f. Construct a UCM model and evaluate its fit
 - g. Determine the best model and perform diagnostics
2. Go to "Multivariate time series dataset for space weather data analytics" at <https://doi.org/10.1038/s41597-020-0548-x>. The authors introduce and make openly accessible a comprehensive, multivariate time series (MVTs) dataset extracted from solar photospheric vector magnetograms in Spaceweather HMI Active Region Patch (SHARP) series. The dataset also includes a cross-checked NOAA solar flare catalog that immediately facilitates solar flare prediction efforts. The dataset covers 4,098 MVTs data collections from active regions occurring between May 2010 and December 2018, includes 51 flare-predictive parameters, and integrates over 10,000 flare reports. Access the SWAN-SF dataset at the Harvard Dataverse (2020 site :
<https://doi.org/10.7910/DVN/EBCFKM> . Select at least four time series to perform multivariate analysis using methods discussed in this chapter.

3. Using the *q.gnp4791* dataset from the FinTS-Package in R:
 - a. Read the documentation and describe the dataset
 - b. Import the dataset into Jupyter and configure it as a times series (Hint: copy and save it as .txt or .csv)
 - c. Analyze the time series using Jupyter, and specify an appropriate ARIMA model
 - d. Build the ARIMA model in Jupyter
 - e. Perform diagnostic and evaluate the ARIMA model
 - f. Build a 12 quarter forecast using iPyhton
 - g. Build a challenger model and repeat (e) and (f)
 - h. Compare the two models
4. Using the *m.urate* dataset from the FinTS-Package in R:
 - i. Read the documentation and describe the dataset
 - j. Import the dataset into Jupyter and configure it as a times series (Hint: copy and save it as .txt or .csv)
 - k. Analyze the time series using Jupyter, and specify an appropriate ARIMA model
 - l. Build the ARIMA model in Jupyter
 - m. Perform diagnostic and evaluate the ARIMA model
 - n. Build a 12-quarter forecast using Jupyter
 - o. Build a challenger model and repeat (e) and (f)
 - p. Compare the two models
5. Using the *EPA.09.Ex.14.3.alkalinity.df* dataset from the EnvStats-Package in R:
 - a. Analyze the time series using R, and specify an appropriate ARIMA model
 - b. Build the ARIMA model
 - c. Perform diagnostic and evaluate the ARIMA model
 - d. Build a 4-period forecast using R
 - e. Build the automated ARIMA model
 - f. Compare the two models
6. Using *co021* dataset from the dplR-Package in R:
 - g. Plot the series using `plot.type="spag"` (Spaghetti Plot)
 - h. Interactively detrend a tree-ring series by using a smoothing spline with the `detrend()` function
 - i. Build Mean Value Chronology using the `chron()` function
 - j. Plot that data created in (c) with spline

- k. Characterize the data using acf() and pacf()
- l. Fit an AR (autoregression) model to the data
- m. Fit an automated ARIMA model to the data
- n. Compare the models
- o. Summarize the fit and perform diagnostics on the best model

Works Cited

- Armstrong, J. S., Green, K. C., & Graefe, A. (2015). Golden Rule of Forecasting: Be Conservative. *Journal of Business Research*, 68(8), 1717-1731.
doi:<http://dx.doi.org/10.1016/j.jbusres.2015.03.031>
- Beiner, D. (2015, 08 31). *ZRA: Dynamic Plots for Time Series Forecasting*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/ZRA/index.html>
- Benjamin, M. A., Rigby, R. A., & Stasinopoulos, D. M. (2003). Generalized Autoregressive Moving Average Models. *Journal of the American Statistical Association*, 98(461), 214–223.
doi:10.1198/016214503388619238
- Benjamin, M. A., Rigby, R. A., & Stasinopoulos, D. M. (2003). Generalized Autoregressive Moving Average Models. *Journal of the American Statistical Association*, 98, 214-223.
- Box, G. E., & Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control*. San Francisco: Holden-Day.
- Box, G. E., & Pierce, D. A. (1970). Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65, 1509-1526.
- Box, G. E., Jenkins, G. M., & Reinsel, G. C. (2008). *Time Series Analysis: Forecasting and Control*. (4th, Ed.) Hoboken, NJ: John Wiley & Sons. Retrieved from 978-0-470-27284-8
- Brockwell, P. J., & Davis, R. A. (2010). *Introduction to Time Series and Forecasting*. (2nd ed.). New York: Springer.
- Brown, R. G. (1956). *Exponential Smoothing for Predicting Demand*. Cambridge, Massachusetts: Arthur D. Little Inc.
- Chan, K.-S., & Ripley, B. (2012, 11 13). *TSA: Time Series Analysis*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/TSA/index.html>

- Chowdhury, K. R. (2015, 11 06). *Implementation of Unobserved Components Model (UCM)*. Retrieved 09 15, 2015, from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/rucm/index.html>
- Coghlan, A. (2016, 01 05). *Using R for Time Series Analysis*. Retrieved from Welcome to a Little Book of R for Time Series!: <http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/index.html>
- Davis, R. A., Dunsmuir, W. T., & Wang, Y. (1999). Modeling Time Series of Count Data. In S. Ghosh, *Asymptotics, Nonparametrics, and Time Series* (Vol. 158, pp. 63–114). New York: Marcel Dekker.
- Dell. (2015, May 8). Time Series Analysis - Statistics Textbook - Dell. (n.d.). Retrieved from Retrieved from <http://documents.software.dell.com/Statistics/Textbook/Time-Series-Analysis>
- Dunsmuir, W. T., Li, C., & LScott, D. J. (2015, 10 03). *glarma: Generalized Linear Autoregressive Moving Average Models*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/glarma/>
- Durbin, J., & Koopman, S. J. (2012). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press.
- Eastwood, B. (2020, Jun 18). The 10 Most Popular Programming Languages to Learn in 2020. *Northeast Universities*, Web Version. Retrieved from <https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/>
- Eichenberg, R. C., & Stoll, R. J. (2008). *The Political Fortunes of War: Iraq and the Domestic Standing of President George W. Bush*. Retrieved 10 8, 2015, from <http://www.owlnet.rice.edu/~stoll/bushpop/>
- Fisher, T. J., & Gallagher, C. M. (2012, 05 24). *WeightedPortTest: Weighted Portmanteau Tests for Time Series Goodness-of-fit*.

- Retrieved from The Comprehensive R Archive Network:
<https://cran.r-project.org/web/packages/WeightedPortTest/index.html>
- Gardner, E., & McKenzie, E. (2011). Why the damped trend works. *Journal of the Operational Research Society*, 62(6). doi:10.1057/jors.2010.37.
- Hafner, R. (2016, 01 06). *stlplus: Enhanced Seasonal Decomposition of Time Series by Loess*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/stlplus/index.html>
- Han, T. T., Nghi, D. H., Diem, M. T., My, N. T., & Minh, H. V. (2015, 12 09). *AnalyzeTS: Analyze Time Series*. Retrieved from The Comprehensive R Archive Network: <https://cran.r-project.org/web/packages/AnalyzeTS/index.html>
- Harvey, A. (1989). *Forecasting, structural time series models and the Kalman filter*. Cambridge: Cambridge University Press.
- Helske, J. (2015, June 26). *KFAS: Kalman filter and Smoothers for Exponential Family State Space Models. R package version 1.0.4-1*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/KFAS/KFAS.pdf>
- Holt, C. C. (2004, January-March). Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1), 5-10. doi:<https://doi.org/10.1016/j.ijforecast.2003.09.015>
- Hyndman, R. J. (2006, 06 04). Another look at measures of forecast accuracy. *FORESIGHT*(4), 46.
- Hyndman, R. J. (2013, 03 14). *fpp: Data for "Forecasting: principles and practice"*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/fpp/index.html>

- Hyndman, R. J. (2015, 04 09). *expsmooth: Data Sets from "Forecasting with Exponential Smoothing"*. Retrieved from The Comprehensive R Archive Network: <https://cran.r-project.org/web/packages/expsmooth/>
- Hyndman, R. J., Athanasopoulos, G., Bergmeir, C., Cinelli, C., Cinelli, Y., Mayer, Z., . . . Zhou, Z. (2015, 10 20). *forecast: Forecasting Functions for Time Series and Linear Models*. Retrieved from The Comprehensive R Archive Network (CRAN): <https://cran.r-project.org/web/packages/forecast/index.html>
- Hyndman, R. J., Koehler, A. B., Ord, J. K., & Snyder, R. D. (2008). *Forecasting with exponential smoothing: the state space approach*. New York: Springer-Verlag. Retrieved from <http://www.exponentialsmoothing.net>.
- Hyndman, R. J., Koehler, A. B., Snyder, R. D., & Grose, S. (2002). A state space framework for automatic forecasting using exponential smoothing methods. *International J. Forecasting*, 18(3), 439--454.
- Johansen, S. (1991). Estimation and Hypothesis Testing of Cointegration Vectors in Gaussian Vector Autoregressive Models. *Econometrica*, 59(6), 1551-1580. doi:DOI: 10.2307/2938278
- Kassambara, A. (2018, November 18). *GGPlot Colors Best Tricks You Will Love*. Retrieved from Datanovia: <https://www.datanovia.com/en/blog/ggplot-colors-best-tricks-you-will-love/#comments>
- Kendall, G. (1976). *Kendall, 1976*. New York: Hafner.
- Kocenda, E., & Cerný, A. (2007). *Elements of Time Series Econometrics: An Applied Approach*. Prague: Karolinum Press.
- Ljung, G. M., & Box, G. E. (1978). On a measure of lack of fit in time series models. *Biometrika*, 65, 297-303.
- Lütkepohl, H. (2005). *New Introduction to Multiple Time Series Analysis*. New York: Springer-Verlag Berlin Heidelberg. doi:10.1007/978-3-540-27752-1

- Mallows, C. L. (1973). Some Comments on CP. *Technometrics*, 15(4), 661–675. doi:10.2307/1267380
- Newton, H. J. (1988). *TIMESLAB: A Time Series Analysis Laboratory*. . Belmont, CA : Wadsworth.
- Petris, G., & Petrone, S. (2011). State Space Models in R. *Journal of Statistical Software*, 41(4), 1-25. Retrieved from <http://www.jstatsoft.org/v41/i04/>
- Petukhova, T. O. (2018). Assessment of autoregressive integrated moving average (ARIMA), generalized linear autoregressive moving average (GLARMA), and random forest (RF) time series regression models for predicting Influenza A virus. *PLoS one*, 13(6). doi:doi: 10.1371/journal.pone.0198313
- Qiu, D. (2015, 07 08). *aTSA: Alternative Time Series Analysis*. Retrieved from The Comprehensive R Archive Network: <https://cran.r-project.org/web/packages/aTSA/index.html>
- R., W. P. (1960, April). Forecasting Sales by Exponentially Weighted Moving Averages. *Management Science*, 6(3), 231-362. doi:<https://doi.org/10.1287/mnsc.6.3.324>
- SAS-Institute. (2010). *SAS/ETS 9.22 User's Guide*. doi:http://support.sas.com/documentation/cdl/en/etsug/63348/HTML/default/viewer.htm#ucm_toc.htm
- Trapletti, A., Hornik, K., & LeBaron, B. (2015, 02 20). *tseries: Time Series Analysis and Computational Finance*. Retrieved from The Comprehensive R Archive Network: <https://cran.r-project.org/web/packages/tseries/index.html>
- USDA. (2016, 01 13). *U.S. Department of Agriculture Economic Research Service*. Retrieved 01 19, 2016, from U.S. Department of Agriculture: <http://www.ers.usda.gov/data-products/livestock-meat-international-trade-data/documentation.aspx#data>
- Velleman, P. F., & Hoaglin, D. C. (1981). *Applications, Basics and Computing of Exploratory Data Analysis*. Boston: Duxbury Press.

- Wang, W. (2006). *Stochasticity, Nonlinearity and Forecasting of Streamflow Processes*. New York: Delft University Press.
- Wilkinson, L., Wills, D., Rope, D., Norton, A., & Dubbs, R. (2005). *The Grammar of Graphics (Statistics and Computing)* (2nd ed.). New Tourk: Springer.
- Yahoo! (2020, October 11). *Yahoo! Finance*. Retrieved from Yahoo!.com: <https://finance.yahoo.com/quote/GOOG/history/>
- Zeger, S. L. (1998). A Regression Model for Time Series of Counts. *Biometrika*, 75, 822-835.
- Zeger, S. L., & Qaqish, B. (1988). Markov Regression Models for Time Series: A Quasi-likelihood Approach. *Biometrics*, 44, 1019-1032.

Index

A

abline	217
abline function.....	29, 214, 216, 262
accuracy() function	219
ACF	33, 94, 95, 97, 223, 235, 236, 344, 348, 349
acf() function.....	169, 205, 206, 207
additive damped trend definition	110, 139
additive Holt-Winters definition	116, 139
additive model.....	114
ADF test.....	190, 191, 192, 193, 206, 233, 247, 376, 377, 378, 380
adjusted R-squared	229, 230
<i>advsales</i> dataset	140, 236
aesthetics ... xvii, 18, 150, 151, 162, 167, 186, 276, 277, 287, 315, 335, 347, 354	
aggregation	148
AIC.....	223, 230, 325, <i>See</i> Akaike Information Criterion
air passenger data.....	108, 210, 213, 214
airpass	<i>See</i> air passenger data
airpass dataset.....	36, 182, 202, 231
Akaike Information Criterion	359, <i>See</i> AIC
AnalyzeTS package.....	224
ARIMA ... 4, 36, 100, 181, 199, 203, 205, 213, 214, 215, 217, 219, 223, 226, 231, 233, 235, 236, 269, 271, 297, 309, 326, 331, 343, 344, 348, 350, 404, 405, 406	
arima() function	205, 213
ARIMA(p, d, q) series definition	213, 235
ARMA	208, 211, 212, 223, 331
ARMA(p,q) model definition	211, 234
aTSA package	127, 185, 194, 205
Augmented Dickey-Fuller test	172, 192, 193, 205, 206, 246, 248
augmented Dickey–Fuller test definition	190, 233
auto.arima() function.....	216
autocorrelation 1, 28, 30, 32, 33, 34, 35, 36, 81, 94, 95, 96, 97, 99, 100, 138, 168, 172, 181, 191, 192, 195, 199, 205, 206, 207, 208, 209, 210, 211, 212, 226, 230, 234, 243, 248, 264, 293, 314, 315, 343, 349, 355, 356, 359, 372, 391, 402	
autocorrelation coefficient definition	33, 41, 94, 138

autocorrelation definition	95, 138
autocorrelation function	<i>See ACF</i>
autoregressive coefficient.....	193, 350
autoregressive component 13, 209, 211, 212, 213, 215, 265, 272, 305, 309, 321, 325, 326, 343, 348, 350, 352, 372	
autoregressive component definition	17, 41, 310, 321
Autoregressive Integrated Moving Average	<i>See ARIMA</i>
autoregressive model.....	81, 190, 212, 246, 249, 250, 325, 406

B

bar plot.....	153, 286, 288
basic structural model.....	<i>See BSM</i>
Bayesian Information Criterion	359
beer sales	102
bias	227, 319, 362
BIC	230, 325, <i>See Bayesian Information Criterion</i>
bicoal dataset.....	141
box plot	22, 23, 161, 162, 163, 288, 289, 338, 339, 341, 342
Box.test() function.....	210, 222
Box-Ljung test.....	<i>See Ljung-Box test</i>
BSM	311, 314, 318

C

cex	48, 315
chicken dataset	140, 235
Clara Peller	<i>See Where's the Beef?</i>
clustermaps.....	291
cointegrated	375
cointegration test.....	192, 247, 372
colormap	281, 282, 284
consumer confidence index	95, 118, 123
copper dataset.....	42, 90, 140, 179, 235
correlation.....	30, 31, 32, 33, 34, 68, 94, 99, 138, 168, 169, 170, 207, 226, 264, 289, 290, 291, 292, 342, 343, 386, 387, 391, 402
correlograms	33, 94, 96, 97, 197, 198, 205, 206, 207, 209
COVID-19 xvii, xviii, 59, 60, 61, 62, 63, 64, 65, 66, 67, 69, 70, 71, 73, 75, 77, 212	
critical value	191, 246, 401, 402
cycle ..	13, 16, 17, 36, 40, 45, 52, 53, 56, 57, 58, 59, 89, 262, 309, 310, 311, 313, 321
cycle definition	16

D

damped Holt-Winters definition	123, 140
data frame	6, 23, 59, 75, 83, 84, 88, 151, 161, 163, 165, 173, 239, 274, 279,
284, 285, 333, 369, 373	
dataframe.mean()	165
datasets package.....	311
date class	6, 7, 9, 10
date-time formatting	5, 6, 9, 10
decompose() function.....	100, 101, 117, 241
decomposition	xvii, xviii, 14, 24, 36, 37, 45, 89, 93, 101, 117, 128, 129, 131,
172, 173, 181, 184, 185, 231, 241, 242, 243, 293, 294, 309, 311, 321, 337,	
395, 400	
decomposition definition	14, 40
DEMA	79
Department of Agriculture.....	15, 143
Descriptives() function.....	224
DIFF function	212, 213, 215, 272, 305, 326
diff() function	199
difference stationary definition	199, 234
differencing.33, 43, 96, 97, 98, 99, 137, 181, 190, 191, 194, 195, 196, 199, 200,	
201, 202, 205, 206, 212, 213, 214, 217, 229, 234, 247, 248, 250, 303, 330,	
374, 376, 380, 388	
dots-per-inch.....	<i>See</i> dpi
double exponential smoothing	105, 107, 109, 133
double exponential smoothing definition	106, 139
double-exponential moving average	<i>See</i> DEMA
dpi ...144, 151, 153, 155, 167, 176, 177, 178, 276, 277, 279, 283, 284, 287, 289,	
291, 295, 299, 335, 338, 347, 357, 364, 369, 389	
dplyr package.....	61, 70, 88
dtype	164, 172, 275, 296, 333, 353, 384
Durbin-Watson statistic	342, 343, 356, 386
Durbin-Watson statistic definition	343, 403
DW statistic.....	<i>See</i> Durbin-Watson statistic

E

Elastic volume-weighted moving average	<i>See</i> EVWMA
elec dataset.....	236
elecequip dataset.....	57
elecsales dataset	51
EMA.....	79, 81, 140

error term.....	13, 31, 211, 235, 309, 321
Error-Trend-Seasonality	81, 271
estimate() function.....	223
EVWMA	79, 81, 141
EWMA	46
exogenous regressors	xviii, 324, 325, 327, 329, 332, 333, 334, 335, 345, 346, 351, 353, 363, 365, 367, 371, 372
explanatory variables	18
Exponential moving average.....	<i>See EMA</i>
exponential smoothing	xviii, 4, 5, 43, 81, 82, 93, 97, 102, 103, 104, 105, 106, 107, 111, 114, 115, 116, 117, 118, 120, 123, 126, 127, 131, 132, 135, 137, 163, 173, 177, 203, 204, 251, 271, 273
exponential smoothing model	204
expsmooth package	118
external regressors.....	326, 354

F

fevd() function.....	399
filter() function	45, 70, 82
first difference definition	199, 234
floor decade function	149, 150, 286
forecast .10, 11, 82, 83, 84, 88, 93, 103, 105, 107, 109, 110, 112, 113, 114, 115, 116, 121, 122, 123, 138, 168, 177, 178, 181, 196, 204, 207, 214, 217, 218, 219, 227, 228, 229, 231, 233, 251, 254, 257, 300, 301, 302, 327, 353, 383, 388, 389	
forecast error	221, 226, 319, 320
forecast model	6, 81
forecast package	118, 181, 205, 213, 216, 219, 229
forecast() function.....	228, 231
forecasting.....	1, 2, 3, 4, 181, 207, 225, 227, 229, 344
fpp package	51, 57

G

Gaussian white noise	401
general state space model definition	271, 305
geofacet package	73, 76
George W. Bush job approval rate	323, 324, 326
ggplot package	146, 147, 154
ggplot() function.....	150
ggplot2 package	19, 23, 74, 129, 150
GLOBAL Land-Ocean Temperature Index.....	xviii, 237

global surface temperature change.....	237, 238, 240
global temperature changes.....	244, 253
GOF	<i>See</i> goodness-of-fit
goodness-of-fit.....	226, 254, 257, 258, 317, 362
Got Milk?.....	xvii, xviii, 5, 18, 80, 81
Granger's Causality Test.....	372

H

heatmap.....	291, 292, 293
Holt's exponential smoothing.....	<i>See</i> exponential smoothing
Holt's linear trend	110
Holt's Linear trend	107
Holt's Linear Trend definition	107, 139
Holt's exponential smoothing.....	178
Holt-Winters' additive method	121, 123
Holt-Winters exponential smoothing	115, 118, 133
Holt-Winters' multiplicative method	121, 123
Holt-Winters smoothing	4, 119
HoltWinters() function.....	206

I

i.i.d.	13, 309, 311, 321, 367
impulse-response function (IRF).....	394
independent variables	143, 226, 313, 332
independent, identically distributed.....	<i>See</i> i.i.d.
index	1, 28, 143, 144, 149, 163, 164, 172, 287, 332, 333
set_index	359
set_index() function	147, 164, 174, 275, 276, 293, 296, 333, 359, 369
info() function	147, 148
international passenger data series (G).....	43
intervention	326, 335
intervention definition	327, 403
irregular component	13, 15, 17, 40, 185, 205, 241, 243, 309, 311, 321
irregular component definition	17, 40

J

<i>Jupyter Notebook</i>	xvii, xviii, xix, 143, 154, 173, 179, 227, 287, 331, 405
-------------------------------	--

K

Kendall Tau	290
KPSS test	194, 195, 234, 246, 248, 249

Kwiatkowski-Phillips-Schmidt-Shin test definition	194, 234
L	
lag	94, 99, 138, 355
lag definition	27, 41
lag operator	35
lag parameter	210
lag() function	71
lag.plot()	29, 31
lagged variable	231, 327, 366
LDV	<i>See</i> lagged variable
least squares regression	160
legend_margin() function	68
level component	15, 89, 114, 115, 131, 133, 136, 186, 311, 312, 313, 314, 317, 318, 320, 324, 404
livestock	111, 113, 153, 154, 156, 173, 273
Ljung-Box Q statistic	95
Ljung-Box test	121, 205, 210, 211, 223, 225, 258, 261, 314, 350, 355, 361
Ljung-Box test definition	210, 234
local averaging	44
locally weighted scatterplot smoothing	<i>See</i> LOWESS
Loess regression definition	21, 41
log transformation	185, 228, 243, 322
long-term interest rate	369
LOWESS	160
M	
m.urate dataset	269, 405
MA	<i>See</i> moving average model, <i>See</i> moving average
MAE	115, 137, 226, 228, 229, 231, 318, 319, 320
MAPE	85, 86, 87, 88, 104, 115, 125, 132, 134, 135, 137, 216, 219, 220, 227, 228, 231, 318, 319, 320, 232, 258, 259, 346, 357, 362, 363, <i>See</i>
MASE	115, 137, 227, 231, 318
matplotlib package	151
Matplotlib package	281, 282, 286, 287, 332, 393
mean absolute error	<i>See</i> MAE
mean absolute percentage error	<i>See</i> MAPE
219, 220, 232, 258, 259, 346, 357, 362	
mean absolute percentage error definition	87, 90
mean absolute scaled error	<i>See</i> MASE
mean forecast error	<i>See</i> MFE

mean percentage error	<i>See MPE</i>
mean squared error	<i>See MSE</i>
meat dataset.....	143, 146, 147
meat type.....	152, 155, 165
MFE	319
monotonous.....	15, 44
monthly beer sales.....	96
monthplot() function	24, 186, 245
moving average....xviii, 43, 45, 46, 47, 48, 49, 50, 51, 53, 54, 56, 57, 58, 59, 60, 62, 63, 64, 65, 67, 68, 70, 71, 72, 73, 76, 77, 78, 79, 80, 81, 93, 117, 126, 140, 141, 145, 163, 166, 167, 168, 169, 170, 172, 179, 209, 211, 212, 213, 215, 223, 224, 233, 235, 251, 254, 264, 265, 272, 294, 295, 296, 301, 305, 321, 325, 326, 327, 352, 371, 372, 403	
moving average definition	59, 89
moving average model ...4, 45, 46, 51, 52, 53, 54, 55, 56, 58, 79, 137, 140, 141, 212, 213, 214, 345, 404	
moving average model definition	45, 89
MPE.....	115, 137, 227, 231, 318
MSE	227, 319, 320
multiplicative Holt-Winters definition	117, 140
multivariate time series	366
N	
NASA	237
Nile dataset.....	311, 312
Nile River..... xviii, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320	
noise.. 17, 18, 33, 36, 40, 45, 59, 63, 95, 116, 156, 157, 159, 161, 181, 188, 196, 197, 211, 223, 226, 229, 235, 251, 293, 297, 401	
nominal home price index	369, 380
Nominal Home Price Index	368
nonparametric smoothers	160
nonstationarity	345
normally distributed	204, 220, 221, 226, 300, 357, 401
null hypothesis 172, 190, 191, 193, 205, 210, 223, 233, 234, 246, 258, 261, 314, 337, 343, 355, 356, 373, 374, 378, 403	
Numpy package	147, 332
O	
Operation Iraqi Freedom 324, 329, 332, 333, 334, 339, 351, 352, 354, 363, 371, 401	
Order of integration.....	374

ordering.....	4
---------------	---

P

PACF	97, 99, 208, 223, 235, 236, 344, 348, 349
pacf() function	205, 206
pandas package.....	143, 144, 147, 148, 149, 161, 164, 166, 173, 274, 279, 282, 332, 368
pandasql package.....	146
par() function	48, 65, 68, 315, <i>See plot parameters</i>
partial autocorrelation	99, 205, 208
Pearson's coefficient	289
percentiles.....	161
PerformanceAnalytics package	67
Phillips-Perron test	191, 192, 193, 248, 269
Phillips-Perron test definition	193, 234
pip	146, 154
plot parameters.....	48, 314, 315, 318, 393
plot() function	232
plot.acf().....	254
plot.ts() function	240
plotForecastErrors.....	220, 221
plotly package	154, 155, 274
plotnine package.....	150, 155, 274
pltSee pyplot	
pmdarima package.....	345, 346
portmanteau test	314
POSIXct	8, 9
POSIXlt.....	9
PP test	193, 194, 248
predict() function	313, 346
p-value.95, 96, 121, 172, 186, 191, 192, 193, 194, 210, 211, 223, 225, 246, 247, 248, 258, 259, 260, 261, 262, 337, 373, 374, 378, 401, 402	
pyplot package	151, 161
Python	1, 143, 149, 184, 331

Q

Q statistic	355
q.gnp4791 dataset	270, 405
Q-Q plot.....	223, 356, 357, 359

R

random error	<i>See</i> irregular component
random walk model	196, 197, 198, 310
RColorBrewer package.....	130, 186, 187, 244
readr package	238
real dividend	369
real price	369
regressive component definition	18, 41
residual diagnostics.....	226
residuals.....	17, 97, 120, 121, 168, 203, 206, 210, 211, 220, 221, 222, 225, 226, 230, 235, 236, 314, 325, 343, 348, 349, 350, 355, 357, 358, 402, 403
RMSE...85, 86, 104, 122, 123, 124, 132, 134, 135, 137, 182, 216, 219, 220, 226, 227, 228, 229, 230, 231, 232, 302, 318, 320,	
rolling	<i>See</i> moving average
rollmean() function	71, 145
root mean squared error	<i>See</i> RMSE
root mean squared error definition	86
R-squared.....	227, 229
R-studio.....	220
rucm package.....	311

S

<i>S&P Composite</i>	xviii, 368, 370
SARIMA	223, 264, 265, 271, 297, 298, 299, 324, 331, 370
sarima() function.....	324, 326
Schwarz' Bayesian Information Criterion.....	<i>See</i> BIC
seaborn package	165, 178, 291, 338, 339, 393, 395, 396, 397
seasonal component.....	181, 185
seasonal index.....	<i>See</i> index
seasonal plot.....	243
seasonal subseries plots	37
seasonal_decompose() function.....	172
seasonality 1, 4, 13, 15, 17, 18, 36, 40, 43, 44, 56, 57, 58, 89, 93, 100, 114, 115, 118, 119, 128, 129, 131, 136, 181, 188, 213, 214, 215, 217, 227, 232, 233, 235, 236, 241, 242, 243, 244, 245, 272, 309, 310, 311, 312, 313, 321, 326	
seasonality definition	16, 40, 94, 138
seasonally adjust.....	101
seasplot() function	186
second difference definition	201
second order differencing.....	200, 201

September 11th	324, 326
serial dependencies.....	96, 99
SES.....	<i>See simple exponential smoothing</i>
set.index()	<i>See index</i>
simple exponential smoothing	xviii, 102, 104, 105, 107, 115, 131, 173, 175, 250, 251
simple exponential smoothing definition	103, 138
Simple moving average	<i>See SMA</i>
slope component	114, 310, 311, 312, 313, 314, 320
SMA.....	79, 223, 224
smoothing ...	5, 15, 44, 45, 97, 118, 119, 120, 131, 133, 134, 135, 136, 137, 156, 204, 207, 235, 236, 309, 405
SOI	128, 130, 131, 133
soi.dataset	127
Southern Oscillation Index	<i>See soi dataset</i>
Spearman rank	290
standard normal distribution	13, 309, 321
state space models.....	309
stationarity	18, 33, 95, 100, 172, 181, 186, 188, 189, 190, 191, 193, 194, 195, 196, 198, 199, 200, 201, 205, 208, 211, 212, 234, 235, 245, 246, 247, 248, 249, 250, 272, 303, 326, 331, 337, 343, 344, 367, 374, 376, 378, 380, 384
stationarity test.....	249
stationary series	181
stationary time series.....	188, 331
stationary times series definition	18, 41, 195, 234
statsmodels package	169, 170, 172, 279, 294, 295, 331, 332, 368, 375, 386, 390, 392, 399, 400, 401, 402
stl() function	128, 185, 231, 241, 243
stlplus package	36
stlplus() function	89
StructTS() function	244, 314

T

testing ...	85, 86, 88, 172, 182, 191, 192, 206, 232, 246, 247, 248, 249, 259, 347, 354, 363, 372, 375, 376, 377, 378, 379, 380, 381, 401, 402
time series analysis	1, 3, 10, 137, 226, 314, 402
time series data.....	1, 2, 3, 4, 10, 11, 13, 15, 16, 17, 18, 40, 41, 43, 44, 80, 127, 128, 137, 143, 179, 181, 200, 213, 216, 227, 233, 235, 236, 238, 240, 244, 269, 309, 310, 326, 344, 404
time series data definition	1, 10

training ..	85, 86, 88, 104, 132, 134, 135, 137, 182, 216, 232, 258, 347, 355, 363, 383
train-test-split	175
trend ...	1, 13, 15, 17, 18, 36, 40, 43, 44, 45, 52, 53, 56, 57, 58, 59, 89, 114, 115, 118, 119, 133, 136, 155, 156, 179, 181, 185, 186, 188, 206, 235, 241, 242, 243, 309, 310, 311, 312, 314, 318, 321, 344, 348, 353
additive trend	15, 107, 126
damped trend.	15, 82, 83, 84, 86, 93, 109, 110, 111, 112, 113, 114, 123, 124, 125, 126, 251
exponential trend	15
multiplicative trend	15
nonseasonal trend	15
trend definition	15, 40
trend stationary definition	195, 234
trend test	185
triple exponential smoothing.....	136, 137, 187
ts() function.....	181, 182
TSA package	36, 96
tsbox package	62
tsdiag() function.....	314
tseries package	194, 205, 248
TSstudio package	64, 207, 208, 209
TTR package	79
U	
UCM	4, 220, 269, 309, 310, 311, 314, 321, 322, 404
ucm() function.....	311, 313, 319
unemployment	118
unemployment rate	143
unit root	xviii, 172, 190, 191, 193, 234, 235, 246, 350, 378, 379, 380, 381
unit root test.....	190, 192, 199, 233, 235
unobserved components model	<i>See UCM</i>
unobserved components model definition	309, 321
unsecured consumer loans	1, 33, 206, 244, 326
V	
VAR.....	<i>See vector auto regression</i>
vector auto regression ...	xviii, 261, 365, 366, 367, 368, 372, 374, 375, 376, 381, 382, 383, 384, 387, 389, 392, 394, 398
vector autoregression definition	367, 403

W

Weighted moving average	<i>See WMA</i>
WeightedPortTest package	210, 223
Wendy's hamburger	157
West Texas Intermediate	<i>See WTI</i>
Where's the Beef?	xvii, xviii, 15, 146, 155, 156, 157, 162, 173, 177, 178, 274, 294
whiteness	400, 402
window() function	82, 214
WMA	79, 140
World Trade Center	324, 326, 327, 328, 329, 331, 363, 372
World Trade Center terrorist attack	324
WTI	144

Z

zero lag exponential moving average	<i>See ZLEMA</i>
ZLEMA	79, 140
zoo	71
zoo package	145
ZRA-package	232

