RAJALAKSHMI ENGINEERING COLLEGE
(Autonomous)

RAJALAKSHMI NAGAR, THANDALAM, CHENNAI-602105

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

AI19642 TIME SERIES ANALYSIS AND FORECASTING

THIRD YEAR

SIXTH SEMESTER

Prepared by

Mrs. I. Eugene Berna, Assistant Professor

RAJALAKSHMI ENGINEERING COLLEGE
(Autonomous)

RAJALAKSHMI NAGAR, THANDALAM, CHENNAI-602105

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

AI19642 TIME SERIES ANALYSIS AND FORECASTING

THIRD YEAR

SIXTH SEMESTER

RAJALAKSHMI ENGINEERING COLLEGE

VISION

To be an institution of excellence in Engineering, Technology and Management Education & Research. To provide competent and ethical professionals with a concern for society.

MISSION

To impart quality technical education imbibed with proficiency and humane values. To provide the right ambience and opportunities for the students to develop into creative, talented and globally competent professionals. To promote research and development in technology and management for the benefit of the society.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

VISION:

- To promote highly Ethical and Innovative Computer Professionals through excellence in teaching, training and research.

MISSION:

- To produce globally competent professionals, motivated to learn the emerging technologies and to be innovative in solving real world problems.
- To promote research activities amongst the students and the members of faculty that could benefit the society.
- To impart moral and ethical values in their profession.

PROGRAMME EDUCATIONAL OBJECTIVES

- To equip students with an essential background in computer science, basic electronics and applied mathematics.
- To prepare students with fundamental knowledge in programming languages, and tools and enable them to develop applications.
- To encourage the research abilities and innovative project development in the field of AI, ML,DL, networking, security, web development, Data Science and also emerging technologies for the cause of social benefit.
- To develop professionally ethical individuals enhanced with analytical skills, communication skills and organizing ability to meet industry requirements.

PROGRAMME OUTCOMES (POs)

- **PO1**: Apply the knowledge of Mathematics, Science, Engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2**: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

- **PO3**: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4**: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions..
- **PO5**: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6**: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **PO7**: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8**: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9**: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10**: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11**: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12**: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

A graduate of the Artificial Intelligence and Machine Learning Program will demonstrate,

- Foundation Skills: Ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, web design, AI, machine learning, deep learning, data science, and networking for efficient design of computer-based systems of varying complexity. Familiarity and practical competence with a broad range of programming language, tools and open source platforms.
- Problem-Solving Skills: Ability to apply mathematical methodologies to solve computational task, model real world problem using appropriate AI and ML algorithms. To understand the standard practices and strategies in project development, using open-ended programming environments to deliver a quality product.
- Successful Progression: Ability to apply knowledge in various domains to identify research gaps and to provide solution to new ideas, inculcate passion towards higher studies, creating innovative career paths to be an entrepreneur and evolve as an ethically social responsible AI and ML professional.

CO-PO-PSO JUSTIFICATION:

CO1-PO1	2	Moderately mapped as students will be learning programming aspects of the course.
CO1-PO2	1	Slightly mapped students can apply the basic knowledge of Machine Learning..
CO1-PO3	1	Slightly mapped as students will be able to design a program based on Autocorrelation models.
CO1-PO4	1	Slightly mapped as students will be able to design a tool for application programs.
CO1-PO5	2	Moderately mapped students can apply techniques and resources and developing a models for time series analysis
CO1-PO12	2	Moderately mapped as students will be able to learn different methods to implement complex series.
CO1-PSO1	3	Strongly mapped as students will be understanding, analyzing, designing the problem in real life.
CO1-PSO2	3	Strongly mapped as students will be able to solve real time problems based on programming skills.

CO1-PSO3	2	Moderately mapped as students will be learning programming aspects of the course and implementing and developing applications in real life.
CO2-PO1	2	Moderately mapped as students will be learning programming aspects of the course.
CO2-PO2	2	Moderately mapped students can apply the basic knowledge of time series analysis based on regression models..
CO2-PO3	1	Slightly mapped as students will be able to design a program based on learning models.
CO2-PO4	1	Slightly mapped as students will be able to design a model for applications.
CO2-PO5	2	Moderately mapped as students will be able to design classes and implement through regression models.
CO2-PO12	2	Moderately mapped as students will be able to learn different methods to implement time series analysis..
CO2-PSO1	3	Strongly mapped as students will be understanding, analyzing, designing the problem based on models.
CO2-PSO2	3	Strongly mapped as students will be able to solve real time problems based on programming skills through regression models.

CO2-PSO3	2	Moderately mapped as students will be learning programming aspects of the course and implementing and developing applications in real life.
CO3-PO1	2	Moderately mapped as students will be learning programming through Linear Regression models.
CO3-PO2	2	Moderately mapped students can apply the basic knowledge of Statistical Inference in Linear Regression.
CO3-PO3	3	Strongly mapped as students will be able to design a model through Linear regression.
CO3-PO4	1	Slightly mapped as students will be able to design a model for building forecasting ARIMA processes.
CO3-PO5	3	Strongly mapped as students will be able to implement the models using ARIMA processes.
CO3-PO9	1	Slightly mapped as students will be able to implement models effectively as an individual and grouping.
CO3-PO11	1	Slightly mapped as students Demonstrate knowledge and understanding of engineering and to manage projects and in time series analysis & forecasting.
CO3-PO12	2	Moderately mapped as students will be able to learn different methods to implement Regression analysis and forecasting and Seasonal processes.

CO3-PSO1	3	Strongly mapped as students will be understanding, analyzing, designing the problem based on ARIMA models.
CO3-PSO2	3	Strongly mapped as students will be able to solve real time problems based on models through ARIMA models.
CO3-PSO3	2	Moderately mapped as students will be learning programming aspects of the course and implementing and developing applications in real life.
CO4-PO1	2	Moderately mapped as students will be learning programming aspects of the course based on ARIMA models..
CO4-PO2	2	Moderately mapped students can apply the basic knowledge of Linear models for stationary time series.
CO4-PO3	3	Strongly mapped as students will be able to design a program based on ARIMA models .
CO4-PO4	1	Slightly mapped as students will be able to design a model for General Time Series Data.
CO4-PO5	3	Strongly mapped as students will be able to design and implement various application programs based on I/O.
CO4-PO9	1	Slightly mapped as students will be able to implement models effectively as an individual and grouping.

CO4-PO11	1	Slightly mapped as students Demonstrate knowledge and understanding of engineering and to manage projects and in time series analysis.
CO4-PO12	2	Moderately mapped as students will be able to learn different methods to implement complex programs.
CO4-PSO1	3	Strongly mapped as students will be understanding, analyzing, designing the problem based on I/O.
CO4-PSO2	3	Strongly mapped as students will be able to solve real time problems based on programming skills.
CO4-PSO3	2	Moderately mapped as students will be learning programming aspects of the course and implementing and developing applications in real life.
CO5-PO1	2	Moderately mapped as students will be learning programming through Multivariate Time series models.
CO5-PO2	2	Moderately mapped students can apply the basic knowledge of Multivariate Time Series Models and Forecasting.
CO5-PO3	3	Strongly mapped as students will be able to design models based on Bayesian Methods in Forecasting.

CO5-PO4	1	Slightly mapped as students will be learning programming through Neural Networks and Forecasting and design experiments based on ARIMA models.
CO5-PO5	3	Strongly mapped as students will be able to design and develop various application programming tools based on ARIMA models.
CO5-PO9	1	Slightly mapped as students will be able to implement models effectively as an individual and grouping.
CO5-PO11	1	Slightly mapped as students will be able to project through these Bayesian Methods in Forecasting.
CO5-PO12	2	Moderately mapped as students will be able to learn different methods to implement Multivariate Time Series Models and Forecasting in lifelong learning.
CO5-PSO1	3	Strongly mapped as students will be understanding, analyzing, designing the problem based on Vector AR (VAR) Models.
CO5-PSO2	3	Strongly mapped as students will be able to solve real time problems based on Multivariate Time Series Models and Forecasting & Bayesian Methods in Forecasting.
CO5-PSO3	2	Moderately mapped as students will be learning programming aspects of the course and implementing and developing applications in real life.

Subject Code	Subject Name (Lab oriented Theory Courses)	Category	L	T	P	C
AI19642	TIME SERIES ANALYSIS AND FORECASTING	PC	3	0	2	4

Objectives:	
●	To understand the basic concepts of time series analysis.
●	To familiarize the basic statistical methods to modeling, analyzing, and forecasting time series data.
●	To learn the application of regression models for forecasting.
●	To explore Autoregressive Integrated Moving Average (ARIMA) Models.
●	To introduce multivariate time series and forecasting models.

UNIT-I	**INTRODUCTION OF TIMESERIES ANALYSIS**	**9**
Time Series and Forecasting -Different types of data-Internal structures of time series-Models for time series analysis-Autocorrelation and Partial Autocorrelation-Examples of Time series- Nature and uses of forecasting-Forecasting Process-Data for forecasting –Resources for forecasting.(T2-CHAPTER NO:1, T1-CHAPTER NO 1)		
UNIT-II	**STATISTICS BACKGROUND FOR FORECASTING**	**9**
Graphical Displays-Time Series Plots - Plotting Smoothed Data - Numerical Description of Time Series Data - Use of Data Transformations and Adjustments- General Approach to Time Series Modelling and Forecasting- Evaluating and Monitoring Forecasting Model Performance. (T1- CHAPTER NO:2)		
UNIT-III	**REGRESSION ANALYSIS AND FORECASTING**	**9**
Introduction - Least Squares Estimation in Linear Regression Models - Statistical Inference in Linear Regression-Prediction of New Observations - Model Adequacy Checking -Variable Selection Methods in Regression - Generalized and Weighted Least Squares- Regression Models for General Time Series Data. (T1- CHAPTER NO:2)		
UNIT-IV	**AUTOREGRESSIVE INTEGRATED MOVING AVERAGE (ARIMA) MODELS**	**9**
Linear models for stationary time series - Finite order moving average processes - Finite order autoregressive processes -Mixed autoregressive–moving average Processes – Non stationary processes - Time series model building forecasting ARIMA processes - Seasonal processes. (T1- CHAPTER NO:5)		
UNIT-V	**MULTIVARIATE TIME SERIES MODELS AND FORECASTING METHODS**	**9**
Multivariate Time Series Models and Forecasting - Multivariate Stationary Process- Vector ARIMA Models - Vector AR (VAR) Models - Neural Networks and Forecasting -Spectral Analysis – Bayesian Methods in Forecasting. (T1-CHAPTER NO:7)		
	Contact Hours :	**45**

List of Experiments	
1.	Implement programs for time series data cleaning, loading and handling times series data and pre-processing techniques.
2.	Implement programs for visualizing time series data.
3.	Implement programs to check stationary of a time series data.
4.	Implement programs for estimating & eliminating trend in time series data- aggregation, smoothing.
5.	Develop a linear regression model for forecasting time series data.
6.	Implement program to apply moving average smoothing for data preparation and time series forecasting.
7.	Implement program for decomposing time series data into trend and seasonality.
8.	Create an ARIMA model for time series forecasting.
9.	Develop neural network-based time series forecasting model.
10.	Develop vector auto regression model for multivariate time series data forecasting.
	Contact Hours : 30
	Total Contact Hours : 75

Course Outcomes: On completion of the course, the students will be able to	
●	Explain the basic concepts in time series analysis and forecasting.
●	Apply various time series models for forecasting.
●	Analyze various time series regression models.
●	Distinguish the ARIMA modelling of stationary and non stationary time series.
●	Compare with multivariate times series and other methods of applications.

Text Books:	
1	Introduction To Time Series Analysis and Forecasting, 2nd Edition, Wiley Series in Probability and Statistics, By Douglas C. Montgomery, Cheryl L. Jen (2015).
2	Master Time Series Data Processing, Visualization, And Modeling Using Python Dr. Avishek PalDr. Pks Prakash (2017) .

Reference Books:	
1	Time Series Analysis and Forecasting by Example Soren Bisgaard Murat Kulahci Technical University of Denmark Copyright c2011 By John Wiley & Sons, Inc.
2	Peter J. Brockwell Richard A. Davis Introduction to Time Series and Forecasting Third Edition. (2016).
3	Multivariate Time Series Analysis and Applications William W.S. Wei Department of Statistical Science Temple University, Philadelphia, PA, SA 2019 John Wiley & Sons Ltd 2019.
4	Time Series Analysis and Forecasting by Example Soren Bisgaard Murat Kulahci Technical University Of Denmark Copyright c 2011 By John Wiley & Sons, Inc.

Web link:

1. https://b-ok.cc/book/3413340/2eb247

2.https://b-ok.cc/book/2542456/2fa941

3.https://b-ok.cc/book/1183901/9be7ed

4. https://www.coursera.org/learn/practical-time-series-analysis

CO - PO – PSO matrices of course

PO/PSO CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
AI19642.1	2	1	1	1	2	-	-	-	-	-	-	2	3	3	2
AI19642.2	2	2	1	1	2	-	-	-	-	-	-	2	3	3	2
AI19642.3	2	2	3	1	3	-	-	-	1	-	1	2	3	3	2
AI19642.4	2	2	3	1	3	-	-	-	1	-	1	2	3	3	2
AI19642.5	2	2	3	1	3	-	-	-	1	-	1	2	3	3	2
Average	2	1.8	2.2	1	2.6	-	-	-	1	-	1	2	3	3	2

Correlation levels 1, 2 or 3 are as defined below:
1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)
No correlation: "-"

RAJALAKSHMI ENGINEERING COLLEGE
DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Lesson Plan - LAB

STAFF NAME AND CODE:	**MS.EUGENE BERNA I /IT185**
SUBJECT CODE / NAME	**AI19642 TIME SERIES ANALYSIS AND FORECASTING**
YEAR / SEMESTER:	**III/VI**

S.No.	**No of Hrs**	**Topic**	**Text / Reference Book**
		List of Experiments	
1	1	Implement programs for time series data cleaning, loading and handling times series data and preprocessing techniques.	T1
2	1	Implement programs for time series data cleaning, loading and handling times series data and preprocessing techniques.	T1
3	1	Implement programs for visualizing time series data.	T1
4	1	Implement programs for visualizing time series data.	T1
5	1	Implement programs to check stationary of a time series data.	T1
6	1	Implement programs to check stationary of a time series data.	T1
7	1	Implement programs for estimating & eliminating trends in time series data- aggregation, smoothing.	T1
8	1	Implement programs for estimating & eliminating trends in time series data- aggregation, smoothing.	T1
9	1	Develop a linear regression model for forecasting time series data.	T1
10	1	Develop a linear regression model for forecasting time series data.	T1
11	1	Implement a program to apply moving average smoothing for data preparation and time series forecasting.	T1
12	1	Implement a program to apply moving average smoothing for data preparation and time series forecasting.	T1
13	1	Implement a program for decomposing time series data into trend and seasonality.	T1

14	1	Implement a program for decomposing time series data into trend and seasonality.	T1
15	1	Create an ARIMA model for time series forecasting.	T1
16	1	Create an ARIMA model for time series forecasting.	T1
17	1	Develop neural network-based time series forecasting models.	T1
18	1	Develop neural network-based time series forecasting models.	T1
19	1	Develop vector auto regression model for multivariate time series data forecasting.	T1
20	1	Develop vector auto regression model for multivariate time series data forecasting.	T1
			TOTAL HOURS:30

Course Outcomes:

- On completion of the course, the students will be able to Explain the basic concepts in time series analysis and forecasting.
- Apply various time series models for forecasting.
- Analyze various time series regression models.
- Distinguish the ARIMA modeling of stationary and non stationary time series.
- Compare with multivariate times series and other methods of applications

Text Books:

1. Introduction To Time Series Analysis and Forecasting, 2nd Edition, Wiley Series in Probability and Statistics, By Douglas C. Montgomery, Cheryl L. Jen (2015).
2. Master Time Series Data Processing, Visualization, And Modeling Using Python Dr. Avishek PalDr. Pks Prakash (2017)

Reference Books:

1. Time Series Analysis and Forecasting by Example Soren Bisgaard Murat Kulahci Technical University of Denmark Copyright c2011 By John Wiley & Sons, Inc.
2. Peter J. Brockwell Richard A. Davis Introduction to Time Series and Forecasting Third Edition. (2016).
3. Multivariate Time Series Analysis and Applications William W.S. Wei Department of Statistical Science Temple University, Philadelphia, PA, SA 2019 John Wiley & Sons Ltd 2019.
4. Time Series Analysis and Forecasting by Example Soren Bisgaard Murat Kulahci Technical University Of Denmark Copyright c 2011 By John Wiley & Sons, Inc.

IMPLEMENT PROGRAMS FOR TIME SERIES DATA CLEANING, LOADING AND HANDLING TIMES SERIES DATA AND PRE-PROCESSING TECHNIQUES.

Aim: To implement programs for time series data cleaning, loading and handling time series data and preprocessing techniques.

Algorithm:

1. Start
2. Import the necessary libraries.
3. Drop columns that aren't useful.
4. Drop rows with missing values.
5. Create dummy variables.
6. Take care of missing data.
7. Convert the data frame to NumPy.
8. Divide the data set into training data and test data.
9. Stop.

Code:

```
import pandas as pd
import numpy as np
df = pd.read_csv("train.csv", parse_dates=["date"], index_col="date")
print(df.head())
df = df.drop(['id'], axis=1)
print(df.head())
df = df.dropna()
df = pd.get_dummies(df, columns=['family'])
print(df.head())
df.fillna(df.mean(), inplace=True)
np_data = df.to_numpy()
print(np_data)
train_data = np_data[:int(0.8 * np_data.shape[0])]
test_data = np_data[int(0.8 * np_data.shape[0]):]
print(train_data)
print(test_data)
```

Output:

```
date          id  store_nbr    family            sales   onpromotion

2013-01-01   0       1        AUTOMOTIVE        0.0          0
2013-01-01   1       1        BABY CARE         0.0          0
```

```
2013-01-01   2        1          BEAUTY              0.0          0
2013-01-01   3        1          BEVERAGES           0.0          0
2013-01-01   4        1          BOOKS               0.0          0

date                store_nbr     family           sales   onpromotion

2013-01-01          1   AUTOMOTIVE      0.0          0
2013-01-01          1   BABY CARE       0.0          0
2013-01-01          1   BEAUTY          0.0          0
2013-01-01          1   BEVERAGES       0.0          0
2013-01-01          1   BOOKS           0.0          0

date                store_nbr    sales   onpromotion       family_BABY CARE  family_BEAUTY

2013-01-01          1   0.0          0                0                     0
2013-01-01          1   0.0          0                1                     0
2013-01-01          1   0.0          0                0                     1
2013-01-01          1   0.0          0                0                     0
2013-01-01          1   0.0          0                0                     0

[[1.000000e+00 0.000000e+00 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]
 [1.000000e+00 0.000000e+00 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]
 [1.000000e+00 0.000000e+00 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]
…………………………

[[  1.      0.      0.    ...  0.      0.      0.   ]
 [  1.      0.      0.    ...  0.      0.      0.   ]
 [  1.      0.      0.    ...  0.      0.      0.   ]
 ………..
[[1.900000e+01 0.000000e+00 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]
 [1.900000e+01 4.000000e+00 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]
 [1.900000e+01 8.568900e+01 0.000000e+00 ... 0.000000e+00 0.000000e+00
  0.000000e+00]

 ……………
```

Result:

Thus the python program To implement programs for time series data cleaning, loading and handling time series data and preprocessing techniques is executed successfully and output is verified.

2.IMPLEMENT PROGRAMS FOR VISUALIZING TIME SERIES DATA

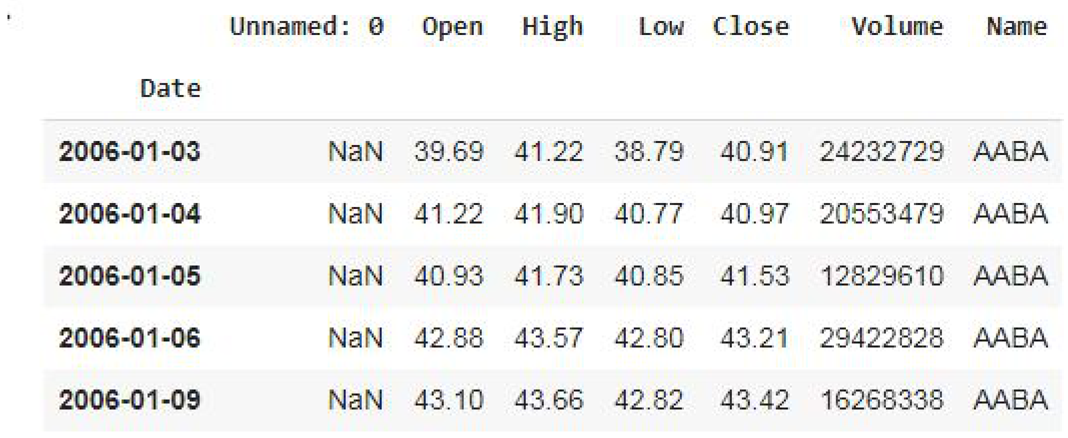
A time series is the series of data points listed in time order. A time series is a sequence of successive equal interval points in time. A time-series analysis consists of methods for analyzing time series data in order to extract meaningful insights and other useful characteristics of data. Time-series data analysis is becoming very important in so many industries like financial industries, pharmaceuticals, social media companies, web service providers, research, and many more. To understand the time-series data, visualizations are essential. Any type of data analysis is not complete without visualizations. Because one good visualization can provide meaningful and interesting insights into data.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Now loading the dataset by creating a dataframe df.

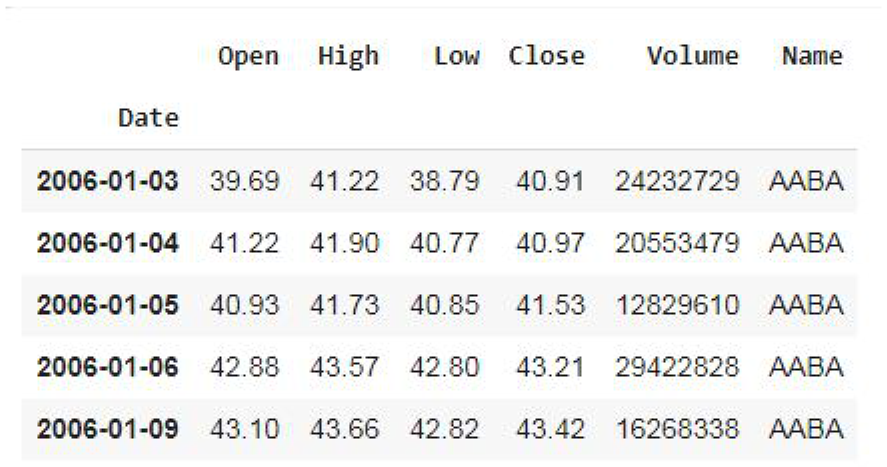
```
reading the dataset using read_csv
df = pd.read_csv("stock_data.csv",
              parse_dates=True,
              index_col="Date")

# displaying the first five rows of dataset
df.head()
```

Output:



```
            Unnamed: 0   Open   High    Low  Close    Volume  Name
Date
2006-01-03         NaN  39.69  41.22  38.79  40.91  24232729  AABA
2006-01-04         NaN  41.22  41.90  40.77  40.97  20553479  AABA
2006-01-05         NaN  40.93  41.73  40.85  41.53  12829610  AABA
2006-01-06         NaN  42.88  43.57  42.80  43.21  29422828  AABA
2006-01-09         NaN  43.10  43.66  42.82  43.42  16268338  AABA
```

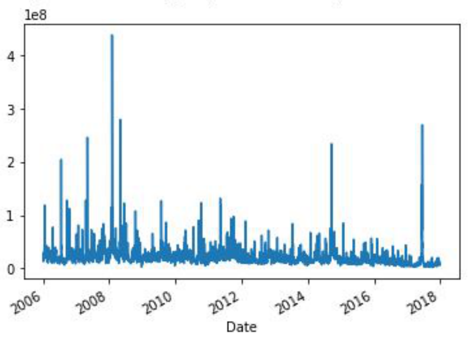
```
# deleting column
df.drop(columns='Unnamed: 0')
```



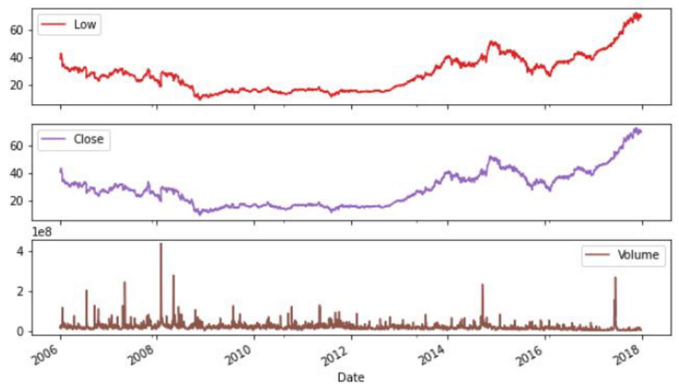
	Open	High	Low	Close	Volume	Name
Date						
2006-01-03	39.69	41.22	38.79	40.91	24232729	AABA
2006-01-04	41.22	41.90	40.77	40.97	20553479	AABA
2006-01-05	40.93	41.73	40.85	41.53	12829610	AABA
2006-01-06	42.88	43.57	42.80	43.21	29422828	AABA
2006-01-09	43.10	43.66	42.82	43.42	16268338	AABA

Example 1: Plotting a simple line plot for time series data.
df['Volume'].plot()

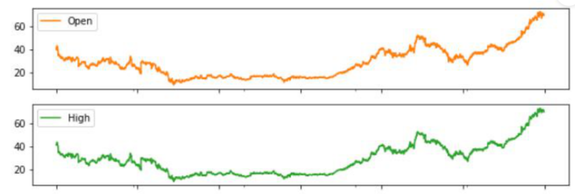
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91c723f1d0>
```



df.plot(subplots=True, figsize=(10, 12))

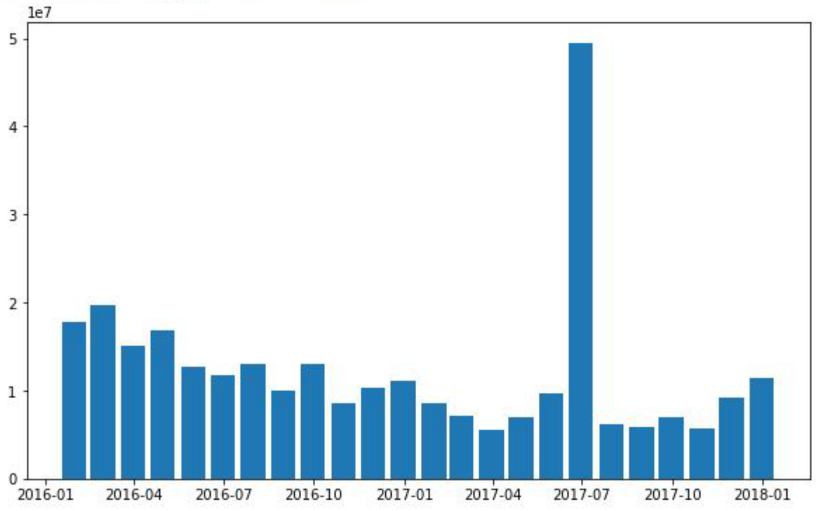


Output:



```
# Resampling the time series data based on monthly 'M' frequency
df_month = df.resample("M").mean()

# using subplot
fig, ax = plt.subplots(figsize=(10, 6))

# plotting bar graph
ax.bar(df_month['2016':].index,
      df_month.loc['2016':, "Volume"],
      width=25, align='center')
```

```
<BarContainer object of 24 artists>
```

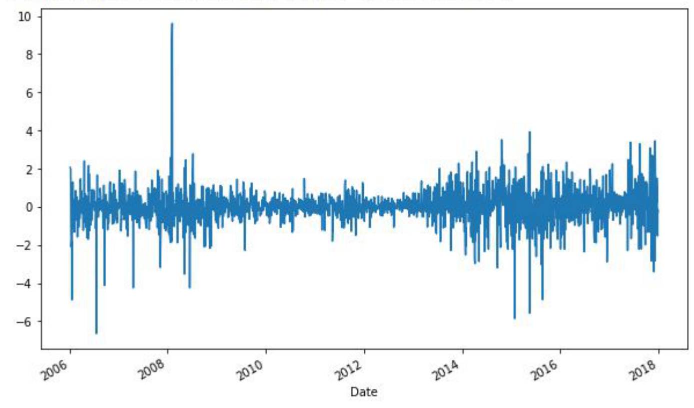


Differencing: Differencing is used to make the difference in values of a specified interval. By default, it's one, we can specify different values for plots. It is the most popular method to remove trends in the data.
Example 4:

- Python3

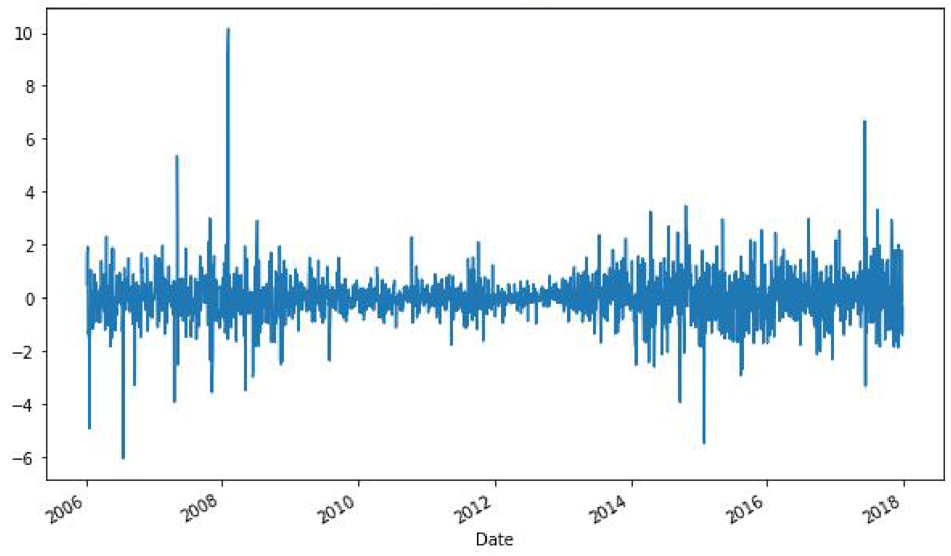
```
df.Low.diff(2).plot(figsize=(10, 6))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91c4ae0150>
```



```
df.High.diff(2).plot(figsize=(10, 6))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91c49fdb10>
```



Plotting the Changes in Data

We can also plot the changes that occurred in data over time. There are a few ways to plot changes in data.

Shift: The shift function can be used to shift the data before or after the specified time interval. We can specify the time, and it will shift the data by one day by default. That means we will get the previous

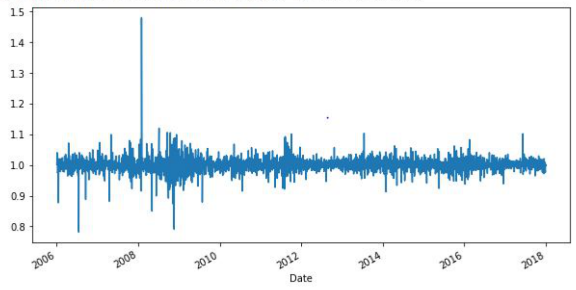
```
df['Change'] = df.Close.div(df.Close.shift())
df['Change'].plot(figsize=(10, 8), fontsize=16)
```

In this code, .div() function helps to fill up the missing data values. Actually, div() means division. If we take df. div(6) it will divide each element in df by 6. We do this to avoid the null or missing values that are created by the 'shift()' operation.

Here, we have taken .div(df.Close.shift()), it will divide each value of df to df.Close.shift() to remove null values.

Output:
day's data. It is helpful to see previous day data and today's data simultaneously side by side.

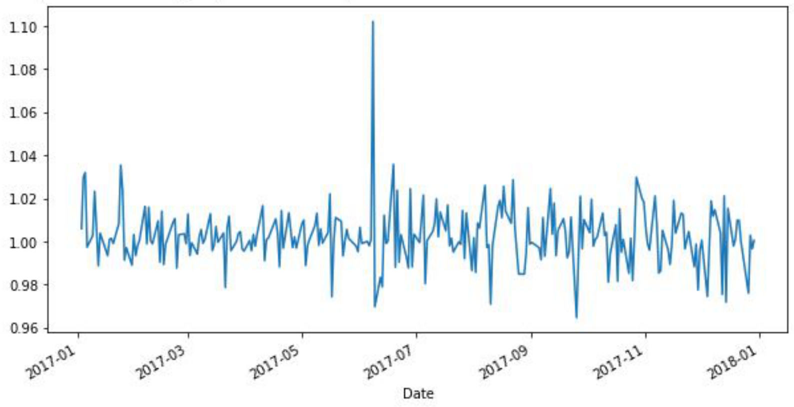
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91c4689850>
```



```
df['2017']['Change'].plot(figsize=(10, 6))
```

Output:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91c45d1fd0>
```



3. IMPLEMENT PROGRAMS TO CHECK STATIONARY OF A TIME SERIES DATA

Stationary Time Series

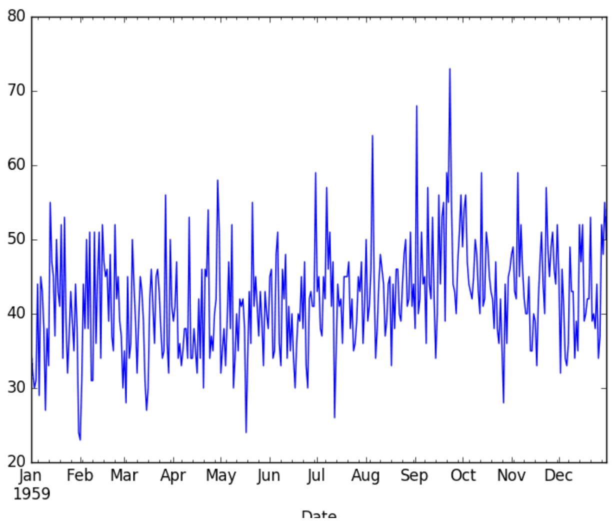
The observations in a stationary time series are not dependent on time.

Time series are stationary if they do not have trend or seasonal effects. Summary statistics calculated on the time series are consistent over time, like the mean or the variance of the observations.
When a time series is stationary, it can be easier to model. Statistical modeling methods assume or require the time series to be stationary to be effective.

- Download the the dataset and save it as: daily-total-female-births.csv.

Below is an example of loading the Daily Female Births dataset that is stationary.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
series.plot()
pyplot.show()
```



Non-Stationary Time Series

Observations from a non-stationary time series show seasonal effects, trends, and other structures that depend on the time index.

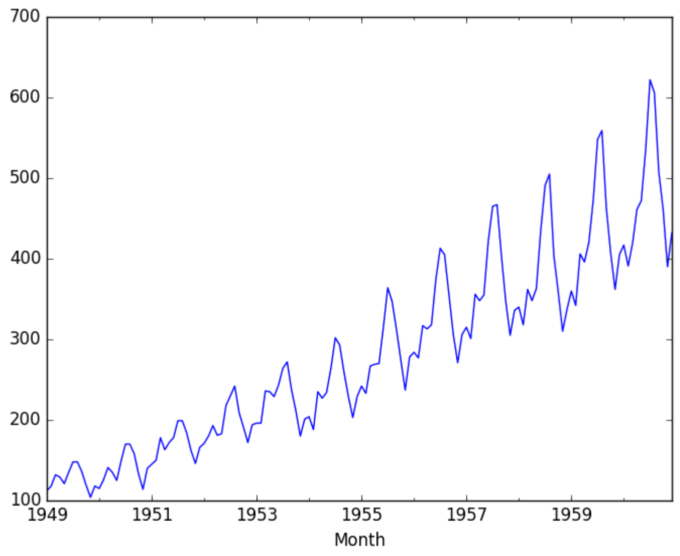
Summary statistics like the mean and variance do change over time, providing a drift in the concepts a model may try to capture.

Classical time series analysis and forecasting methods are concerned with making non-stationary time series data stationary by identifying and removing trends and removing seasonal effects.

- Download the dataset and saved it as: international-airline-passengers.csv.

Below is an example of the Airline Passengers dataset that is non-stationary, showing both trend and seasonal components.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('international-airline-passengers.csv', header=0, index_col=0)
series.plot()
pyplot.show()
```



Checks for Stationarity

There are many methods to check whether a time series (direct observations, residuals, otherwise) is stationary or non-stationary.

1. **Look at Plots**: You can review a time series plot of your data and visually check if there are any obvious trends or seasonality.
2. **Summary Statistics**: You can review the summary statistics for your data for seasons or random partitions and check for obvious or significant differences.
3. **Statistical Tests**: You can use statistical tests to check if the expectations of stationarity are met or have been violated.

Above, we have already introduced the Daily Female Births and Airline Passengers datasets as stationary and non-stationary respectively with plots showing an obvious lack and presence of trend and seasonality components.

Next, we will look at a quick and dirty way to calculate and review summary statistics on our time series dataset for checking to see if it is stationary.

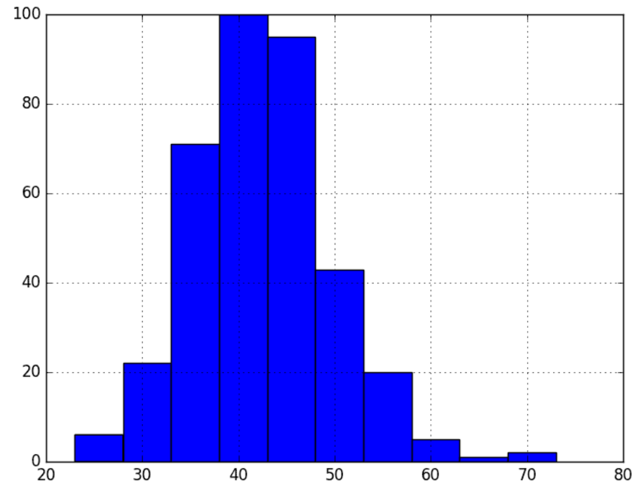
Daily Births Dataset

Because we are looking at the mean and variance, we are assuming that the data conforms to a Gaussian (also called the bell curve or normal) distribution.

We can also quickly check this by eyeballing a histogram of our observations.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
series.hist()
pyplot.show()
```

Running the example plots a histogram of values from the time series. We clearly see the bell curve-like shape of the Gaussian distribution, perhaps with a longer right tail.



How to Check if Time Series Data is Stationary with Python

by **Jason Brownlee** on December 30, 2016 in **Time Series**
Tweet Tweet **Share Share**
Last Updated on August 15, 2020

Time series is different from more traditional classification and regression predictive modeling problems.

The temporal structure adds an order to the observations. This imposed order means that important assumptions about the consistency of those observations needs to be handled specifically.

For example, when modeling, there are assumptions that the summary statistics of observations are consistent. In time series terminology, we refer to this expectation as the time series being stationary.

These assumptions can be easily violated in time series by the addition of a trend, seasonality, and other time-dependent structures.

In this tutorial, you will discover how to check if your time series is stationary with Python.

After completing this tutorial, you will know:

- How to identify obvious stationary and non-stationary time series using line plot.
- How to spot check summary statistics like mean and variance for a change over time.

- How to use statistical tests with statistical significance to check if a time series is stationary.

Kick-start your project with my new book Time Series Forecasting With Python, including *step-by-step tutorials* and the *Python source code* files for all examples.
Let's get started.

- **Updated Feb/2017**: Fixed typo in interpretation of p-value, added bullet points to make it clearer.
- **Updated May/2018**: Improved language around reject vs fail to reject of statistical tests.
- **Updated Apr/2019**: Updated the link to dataset.
- **Updated Aug/2019**: Updated data loading to use new API.
- **Updated Nov/2019**: Updated mean/variance example for Python 3, also updated bug in data loading (thanks John).

How to Check if Time Series Data is Stationary with Python
Photo by Susanne Nilsson, some rights reserved.

Stationary Time Series

The observations in a stationary time series are not dependent on time.

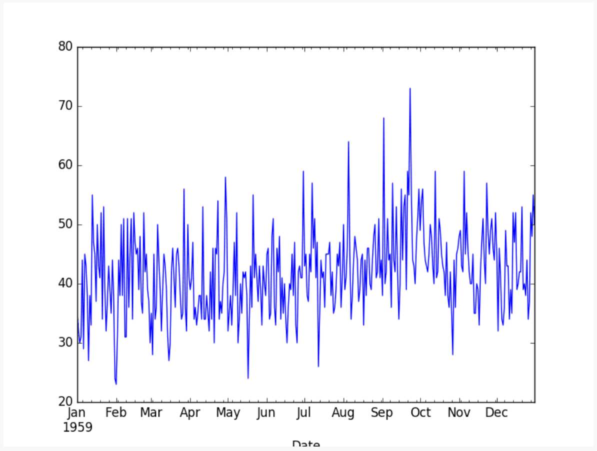
Time series are stationary if they do not have trend or seasonal effects. Summary statistics calculated on the time series are consistent over time, like the mean or the variance of the observations.
When a time series is stationary, it can be easier to model. Statistical modeling methods assume or require the time series to be stationary to be effective.

- Download the the dataset and save it as: daily-total-female-births.csv.

Below is an example of loading the Daily Female Births dataset that is stationary.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
series.plot()
pyplot.show()
```

Running the example creates the following plot.



Daily Female Births Dataset Plot

Non-Stationary Time Series

Observations from a non-stationary time series show seasonal effects, trends, and other structures that depend on the time index.

Summary statistics like the mean and variance do change over time, providing a drift in the concepts a model may try to capture.

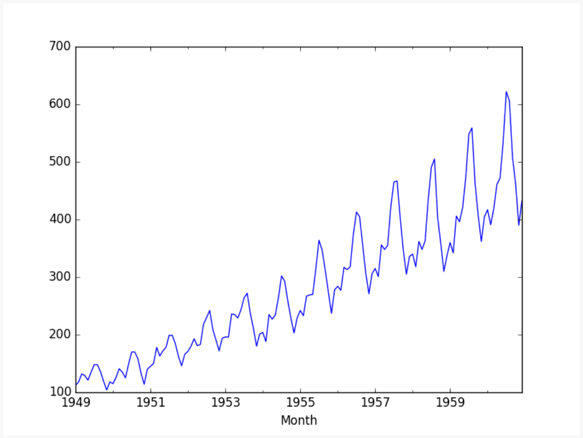
Classical time series analysis and forecasting methods are concerned with making non-stationary time series data stationary by identifying and removing trends and removing seasonal effects.

- Download the dataset and saved it as: international-airline-passengers.csv.

Below is an example of the Airline Passengers dataset that is non-stationary, showing both trend and seasonal components.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('international-airline-passengers.csv', header=0, index_col=0)
series.plot()
pyplot.show()
```

Running the example creates the following plot.



Non-Stationary Airline Passengers Dataset

Types of Stationary Time Series

The notion of stationarity comes from the theoretical study of time series and it is a useful abstraction when forecasting.

There are some finer-grained notions of stationarity that you may come across if you dive deeper into this topic. They are:

They are:

- **Stationary Process**: A process that generates a stationary series of observations.
- **Stationary Model**: A model that describes a stationary series of observations.
- **Trend Stationary**: A time series that does not exhibit a trend.
- **Seasonal Stationary**: A time series that does not exhibit seasonality.
- **Strictly Stationary**: A mathematical definition of a stationary process, specifically that the joint distribution of observations is invariant to time shift.

Stationary Time Series and Forecasting

Should you make your time series stationary?

Generally, yes.

If you have clear trend and seasonality in your time series, then model these components, remove them from observations, then train models on the residuals.

If we fit a stationary model to data, we assume our data are a realization of a stationary process. So our first step in an analysis should be to check whether there is any evidence of a trend or seasonal effects and, if there is, remove them.

— Page 122, Introductory Time Series with R.
Statistical time series methods and even modern machine learning methods will benefit from the clearer signal in the data.

But…

We turn to machine learning methods when the classical methods fail. When we want more or better results. We cannot know how to best model unknown nonlinear relationships in time series data and some methods may result in better performance when working with non-stationary observations or some mixture of stationary and non-stationary views of the problem.

The suggestion here is to treat properties of a time series being stationary or not as another source of information that can be used in feature engineering and feature selection on your time series problem when using machine learning methods.

Checks for Stationarity

There are many methods to check whether a time series (direct observations, residuals, otherwise) is stationary or non-stationary.

1. **Look at Plots**: You can review a time series plot of your data and visually check if there are any obvious trends or seasonality.
2. **Summary Statistics**: You can review the summary statistics for your data for seasons or random partitions and check for obvious or significant differences.
3. **Statistical Tests**: You can use statistical tests to check if the expectations of stationarity are met or have been violated.

Above, we have already introduced the Daily Female Births and Airline Passengers datasets as stationary and non-stationary respectively with plots showing an obvious lack and presence of trend and seasonality components.

Next, we will look at a quick and dirty way to calculate and review summary statistics on our time series dataset for checking to see if it is stationary.

Summary Statistics

A quick and dirty check to see if your time series is non-stationary is to review summary statistics.

You can split your time series into two (or more) partitions and compare the mean and variance of each group. If they differ and the difference is statistically significant, the time series is likely non-stationary.

Next, let’s try this approach on the Daily Births dataset.

Daily Births Dataset

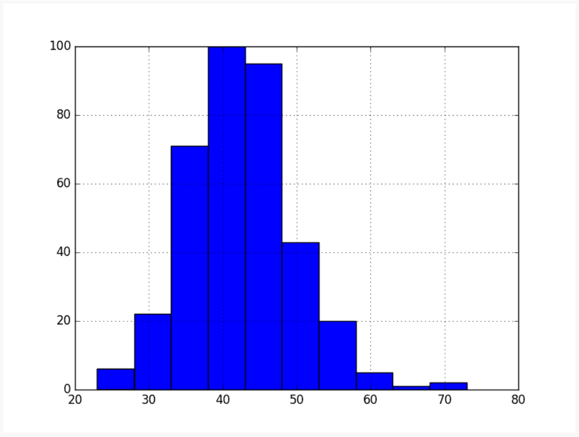
Because we are looking at the mean and variance, we are assuming that the data conforms to a Gaussian (also called the bell curve or normal) distribution.

We can also quickly check this by eyeballing a histogram of our observations.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
```

```
series.hist()
pyplot.show()
```

Running the example plots a histogram of values from the time series. We clearly see the bell curve-like shape of the Gaussian distribution, perhaps with a longer right tail.



Histogram of Daily Female Births

Next, we can split the time series into two contiguous sequences. We can then calculate the mean and variance of each group of numbers and compare the values.

```
from pandas import read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
X = series.values
split = round(len(X) / 2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
var1, var2 = X1.var(), X2.var()
print('mean1=%f, mean2=%f' % (mean1, mean2))
print('variance1=%f, variance2=%f' % (var1, var2))
```

Running this example shows that the mean and variance values are different, but in the same ball-park.

```
mean1=39.763736, mean2=44.185792
variance1=49.213410, variance2=48.708651
```

4. IMPLEMENT PROGRAMS FOR ESTIMATING & ELIMINATING TREND IN TIME SERIES DATA- AGGREGATION, SMOOTHING

A trend is a continued increase or decrease in the series over time. There can be benefit in identifying, modeling, and even removing trend information from your time series dataset.

In this tutorial, you will discover how to model and remove trend information from time series data in Python.

After completing this tutorial, you will know:

- The importance and types of trends that may exist in time series and how to identify them.
- How to use a simple differencing method to remove a trend.
- How to model a linear trend and remove it from a sales time series dataset.
- **Faster Modeling**: Perhaps the knowledge of a trend or lack of a trend can suggest methods and make model selection and evaluation more efficient.
- **Simpler Problem**: Perhaps we can correct or remove the trend to simplify modeling and improve model performance.
- **More Data**: Perhaps we can use trend information, directly or as a summary, to provide additional information to the model and improve model performance.

Types of Trends

There are all kinds of trends.

Two general classes that we may think about are:

- **Deterministic Trends**: These are trends that consistently increase or decrease.
- **Stochastic Trends**: These are trends that increase and decrease inconsistently.

In general, deterministic trends are easier to identify and remove, but the methods discussed in this tutorial can still be useful for stochastic trends.

We can think about trends in terms of their scope of observations.

- **Global Trends**: These are trends that apply to the whole time series.
- **Local Trends**: These are trends that apply to parts or subsequences of a time series.

dentifying a Trend

You can plot time series data to see if a trend is obvious or not.

The difficulty is that in practice, identifying a trend in a time series can be a subjective process. As such, extracting or removing it from the time series can be just as subjective.

Create line plots of your data and inspect the plots for obvious trends.

Add linear and nonlinear trend lines to your plots and see if a trend is obvious.

Removing a Trend

A time series with a trend is called non-stationary.

An identified trend can be modeled. Once modeled, it can be removed from the time series dataset. This is called detrending the time series.

If a dataset does not have a trend or we successfully remove the trend, the dataset is said to be trend stationary.

Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3 year period.

The units are a sales count and there are 36 observations. The original dataset is credited to Makridakis, Wheelwright, and Hyndman (1998).

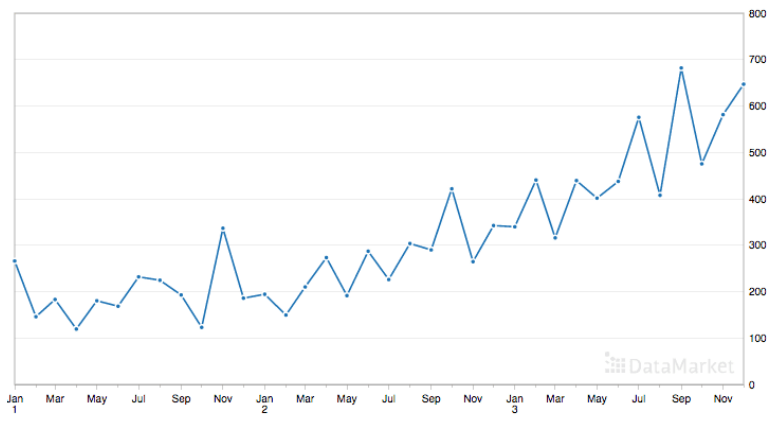
- Download the dataset.

Below is a sample of the first 5 rows of data, including the header row.

```
"Month","Sales"
"1-01",266.0
"1-02",145.9
"1-03",183.1
"1-04",119.3
"1-05",180.3
```

Below is a plot of the entire dataset, where you can learn more and download the dataset.

The dataset shows an increasing trend.



How to Use and Remove Trend Information from Time Series Data in Python

Our time series dataset may contain a trend.

A trend is a continued increase or decrease in the series over time. There can be benefit in identifying, modeling, and even removing trend information from your time series dataset.

In this tutorial, you will discover how to model and remove trend information from time series data in Python.

After completing this tutorial, you will know:

- The importance and types of trends that may exist in time series and how to identify them.
- How to use a simple differencing method to remove a trend.
- How to model a linear trend and remove it from a sales time series dataset.

Kick-start your project with my new book Time Series Forecasting With Python, including *step-by-step tutorials* and the *Python source code* files for all examples.
Let's get started.

Trends in Time Series

A trend is a long-term increase or decrease in the level of the time series.

In general, a systematic change in a time series that does not appear to be periodic is known as a trend.

— Page 5, Introductory Time Series with R
Identifying and understanding trend information can aid in improving model performance; below are a few reasons:

- **Faster Modeling**: Perhaps the knowledge of a trend or lack of a trend can suggest methods and make model selection and evaluation more efficient.
- **Simpler Problem**: Perhaps we can correct or remove the trend to simplify modeling and improve model performance.
- **More Data**: Perhaps we can use trend information, directly or as a summary, to provide additional information to the model and improve model performance.

Types of Trends

There are all kinds of trends.

Two general classes that we may think about are:

- **Deterministic Trends**: These are trends that consistently increase or decrease.
- **Stochastic Trends**: These are trends that increase and decrease inconsistently.

In general, deterministic trends are easier to identify and remove, but the methods discussed in this tutorial can still be useful for stochastic trends.

We can think about trends in terms of their scope of observations.

- **Global Trends**: These are trends that apply to the whole time series.
- **Local Trends**: These are trends that apply to parts or subsequences of a time series.

Generally, global trends are easier to identify and address.

Identifying a Trend

You can plot time series data to see if a trend is obvious or not.

The difficulty is that in practice, identifying a trend in a time series can be a subjective process. As such, extracting or removing it from the time series can be just as subjective.

Create line plots of your data and inspect the plots for obvious trends.

Add linear and nonlinear trend lines to your plots and see if a trend is obvious.

Removing a Trend

A time series with a trend is called non-stationary.

An identified trend can be modeled. Once modeled, it can be removed from the time series dataset. This is called detrending the time series.

If a dataset does not have a trend or we successfully remove the trend, the dataset is said to be trend stationary.

Using Time Series Trends in Machine Learning

From a machine learning perspective, a trend in your data represents two opportunities:

1. **Remove Information**: To remove systematic information that distorts the relationship between input and output variables.
2. **Add Information**: To add systematic information to improve the relationship between input and output variables.

Specifically, a trend can be removed from your time series data (and data in the future) as a data preparation and cleaning exercise. This is common when using statistical methods for time series forecasting, but does not always improve results when using machine learning models.

Alternately, a trend can be added, either directly or as a summary, as a new input variable to the supervised learning problem to predict the output variable.

One or both approaches may be relevant for your time series forecasting problem and may be worth investigating.

Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3 year period.

The units are a sales count and there are 36 observations. The original dataset is credited to Makridakis, Wheelwright, and Hyndman (1998).

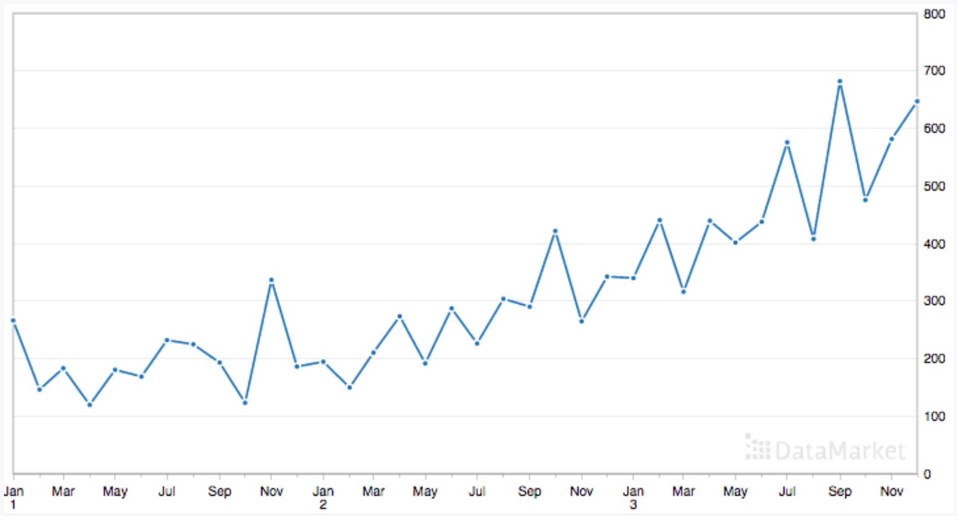
- Download the dataset.

Below is a sample of the first 5 rows of data, including the header row.

```
"Month","Sales"
"1-01",266.0
"1-02",145.9
"1-03",183.1
"1-04",119.3
"1-05",180.3
```

Below is a plot of the entire dataset, where you can learn more and download the dataset.

The dataset shows an increasing trend.



Shampoo Sales Dataset

Load the Shampoo Sales Dataset

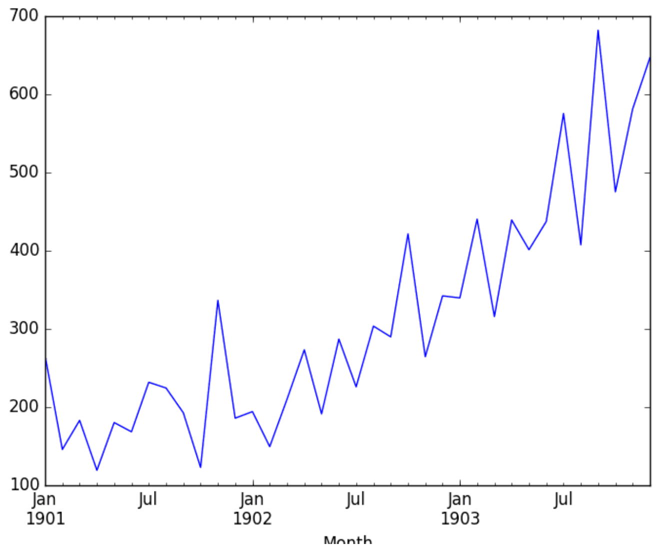
Download the dataset and place it in the current working directory with the filename "*shampoo-sales.csv*".

- Download the dataset.

The dataset can be loaded with a custom date parsing routine as follows:

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')


```

```
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
squeeze=True, date_parser=parser)
series.plot()
pyplot.show()
```



Detrend by Differencing

Perhaps the simplest method to detrend a time series is by differencing.

Specifically, a new series is constructed where the value at the current time step is calculated as the difference between the original observation and the observation at the previous time step.

```
value(t) = observation(t) - observation(t-1)
```

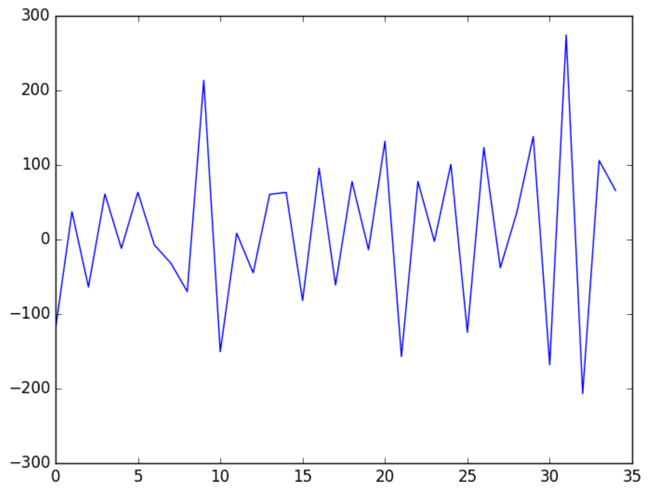
This has the effect of removing a trend from a time series dataset.

We can create a new difference dataset in Python by implementing this directly. A new list of observations can be created.

Below is an example that creates the difference detrended version of the Shampoo Sales dataset.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
squeeze=True, date_parser=parser)
X = series.values
diff = list()
```

```
for i in range(1, len(X)):
    value = X[i] - X[i - 1]
    diff.append(value)
pyplot.plot(diff)
pyplot.show()
```



Detrend by Model Fitting

A trend is often easily visualized as a line through the observations.

Linear trends can be summarized by a linear model, and nonlinear trends may be best summarized using a polynomial or other curve-fitting method.

Because of the subjective and domain-specific nature of identifying trends, this approach can help to identify whether a trend is present. Even fitting a linear model to a trend that is clearly super-linear or exponential can be helpful.

In addition to being used as a trend identification tool, these fit models can also be used to detrend a time series.

For example, a linear model can be fit on the time index to predict the observation. This dataset would look as follows:

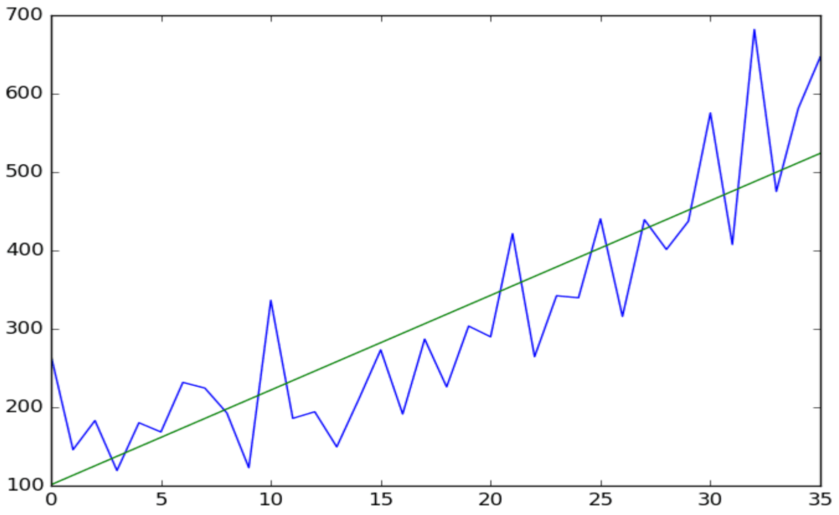
```
X,     y
1,     obs1
2,     obs2
3,     obs3
4,     obs4
5,     obs5
```

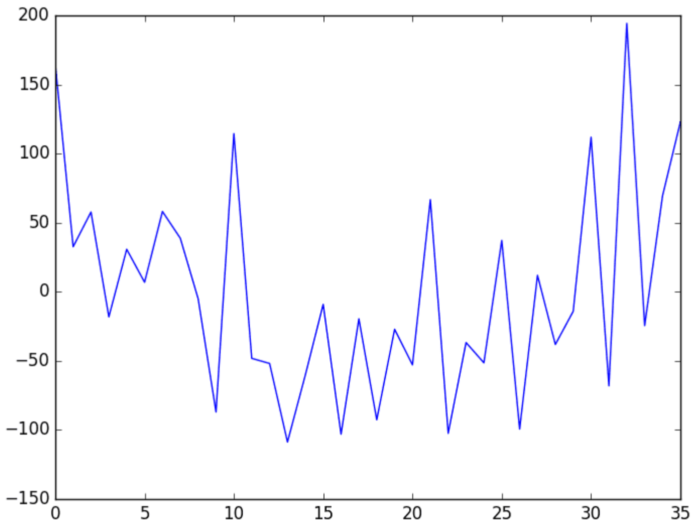
The predictions from this model will form a straight line that can be taken as the trend line for the dataset. These predictions can also be subtracted from the original time series to provide a detrended version of the dataset.

```
1value(t) = observation(t) - prediction(t)
```
The residuals from the fit of the model are a detrended form of the dataset. Polynomial curve fitting and other nonlinear models can also be used.

We can implement this in Python by training a scikit-learn LinearRegression model on the data.

```
from pandas import datetime
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot
import numpy

def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True,
date_parser=parser)
# fit linear model
X = [i for i in range(0, len(series))]
X = numpy.reshape(X, (len(X), 1))
y = series.values
model = LinearRegression()
model.fit(X, y)
# calculate trend
trend = model.predict(X)
# plot trend
pyplot.plot(y)
pyplot.plot(trend)
pyplot.show()
# detrend
detrended = [y[i]-trend[i] for i in range(0, len(series))]
# plot detrended
pyplot.plot(detrended)
pyplot.show()
```


700
600
500
400
300
200
100
0
5
10
15
20
25
30
35


200
150
100
50
0
−50
−100
−150
0
5
10
15
20
25
30
35

5.DEVELOP A LINEAR REGRESSION MODEL FOR FORECASTING TIME SERIES DATA

For the first part of this course, we'll use the linear regression algorithm to construct forecasting models. Linear regression is widely used in practice and adapts naturally to even complex forecasting tasks.

The **linear regression** algorithm learns how to make a weighted sum from its input features. For two features, we would have:

target = weight_1 * feature_1 + weight_2 * feature_2 + bias

During training, the regression algorithm learns values for the parameters weight_1, weight_2, and bias that best fit the target. (This algorithm is often called *ordinary least squares* since it chooses values that minimize the squared error between the target and the predictions.) The weights are also called *regression coefficients* and the bias is also called the *intercept* because it tells you where the graph of this function crosses the y-axis.

Time-step features

There are two kinds of features unique to time series: time-step features and lag features.

Time-step features are features we can derive directly from the time index. The most basic time-step feature is the **time dummy**, which counts off time steps in the series from beginning to end.

```
import numpy as np

df['Time'] = np.arange(len(df.index))
```

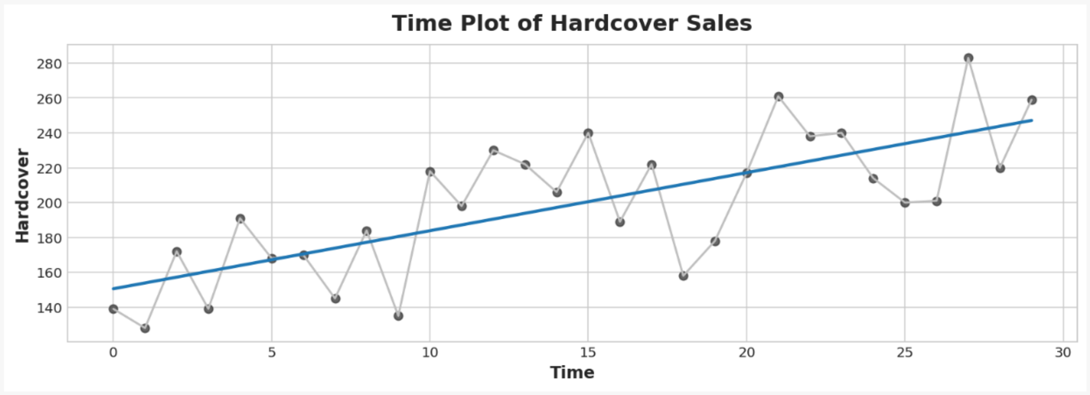
```
df.head()
Out[2]:
```

	Hardcover	Time
Date		
2000-04-01	139	0
2000-04-02	128	1
2000-04-03	172	2
2000-04-04	139	3
2000-04-05	191	4

Linear regression with the time dummy produces the model:

```
target = weight * time + bias
```

The time dummy then lets us fit curves to time series in a time plot, where Time forms the x-axis.



Time-step features let you model time dependence. A series is time dependent if its values can be predicted from the time they occured. In the Hardcover Sales series, we can predict that sales later in the month are generally higher than sales earlier in the month.

Lag features

To make a lag feature we shift the observations of the target series so that they appear to have occured later in time. Here we've created a 1-step lag feature, though shifting by multiple steps is possible too.

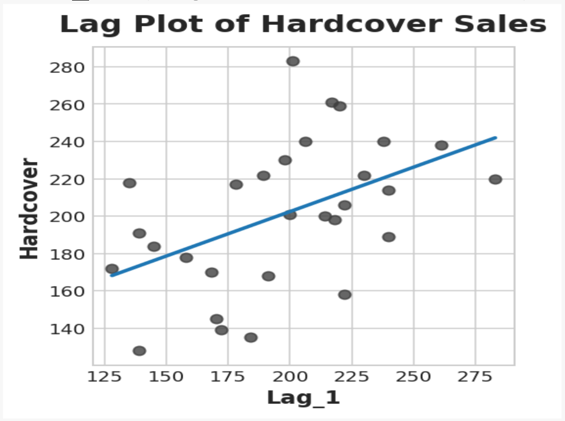
```
df['Lag_1'] = df['Hardcover'].shift(1)
df = df.reindex(columns=['Hardcover', 'Lag_1'])

df.head()
Out[4]:
```

	Hardcover	Lag_1
Date		
2000-04-01	139	NaN
2000-04-02	128	139.0

	Hardcover	Lag_1
Date		
2000-04-03	172	128.0
2000-04-04	139	172.0
2000-04-05	191	139.0

Linear regression with a lag feature produces the model:

```
target = weight * lag + bias
fig, ax = plt.subplots()
ax = sns.regplot(x='Lag_1', y='Hardcover', data=df, ci=None, scatter_kws=dict(color='0.25'))
ax.set_aspect('equal')
ax.set_title('Lag Plot of Hardcover Sales');
```



you can see from the lag plot that sales on one day (Hardcover) are correlated with sales from the previous day (Lag_1). When you see a relationship like this, you know a lag feature will be useful.

More generally, lag features let you model serial dependence. A time series has serial dependence when an observation can be predicted from previous observations. In Hardcover Sales, we can predict that high sales on one day usually mean high sales the next day.

Adapting machine learning algorithms to time series problems is largely about feature engineering with the time index and lags. For most of the course, we use linear regression for its simplicity, but these features will be useful whichever algorithm you choose for your forecasting task.

```
from pathlib import Path
from warnings import simplefilter

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

simplefilter("ignore")  # ignore warnings to clean up output cells

# Set Matplotlib defaults
```

```
plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True, figsize=(11, 4))
plt.rc(
    "axes",
    labelweight="bold",
    labelsize="large",
    titleweight="bold",
    titlesize=14,
    titlepad=10,
)
plot_params = dict(
    color="0.75",
    style=".-",
    markeredgecolor="0.25",
    markerfacecolor="0.25",
    legend=False,
)
%config InlineBackend.figure_format = 'retina'


# Load Tunnel Traffic dataset
data_dir = Path("../input/ts-course-data")
tunnel = pd.read_csv(data_dir / "tunnel.csv", parse_dates=["Day"])

# Create a time series in Pandas by setting the index to a date
# column. We parsed "Day" as a date type by using `parse_dates` when
# loading the data.
tunnel = tunnel.set_index("Day")

# By default, Pandas creates a `DatetimeIndex` with dtype `Timestamp`
# (equivalent to `np.datetime64`, representing a time series as a
# sequence of measurements taken at single moments. A `PeriodIndex`,
# on the other hand, represents a time series as a sequence of
# quantities accumulated over periods of time. Periods are often
# easier to work with, so that's what we'll use in this course.
tunnel = tunnel.to_period()

tunnel.head()
Out[6]:
```

	NumVehicles
Day	
2003-11-01	103536

	NumVehicles
Day	
2003-11-02	92051
2003-11-03	100795
2003-11-04	102352
2003-11-05	106569

Time-step feature

Provided the time series doesn't have any missing dates, we can create a time dummy by counting out the length of the series.

In [7]:

```
df = tunnel.copy()

df['Time'] = np.arange(len(tunnel.index))

df.head()
```

Out[7]:

	NumVehicles	Time
Day		
2003-11-01	103536	0
2003-11-02	92051	1
2003-11-03	100795	2
2003-11-04	102352	3
2003-11-05	106569	4

The procedure for fitting a linear regression model follows the standard steps for scikit-learn.

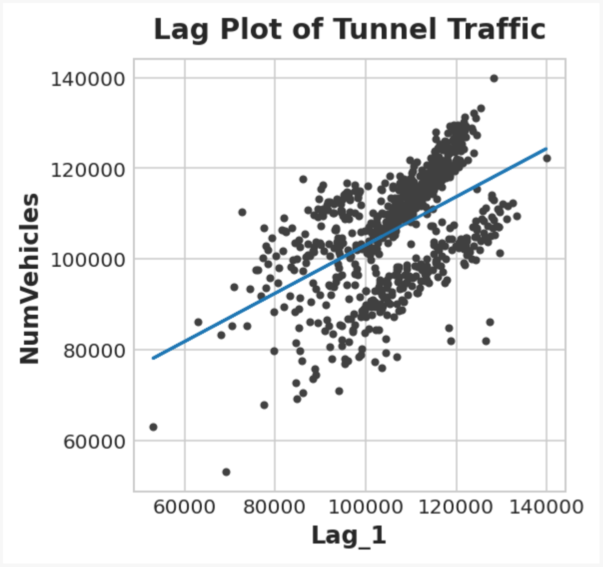
In [8]:
```
from sklearn.linear_model import LinearRegression

# Training data
X = df.loc[:, ['Time']]  # features
y = df.loc[:, 'NumVehicles']  # target

# Train the model
model = LinearRegression()
model.fit(X, y)

# Store the fitted values as a time series with the same time index as
# the training data
y_pred = pd.Series(model.predict(X), index=X.index)
```
linkcode
The model actually created is (approximately): Vehicles = 22.5 * Time + 98176. Plotting the fitted values over time shows us how fitting linear regression to the time dummy creates the trend line defined by this equation.
Lag feature
Pandas provides us a simple method to lag a series, the shift method.
In [10]:
```
df['Lag_1'] = df['NumVehicles'].shift(1)
df.head()
```
Out[10]:

	NumVehicles	Time	Lag_1
Day			
2003-11-01	103536	0	NaN
2003-11-02	92051	1	103536.0
2003-11-03	100795	2	92051.0
2003-11-04	102352	3	100795.0
2003-11-05	106569	4	102352.0

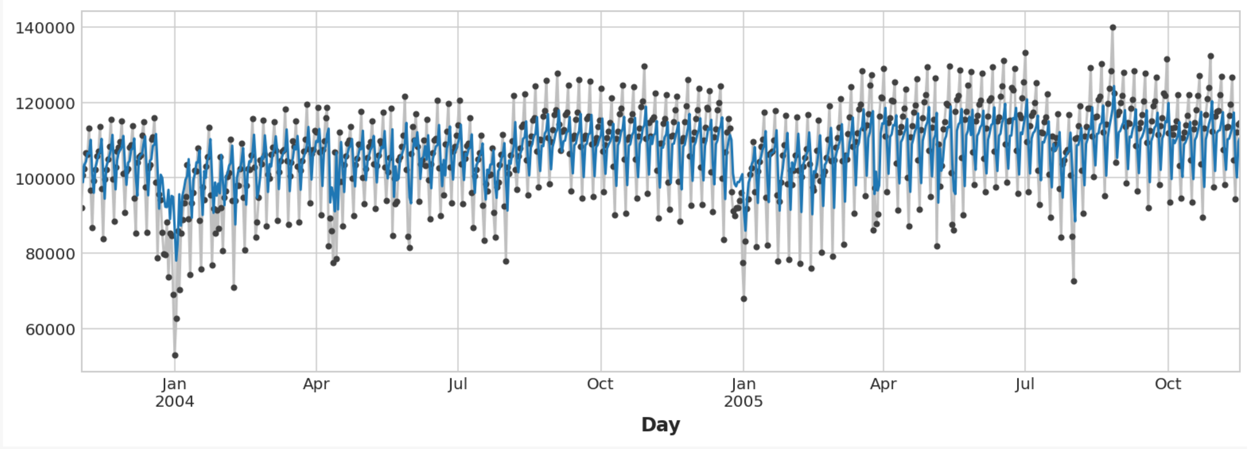
When creating lag features, we need to decide what to do with the missing values produced. Filling them in is one option, maybe with 0.0 or "backfilling" with the first known value. Instead, we'll just drop the missing values, making sure to also drop values in the target from corresponding dates.
In [11]:

```
from sklearn.linear_model import LinearRegression

X = df.loc[:, ['Lag_1']]
X.dropna(inplace=True)  # drop missing values in the feature set
y = df.loc[:, 'NumVehicles']  # create the target
y, X = y.align(X, join='inner')  # drop corresponding values in target

model = LinearRegression()
model.fit(X, y)

y_pred = pd.Series(model.predict(X), index=X.index)
```

The lag plot shows us how well we were able to fit the relationship between the number of vehicles one day and the number the previous day.



What does this prediction from a lag feature mean about how well we can predict the series across time? The following time plot shows us how our forecasts now respond to the behavior of the series in the recent past.



The best time series models will usually include some combination of time-step features and lag features. Over the next few lessons, we'll learn how to engineer features modeling the most common patterns in time series using the features from this lesson as a starting point.

6.IMPLEMENT PROGRAM TO APPLY MOVING AVERAGE SMOOTHING FOR DATA PREPARATION AND TIME SERIES FORECASTING

Moving average smoothing is a naive and effective technique in time series forecasting.
It can be used for data preparation, feature engineering, and even directly for making predictions.
In this tutorial, you will discover how to use moving average smoothing for time series forecasting with Python.
After completing this tutorial, you will know:
How moving average smoothing works and some expectations of your data before you can use it.
How to use moving average smoothing for data preparation and feature engineering.
How to use moving average smoothing to make predictions.
Moving Average Smoothing
Smoothing is a technique applied to time series to remove the fine-grained variation between time steps.
The hope of smoothing is to remove noise and better expose the signal of the underlying causal processes. Moving averages are a simple and common type of smoothing used in time series analysis and time series forecasting.
Calculating a moving average involves creating a new series where the values are comprised of the average of raw observations in the original time series.
A moving average requires that you specify a window size called the window width. This defines the number of raw observations used to calculate the moving average value.
The "moving" part in the moving average refers to the fact that the window defined by the window width is slid along the time series to calculate the average values in the new series.
There are two main types of moving average that are used: Centered and Trailing Moving Average.
Moving Average Smoothing for Data Preparation and Time Series Forecasting in Python
by Jason Brownlee on December 28, 2016 in Time Series
Tweet Tweet Share Share
Last Updated on August 15, 2020
Moving average smoothing is a naive and effective technique in time series forecasting.
It can be used for data preparation, feature engineering, and even directly for making predictions.
In this tutorial, you will discover how to use moving average smoothing for time series forecasting with Python.
After completing this tutorial, you will know:
How moving average smoothing works and some expectations of your data before you can use it.
How to use moving average smoothing for data preparation and feature engineering.

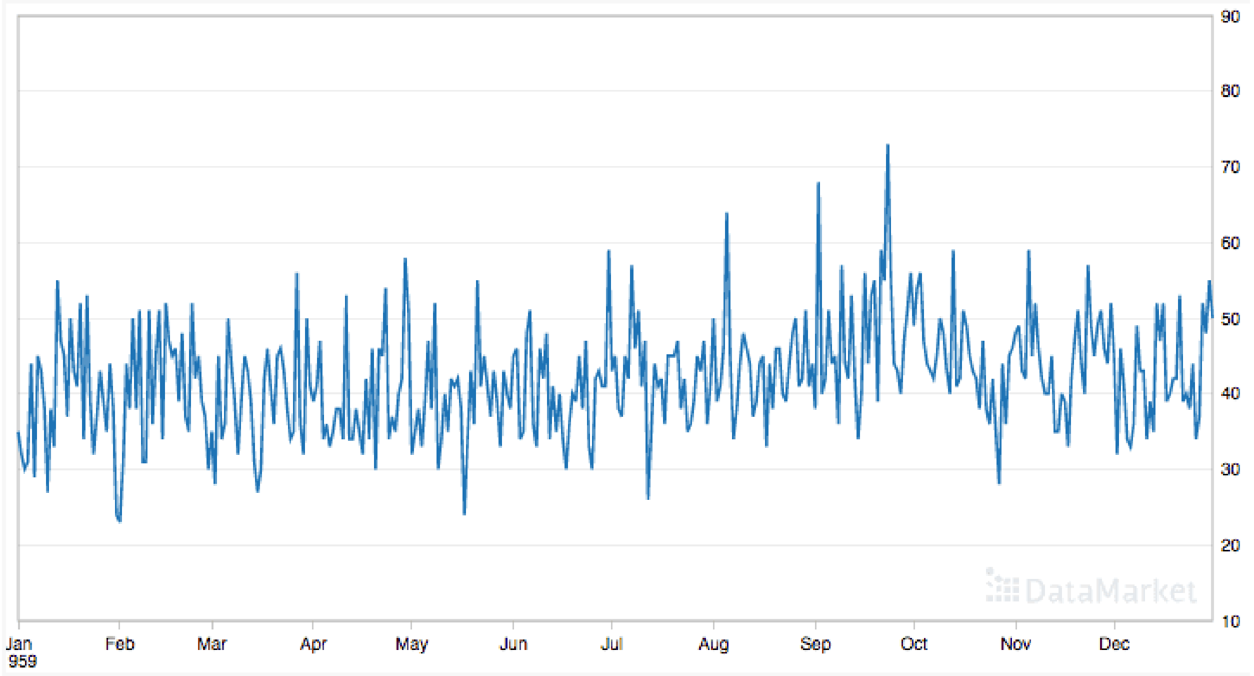
How to use moving average smoothing to make predictions.

Moving Average Smoothing
Smoothing is a technique applied to time series to remove the fine-grained variation between time steps.
The hope of smoothing is to remove noise and better expose the signal of the underlying causal processes. Moving averages are a simple and common type of smoothing used in time series analysis and time series forecasting.
Calculating a moving average involves creating a new series where the values are comprised of the average of raw observations in the original time series.
A moving average requires that you specify a window size called the window width. This defines the number of raw observations used to calculate the moving average value.
The “moving” part in the moving average refers to the fact that the window defined by the window width is slid along the time series to calculate the average values in the new series.
There are two main types of moving average that are used: Centered and Trailing Moving Average.
Centered Moving Average
The value at time (t) is calculated as the average of raw observations at, before, and after time (t).
For example, a center moving average with a window of 3 would be calculated as:
1center_ma(t) = mean(obs(t-1), obs(t), obs(t+1))
This method requires knowledge of future values, and as such is used on time series analysis to better understand the dataset.
A center moving average can be used as a general method to remove trend and seasonal components from a time series, a method that we often cannot use when forecasting.
Trailing Moving Average
The value at time (t) is calculated as the average of the raw observations at and before the time (t).
For example, a trailing moving average with a window of 3 would be calculated as:
1trail_ma(t) = mean(obs(t-2), obs(t-1), obs(t))
Trailing moving average only uses historical observations and is used on time series forecasting.
It is the type of moving average that we will focus on in this tutorial.
Data Expectations
Calculating a moving average of a time series makes some assumptions about your data.
It is assumed that both trend and seasonal components have been removed from your time series.
This means that your time series is stationary, or does not show obvious trends (long-term increasing or decreasing movement) or seasonality (consistent periodic structure).
There are many methods to remove trends and seasonality from a time series dataset when forecasting. Two good methods for each are to use the differencing method and to model the behavior and explicitly subtract it from the series.
Moving average values can be used in a number of ways when using machine learning algorithms on time series problems.
In this tutorial, we will look at how we can calculate trailing moving average values for use as data preparation, feature engineering, and for directly making predictions.
Before we dive into these examples, let’s look at the Daily Female Births dataset that we will use in each example.
Daily Female Births Dataset

This dataset describes the number of daily female births in California in 1959.
The units are a count and there are 365 observations. The source of the dataset is credited to Newton (1988).
Download the dataset.
Below is a sample of the first 5 rows of data, including the header row.

```
"Date","Births"
"1959-01-01",35
"1959-01-02",32
"1959-01-03",30
"1959-01-04",31
"1959-01-05",44
```

Below is a plot of the entire dataset.



Daily Female Births Dataset

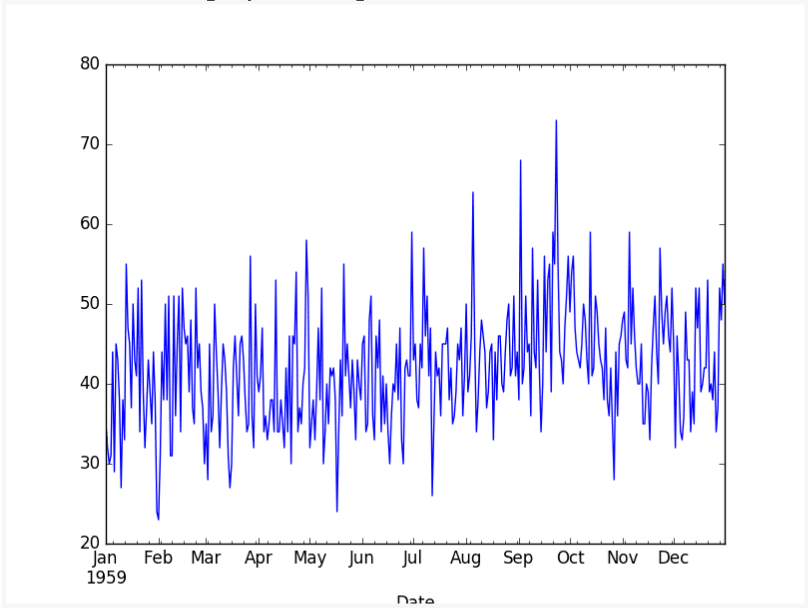
This dataset is a good example for exploring the moving average method as it does not show any clear trend or seasonality.
Load the Daily Female Births Dataset
Download the dataset and place it in the current working directory with the filename “daily-total-female-births.csv“.
The snippet below loads the dataset as a Series, displays the first 5 rows of the dataset, and graphs the whole series as a line plot.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
print(series.head())
series.plot()
pyplot.show()
```

Running the example prints the first 5 rows as follows:

```
Date
```

```
1959-01-01 35
1959-01-02 32
1959-01-03 30
1959-01-04 31
1959-01-05 44
```
Below is the displayed line plot of the loaded data.



Daily Female Births Dataset Plot

Moving Average as Data Preparation

Moving average can be used as a data preparation technique to create a smoothed version of the original dataset.

Smoothing is useful as a data preparation technique as it can reduce the random variation in the observations and better expose the structure of the underlying causal processes.

The rolling() function on the Series Pandas object will automatically group observations into a window. You can specify the window size, and by default a trailing window is created. Once the window is created, we can take the mean value, and this is our transformed dataset.

New observations in the future can be just as easily transformed by keeping the raw values for the last few observations and updating a new average value.

To make this concrete, with a window size of 3, the transformed value at time (t) is calculated as the mean value for the previous 3 observations (t-2, t-1, t), as follows:

```
obs(t) = 1/3 * (t-2 + t-1 + t)
```
For the Daily Female Births dataset, the first moving average would be on January 3rd, as follows:

```
obs(t) = 1/3 * (t-2 + t-1 + t)
obs(t) = 1/3 * (35 + 32 + 30)
obs(t) = 32.333
```
Below is an example of transforming the Daily Female Births dataset into a moving average with a window size of 3 days, chosen arbitrarily.

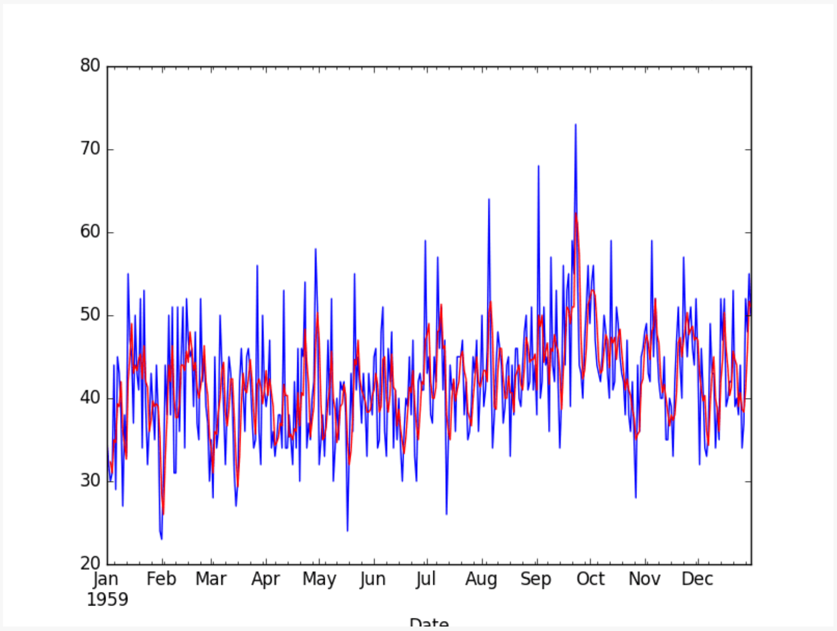
```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
```

```
# Tail-rolling average transform
rolling = series.rolling(window=3)
rolling_mean = rolling.mean()
print(rolling_mean.head(10))
# plot original and transformed dataset
series.plot()
rolling_mean.plot(color='red')
pyplot.show()
```

Running the example prints the first 10 observations from the transformed dataset.
We can see that the first 2 observations will need to be discarded.

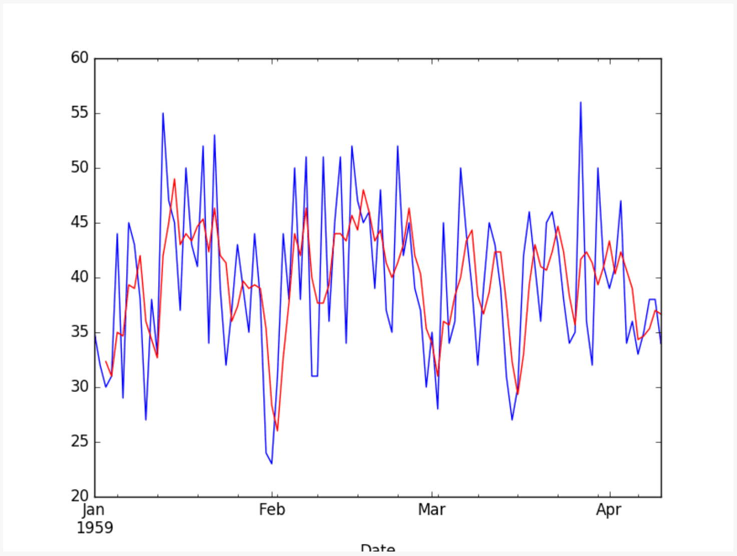
```
Date
1959-01-01          NaN
1959-01-02          NaN
1959-01-03    32.333333
1959-01-04    31.000000
1959-01-05    35.000000
1959-01-06    34.666667
1959-01-07    39.333333
1959-01-08    39.000000
1959-01-09    42.000000
1959-01-10    36.000000
```

The raw observations are plotted (blue) with the moving average transform overlaid (red).



Moving Average Transform

To get a better idea of the effect of the transform, we can zoom in and plot the first 100 observations.



Zoomed Moving Average Transform

Here, you can clearly see the lag in the transformed dataset.

Next, let’s take a look at using the moving average as a feature engineering method.

Moving Average as Feature Engineering

The moving average can be used as a source of new information when modeling a time series forecast as a supervised learning problem.

In this case, the moving average is calculated and added as a new input feature used to predict the next time step.

First, a copy of the series must be shifted forward by one time step. This will represent the input to our prediction problem, or a lag=1 version of the series. This is a standard supervised learning view of the time series problem. For example:

```
X, y
NaN,   obs1
obs1,   obs2
obs2,   obs3
```

Next, a second copy of the series needs to be shifted forward by one, minus the window size. This is to ensure that the moving average summarizes the last few values and does not include the value to be predicted in the average, which would be an invalid framing of the problem as the input would contain knowledge of the future being predicted.

For example, with a window size of 3, we must shift the series forward by 2 time steps. This is because we want to include the previous two observations as well as the current observation in the moving average in order to predict the next value. We can then calculate the moving average from this shifted series.

Below is an example of how the first 5 moving average values are calculated. Remember, the dataset is shifted forward 2 time steps and as we move along the time series, it takes at least 3 time steps before we even have enough data to calculate a window=3 moving average.

```
Observations, Mean
NaN NaN
NaN, NaN NaN
NaN, NaN, 35 NaN
NaN, 35, 32 NaN
```

630, 32, 35 32

Below is an example of including the moving average of the previous 3 values as a new feature, as wellas a lag-1 input feature for the Daily Female Births dataset.

```
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
df = DataFrame(series.values)
width = 3
lag1 = df.shift(1)
lag3 = df.shift(width - 1)
window = lag3.rolling(window=width)
means = window.mean()
dataframe = concat([means, lag1, df], axis=1)
dataframe.columns = ['mean', 't-1', 't+1']
print(dataframe.head(10))
```

Running the example creates the new dataset and prints the first 10 rows.

We can see that the first 3 rows cannot be used and must be discarded. The first row of the lag1 dataset cannot be used because there are no previous observations to predict the first observation, therefore a NaN value is used.

```
        mean   t-1  t+1
0        NaN   NaN   35
1        NaN  35.0   32
2        NaN  32.0   30
3        NaN  30.0   31
4  32.333333  31.0   44
5  31.000000  44.0   29
6  35.000000  29.0   45
7  34.666667  45.0   43
8  39.333333  43.0   38
9  39.000000  38.0   27
```

The next section will look at how to use the moving average as a naive model to make predictions.

Moving Average as Prediction

The moving average value can also be used directly to make predictions.

It is a naive model and assumes that the trend and seasonality components of the time series have already been removed or adjusted for.

The moving average model for predictions can easily be used in a walk-forward manner. As new observations are made available (e.g. daily), the model can be updated and a prediction made for the next day.

We can implement this manually in Python. Below is an example of the moving average model used in a walk-forward manner.

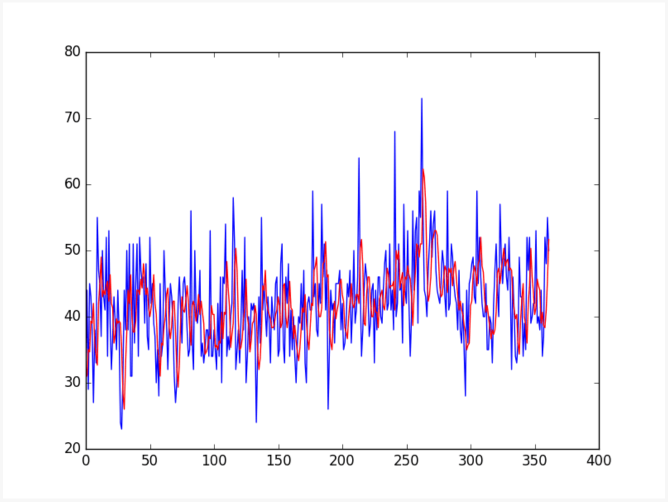
```
from pandas import read_csv
from numpy import mean
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
```

```
# prepare situation
X = series.values
window = 3
history = [X[i] for i in range(window)]
test = [X[i] for i in range(window, len(X))]
predictions = list()
# walk forward over time steps in test
for t in range(len(test)):
 length = len(history)
 yhat = mean([history[i] for i in range(length-window,length)])
 obs = test[t]
 predictions.append(yhat)
 history.append(obs)
 print('predicted=%f, expected=%f' % (yhat, obs))
error = mean_squared_error(test, predictions)
print('Test MSE: %.3f' % error)
# plot
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
# zoom plot
pyplot.plot(test[0:100])
pyplot.plot(predictions[0:100], color='red')
pyplot.show()
```

Running the example prints the predicted and expected value each time step moving forward, starting from time step 4 (1959-01-04).
Finally, the mean squared error (MSE) is reported for all predictions made.

```
predicted=32.333333, expected=31.000000
predicted=31.000000, expected=44.000000
predicted=35.000000, expected=29.000000
predicted=34.666667, expected=45.000000
predicted=39.333333, expected=43.000000
...
predicted=38.333333, expected=52.000000
predicted=41.000000, expected=48.000000
predicted=45.666667, expected=55.000000
predicted=51.666667, expected=50.000000
Test MSE: 61.379
```

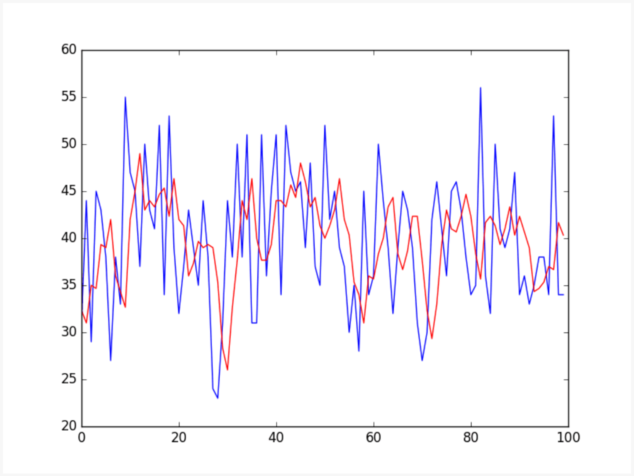
The example ends by plotting the expected test values (blue) compared to the predictions (red).



Moving Average Predictions

Again, zooming in on the first 100 predictions gives an idea of the skill of the 3-day moving average predictions.

Note the window width of 3 was chosen arbitrary and was not optimized.



Zoomed Moving Average Predictions

7. IMPLEMENT PROGRAM FOR DECOMPOSING TIME SERIES DATA INTO TREND AND SEASONALITY

Time series decomposition involves thinking of a series as a combination of level, trend, seasonality, and noise components.

Decomposition provides a useful abstract model for thinking about time series generally and for better understanding problems during time series analysis and forecasting.

In this tutorial, you will discover time series decomposition and how to automatically split a time series into its components with Python.

After completing this tutorial, you will know:

- The time series decomposition method of analysis and how it can help with forecasting.
- How to automatically decompose time series data in Python.
- How to decompose additive and multiplicative time series problems and plot the results.

Time Series Components

A useful abstraction for selecting forecasting methods is to break a time series down into systematic and unsystematic components.

- **Systematic**: Components of the time series that have consistency or recurrence and can be described and modeled.
- **Non-Systematic**: Components of the time series that cannot be directly modeled.

A given time series is thought to consist of three systematic components including level, trend, seasonality, and one non-systematic component called noise.

These components are defined as follows:

- **Level**: The average value in the series.
- **Trend**: The increasing or decreasing value in the series.
- **Seasonality**: The repeating short-term cycle in the series.
- **Noise**: The random variation in the series.
- y has an increasing or decreasing frequency and/or amplitude over time.

- Decomposition as a Tool
- This is a useful abstraction.

- Decomposition is primarily used for time series analysis, and as an analysis tool it can be used to inform forecasting models on your problem.

- It provides a structured way of thinking about a time series forecasting problem, both generally in terms of modeling complexity and specifically in terms of how to best capture each of these components in a given model.

- Each of these components are something you may need to think about and address during data preparation, model selection, and model tuning. You may address it explicitly in terms of modeling the trend and subtracting it from your data, or implicitly by providing enough history for an algorithm to model a trend if it may exist.

- You may or may not be able to cleanly or perfectly break down your specific time series as an additive or multiplicative model.

- Real-world problems are messy and noisy. There may be additive and multiplicative components. There may be an increasing trend followed by a decreasing trend. There may be non-repeating cycles mixed in with the repeating seasonality components.

- Nevertheless, these abstract models provide a simple framework that you can use to analyze your data and explore ways to think about and forecast your problem.

- Automatic Time Series Decomposition

- There are methods to automatically decompose a time series.
- The statsmodels library provides an implementation of the naive, or classical, decomposition method in a function called seasonal_decompose(). It requires that you specify whether the model is additive or multiplicative.
- Both will produce a result and you must be careful to be critical when interpreting the result. A review of a plot of the time series and some summary statistics can often be a good start to get an idea of whether your time series problem looks additive or multiplicative.

- The *seasonal_decompose()* function returns a result object. The result object contains arrays to access four pieces of data from the decomposition.
- For example, the snippet below shows how to decompose a series into trend, seasonal, and residual components assuming an additive model.

- The result object provides access to the trend and seasonal series as arrays. It also provides access to the residuals, which are the time series after the trend, and seasonal components are removed. Finally, the original or observed data is also stored.

```
from statsmodels.tsa.seasonal import seasonal_decompose
series = ...
result = seasonal_decompose(series, model='additive')
print(result.trend)
print(result.seasonal)
print(result.resid)
print(result.observed)
```

- These four time series can be plotted directly from the result object by calling the *plot()* function. For example:

```
from statsmodels.tsa.seasonal import seasonal_decompose
from matplotlib import pyplot
series = ...
result = seasonal_decompose(series, model='additive')
result.plot()
pyplot.show()
```

- Let's look at some examples.

- Additive Decomposition

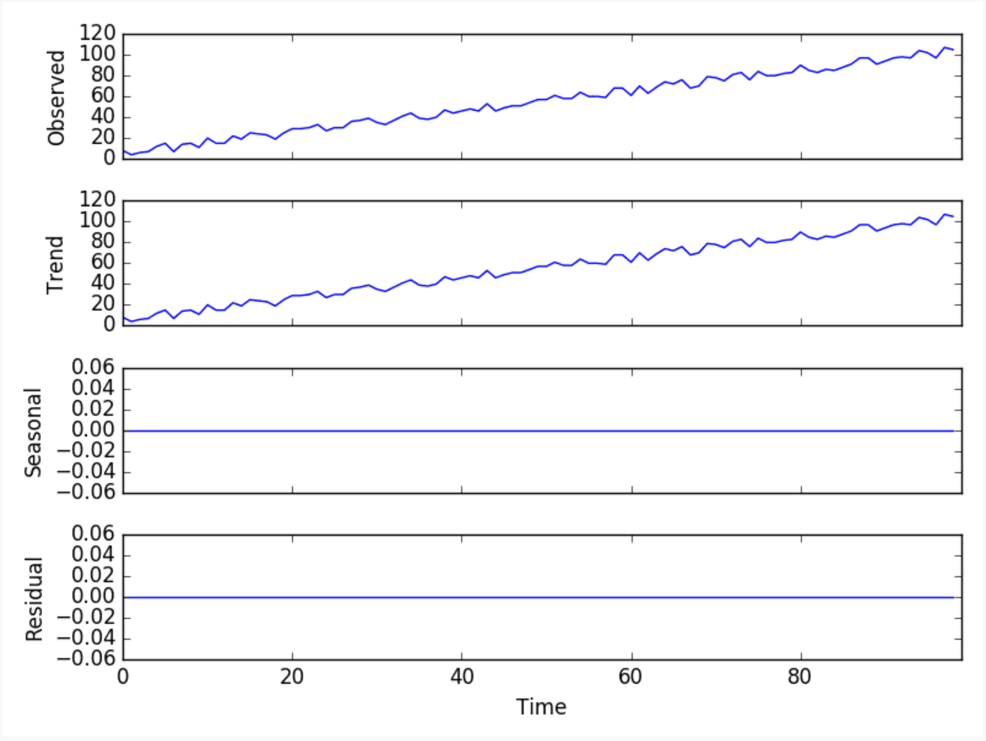
- We can create a time series comprised of a linearly increasing trend from 1 to 99 and some random noise and decompose it as an additive model.

- Because the time series was contrived and was provided as an array of numbers, we must specify the frequency of the observations (the *period=1* argument). If a Pandas Series object is provided, this argument is not required.

```
from random import randrange
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.seasonal import seasonal_decompose
series = [i+randrange(10) for i in range(1,100)]
result = seasonal_decompose(series, model='additive', period=1)
result.plot()
pyplot.show()
```

- Running the example creates the series, performs the decomposition, and plots the 4 resulting series.

- We can see that the entire series was taken as the trend component and that there was no seasonality.

- 

- Additive Model Decomposition Plot

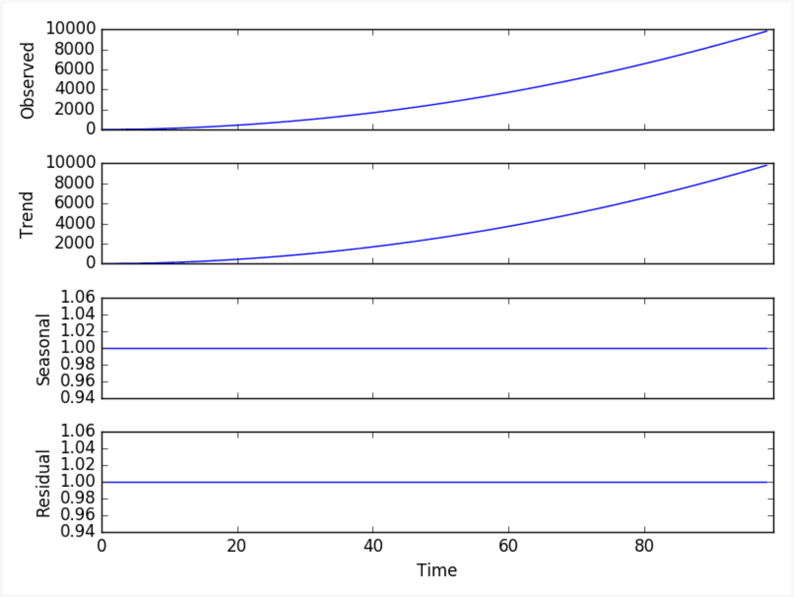
- We can also see that the residual plot shows zero. This is a good example where the naive, or classical, decomposition was not able to separate the noise that we added from the linear trend.

- The naive decomposition method is a simple one, and there are more advanced decompositions available, like Seasonal and Trend decomposition using Loess or STL decomposition.
- Caution and healthy skepticism is needed when using automated decomposition methods.

Multiplicative Decomposition

We can contrive a quadratic time series as a square of the time step from 1 to 99, and then decompose it assuming a multiplicative model.

```
from pandas import Series
from matplotlib import pyplot
from statsmodels.tsa.seasonal import seasonal_decompose
series = [i**2.0 for i in range(1,100)]
result = seasonal_decompose(series, model='multiplicative', period=1)
result.plot()
pyplot.show()
```
Running the example, we can see that, as in the additive case, the trend is easily extracted and wholly characterizes the time series.



Multiplicative Model Decomposition Plot

Exponential changes can be made linear by data transforms. In this case, a quadratic trend can be made linear by taking the square root. An exponential growth in seasonality may be made linear by taking the natural logarithm.

Again, it is important to treat decomposition as a potentially useful analysis tool, but to consider exploring the many different ways it could be applied for your problem, such as on data after it has been transformed or on residual model errors.

Airline Passengers Dataset

The Airline Passengers dataset describes the total number of airline passengers over a period of time.

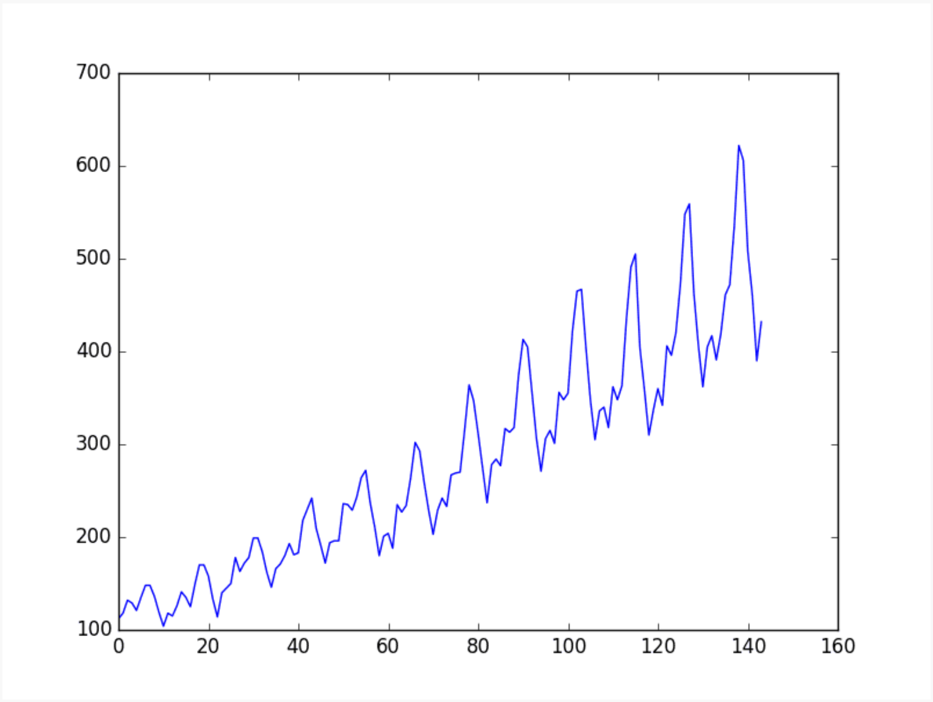
The units are a count of the number of airline passengers in thousands. There are 144 monthly observations from 1949 to 1960.

- Download the dataset.

Download the dataset to your current working directory with the filename "*airline-passengers.csv*".
First, let's graph the raw observations.

```
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0)
series.plot()
pyplot.show()
```

Reviewing the line plot, it suggests that there may be a linear trend, but it is hard to be sure from eye-balling. There is also seasonality, but the amplitude (height) of the cycles appears to be increasing, suggesting that it is multiplicative.



Plot of the Airline Passengers Dataset

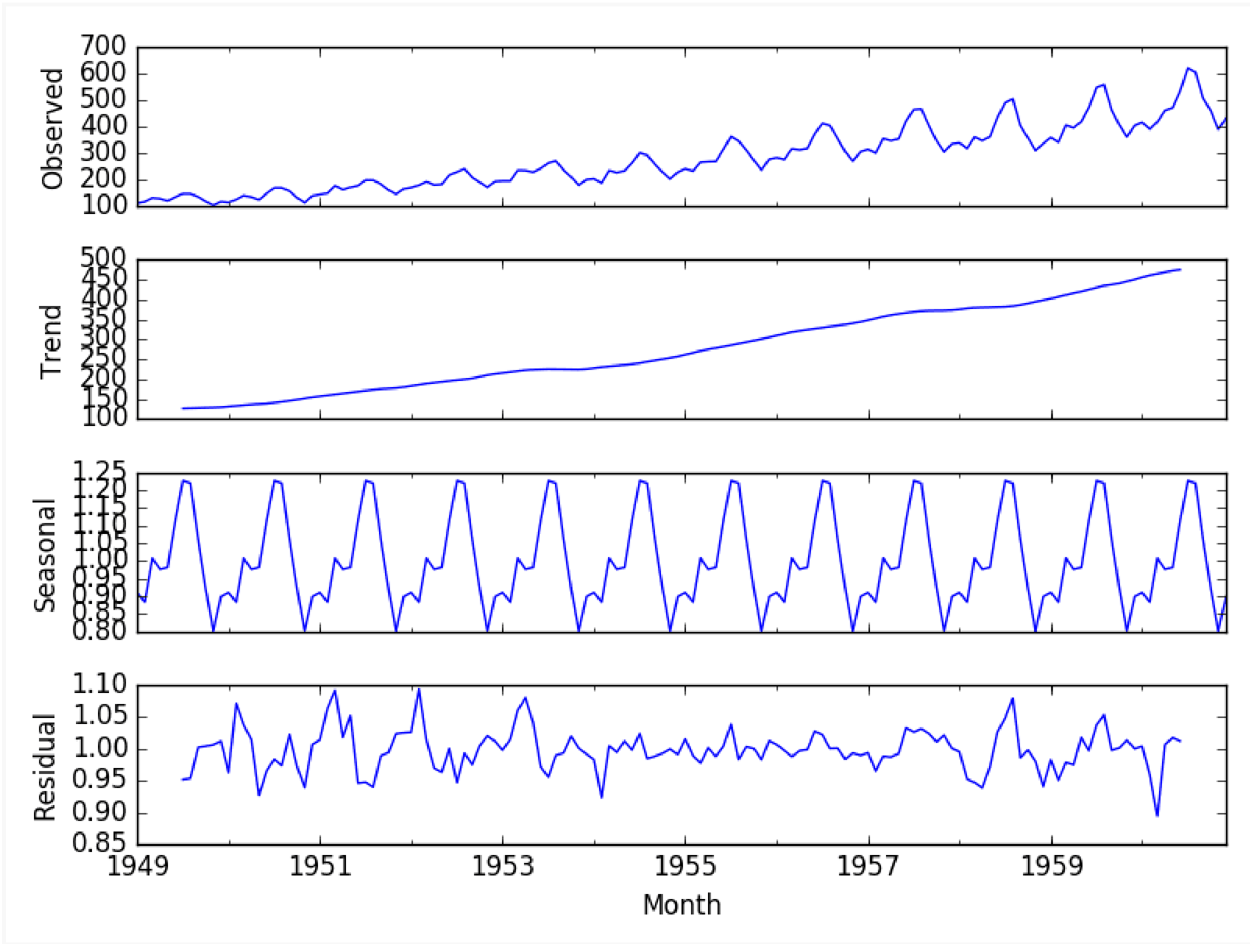
We will assume a multiplicative model.

The example below decomposes the airline passengers dataset as a multiplicative model.

```
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.seasonal import seasonal_decompose
series = read_csv('airline-passengers.csv', header=0, index_col=0)
result = seasonal_decompose(series, model='multiplicative')
result.plot()
pyplot.show()
```

Running the example plots the observed, trend, seasonal, and residual time series.

We can see that the trend and seasonality information extracted from the series does seem reasonable. The residuals are also interesting, showing periods of high variability in the early and later years of the series.



Multiplicative Decomposition of Airline Passenger Dataset

8. CREATE AN ARIMA MODEL FOR TIME SERIES FORECASTING

A popular and widely used statistical method for time series forecasting is the ARIMA model.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a class of model that captures a suite of different standard temporal structures in time series data.

In this tutorial, you will discover how to **develop an ARIMA model for time series** forecasting in Python.
After completing this tutorial, you will know:

- About the ARIMA model the parameters used and assumptions made by the model.
- How to fit an ARIMA model to data and use it to make forecasts.
- How to configure the ARIMA model on your time series problem.

Autoregressive Integrated Moving Average Model

An ARIMA model is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration.

This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of ARIMA(p,d,q) where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

- **p**: The number of lag observations included in the model, also called the lag order.
- **d**: The number of times that the raw observations are differenced, also called the degree of differencing.
- **q**: The size of the moving average window, also called the order of moving average.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model.

Next, let's take a look at how we can use the ARIMA model in Python. We will start with loading a simple univariate time series.

Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3 year period.

The units are a sales count and there are 36 observations. The original dataset is credited to Makridakis, Wheelwright, and Hyndman (1998).

- Download the dataset.

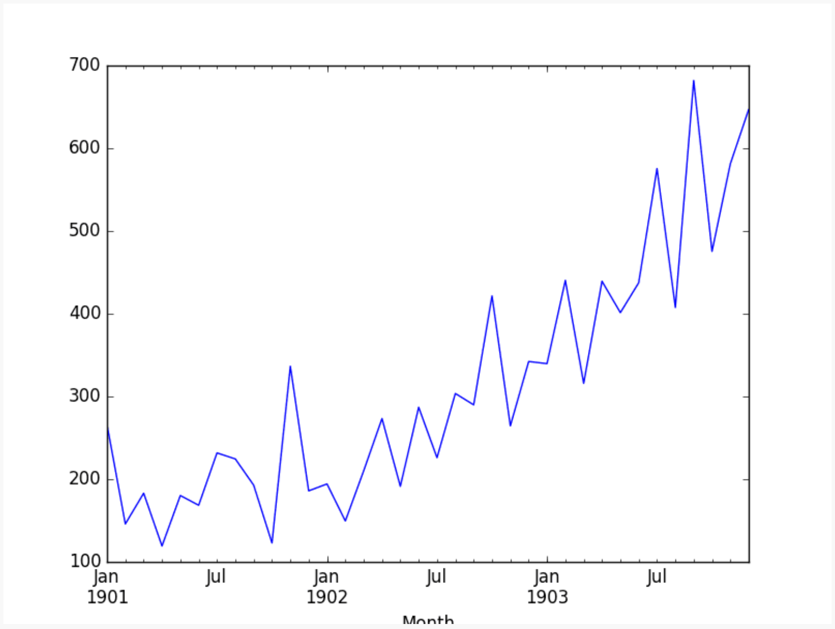
Download the dataset and place it in your current working directory with the filename "*shampoo-sales.csv*".
Below is an example of loading the Shampoo Sales dataset with Pandas with a custom function to parse the date-time field. The dataset is baselined in an arbitrary year, in this case 1900.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
squeeze=True, date_parser=parser)
print(series.head())
series.plot()
pyplot.show()
```

Running the example prints the first 5 rows of the dataset.

```
Month
1901-01-01 266.0
1901-02-01 145.9
1901-03-01 183.1
1901-04-01 119.3
1901-05-01 180.3
Name: Sales, dtype: float64
```

The data is also plotted as a time series with the month along the x-axis and sales figures on the y-axis.



Shampoo Sales Dataset Plot

We can see that the Shampoo Sales dataset has a clear trend.

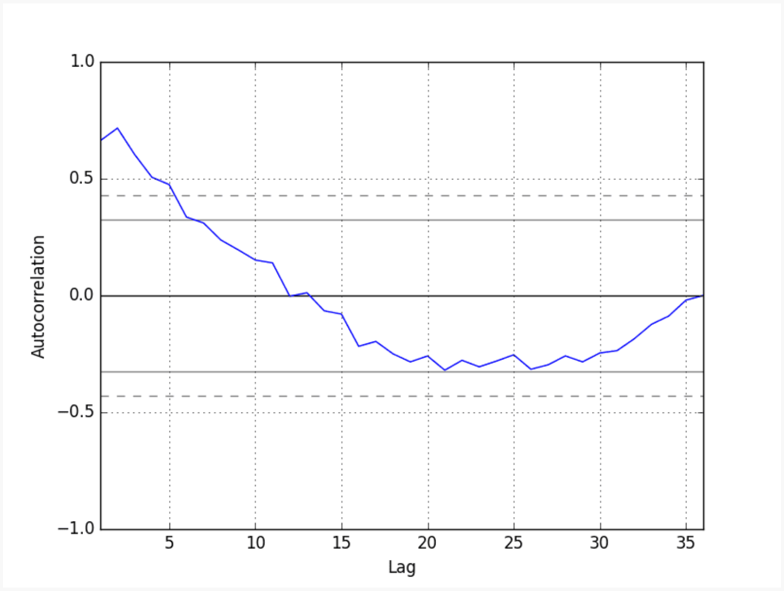
This suggests that the time series is not stationary and will require differencing to make it stationary, at least a difference order of 1.

Let’s also take a quick look at an autocorrelation plot of the time series. This is also built-in to Pandas. The example below plots the autocorrelation for a large number of lags in the time series.

```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot

def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
squeeze=True, date_parser=parser)
autocorrelation_plot(series)
pyplot.show()
```

Running the example, we can see that there is a positive correlation with the first 10-to-12 lags that is perhaps significant for the first 5 lags.

A good starting point for the AR parameter of the model may be 5.



Autocorrelation Plot of Shampoo Sales Data

ARIMA with Python

The statsmodels library provides the capability to fit an ARIMA model.

An ARIMA model can be created using the statsmodels library as follows:

1. Define the model by calling ARIMA() and passing in the *p*, *d*, and *q* parameters.
2. The model is prepared on the training data by calling the fit() function.
3. Predictions can be made by calling the predict() function and specifying the index of the time or times to be predicted.

Let's start off with something simple. We will fit an ARIMA model to the entire Shampoo Sales dataset and review the residual errors.

First, we fit an ARIMA(5,1,0) model. This sets the lag value to 5 for autoregression, uses a difference order of 1 to make the time series stationary, and uses a moving average model of 0.

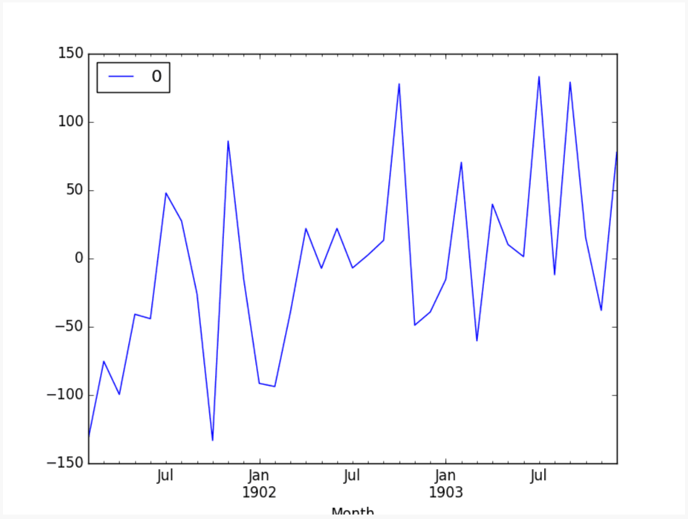
```
# fit an ARIMA model and plot residual errors
from pandas import datetime
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima.model import ARIMA
from matplotlib import pyplot
# load dataset
def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
squeeze=True, date_parser=parser)
series.index = series.index.to_period('M')
# fit model
```

```
model = ARIMA(series, order=(5,1,0))
model_fit = model.fit()
# summary of fit model
print(model_fit.summary())
# line plot of residuals
residuals = DataFrame(model_fit.resid)
residuals.plot()
pyplot.show()
# density plot of residuals
residuals.plot(kind='kde')
pyplot.show()
# summary stats of residuals
print(residuals.describe())
```

Running the example prints a summary of the fit model. This summarizes the coefficient values used as well as the skill of the fit on the on the in-sample observations.

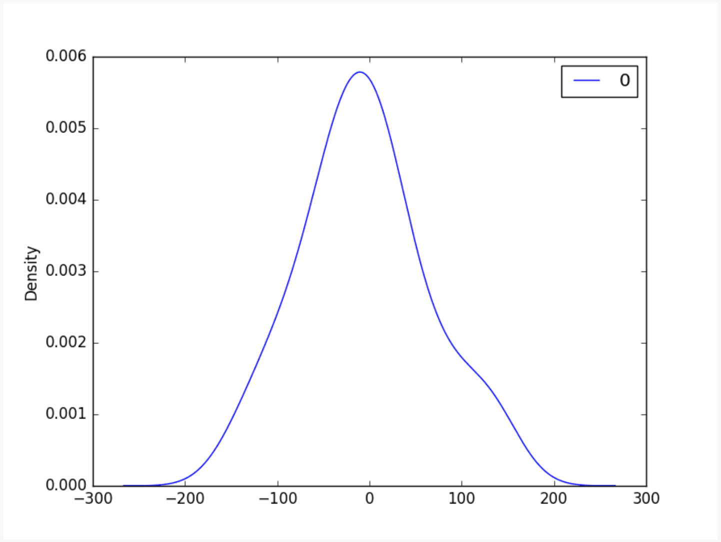
```
                               SARIMAX Results
==============================================================================
==========
Dep. Variable:                  Sales   No. Observations:                   36
Model:               ARIMA(5, 1, 0)   Log Likelihood                -198.485
Date:              Thu, 10 Dec 2020   AIC                            408.969
Time:                      09:15:01   BIC                            418.301
Sample:                  01-31-1901   HQIC                           412.191
                       - 12-31-1903
Covariance Type:                opg
==============================================================================
==========
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.9014      0.247     -3.647      0.000      -1.386      -0.417
ar.L2         -0.2284      0.268     -0.851      0.395      -0.754       0.298
ar.L3          0.0747      0.291      0.256      0.798      -0.497       0.646
ar.L4          0.2519      0.340      0.742      0.458      -0.414       0.918
ar.L5          0.3344      0.210      1.593      0.111      -0.077       0.746
sigma2      4728.9608   1316.021      3.593      0.000    2149.607    7308.314
==============================================================================
===============
Ljung-Box (L1) (Q):                   0.61   Jarque-Bera (JB):                 0.96
Prob(Q):                              0.44   Prob(JB):                         0.62
Heteroskedasticity (H):               1.07   Skew:                             0.28
Prob(H) (two-sided):                  0.90   Kurtosis:                         2.41
==============================================================================
===============
```

First, we get a line plot of the residual errors, suggesting that there may still be some trend information not captured by the model.



ARMA Fit Residual Error Line Plot

Next, we get a density plot of the residual error values, suggesting the errors are Gaussian, but may not be centered on zero.



ARMA Fit Residual Error Density Plot

The distribution of the residual errors is displayed. The results show that indeed there is a bias in the prediction (a non-zero mean in the residuals).

```
count   36.000000
mean    21.936144
std     80.774430
min   -122.292030
25%    -35.040859
50%     13.147219
75%     68.848286
max    266.000000
```

Note, that although above we used the entire dataset for time series analysis, ideally we would perform this analysis on just the training dataset when developing a predictive model.

Next, let’s look at how we can use the ARIMA model to make forecasts.

Rolling Forecast ARIMA Model

The ARIMA model can be used to forecast future time steps.

We can use the predict() function on the ARIMAResults object to make predictions. It accepts the index of the time steps to make predictions as arguments. These indexes are relative to the start of the training dataset used to make predictions.
If we used 100 observations in the training dataset to fit the model, then the index of the next time step for making a prediction would be specified to the prediction function as *start=101, end=101*. This would return an array with one element containing the prediction.
We also would prefer the forecasted values to be in the original scale, in case we performed any differencing (*d>0* when configuring the model). This can be specified by setting the *typ* argument to the value *‘levels’*: *typ=’levels’*.
Alternately, we can avoid all of these specifications by using the forecast() function, which performs a one-step forecast using the model.

We can split the training dataset into train and test sets, use the train set to fit the model, and generate a prediction for each element on the test set.

A rolling forecast is required given the dependence on observations in prior time steps for differencing and the AR model. A crude way to perform this rolling forecast is to re-create the ARIMA model after each new observation is received.

We manually keep track of all observations in a list called history that is seeded with the training data and to which new observations are appended each iteration.

Putting this all together, below is an example of a rolling forecast with the ARIMA model in Python.

```
# evaluate an ARIMA model using a walk-forward validation
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
squeeze=True, date_parser=parser)
series.index = series.index.to_period('M')
```

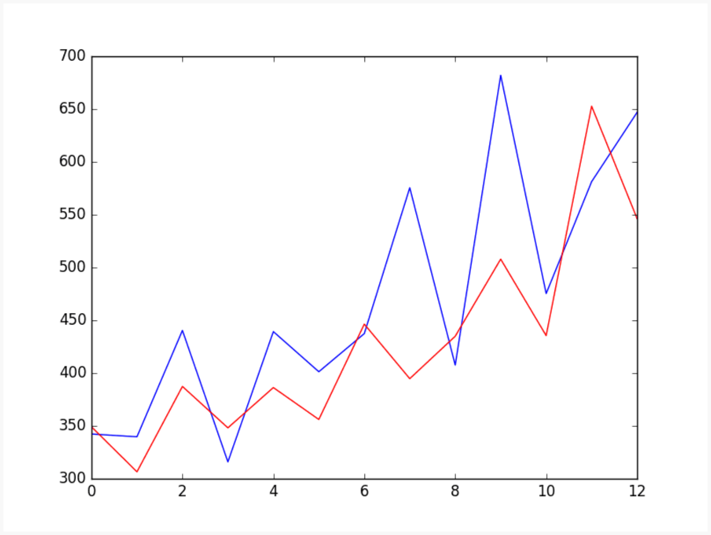
```
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
 model = ARIMA(history, order=(5,1,0))
 model_fit = model.fit()
 output = model_fit.forecast()
 yhat = output[0]
 predictions.append(yhat)
 obs = test[t]
 history.append(obs)
 print('predicted=%f, expected=%f' % (yhat, obs))
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Running the example prints the prediction and expected value each iteration.

We can also calculate a final root mean squared error score (RMSE) for the predictions, providing a point of comparison for other ARIMA configurations.

```
predicted=343.272180, expected=342.300000
predicted=293.329674, expected=339.700000
predicted=368.668956, expected=440.400000
predicted=335.044741, expected=315.900000
predicted=363.220221, expected=439.300000
predicted=357.645324, expected=401.300000
predicted=443.047835, expected=437.400000
predicted=378.365674, expected=575.500000
predicted=459.415021, expected=407.600000
predicted=526.890876, expected=682.000000
predicted=457.231275, expected=475.300000
predicted=672.914944, expected=581.300000
predicted=531.541449, expected=646.900000
Test RMSE: 89.021
```

A line plot is created showing the expected values (blue) compared to the rolling forecast predictions (red). We can see the values show some trend and are in the correct scale.



ARIMA Rolling Forecast Line Plot

The model could use further tuning of the p, d, and maybe even the q parameters.

Configuring an ARIMA Model

The classical approach for fitting an ARIMA model is to follow the Box-Jenkins Methodology. This is a process that uses time series analysis and diagnostics to discover good parameters for the ARIMA model.

In summary, the steps of this process are as follows:

1. **Model Identification**. Use plots and summary statistics to identify trends, seasonality, and autoregression elements to get an idea of the amount of differencing and the size of the lag that will be required.
2. **Parameter Estimation**. Use a fitting procedure to find the coefficients of the regression model.
3. **Model Checking**. Use plots and statistical tests of the residual errors to determine the amount and type of temporal structure not captured by the model.

The process is repeated until either a desirable level of fit is achieved on the in-sample or out-of-sample observations (e.g. training or test datasets).

The process was described in the classic 1970 textbook on the topic titled Time Series Analysis: Forecasting and Control by George Box and Gwilym Jenkins. An updated 5th edition is now available if you are interested in going deeper into this type of model and methodology.
Given that the model can be fit efficiently on modest-sized time series datasets, grid searching parameters of the model can be a valuable approach.

9.DEVELOP NEURAL NETWORK-BASED TIME SERIES FORECASTING MODEL

Convolutional Neural Network models, or CNNs for short, can be applied to time series forecasting.
There are many types of CNN models that can be used for each specific type of time series forecasting problem.

In this tutorial, you will discover how to develop a suite of CNN models for a range of standard time series forecasting problems.

The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem.

After completing this tutorial, you will know:

- How to develop CNN models for univariate time series forecasting.
- How to develop CNN models for multivariate time series forecasting.
- How to develop CNN models for multi-step time series forecasting.

Univariate CNN Models

Although traditionally developed for two-dimensional image data, CNNs can be used to model univariate time series forecasting problems.

Univariate time series are datasets comprised of a single series of observations with a temporal ordering and a model is required to learn from the series of past observations to predict the next value in the sequence.

This section is divided into two parts; they are:

1. Data Preparation
2. CNN Model

Data Preparation

Before a univariate series can be modeled, it must be prepared.

The CNN model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the model can learn.

Consider a given univariate sequence:

```
1[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X, y
10, 20, 30 40
20, 30, 40 50
30, 40, 50 60
...
```

The *split_sequence()* function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the sequence
 if end_ix > len(sequence)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can demonstrate this function on our small contrived dataset above.

The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the sequence
 if end_ix > len(sequence)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```
Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```
Now that we know how to prepare a univariate series for modeling, let's look at developing a CNN model that can learn the mapping of inputs to outputs.

CNN Model

A one-dimensional CNN is a CNN model that has a convolutional hidden layer that operates over a 1D sequence. This is followed by perhaps a second convolutional layer in some cases, such as very long input sequences, and then a pooling layer whose job it is to distill the output of the convolutional layer to the most salient elements.
The convolutional and pooling layers are followed by a dense fully connected layer that interprets the features extracted by the convolutional part of the model. A flatten layer is used between the convolutional layers and the dense layer to reduce the feature maps to a single one-dimensional vector.

We can define a 1D CNN Model for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```
Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features.

We are working with a univariate series, so the number of features is one, for one variable.

The number of time steps as input is the number we chose when preparing our dataset as an argument to the *split_sequence()* function.
The input shape for each sample is specified in the *input_shape* argument on the definition of the first hidden layer.
We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape:

```
[samples, timesteps, features]
```

Our *split_sequence()* function in the previous section outputs the X with the shape [*samples, timesteps*], so we can easily reshape it to have an additional dimension for the one feature.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

The CNN does not actually view the data as having time steps, instead, it is treated as a sequence over which convolutional read operations can be performed, like a one-dimensional image.

In this example, we define a convolutional layer with 64 filter maps and a kernel size of 2. This is followed by a max pooling layer and a dense layer to interpret the input feature. An output layer is specified that predicts a single numerical value.

The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or '*mse*', loss function.
Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=1000, verbose=0)
```

After the model is fit, we can use it to make a prediction.

We can predict the next value in the sequence by providing the input:

```
[70, 80, 90]
```

And expecting the model to predict something like:

```
[100]
```

The model expects the input shape to be three-dimensional with [*samples, timesteps, features*], therefore, we must reshape the single input sample before making the prediction.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

We can tie all of this together and demonstrate how to develop a 1D CNN model for univariate time series forecasting and make a single prediction.

```
# univariate cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the sequence
 if end_ix > len(sequence)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```
Running the example prepares the data, fits the model, and makes a prediction.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
We can see that the model predicts the next value in the sequence.

```
[[101.67965]]
```

Multivariate CNN Models

Multivariate time series data means data where there is more than one observation for each time step.

There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.
2. Multiple Parallel Series.

Let’s take a look at each in turn.

Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series.

The input time series are parallel because each series has observations at the same time steps.

We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

We can reshape these three arrays of data as a single dataset where each row is a time step and each column is a separate time series.

This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
```

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

As with the univariate time series, we must structure these data into samples with input and output samples.

A 1D CNN model needs sufficient context to learn a mapping from an input sequence to an output value. CNNs can support parallel input time series as separate channels, like red, green, and blue components of an image. Therefore, we need to split the data into samples maintaining the order of observations across the two input sequences.

If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Output:

```
65
```

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65.

We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used.

We can define a function named *split_sequences()* that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can test this function on our dataset using three time steps for each input time series as input.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

Running the example first prints the shape of the *X* and *y* components.
We can see that the *X* component has a three-dimensional structure.
The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series.

This is the exact three-dimensional structure expected by a 1D CNN as input. The data is ready to use without further reshaping.

We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

We are now ready to fit a 1D CNN model on this data, specifying the expected number of time steps and features to expect for each input sample, in this case three and two respectively.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

When making a prediction, the model expects three time steps for two input time series.

We can predict the next value in the output series providing the input values of:

```
80, 85
90, 95
100, 105
```

The shape of the one sample with three time steps and two variables must be [1, 3, 2].

We would expect the next value in the sequence to be 100 + 105 or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

The complete example is listed below.

```
# multivariate cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
```

```
end_ix = i + n_steps
# check if we are beyond the dataset
if end_ix > len(sequences):
break
# gather input and output parts of the pattern
seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
X.append(seq_x)
y.append(seq_y)
return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prepares the data, fits the model, and makes a prediction.

```
[[206.0161]]
```
There is another, more elaborate way to model the problem.

Each input series can be handled by a separate CNN and the output of each of these submodels can be combined before a prediction is made for the output sequence.

We can refer to this as a multi-headed CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled. For example, it allows you to configure each sub-model differently for each input series, such as the number of filter maps and the kernel size.

This type of model can be defined in Keras using the Keras functional API.
First, we can define the first input model as a 1D CNN with an input layer that expects vectors with *n_steps* and 1 feature.

```
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
```

We can define the second input submodel in the same way.

```
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
cnn2 = Flatten()(cnn2)
```

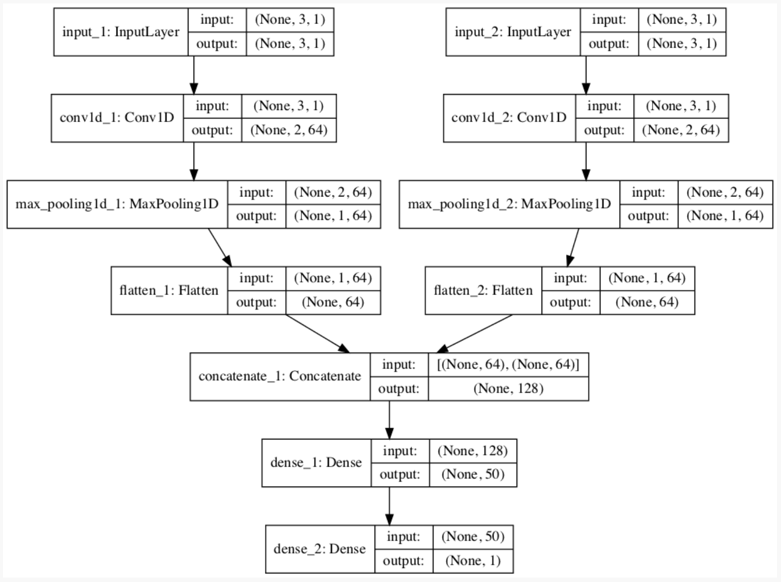
Now that both input submodels have been defined, we can merge the output from each model into one long vector which can be interpreted before making a prediction for the output sequence.

```
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
```

We can then tie the inputs and outputs together.

```
model = Model(inputs=[visible1, visible2], outputs=output)
```

The image below provides a schematic for how this model looks, including the shape of the inputs and outputs of each layer.



Plot of Multi-Headed 1D CNN for Multivariate Time Series Forecasting

This model requires input to be provided as a list of two elements where each element in the list contains data for one of the submodels.

In order to achieve this, we can split the 3D input data into two separate arrays of input data; that is from one array with the shape [7, 3, 2] to two 3D arrays with [7, 3, 1]

```
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
```

These data can then be provided in order to fit the model.

```
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
```

Similarly, we must prepare the data for a single sample as two separate two-dimensional arrays when making a single one-step prediction.

```
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
```

We can tie all of this together; the complete example is listed below.

```
# multivariate multi-headed 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
# second input model
visible2 = Input(shape=(n_steps, n_features))
```

```
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
cnn2 = Flatten()(cnn2)
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
model = Model(inputs=[visible1, visible2], outputs=output)
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
yhat = model.predict([x1, x2], verbose=0)
print(yhat)
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prepares the data, fits the model, and makes a prediction.

```
[[205.871]]
```

Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each.

For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

We may want to predict the value for each of the three time series for the next time step.

This might be referred to as multivariate forecasting.

Again, the data must be split into input/output samples in order to train a model.

The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Output:

```
40, 45, 85
```

The *split_sequences()* function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
```

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

Running the example first prints the shape of the prepared X and y components.

The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3).

The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3).

The data is ready to use in a 1D CNN model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample.

Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50  55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60  65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70  75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80  85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90  95 185]
```

We are now ready to fit a 1D CNN model on this data.

In this model, the number of time steps and parallel series (features) are specified for the input layer via the *input_shape* argument.
The number of parallel series is also used in the specification of the number of values to predict by the model in the output layer; again, this is three.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
```

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

The shape of the input for making a single prediction must be 1 sample, 3 time steps, and 3 features, or [1, 3, 3].

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

We would expect the vector output to be:

```
[100, 105, 205]
```

We can tie all of this together and demonstrate a 1D CNN for multivariate output time series forecasting below.

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=3000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prepares the data, fits the model and makes a prediction.

```
[[100.11272 105.32213 205.53436]]
```

As with multiple input series, there is another more elaborate way to model the problem.

Each output series can be handled by a separate output CNN model.

We can refer to this as a multi-output CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled.

This type of model can be defined in Keras using the Keras functional API.

First, we can define the first input model as a 1D CNN model.

```
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
```

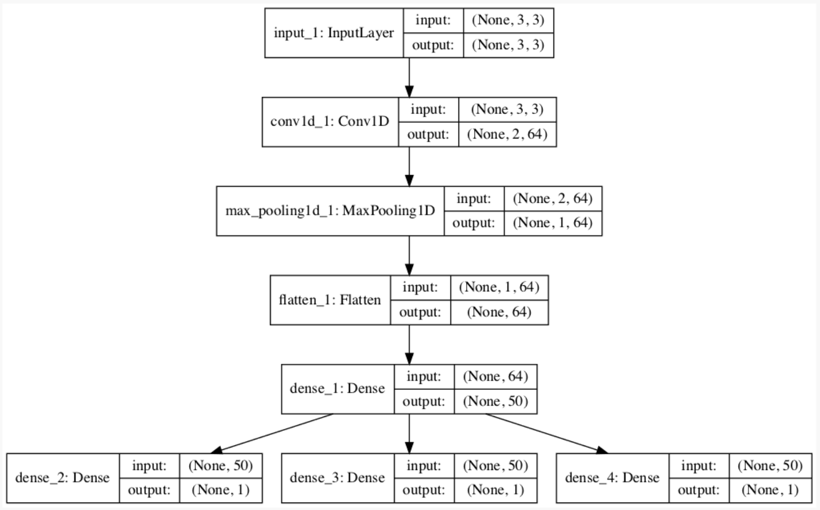
We can then define one output layer for each of the three series that we wish to forecast, where each output submodel will forecast a single time step.

```
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
```

We can then tie the input and output layers together into a single model.

```
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
```

To make the model architecture clear, the schematic below clearly shows the three separate output layers of the model and the input and output shapes of each layer.



Plot of Multi-Output 1D CNN for Multivariate Time Series Forecasting

When training the model, it will require three separate output arrays per sample. We can achieve this by converting the output training data that has the shape [7, 3] to three arrays with the shape [7, 1].

```
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

These arrays can be provided to the model during training.

```
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Tying all of this together, the complete example is listed below.

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps
 # check if we are beyond the dataset
 if end_ix > len(sequences)-1:
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
# demonstrate prediction
```

```
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
Running the example prepares the data, fits the model, and makes a prediction.

```
[array([[100.96118]], dtype=float32),
 array([[105.502686]], dtype=float32),
 array([[205.98045]], dtype=float32)]
```

Multi-Step CNN Models

In practice, there is little difference to the 1D CNN model in predicting a vector output that represents different output variables (as in the previous example), or a vector output that represents multiple time steps of one variable.

Nevertheless, there are subtle and important differences in the way the training data is prepared. In this section, we will demonstrate the case of developing a multi-step forecast model using a vector model.

Before we look at the specifics of the model, let's first look at the preparation of data for multi-step forecasting.

Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components.

Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps.

For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

We could use the last three time steps as input and forecast the next two time steps.

The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Output:

```
[40, 50]
```

The *split_sequence()* function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the sequence
 if out_end_ix > len(sequence):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can demonstrate this function on the small contrived dataset.

The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the sequence
 if out_end_ix > len(sequence):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
```

```
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Now that we know how to prepare data for multi-step forecasting, let's look at a 1D CNN model that can learn this mapping.

Vector Output Model

The 1D CNN can output a vector directly that can be interpreted as a multi-step forecast.

This approach was seen in the previous section were one time step of each output time series was forecasted as a vector.

As with the 1D CNN models for univariate data in a prior section, the prepared samples must first be reshaped. The CNN expects data to have a three-dimensional structure of [*samples, timesteps, features*], and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

With the number of input and output steps specified in the *n_steps_in* and *n_steps_out* variables, we can define a multi-step time-series forecasting model.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

We would expect the predicted output to be:

```
[100, 110]
```

As expected by the model, the shape of the single sample of input data when making the prediction must be [1, 3, 1] for the 1 sample, 3 time steps of the input, and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Tying all of this together, the 1D CNN for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output 1d cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the sequence
 if out_end_ix > len(sequence):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example forecasts and prints the next two time steps in the sequence.

```
[[102.86651 115.08979]]
```

Multivariate Multi-Step CNN Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting.

It is possible to mix and match the different types of 1D CNN models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging.

In this section, we will explore short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-Step Output.
2. Multiple Parallel Input and Multi-Step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

Multiple Input Multi-Step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series.

For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Output:

```
65
85
```

The *split_sequences()* function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out-1
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out-1
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
```

```
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

Running the example first prints the shape of the prepared training data.

We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps and two variables for the two input time series.

The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted.

The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
```

```
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

We can now develop a 1D CNN model for multi-step predictions.

In this case, we will demonstrate a vector output model. The complete example is listed below.

```
# multivariate multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out-1
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
```

```
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset.

We would expect the next two steps to be [185, 205].

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
It is a challenging framing of the problem with very little data, and the arbitrarily configured version of the model gets close.

```
[[185.57011 207.77893]]
```

Multiple Parallel Input and Multi-Step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series.

For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

We may use the last three time steps from each of the three time series as input to the model, and predict the next time steps of each of the three time series as output.

The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Output:

```
40, 45, 85
50, 55, 105
```

The *split_sequences()* function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)
```

We can demonstrate this function on the small contrived dataset.

The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
```

```python
from keras.layers import TimeDistributed

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

Running the example first prints the shape of the prepared training dataset.

We can see that both the input (*X*) and output (*Y*) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively.
The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
```

```
 [20 25 45]
 [30 35 65]] [[ 40  45  85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50  55 105]
 [ 60  65 125]]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [[ 60  65 125]
 [ 70  75 145]]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [[ 70  75 145]
 [ 80  85 165]]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [[ 80  85 165]
 [ 90  95 185]]
```
We can now develop a 1D CNN model for this dataset.

We will use a vector-output model in this case. As such, we must flatten the three-dimensional structure of the output portion of each sample in order to train the model. This means, instead of predicting two steps for each series, the model is trained on and expected to predict a vector of six numbers directly.

```
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
```
The complete example is listed below.

```
# multivariate output multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
 # find the end of this pattern
 end_ix = i + n_steps_in
 out_end_ix = end_ix + n_steps_out
 # check if we are beyond the dataset
 if out_end_ix > len(sequences):
 break
 # gather input and output parts of the pattern
 seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
 X.append(seq_x)
 y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=7000, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

10.DEVELOP VECTOR AUTO REGRESSION MODEL FOR MULTIVARIATE TIME SERIES DATA FORECASTING

1. Introduction

Vector Autoregression (VAR) is a multivariate forecasting algorithm that is used when two or more time series influence each other.That means, the basic requirements in order to use VAR are:

1. You need at least two time series (variables)
2. The time series should influence each other.

why is it called 'Autoregressive'?

It is considered as an Autoregressive model because, each variable (Time Series) is modeled as a function of the past values, that is the predictors are nothing but the lags (time delayed value) of the series.

2. Intuition behind VAR Model Formula

If you remember in Autoregression models, the time series is modeled as a linear combination of it's own lags. That is, the past values of the series are used to forecast the current and future.

A typical AR(p) model equation looks something like this:

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \cdots + \beta_p Y_{t-p} + \epsilon_t$$

where α is the intercept, a constant and β1, β2 till βp are the coefficients of the lags of Y till order p.

Order 'p' means, up to p-lags of Y is used and they are the predictors in the equation. The ε_{t} is the error, which is considered as white noise.

Alright. So, how does a VAR model's formula look like?

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 you have multiple time series that influence each other, it is modeled as a system of equations with one equation per variable (time series).

That is, if you have 5 time series that influence each other, we will have a system of 5 equations.

Well, how is the equation exactly framed?

Let’s suppose, you have two variables (Time series) Y1 and Y2, and you need to forecast the values of these variables at time (t).

To calculate Y1(t), VAR will use the past values of both Y1 as well as Y2. Likewise, to compute Y2(t), the past values of both Y1 and Y2 be used.

For example, the system of equations for a VAR(1) model with two time series (variables `Y1` and `Y2`) is as follows:

$$Y_{1,t} = \alpha_1 + \beta_{11,1} Y_{1,t-1} + \beta_{12,1} Y_{2,t-1} + \epsilon_{1,t}$$
$$Y_{2,t} = \alpha_2 + \beta_{21,1} Y_{1,t-1} + \beta_{22,1} Y_{2,t-1} + \epsilon_{2,t}$$

Where, Y{1,t-1} and Y{2,t-1} are the first lag of time series Y1 and Y2 respectively.

The above equation is referred to as a VAR(1) model, because, each equation is of order 1, that is, it contains up to one lag of each of the predictors (Y1 and Y2).

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 (Y1 and Y2).

$$Y_{1,t} = \alpha_1 + \beta_{11,1} Y_{1,t-1} + \beta_{12,1} Y_{2,t-1} + \beta_{11,2} Y_{1,t-2} + \beta_{12,2} Y_{2,t-2} + \epsilon_{1,t}$$
$$Y_{2,t} = \alpha_2 + \beta_{21,1} Y_{1,t-1} + \beta_{22,1} Y_{2,t-1} + \beta_{21,2} Y_{1,t-2} + \beta_{22,2} Y_{2,t-2} + \epsilon_{2,t}$$

Can you imagine what a second order VAR(2) model with three variables (Y1, Y2 and Y3) would look like?

$$Y_{1,t} = \alpha_1 + \beta_{11,1} Y_{1,t-1} + \beta_{12,1} Y_{2,t-1} + \beta_{13,1} Y_{3,t-1} + \beta_{11,2} Y_{1,t-2} + \beta_{12,2} Y_{2,t-2} + \beta_{13,2} Y_{3,t-2} + \epsilon_{1,t}$$
$$Y_{2,t} = \alpha_2 + \beta_{21,1} Y_{1,t-1} + \beta_{22,1} Y_{2,t-1} + \beta_{23,1} Y_{3,t-1} + \beta_{21,2} Y_{1,t-2} + \beta_{22,2} Y_{2,t-2} + \beta_{23,2} Y_{3,t-2} + \epsilon_{2,t}$$
$$Y_{3,t} = \alpha_3 + \beta_{31,1} Y_{1,t-1} + \beta_{32,1} Y_{2,t-1} + \beta_{33,1} Y_{3,t-1} + \beta_{31,2} Y_{1,t-2} + \beta_{32,2} Y_{2,t-2} + \beta_{33,2} Y_{3,t-2} + \epsilon_{3,t}$$

As you increase the number of time series (variables) in the model the system of equations become larger.

3. Building a VAR model in Python

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

```
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
```

4. Import the datasets

For this article let's use the time series used in Yash P Mehra's 1994 article: "Wage Growth and the Inflation Process: An Empirical Approach".

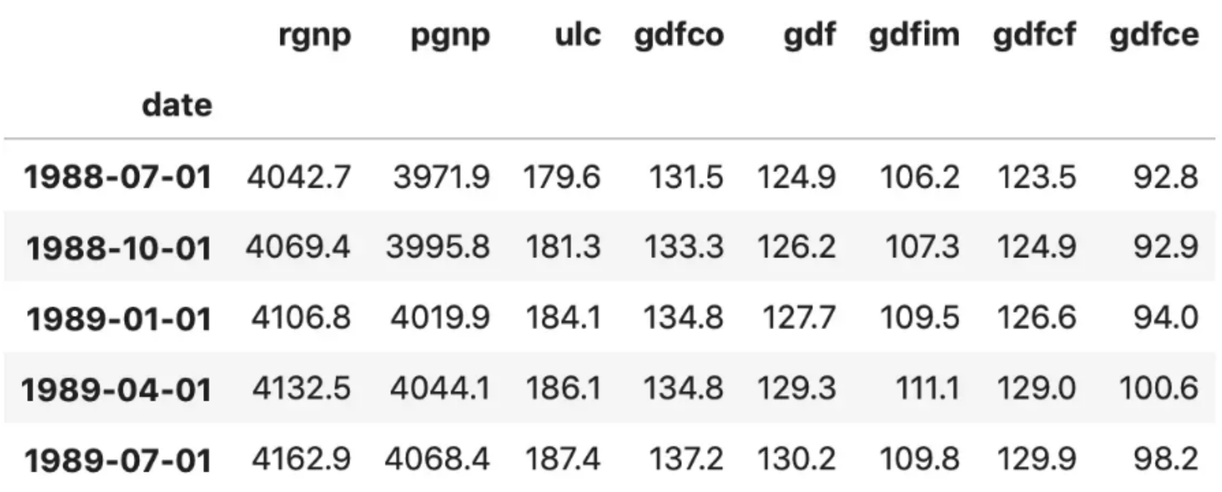
This dataset has the following 8 quarterly time series:

```
1. rgnp  : Real GNP.

2. pgnp  : Potential real GNP.

3. ulc   : Unit labor cost.

4. gdfco : Fixed weight deflator for personal consumption expenditure excluding food and energy.

5. gdf   : Fixed weight GNP deflator.

6. gdfim : Fixed weight import deflator.

7. gdfcf : Fixed weight deflator for food in personal consumption expenditure.

8. gdfce : Fixed weight deflator for energy in personal consumption expenditure.
```

Let's import the data.

```
filepath = 'https://raw.githubusercontent.com/selva86/datasets/master/Raotbl6.csv'
```

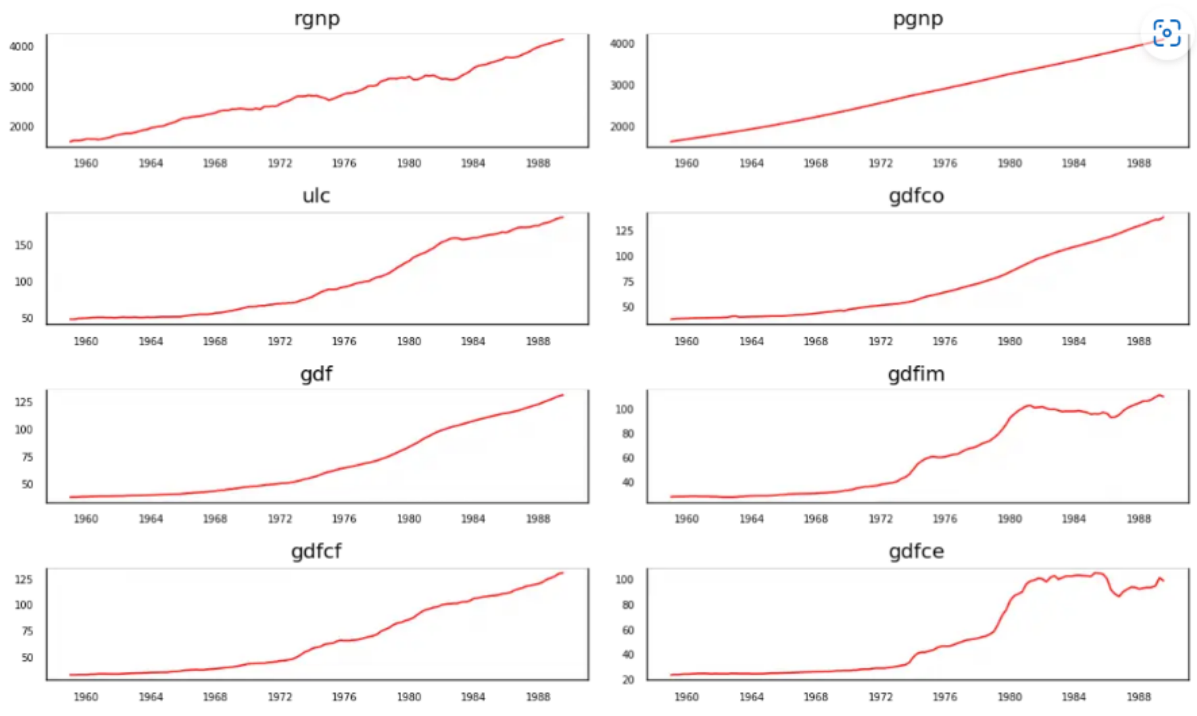
```
df = pd.read_csv(filepath, parse_dates=['date'], index_col='date')

print(df.shape)  # (123, 8)

df.tail()
```



date	rgnp	pgnp	ulc	gdfco	gdf	gdfim	gdfcf	gdfce
1988-07-01	4042.7	3971.9	179.6	131.5	124.9	106.2	123.5	92.8
1988-10-01	4069.4	3995.8	181.3	133.3	126.2	107.3	124.9	92.9
1989-01-01	4106.8	4019.9	184.1	134.8	127.7	109.5	126.6	94.0
1989-04-01	4132.5	4044.1	186.1	134.8	129.3	111.1	129.0	100.6
1989-07-01	4162.9	4068.4	187.4	137.2	130.2	109.8	129.9	98.2

5. Visualize the Time Series

```
# Plot

fig, axes = plt.subplots(nrows=4, ncols=2, dpi=120, figsize=(10,6))

for i, ax in enumerate(axes.flatten()):

    data = df[df.columns[i]]

    ax.plot(data, color='red', linewidth=1)

    # Decorations

    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=6)



plt.tight_layout();
```



Actual Multi Dimensional Time Series for VAR model

Actual Multi Dimensional Time Series for VAR model

6. Testing Causation using Granger's Causality Test

The basis behind Vector AutoRegression is that each of the time series in the system influences each other. That is, you can predict the series with past values of itself along with other series in the system.

Using Granger’s Causality Test, it’s possible to test this relationship before even building the model.

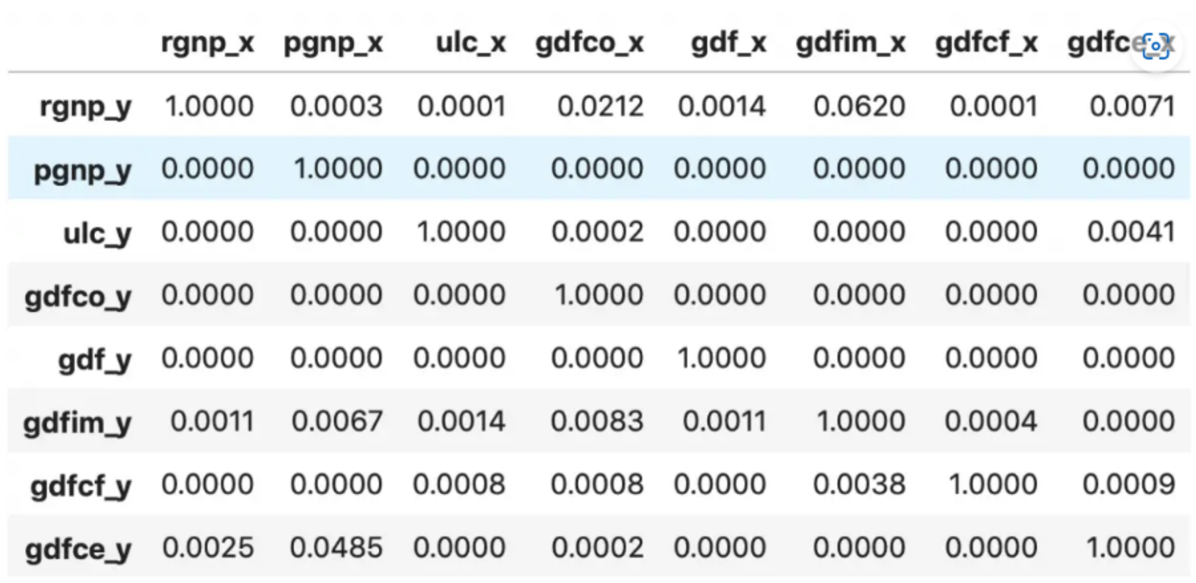
So what does Granger’s Causality really test?

Granger’s causality tests the null hypothesis that the coefficients of past values in the regression equation is zero.

In simpler terms, the past values of time series (X) do not cause the other series (Y). So, if the p-value obtained from the test is lesser than the significance level of 0.05, then, you can safely reject the null hypothesis.

The below code implements the Granger’s Causality test for all possible combinations of the time series in a given dataframe and stores the p-values of each combination in the output matrix.

```
from statsmodels.tsa.stattools import grangercausalitytests

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 is

    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



grangers_causation_matrix(df, variables = df.columns)
```



	rgnp_x	pgnp_x	ulc_x	gdfco_x	gdf_x	gdfim_x	gdfcf_x	gdfce_x
rgnp_y	1.0000	0.0003	0.0001	0.0212	0.0014	0.0620	0.0001	0.0071
pgnp_y	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ulc_y	0.0000	0.0000	1.0000	0.0002	0.0000	0.0000	0.0000	0.0041
gdfco_y	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000
gdf_y	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000
gdfim_y	0.0011	0.0067	0.0014	0.0083	0.0011	1.0000	0.0004	0.0000
gdfcf_y	0.0000	0.0000	0.0008	0.0008	0.0000	0.0038	1.0000	0.0009
gdfce_y	0.0025	0.0485	0.0000	0.0002	0.0000	0.0000	0.0000	1.0000

For example, if you take the value 0.0003 in (row 1, column 2), it refers to the p-value of pgnp_x causing rgnp_y. Whereas, the 0.000 in (row 2, column 1) refers to the p-value of rgnp_y causing pgnp_x.

So, how to interpret the p-values?

If a given p-value is < significance level (0.05), then, the corresponding X series (column) causes the Y (row).

For example, P-Value of 0.0003 at (row 1, column 2) represents the p-value of the Grangers Causality test for pgnp_x causing rgnp_y, which is less that the significance level of 0.05.

So, you can reject the null hypothesis and conclude pgnp_x causes rgnp_y.

Looking at the P-Values in the above table, you can pretty much observe that all the variables (time series) in the system are interchangeably causing each other.

This makes this system of multi time series a good candidate for using VAR models to forecast.

Next, let’s do the Cointegration test.

7. Cointegration Test

Cointegration test helps to establish the presence of a statistically significant connection between two or more time series.

But, what does Cointegration mean?
It is fairly straightforward to implement in python's statsmodels, as you can see below.

```
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.lr1
    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)
Results:
ame   ::  Test Stat > C(95%)    =>   Signif
 ----------------------------------------
rgnp   ::  248.0     > 143.6691  =>    True
pgnp   ::  183.12    > 111.7797  =>    True
ulc    ::  130.01    > 83.9383   =>    True
gdfco  ::  85.28     > 60.0627   =>    True
gdf    ::  55.05     > 40.1749   =>    True
gdfim  ::  31.59     > 24.2761   =>    True
gdfcf  ::  14.06     > 12.3212   =>    True
gdfce  ::  0.45      > 4.1296    =>    False
8. Split the Series into Training and Testing Data
Splitting the dataset into training and test data.
The VAR model will be fitted on df_train and then used to forecast the next 4
observations. These forecasts will be compared against the actuals present in
test data.
To do the comparisons, we will use multiple forecast accuracy metrics, as seen
later in this article.
nobs = 4
df_train, df_test = df[0:-nobs], df[-nobs:]

# Check size
print(df_train.shape)  # (119, 8)
print(df_test.shape)  # (4, 8)
```

9. Check for Stationarity and Make the Time Series Stationary

Since the VAR model requires the time series you want to forecast to be stationary, it is customary to check all the time series in the system for stationarity.

Just to refresh, a stationary time series is one whose characteristics like mean and variance does not change over time.

So, how to test for stationarity?

There is a suite of tests called unit-root tests. The popular ones are:

Augmented Dickey-Fuller Test (ADF Test)

KPSS test

Philip-Perron test

Let’s use the ADF test for our purpose.

By the way, if a series is found to be non-stationary, you make it stationary by differencing the series once and repeat the test again until it becomes stationary.

Since, differencing reduces the length of the series by 1 and since all the time series has to be of the same length, you need to difference all the series in the system if you choose to difference at all.

Let’s implement the ADF Test.

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.

```
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}"', "\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.")
```

Call the adfuller_test() on each series.

```
# ADF Test on each column
for name, column in df_train.iteritems():
    adfuller_test(column, name=column.name)
    print('\n')
```

Results:

```
    Augmented Dickey-Fuller Test on "rgnp"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
```

```
    Significance Level    = 0.05
    Test Statistic        = 0.5428
    No. Lags Chosen       = 2
    Critical value 1%     = -3.488
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.9861. Weak evidence to reject the Null Hypothesis.
    => Series is Non-Stationary.


     Augmented Dickey-Fuller Test on "pgnp"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = 1.1556
    No. Lags Chosen       = 1
    Critical value 1%     = -3.488
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.9957. Weak evidence to reject the Null Hypothesis.
    => Series is Non-Stationary.


       Augmented Dickey-Fuller Test on "ulc"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = 1.2474
    No. Lags Chosen       = 2
    Critical value 1%     = -3.488
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.9963. Weak evidence to reject the Null Hypothesis.
    => Series is Non-Stationary.


       Augmented Dickey-Fuller Test on "gdfco"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = 1.1954
    No. Lags Chosen       = 3
    Critical value 1%     = -3.489
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.996. Weak evidence to reject the Null Hypothesis.
    => Series is Non-Stationary.


       Augmented Dickey-Fuller Test on "gdf"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = 1.676
    No. Lags Chosen       = 7
    Critical value 1%     = -3.491
```

```
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.9981. Weak evidence to reject the Null Hypothesis.
 => Series is Non-Stationary.


    Augmented Dickey-Fuller Test on "gdfim"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -0.0799
 No. Lags Chosen       = 1
 Critical value 1%     = -3.488
 Critical value 5%     = -2.887
 Critical value 10%    = -2.58
 => P-Value = 0.9514. Weak evidence to reject the Null Hypothesis.
 => Series is Non-Stationary.


    Augmented Dickey-Fuller Test on "gdfcf"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = 1.4395
 No. Lags Chosen       = 8
 Critical value 1%     = -3.491
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.9973. Weak evidence to reject the Null Hypothesis.
 => Series is Non-Stationary.


    Augmented Dickey-Fuller Test on "gdfce"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -0.3402
 No. Lags Chosen       = 8
 Critical value 1%     = -3.491
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.9196. Weak evidence to reject the Null Hypothesis.
 => Series is Non-Stationary.
The ADF test confirms none of the time series is stationary. Let’s difference
all of them once and check again.
# 1st difference
df_differenced = df_train.diff().dropna()
Re-run ADF test on each differenced series.
# ADF Test on each column of 1st Differences Dataframe
for name, column in df_differenced.iteritems():
    adfuller_test(column, name=column.name)
    print('\n')
    Augmented Dickey-Fuller Test on "rgnp"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -5.3448
```

```
No. Lags Chosen       = 1
Critical value 1%     = -3.488
Critical value 5%     = -2.887
Critical value 10%    = -2.58
=> P-Value = 0.0. Rejecting Null Hypothesis.
=> Series is Stationary.


   Augmented Dickey-Fuller Test on "pgnp"
   -----------------------------------------------
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic        = -1.8282
No. Lags Chosen       = 0
Critical value 1%     = -3.488
Critical value 5%     = -2.887
Critical value 10%    = -2.58
=> P-Value = 0.3666. Weak evidence to reject the Null Hypothesis.
=> Series is Non-Stationary.


   Augmented Dickey-Fuller Test on "ulc"
   -----------------------------------------------
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic        = -3.4658
No. Lags Chosen       = 1
Critical value 1%     = -3.488
Critical value 5%     = -2.887
Critical value 10%    = -2.58
=> P-Value = 0.0089. Rejecting Null Hypothesis.
=> Series is Stationary.


   Augmented Dickey-Fuller Test on "gdfco"
   -----------------------------------------------
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic        = -1.4385
No. Lags Chosen       = 2
Critical value 1%     = -3.489
Critical value 5%     = -2.887
Critical value 10%    = -2.58
=> P-Value = 0.5637. Weak evidence to reject the Null Hypothesis.
=> Series is Non-Stationary.


   Augmented Dickey-Fuller Test on "gdf"
   -----------------------------------------------
Null Hypothesis: Data has unit root. Non-Stationary.
Significance Level    = 0.05
Test Statistic        = -1.1289
No. Lags Chosen       = 2
Critical value 1%     = -3.489
Critical value 5%     = -2.887
Critical value 10%    = -2.58
=> P-Value = 0.7034. Weak evidence to reject the Null Hypothesis.
```

```
 => Series is Non-Stationary.


    Augmented Dickey-Fuller Test on "gdfim"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -4.1256
 No. Lags Chosen       = 0
 Critical value 1%     = -3.488
 Critical value 5%     = -2.887
 Critical value 10%    = -2.58
 => P-Value = 0.0009. Rejecting Null Hypothesis.
 => Series is Stationary.


    Augmented Dickey-Fuller Test on "gdfcf"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -2.0545
 No. Lags Chosen       = 7
 Critical value 1%     = -3.491
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.2632. Weak evidence to reject the Null Hypothesis.
 => Series is Non-Stationary.


    Augmented Dickey-Fuller Test on "gdfce"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -3.1543
 No. Lags Chosen       = 7
 Critical value 1%     = -3.491
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.0228. Rejecting Null Hypothesis.
 => Series is Stationary.
After the first difference, Real Wages (Manufacturing) is still not stationary.
It's critical value is between 5% and 10% significance level.
All of the series in the VAR model should have the same number of observations.
So, we are left with one of two choices.
That is, either proceed with 1st differenced series or difference all the
series one more time.
# Second Differencing
df_differenced = df_differenced.diff().dropna()
Re-run ADF test again on each second differenced series.
# ADF Test on each column of 2nd Differences Dataframe
for name, column in df_differenced.iteritems():
    adfuller_test(column, name=column.name)
    print('\n')
Results:
    Augmented Dickey-Fuller Test on "rgnp"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
```

```
    Significance Level    = 0.05
    Test Statistic        = -9.0123
    No. Lags Chosen       = 2
    Critical value 1%     = -3.489
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.0. Rejecting Null Hypothesis.
    => Series is Stationary.


       Augmented Dickey-Fuller Test on "pgnp"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = -10.9813
    No. Lags Chosen       = 0
    Critical value 1%     = -3.488
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.0. Rejecting Null Hypothesis.
    => Series is Stationary.


       Augmented Dickey-Fuller Test on "ulc"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = -8.769
    No. Lags Chosen       = 2
    Critical value 1%     = -3.489
    Critical value 5%     = -2.887
    Critical value 10%    = -2.58
    => P-Value = 0.0. Rejecting Null Hypothesis.
    => Series is Stationary.


       Augmented Dickey-Fuller Test on "gdfco"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = -7.9102
    No. Lags Chosen       = 3
    Critical value 1%     = -3.49
    Critical value 5%     = -2.887
    Critical value 10%    = -2.581
    => P-Value = 0.0. Rejecting Null Hypothesis.
    => Series is Stationary.


       Augmented Dickey-Fuller Test on "gdf"
       -----------------------------------------------
    Null Hypothesis: Data has unit root. Non-Stationary.
    Significance Level    = 0.05
    Test Statistic        = -10.0351
    No. Lags Chosen       = 1
    Critical value 1%     = -3.489
    Critical value 5%     = -2.887
```

```
 Critical value 10%    = -2.58
 => P-Value = 0.0. Rejecting Null Hypothesis.
 => Series is Stationary.


    Augmented Dickey-Fuller Test on "gdfim"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -9.4059
 No. Lags Chosen       = 1
 Critical value 1%     = -3.489
 Critical value 5%     = -2.887
 Critical value 10%    = -2.58
 => P-Value = 0.0. Rejecting Null Hypothesis.
 => Series is Stationary.


    Augmented Dickey-Fuller Test on "gdfcf"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -6.922
 No. Lags Chosen       = 5
 Critical value 1%     = -3.491
 Critical value 5%     = -2.888
 Critical value 10%    = -2.581
 => P-Value = 0.0. Rejecting Null Hypothesis.
 => Series is Stationary.


    Augmented Dickey-Fuller Test on "gdfce"
    -----------------------------------------------
 Null Hypothesis: Data has unit root. Non-Stationary.
 Significance Level    = 0.05
 Test Statistic        = -5.1732
 No. Lags Chosen       = 8
 Critical value 1%     = -3.492
 Critical value 5%     = -2.889
 Critical value 10%    = -2.581
 => P-Value = 0.0. Rejecting Null Hypothesis.
 => Series is Stationary.
```

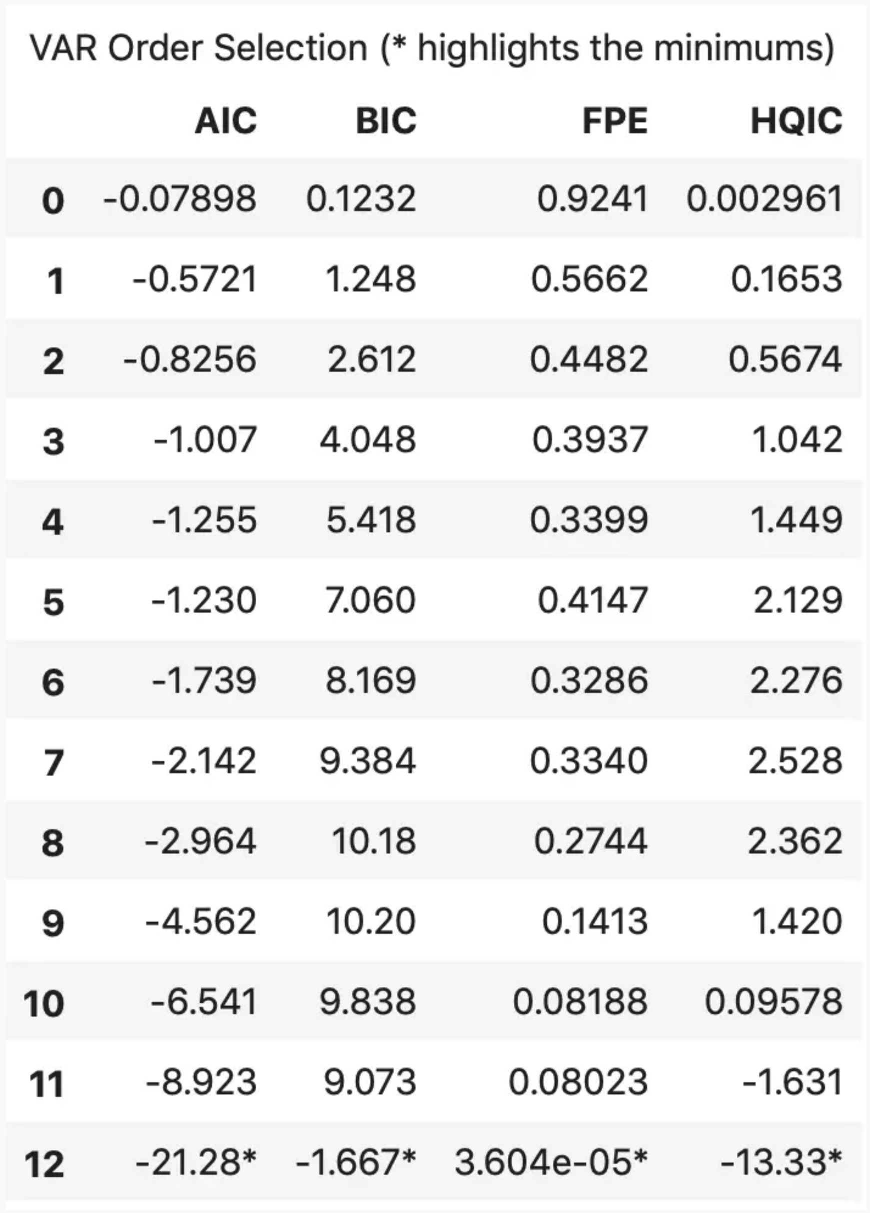
All the series are now stationary.
Let’s prepare the training and test datasets.

10. How to Select the Order (P) of VAR model

To select the right order of the VAR model, we iteratively fit increasing orders of VAR model and pick the order that gives a model with least AIC. Though the usual practice is to look at the AIC, you can also check other best fit comparison estimates of BIC, FPE and HQIC.

```
model = VAR(df_differenced)
for i in [1,2,3,4,5,6,7,8,9]:
    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')
```

```
Results:
Lag Order = 1
AIC :  -1.3679402315450664
BIC :  0.3411847146588838
FPE :  0.2552682517347198
HQIC:  -0.6741331335699554

Lag Order = 2
AIC :  -1.621237394447824
BIC :  1.6249432095295848
FPE :  0.2011349437137139
HQIC:  -0.3036288826795923

Lag Order = 3
AIC :  -1.7658008387012791
BIC :  3.0345473163767833
FPE :  0.18125103746164364
HQIC:  0.18239143783963296

Lag Order = 4
AIC :  -2.000735164470318
BIC :  4.3712151376540875
FPE :  0.15556966521481097
HQIC:  0.5849359332771069

Lag Order = 5
AIC :  -1.9619535608363954
BIC :  5.9993645622420955
FPE :  0.18692794389114886
HQIC:  1.268206331178333

Lag Order = 6
AIC :  -2.3303386524829053
BIC :  7.2384526890885805
FPE :  0.16380374017443664
HQIC:  1.5514371669548073

Lag Order = 7
AIC :  -2.592331352347129
BIC :  8.602387254937796
FPE :  0.1823868583715414
HQIC:  1.9483069621146551

Lag Order = 8
AIC :  -3.317261976458205
BIC :  9.52219581032303
FPE :  0.15573163248209088
HQIC:  1.8896071386220985

Lag Order = 9
AIC :  -4.804763125958631
BIC :  9.698613139231597
FPE :  0.08421466682671915
HQIC:  1.0758291640834052
In the above output, the AIC drops to lowest at lag 4, then increases at lag 5
and then continuously drops further.
```



VAR Order Selection (* highlights the minimums)

	AIC	BIC	FPE	HQIC
0	-0.07898	0.1232	0.9241	0.002961
1	-0.5721	1.248	0.5662	0.1653
2	-0.8256	2.612	0.4482	0.5674
3	-1.007	4.048	0.3937	1.042
4	-1.255	5.418	0.3399	1.449
5	-1.230	7.060	0.4147	2.129
6	-1.739	8.169	0.3286	2.276
7	-2.142	9.384	0.3340	2.528
8	-2.964	10.18	0.2744	2.362
9	-4.562	10.20	0.1413	1.420
10	-6.541	9.838	0.08188	0.09578
11	-8.923	9.073	0.08023	-1.631
12	-21.28*	-1.667*	3.604e-05*	-13.33*

```
        11. Train the VAR Model of Selected Order(p)
model_fitted = model.fit(4)
model_fitted.summary()
Results:
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Sat, 18, May, 2019
Time:                     11:35:15
--------------------------------------------------------------------
No. of Equations:         8.00000    BIC:                    4.37122
```

```
Nobs:                      113.000    HQIC:                   0.584936
Log likelihood:           -905.679    FPE:                    0.155570
AIC:                      -2.00074    Det(Omega_mle):        0.0200322
--------------------------------------------------------------------
Results for equation rgnp
==========================================================================
              coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------
const            2.430021         2.677505            0.908           0.364
L1.rgnp         -0.750066         0.159023           -4.717           0.000
L1.pgnp         -0.095621         4.938865           -0.019           0.985
L1.ulc          -6.213996         4.637452           -1.340           0.180
L1.gdfco        -7.414768        10.184884           -0.728           0.467
L1.gdf         -24.864063        20.071245           -1.239           0.215
L1.gdfim         1.082913         4.309034            0.251           0.802
L1.gdfcf        16.327252         5.892522            2.771           0.006
L1.gdfce         0.910522         2.476361            0.368           0.713
L2.rgnp         -0.568178         0.163971           -3.465           0.001
L2.pgnp         -1.156201         4.931931           -0.234           0.815
L2.ulc         -11.157111         5.381825           -2.073           0.038
L2.gdfco         3.012518        12.928317            0.233           0.816
L2.gdf         -18.143523        24.090598           -0.753           0.451
L2.gdfim        -4.438115         4.410654           -1.006           0.314
L2.gdfcf        13.468228         7.279772            1.850           0.064
L2.gdfce         5.130419         2.805310            1.829           0.067
L3.rgnp         -0.514985         0.152724           -3.372           0.001
L3.pgnp        -11.483607         5.392037           -2.130           0.033
L3.ulc         -14.195308         5.188718           -2.736           0.006
L3.gdfco       -10.154967        13.105508           -0.775           0.438
L3.gdf         -15.438858        21.610822           -0.714           0.475
L3.gdfim        -6.405290         4.292790           -1.492           0.136
L3.gdfcf         9.217402         7.081652            1.302           0.193
L3.gdfce         5.279941         2.833925            1.863           0.062
L4.rgnp         -0.166878         0.138786           -1.202           0.229
L4.pgnp          5.329900         5.795837            0.920           0.358
L4.ulc          -4.834548         5.259608           -0.919           0.358
L4.gdfco        10.841602        10.526530            1.030           0.303
L4.gdf         -17.651510        18.746673           -0.942           0.346
L4.gdfim        -1.971233         4.029415           -0.489           0.625
L4.gdfcf         0.617824         5.842684            0.106           0.916
L4.gdfce        -2.977187         2.594251           -1.148           0.251
==========================================================================

Results for equation pgnp
==========================================================================
              coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------
const            0.094556         0.063491            1.489           0.136
L1.rgnp         -0.004231         0.003771           -1.122           0.262
L1.pgnp          0.082204         0.117114            0.702           0.483
L1.ulc          -0.097769         0.109966           -0.889           0.374

(... TRUNCATED because of long output....)
(... TRUNCATED because of long output....)
(... TRUNCATED because of long output....)

Correlation matrix of residuals
```

```
             rgnp      pgnp       ulc     gdfco       gdf     gdfim     gdfcf
gdfce
rgnp     1.000000  0.248342 -0.668492 -0.160133 -0.047777  0.084925  0.009962
0.205557
pgnp     0.248342  1.000000 -0.148392 -0.167766 -0.134896  0.007830 -0.169435
0.032134
ulc     -0.668492 -0.148392  1.000000  0.268127  0.327761  0.171497  0.135410
-0.026037
gdfco   -0.160133 -0.167766  0.268127  1.000000  0.303563  0.232997 -0.035042
0.184834
gdf     -0.047777 -0.134896  0.327761  0.303563  1.000000  0.196670  0.446012
0.309277
gdfim    0.084925  0.007830  0.171497  0.232997  0.196670  1.000000 -0.089852
0.707809
gdfcf    0.009962 -0.169435  0.135410 -0.035042  0.446012 -0.089852  1.000000
-0.197099
gdfce    0.205557  0.032134 -0.026037  0.184834  0.309277  0.707809 -0.197099
1.000000
```

12. Check for Serial Correlation of Residuals (Errors) using Durbin Watson Statistic

If there is any correlation left in the residuals, then, there is some pattern in the time series that is still left to be explained by the model. In that case, the 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 to ensure that the model is sufficiently able to explain the variances and patterns in the time series.

Alright, coming back to topic.

A common way of checking for serial correlation of errors can be measured using the Durbin Watson's Statistic

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.

```
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))
```

Results:

```
rgnp   : 2.09
pgnp   : 2.02
ulc    : 2.17
gdfco  : 2.05
gdf    : 2.25
gdfim  : 1.99
gdfcf  : 2.2
gdfce  : 2.17
```

The serial correlation seems quite alright. Let's proceed with the forecast.

13. How to Forecast VAR model using statsmodels

```
# Get the lag order
lag_order = model_fitted.k_ar
print(lag_order)  #> 4

# Input data for forecasting
forecast_input = df_differenced.values[-lag_order:]
forecast_input
4

array([[ 13.5,   0.1,   1.4,   0.1,   0.1,  -0.1,   0.4,  -2. ],
       [-23.6,   0.2,  -2. ,  -0.5,  -0.1,  -0.2,  -0.3,  -1.2],
       [ -3.3,   0.1,   3.1,   0.5,   0.3,   0.4,   0.9,   2.2],
       [ -3.9,   0.2,  -2.1,  -0.4,   0.2,  -1.5,   0.9,  -0.3]])
```

Let’s forecast.

```
# Forecast
fc = model_fitted.forecast(y=forecast_input, steps=nobs)
df_forecast = pd.DataFrame(fc, index=df.index[-nobs:], columns=df.columns +
'_2d')
df_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.
14. Invert the transformation to get the real forecast

```
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
df_results = invert_transformation(train, df_forecast, second_diff=True)
df_results.loc[:, ['rgnp_forecast', 'pgnp_forecast', 'ulc_forecast',
'gdfco_forecast',
                   'gdf_forecast', 'gdfim_forecast', 'gdfcf_forecast',
'gdfce_forecast']]
```

```
The forecasts are back to the original scale. Let’s plot the forecasts against
the actuals from test data.
15. Plot of Forecast vs Actuals
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=True)
    df_test[col][-nobs:].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=6)

plt.tight_layout();
Forecast vs Actuals comparison of VAR model
16. Evaluate the Forecasts
To evaluate the forecasts, let’s compute a comprehensive set of metrics,
namely, the MAPE, ME, MAE, MPE, RMSE, corr and minmax.
from statsmodels.tsa.stattools import acf
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual))  # MAPE
    me = np.mean(forecast - actual)             # ME
    mae = np.mean(np.abs(forecast - actual))    # MAE
    mpe = np.mean((forecast - actual)/actual)   # MPE
    rmse = np.mean((forecast - actual)**2)**.5  # RMSE
    corr = np.corrcoef(forecast, actual)[0,1]   # corr
    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)             # minmax
    return({'mape':mape, 'me':me, 'mae': mae,
            'mpe': mpe, 'rmse':rmse, 'corr':corr, 'minmax':minmax})

print('Forecast Accuracy of: rgnp')
accuracy_prod = forecast_accuracy(df_results['rgnp_forecast'].values,
df_test['rgnp'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: pgnp')
accuracy_prod = forecast_accuracy(df_results['pgnp_forecast'].values,
df_test['pgnp'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: ulc')
accuracy_prod = forecast_accuracy(df_results['ulc_forecast'].values,
df_test['ulc'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: gdfco')
accuracy_prod = forecast_accuracy(df_results['gdfco_forecast'].values,
df_test['gdfco'])
```

```
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: gdf')
accuracy_prod = forecast_accuracy(df_results['gdf_forecast'].values,
df_test['gdf'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: gdfim')
accuracy_prod = forecast_accuracy(df_results['gdfim_forecast'].values,
df_test['gdfim'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: gdfcf')
accuracy_prod = forecast_accuracy(df_results['gdfcf_forecast'].values,
df_test['gdfcf'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))

print('\nForecast Accuracy of: gdfce')
accuracy_prod = forecast_accuracy(df_results['gdfce_forecast'].values,
df_test['gdfce'])
for k, v in accuracy_prod.items():
    print(adjust(k), ': ', round(v,4))
Forecast Accuracy of: rgnp
mape   :  0.0192
me     :  79.1031
mae    :  79.1031
mpe    :  0.0192
rmse   :  82.0245
corr   :  0.9849
minmax :  0.0188

Forecast Accuracy of: pgnp
mape   :  0.0005
me     :  2.0432
mae    :  2.0432
mpe    :  0.0005
rmse   :  2.146
corr   :  1.0
minmax :  0.0005

Forecast Accuracy of: ulc
mape   :  0.0081
me     :  -1.4947
mae    :  1.4947
mpe    :  -0.0081
rmse   :  1.6856
corr   :  0.963
minmax :  0.0081

Forecast Accuracy of: gdfco
mape   :  0.0033
me     :  0.0007
mae    :  0.4384
```

```
mpe    :  0.0
rmse   :  0.5169
corr   :  0.9407
minmax :  0.0032

Forecast Accuracy of: gdf
mape   :  0.0023
me     :  0.2554
mae    :  0.29
mpe    :  0.002
rmse   :  0.3392
corr   :  0.9905
minmax :  0.0022

Forecast Accuracy of: gdfim
mape   :  0.0097
me     :  -0.4166
mae    :  1.06
mpe    :  -0.0038
rmse   :  1.0826
corr   :  0.807
minmax :  0.0096

Forecast Accuracy of: gdfcf
mape   :  0.0036
me     :  -0.0271
mae    :  0.4604
mpe    :  -0.0002
rmse   :  0.5286
corr   :  0.9713
minmax :  0.0036

Forecast Accuracy of: gdfce
mape   :  0.0177
me     :  0.2577
mae    :  1.72
mpe    :  0.0031
rmse   :  2.034
corr   :  0.764
minmax :  0.0175
```

17. Conclusion

In this article we covered VAR from scratch beginning from the intuition behind it, interpreting the formula, causality tests, finding the optimal order of the VAR model, preparing the data for forecasting, build the model, checking for serial autocorrelation, inverting the transform to get the actual forecasts, plotting the results and computing the accuracy metrics.

Hope you enjoyed reading this as much as I did writing it. I will see you in the next one.