POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Python and Quality Data Analysis tutorials

Edited by:
Giorgio Capuana

Prof: Bianca Maria Colosimo
Tutor: Prof. Matteo Bugatti
Year 2023/2024 - 2nd semester

Contents

Contents **i**

Contents **i**

List of Figures **iii**

List of Figures **iii**

List of Codes **vii**

1 Introduction **1**

1.1 Python basic functions . . . 1

1.1.1 Numpy enviroment, tools and print functions . . . 1

1.1.2 Pandas environment and Series/DataFrame management . . . 3

1.1.3 List of basic needed commands . . . 5

1.1.4 Graphical representation: matplotlib to support stastical analysis . . . 5

1.2 Data modeling and preparation: Box Cox normality transformation + gapping algorithm + batching algorithm . . . 14

1.2.1 Box Cox transformation . . . 14

1.2.2 Data gapping algorithm . . . 15

1.2.3 Data batching . . . 15

2 Inferencial statistic **16**

2.1 Hypothesis tests in presence of one sample (on one population) . . . 16

2.1.1 Z test on the mean: mean is unkown, variance is known . . . 16

2.1.2 T test on the mean: mean and variance are unkown . . . 19

2.1.3 Chi2 test on the variance and standard deviation . . . 21

2.2 Hypothesis tests in presence of two samples (on two populations) . . . 24

2.2.1 Z test on the difference in means: variances are known . . . 24

2.2.2 T test of the difference in means: variances are unknown but EQUAL . . . 25

2.2.3 T test of the difference in means (Welch's T-test): variances are unkown but DIFFERENT . . . 25

2.2.4 Paired T-test for the mean of delta vector "D" = Dj = X1j-X2j : variances of the differences between pairs in unkown . . . 26

2.2.5 F-test for equality of variances: variances are unkown . . . 27

2.3 Test of hypothesis for assumption compliance . . . 29

2.3.1 Runs test . . . 29

2.3.2 Barlett test . . . 30

2.3.3 Ljung Box Pierce or LBQ test . . . 31

3 Time series modeling: linear models and ARIMA **33**

3.1 How to create and assess linear models . . . 33

3.1.1 Test of hypothesis on the significance of the simple linear regression coefficients . . . 36

3.1.2 Confidence and prediction intervals respectively for the mean and single next process outcome . . . 37

3.1.3 STEPWISE REGRESSION APPROACH . . . 40

3.1.4 Confidence and prediction intervals on the step-wise regression output . . . 42

3.2 ARIMA . . . 43

4 PCA **46**

4.1 Principal components analysis on python . . . 48

4.1.1 PCA with covariance matrix . . . 48

4.1.2 PCA with correlation matrix . . . 49

4.1.3 Main comments on loadings plots . . . 51

4.2 Data reconstruction . . . 52

4.2.1 How to reconstruct the data with k < P principal components . . . 52

4.2.2 How to reconstruct the data with python . . . 53

5 SPC iid **54**

5.1 Shewhart's approach based SPC . . . 54

5.1.1 Xbar-R control charts with known parameters . . . 54

5.1.2 Xbar-R control charts with unknown parameters . . . 54

5.1.3 OC curve for Xbar and R chart + ARL + ATS . . . 57

5.1.4 Xbar-S charts (n constant) . . . 58

5.1.5 Preparation of the DataFrame in case of normality violation for n = const > 2 . . . 58

5.1.6 Xbar-S control charts (n variable) . . . 60

5.1.7 X and I-MR control charts for individuals . . . 61

5.2 Probabilistic control chart . . . 63

6 SPC non iid **66**

6.1 Trend control charts and model-based control charts . . . 66

6.2 Fitted Value control chart and Special Cause control chart . . . 68

6.2.1 SCC charts: handling OOCs in case of assignable causes . . . 69

6.2.2 SCC charts: handling OOCs in case of NO ASSIGNABLE CAUSES . . . 69

6.3 Between group control charts for SPC non iid . . . 71

7 Multivariate control charts **73**

7.0.1 Multivariate control charts for batched process: n > 1 . . . 73

7.0.2 Multivariate control charts for individuals: n = 1 . . . 76

7.1 Univariate control charts for multivariate processes . . . 80

8 Small shift control charts **81**

8.1 Application field and main differences respect to Shewhart's approach . . . 81

8.1.1 EWMA Control Chart for $n > 1$ . . . 81

8.1.2 EWMA Control Chart for Individual Observations ($n = 1$) . . . 84

9 Predefined algorithm to handle data **88**

9.1 Outliers detection function . . . 88

List of Figures

1.1 Time series plot from a pandas dataframe . . . 6
1.2 example of Chi2 random generated dataset . . . 6
1.3 Time series original vs fitted data . . . 6
1.4 Time series of "n" batches overlapped on one single plot . . . 6
1.5 Simplest type of histogram plot . . . 7
1.6 Histograms plot with KDE . . . 7
1.7 QQ plot in case of normally distributed data . . . 8
1.8 QQ plot in case of skewed distributions . . . 8
1.9 Scatter plot to check on autocorrelation (+ trend line) . . . 8
1.10 Scatter plot to check on correlation between two variables . . . 8
1.11 PCA: Scatter plot PCs correlation (+ cross validation) . . . 8
1.12 Batch process stacked data . . . 9
1.13 Batch process unstacked data . . . 9
1.14 BATCH PROCESS: Piled points scatter plot sorted by batches index . . . 10
1.15 Typical dataframe for CC with n > 1 . . . 10
1.16 BATCH PROCESS: scattered data sorted by the position index within the batches . . . 10
1.17 Scatter matrix for covariance (uncorrelated variables) . . . 11
1.18 Scatter matrix for covariance (correlated variables) . . . 11
1.19 2 Boxplots in 1 plot . . . 11
1.20 2 Boxplots in 2 subplots . . . 11
1.21 Autocorrelation Function . . . 12
1.22 Partial Autocorrelation Function . . . 12
1.23 Box Cox transformation . . . 14
1.24 Box Cox normality plot . . . 14
1.25 Gapping size effect . . . 15
1.26 Batch size effect . . . 15

2.1 Hypothesis testing procedure . . . 16
2.2 Types of tests introduced . . . 16
2.3 Error types (conceptually) . . . 16
2.4 Error types (visually) . . . 16
2.5 Z-test 1 sample structure . . . 16
2.6 Z test acceptance and rejection region . . . 16
2.7 Pvalue concept and computation for Z tests . . . 17
2.8 Pvalue region visual representation . . . 17
2.9 Confidence interval Z test 1 sample . . . 18
2.10 Pvalue region visual representation . . . 18
2.11 Example of rejection criteria using confidence interval . . . 18
2.12 Scatter matrix for covariance (uncorrelated variables) . . . 18
2.13 2nd type error computation (use it for the power of the curve) . . . 18
2.14 Operating characteristic curve (OC CURVE) . . . 18
2.15 T test statistic . . . 19
2.16 Probability density function t distribution . . . 19
2.17 T test 1 sample structure . . . 19

2.18 T test acceptance and rejection region . . . 19
2.19 Pvalue concept and computation for T tests) . . . 19
2.20 Pvalue region visual representation) . . . 19
2.21 Confidence intervals for t student . . . 20
2.22 One-sided confidence interval for t student . . . 20
2.23 Two-sided confidence interval on t student . . . 20
2.24 Chi2 test statistic . . . 21
2.25 Probability density function Chi2 distribution . . . 21
2.26 Chi2 test structure . . . 22
2.27 Chi2 test acceptance and rejection region . . . 22
2.28 Chi2 critical values nomenclature and computation . . . 22
2.29 Chi2 nomenclature visually . . . 22
2.30 Chi2 pvalue . . . 22
2.31 Chi2 test lower bounded . . . 22
2.32 Chi2 test upper bounded . . . 22
2.33 Confidence intervals for variance with Chi2) . . . 24
2.34 Confidence intervals for standard deviation with Chi2) . . . 24
2.35 Critical values to be place in Chi2 confidence intervals) . . . 24
2.36 Z statistic 2 samples test . . . 24
2.37 Z test 2 samples structure . . . 24
2.38 Confidence interval Z-test 2 samples . . . 25
2.39 T statistic 2 samples test equal variances . . . 25
2.40 T test 2 samples structure . . . 25
2.41 Confidence interval T-test 2 samples equal variances . . . 25
2.42 T statistic 2 samples test equal variances . . . 26
2.43 T test 2 samples structure . . . 26
2.44 Confidence interval T-test 2 samples different variances . . . 26
2.45 T statistic 2 samples test equal variances . . . 26
2.46 T test 2 samples structure . . . 26
2.47 Confidence interval Paired T-test . . . 27
2.48 F distribution . . . 27
2.49 F distribution representation . . . 27
2.50 F test statistic for equality of variances . . . 27
2.51 F-test structure . . . 27
2.52 Steps to get the confidence interval . . . 28
2.53 Confidence interval F test on variances ratio . . . 28
2.54 2nd type error F-test . . . 29
2.55 Runs test structure) . . . 29
2.56 Runs test plot . . . 29
2.57 Barlett test for autocorrelation at a specific lag "k" . . . 31
2.58 Bonferroni inequality . . . 31
2.59 Barlett test stastic comparison with one or more lags (Bonferroni correction is applied) . . . 31
2.60 LBQ test . . . 32

3.1 Example on distributional model application) . . . 33
3.2 Example on linear model application . . . 33
3.3 Reminder (non random processes) . . . 33
3.4 Main types of non random processes . . . 33
3.5 Short recap on linear models) . . . 34
3.6 Output sm.OLS(y,x).fit() . . . 34

3.7 Test of hypothesis on the significance of a regressor) . . . 36
3.8 Test of hypothesis on the whole regression model significance . . . 36
3.9 Confidence interval on coefficients . . . 36
3.10 Confidence and prediction intervals definition) . . . 37
3.11 Confidence and prediction intervals on a linear model . . . 37
3.12 Confidence and prediction intervals on the next observation . . . 37
3.13 Time series fitting: scatter plot with confidence interval for next mean process outcome . . . 39
3.14 Time series fitting: scatter plot with confidence and prediction intervals for the next mean and single process outcome . . . 39
3.15 Original vs Step-wise regression model fits . . . 40
3.16 Example of ACF + PACF of a model solved by stepwise regession . . . 40
3.17 Step-wise regression approach . . . 40
3.18 Shifted DataFrame and preparation . . . 41
3.19 y: original observations series prepared for step-wise regression . . . 41
3.20 x: regressors DataFrame prepared for step-wise regression . . . 41
3.21 ARIMA models introduction . . . 43
3.22 Steps to fit an ARIMA: Box-Jenkis approach . . . 43

4.1 Principal component analysis purposes . . . 46
4.2 Variance-Covariance matrix . . . 46
4.3 Correlation matrix . . . 46
4.4 Variance-Covariance matrix . . . 47
4.5 Correlation matrix . . . 47
4.6 Orthogonal bases changes in PCA . . . 47
4.7 PCA vs regressions in case of p = 2 variables . . . 47
4.8 PCA steps . . . 48
4.9 Example of loading plot + comments . . . 51
4.10 Additional remarks on the loadings . . . 51
4.11 How to get the principal components . . . 52
4.12 How to reconstruct data based on k < P principal components . . . 52

5.1 SPC based on Shewhart's approach and assumptions . . . 54
5.2 Types of control charts based on Shewhart's approach . . . 54
5.3 Xbar-R control charts with known parameters . . . 54
5.4 Xbar-R control charts with unknown parameters . . . 55
5.5 Xbar chart: OC curve + ARLo + ARL + ATS . . . 57
5.6 R chart: OC curve (only shape) . . . 57
5.7 Xbar-S control chart with n = constant . . . 58
5.8 Xbar-S control charts with "n" variable . . . 60
5.9 X and I-MR control charts for individuals . . . 61
5.10 Table of factors for control charts in SPC . . . 62
5.11 Probabilistic control chart on Moving Range (half-normal approximation) . . . 63
5.12 Example of MR distribution (note the half normal) . . . 63
5.13 Probabilistic control chart outcome . . . 63

6.1 Trend control chart . . . 66
6.2 Trend control chart on non negative values . . . 67
6.3 Model-based control chart . . . 67
6.4 Fitted Value chart . . . 69
6.5 Special Cause control chart . . . 69
6.6 Model-based control chart . . . 71
6.7 Example of iid violation and overdispersion . . . 71

6.8 I-MR-R control charts for non iid process (n > 1) 72

7.1 Multivariate SPC 74
7.2 Chi2 control chart design for multivariate SPC (n > 1) 74
7.3 Hotelling control chart design for multivariate SPC (n > 1) 74
7.4 Chi2 control chart design for multivariate SPC (n = 1) 76
7.5 Example of Chi2 control chart plot for multivariate SPC (n = 1) 77
7.6 Hotelling's multivariate control chart for individuals (n = 1) with unknown parameters 78

List of Codes

1.1 List of libraries to be imported . . . 1
1.2 Print function and Numpy environment . . . 1
1.3 Usefull numpy function for DataFrame preparation: np.tile and np.repeat . . . 2
1.4 Pandas and DataFrame management . . . 3
1.5 Slicing example . . . 4
1.6 loc function example . . . 4
1.7 iloc example (1) . . . 4
1.8 iloc example (2) . . . 4
1.9 Pandas and DataFrame management . . . 5
1.10 Dataframe (time series) plot and random data generation from specific distributions . . . 6
1.11 "Fitted vs original data plot" and plotting of "n" time series within one graph (BATCHED PROCESS) . . . 6
1.12 Histograms code for both one-column or either multivariate dataset . . . 7
1.13 Q-Q plot code for both one-column or either multivariate dataset . . . 8
1.14 Scatter plot for (auto)correlation . . . 8
1.15 Scatter plot and data subset scatter plot . . . 9
1.16 BATCH PROCESS: Piled points scatter plot sorted by batches index . . . 10
1.17 BATCH PROCESS: scattered data sorted by the position index within the batches . . . 10
1.18 BATCH PROCESS: scattered data sorted by the position index within the batches . . . 11
1.19 Box plot . . . 11
1.20 How to create SUBPLOTS . . . 12
1.21 ACF and PACF PLOTS . . . 13
1.22 ACF and PACF functions' values stored in two vectors . . . 13
1.23 Box Cox tranformation . . . 14
1.24 Box Cox anti-tranformation . . . 14
1.25 Box Cox anti-tranformation . . . 15
1.26 Box Cox anti-tranformation . . . 15
2.1 Gussian plots and representations for hypothesis testing . . . 17
2.2 Power of the test and OC curve . . . 18
2.3 T-student distribution plot and representations for hypothesis testing . . . 19
2.4 Scatter plot for confidence intervals . . . 20
2.5 CODE TO AUTOMATICALLY RUN A 1 SAMPLE T-TEST . . . 21
2.6 Chi2 test code, CI code and rejection region representation . . . 23
2.7 CODE TO AUTOMATICALLY RUN A 2 SAMPLE T-TEST (BOTH WITH EQUAL OR NON EQUAL VARIANCES . . . 26
2.8 CODE TO AUTOMATICALLY RUN A PAIRED T-TEST . . . 27
2.9 F-TEST CODE + VISUAL REPRESENTATION . . . 27
2.10 F-test 2nd type error (and trial of OC curve) . . . 29
2.11 Runs test codes for both plot and test . . . 30
2.12 Code for automatic Runs Test . . . 30
2.13 Code for AUTOCORRELATION FUNCTION and BARLETT TEST . . . 31
2.14 Code for AUTOCORRELATION FUNCTION and LBQ test . . . 32
2.15 Code for AUTOMATIC LBQ test . . . 32
3.1 AUTOMATIC ORDINARY LEAST SQUARE LINEAR MODEL FITTING + RESIDUALS CHECK . . . 35
3.2 Confidence interval on a regressor coefficient . . . 36
3.3 Prediction next observation + confidence interval + prediction interval on it . . . 37
3.4 Automatic prediction next observation + CI + PI in case of more predictors models . . . 38
3.5 Automatic prediction next observation + CI + PI in case of ARIMA(1,1,0) . . . 38
3.6 How to get a vector of future prediction and residuals computation with new data . . . 38
3.7 CONFIDENCE AND PREDICTION INTERVAL plotted on a time series model . . . 39
3.8 Original dataframe preparation for stepwise regression . . . 41
3.9 Step-wise object creation and model fitting . . . 41
3.10 Model result storing and displaying . . . 42
3.11 Residuals plots and tests . . . 42
3.12 Next observation, confidence and prediction interval on step-wise output . . . 42
3.13 ARIMA in case of non stationarity . . . 44
3.14 ARIMA in case of non stationarity . . . 44
3.15 residuals check . . . 44
3.16 ARIMA original data vs fits . . . 44
4.1 Checks and computation before PCA (covariance matrix) . . . 48
4.2 PCA (covariance) fitting . . . 48
4.3 Scree plot + cumulative explained variance . . . 48
4.4 PCA (covariance) scores computation . . . 49
4.5 Checks post PCA: scores scatter and loadings plot . . . 49

4.6 Checks and computation before PCA (covariance matrix) . . . 49
4.7 PCA (correlation) fitting . . . 49
4.8 Scree plot + cumulative explained variance . . . 50
4.9 PCA (correlation) scores computation . . . 50
4.10 Checks post PCA: PCs scatter and loadings plot . . . 50
4.11 Data reconstruction with "k2 of the "p" PCs on python (covariance) . . . 53
4.12 Data reconstruction with "k2 of the "p" PCs on python (correlation) . . . 53
5.1 Xbar-R control charts manual computation . . . 56
5.2 Xbar-R control charts automatic computation . . . 57
5.3 How to get OOC's index in Xbar-R . . . 57
5.4 OC and ARL curves for Xbar control chart . . . 57
5.5 Xbar-S control charts manual computation . . . 58
5.6 Xbar-S control charts automatic computation . . . 59
5.7 DataFrame preparation in case of normality violation . . . 59
5.8 DataFrame preparation in case of normality violation . . . 60
5.9 I-MR control charts manual computation . . . 61
5.10 I-MR control charts automatic computation . . . 62
5.11 How to get OOC's index in I-MR . . . 62
5.12 Moving range probabilistic control chart . . . 63
5.13 Normality transformation on moving range statistic + I-chart on the outcome . . . 64
6.1 Trend control chart plot on non-negative values and flattening control limits . . . 66
6.2 Trend control chart plot on non-negative values and flattening control limits . . . 67
6.3 Special Cause control chart manual computation . . . 68
6.4 Special Cause control chart automatic computation . . . 68
6.5 Normality transformation on residuals' moving range statistic + I-chart on the outcome . . . 70
6.6 Normality transformation on residuals' moving range statistic + I-chart on the outcome . . . 72
7.1 Hotelling's control chart design . . . 75
7.2 Hotelling's control chart: check if a single sample is in-control . . . 76
7.3 Future observation UCL control limit . . . 76
7.4 Chi2 control chart design n = 1 . . . 77
7.5 Hotelling's control chart design n = 1 . . . 79
7.6 Hotelling's control chart design n = 1 (phase2) . . . 79
8.1 EWMA control chart (n > 1) . . . 82
8.2 Automatic EWMA control chart (n > 1) . . . 82
8.3 CUSUM control chart (n > 1) . . . 85
8.4 Automatic CUSUM control chart (n > 1) . . . 86
9.1 Shapiro Wilk test + Scatter plot + histogram + QQ plot + boxplot of each column of the dataframe (TO BE REVISED . . . 88
9.2 Outliers detection (NEED TO BE REVISED) . . . 88
9.3 SHAPIRO WILK TEST => BOX-COX => SHAPIRO WILK TEST => RETURN ERROR (NEED TO BE REVISED) . . . 88
9.4 Apply PCA (NEED TO BE REVISED) . . . 89
9.5 Dataframe standardization . . . 89

1 | Introduction

Before starting any exercise remember to import the libraries. More libraries or packages will be introduced in further sections when needed for specifically for the topic discussed and the tools applied.

```
# Import the necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
import seaborn as sns
import qda
```

Code 1.1: List of libraries to be imported

1.1. Python basic functions

1.1.1. Numpy enviroment, tools and print functions

These functions are embedded in Python and do not require importing libraries.

The **print()** function allows to print out a string or a combination of string and numbers. The printed strings can be also constructed using the appropriate syntax:

- integers %d
- float %f
- strings %s

Numpy is a powerful library for Python that is widely used for scientific and technical computing.

It provides a powerful array object, as well as a variety of functions for working with arrays, such as mathematical operations, and linear algebra.

The main advantages of using Numpy are its speed, convenience, and compatibility with other scientific libraries.

You can access individual elements of an array using brackets [ ].

Remind: elements are numbered starting from 0

We can use **np.arange(start, stop, step)** function to create an evenly spaced array of values within a given range:

- **start** is the first value of the array
- **stop** is the last value of the array **(the array will not include this value)**
- **step** is the difference between each value in the array (default is 1)

Or, if you are not interested in a specific step size and you just want to evenly cover a range with n values, you can use **np.linspace(start, stop, n)** function.

Note: np.arange allows you to define the step, so the amount of values composing the vector will be fixed by consequence. whereas np.linspace allows you to control the amount of values composing the vector, so the step will be fixed by consequence.

```
# Import the libraries
import numpy as np

name = 'Mark'
age = 23

# Creating a one-dimensional array
a = np.array([1, 2, 3, 4])
# Creating a two-dimensional array
b = np.array([[1, 2], [3, 4]])

#Creating evenly spaced values within a range by np.arange
c = np.arange(-5, 5, 1)      #CONTROL THE STEP
#Creating random values within a range by np.linspace
d = np.linspace(-5, 5, 11)  #CONTROL THE AMOUNT OF VALUES

#print strings or numbers (different options)
print('Hi, my name is', name, 'and I am', age, 'years old.')
print('Hi, my name is %s and I am %d years old.' % (name, age))

#print vectors or matrixes
print('a =', a) #vectors print syntax
print('b =', b) #matrix print syntax

#print rows/columns or specific values of vectors and matrixes
```

```
print('first element of a is: ', a[0])
print('element in position [0,1] of b is: ', b[0,1])
print('the first column of b is: ', b[:,0])
print('the second row of b is: ', b[1,:])
```

Code 1.2: Print function and Numpy environment

The `np.tile` function repeats an array a specified number of times.

Syntax:

`np.tile(A, reps)`

Parameters:

- `A`: The input array.
- `reps`: The number of repetitions of `A` along each axis.

Example:

`np.tile(np.arange(1, 4), 3)`

Output:

`array([1, 2, 3, 1, 2, 3, 1, 2, 3])`

The `np.repeat` function repeats elements of an array.

Syntax:

`np.repeat(a, repeats, axis=None)`

Parameters:

- `a`: The input array.
- `repeats`: The number of repetitions for each element.
- `axis` (optional): The axis along which to repeat values.

Example:

`np.repeat(np.arange(1, 4), 2)`

Output:

`array([1, 1, 2, 2, 3, 3])`

Example of code:

```
# the following is an example with batches of size 5, change them opportunely
# len(df)/num batches = num days
df_stack['Batch'] = np.tile(np.arange(1, 6), int(len(df_stack) / 5) + 1)[:len(df_stack)]

# len(df)/num batches = num days, but then + 1 is needed because np.arange leave out the last value!
df_stack['Day'] = np.repeat(np.arange(1, int(len(df_stack) / 5) + 1), 5)[:len(df_stack)]
```

Code 1.3: Usefull numpy function for DataFrame preparation: np.tile and np.repeat

Outcome:

	index	Diame...	Batch	Day
0	0	2.92	1	1
1	1	2.9	2	1
2	2	4.07	3	1
3	3	3.09	4	1
4	4	3.63	1	2
5	5	2.87	2	2
6	6	4.4	3	2
7	7	1.9	4	2
8	8	2.61	1	3
9	9	2.3	2	3
10	10	4.31	3	3
11	11	3.26	4	3

1.1.2. Pandas environment and Series/DataFrame management

Pandas is a powerful library for data manipulation and analysis in Python.

It provides data structures such as Series (1-dimensional) and DataFrame (2-dimensional) for storing and manipulating data.

This is the library we will use to import the CSV files.

To use Pandas, we first need to import it.

One of the most basic and important functions of Pandas is the **raed_csv()** function, which is used to load a CSV file into a DataFrame.

Another important function is the **head()** function, which is used to display the first "n" rows of a DataFrame. Therefore you can input a value within the parethesis in order to define how many rows should be displayed. If you do not input any value python displays as many as it can.

You can also **access and modify individual cells** in the DataFrame using the **loc [ ]** function.

You can combine loc [ ] with conditions in order to access specific values. This conditions can be equalities/inequalities with numbers or strings.

You can use the built-in functions to find the index of the maximum value in a column and use it as a condition.

THE DATAFRAME OF THE EXAMPLE WILL BE CALLED "df", SO WHENEVER YOU ENTOUNTER "df" REMEMBER IT SHOULD BE SUBSTITUTED BY THE NAME OF YOUR DATAFRAME !!!

```
import pandas as pd

# Loading a CSV file into a DataFrame
df = pd.read_csv('filename.csv')

# Alternatevely you can make it slightly parametric in this way
#You can specify delimiters and decimal settings as well
file_name = 'image_statistics.csv'
data = pd.read_csv(file_name, delimiter=',', decimal='.')

# Display the first "3" rows of a DataFrame or let python defining how many
df.head(3)
df.head()

# Create the copy of a dataframe, assume df_new is the name of the new dataset
df_new = df.copy()

# Define a new DataFrame with its columns names and then fill it.
df = pd.DataFrame(columns=['Colonna1', 'Colonna2', 'Colonna3'])
df['Colonna1'] = #define values in column 1
df['Colonna2'] = #define values in column 2 and so on

# Change the value of a cell
df.loc[row,'column name'] = new_value
#Change the value of the cells which meet the condition; value can be a number of a string
#Recall that if value is a string you must include it within the quotation marks: 'text'
df.loc[df['column name'] == value, 'column name'] = new value       #value can even be df.['column name'].max() or .min()
# Find the row with the maximum value in a column and change the value
df.loc[df['column name'].idxmax(), 'column name'] = new value
```

Code 1.4: Pandas and DataFrame management

Python provides several data structures to handle different types of data, such as arrays, DataFrames, lists, and Series. Each of these data structures has unique characteristics and is used in specific contexts. Below is a brief overview:

- **Arrays**: Arrays are a collection of items stored at contiguous memory locations. The major advantage of using an array is that it allows random access and has a fixed size.
- **DataFrames**: DataFrames are a part of the pandas library and are used for handling and manipulating tabular data. They consist of rows and columns, similar to a table in a database.
- **Lists**: Lists are a built-in Python data structure that can hold an ordered collection of items, which can be of different types. Lists are mutable and dynamic in size.
- **Series**: Series are one-dimensional arrays in pandas that can hold any data type. They are similar to columns in a DataFrame.

Slicing and Indexing

Slicing is a technique used to select a subset of elements from an array, list, or Series. Here is an example of how to use slicing with a pandas DataFrame.

The command `x = data['lag1'][1:]` in Python utilizes the slicing functionality of arrays and lists to select a specific part of the data within a column of the pandas DataFrame.

Firstly, `data['lag1']` extracts the column named "lag1" from the DataFrame `data`. The result is a pandas Series, which is similar to a one-dimensional array. The `[1:]` part is an example of slicing. In this context, it instructs Python to take all the elements from the Series starting from index 1 to the end. In practice, this operation skips the first element of the Series (the element at index 0) and includes everything else.

For instance, if the "lag1" column contains values `[10, 20, 30, 40, 50]`, the command `data['lag1'][1:]` would return `[20, 30, 40, 50]`. This demonstrates how slicing can be used to manipulate and access specific parts of data within a pandas DataFrame column.

```
import pandas as pd

# Example DataFrame
data = pd.DataFrame({
    'lag1': [10, 20, 30, 40, 50]
})

# Selecting a specific part of the data within the column
x = data['lag1'][1:]
print(x)
```

Code 1.5: Slicing example

Using `loc` and `iloc`

The pandas library provides two powerful indexing functions: `loc` and `iloc`.

NOTE: loc and iloc input values should be place within brackets **`loc`** The `loc` function is label-based. You can access a group of rows and columns by labels or a boolean array.

syntax to loc of one value: df.loc[row_number, 'column name']

syntax to loc of a range of values: df.loc[a:b, ['column name 1', 'column name 2', ... , 'last column you need']]

For example:

```
import pandas as pd

# Example DataFrame
data = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': [9, 10, 11, 12]
})

# Selecting rows by labels
print(data.loc[0:2, ['A', 'B']])
```

Code 1.6: loc function example

This will print:

```
   A  B
0  1  5
1  2  6
2  3  7
```

The `iloc` function is integer position-based. You can access a group of rows and columns by their integer positions.

THE SYNTAX STRUCTURE IS THE SAME AS ".loc[]" BUT NO COLUMN NAME HAS TO BE INPUT SINCE IT IS INTEGER POSITION-BASED. THEREFORE USE THE INDEXES TO REFERE TO A SINGLE/RANGE OF ROW/S AND COLUMN/S

Short example: this two formulations are exactly equal "last_obs1 = df.loc[34, 'column name']" ; "last_obs2 = df.iloc[34, colun index]"

For example:

```
import pandas as pd

# Example DataFrame
data = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': [9, 10, 11, 12]
})

# Selecting rows by integer position
print(data.iloc[0:2, 0:2])
```

Code 1.7: iloc example (1)

This will print:

```
   A  B
0  1  5
1  2  6
```

In summary, while `loc` is used for label-based indexing and can handle boolean arrays, `iloc` is used for positional indexing. Both are essential for efficient data manipulation in pandas.

There is other useful way to apply "iloc". Suppose for example you want to pick the last value of a specific column within a dataframe. It might be very crucial when working with autoregressive models and you want to predict the next process outcome, for example. So in that case you would need to access the last observation value and input it in the model equation.

```
# how to pic the last observation from a dataframe
last_obs = df['column name'].iloc[-1]
```

```
last_obs = df
```

Code 1.8: iloc example (2)

1.1.3. List of basic needed commands

- **.head()** is usually applied to display a portion of the dataframe
- **.info()** provides you information on thenumber of rows for each column and tells you if there are null values. Use at the beginning to understand if the size of the provided data are consistent in the imported csv, or whether there are "nan" values
- **.dropna()** allows you to neglet "nan" values. It is useful in the case previously descripted. Suggestion: if it make sense, split the csv in different dataframe, drops "nan" values e go ahead with the analysis
- **.drop()** helps you to get rid of an observation. You can state index of the row within the parenthesis
- **.idxmax() , .idxmin()** are used to find the index of the max or a min value within a dataframe
- **p**d.concat([df1, df2], ignore_index = true) allows you to joint two dataframe one after the other

```
#Display the firsts "n" rows of your dataframe df
df.head(insert a number) #visual code is able to display max the first 10 observations

#Get info on your data about #rows
df.info()

# DESCRIBE THE MAIN DESPRIPTIVE STATISTICS (number of observations, mean, std, min, max and quartiles) of a dataframe called df
print(df['supp1'].describe())

# Erase from your data the "nan" (not a number) values
# USE .dropna()
df1 = df['insert column name'].dropna()
plt.boxblot(df['supp1'].dropna())

# Make a copy of your dataset and get rid of specific row
df_correct = df.drop(index=29)

# Make a copy of your dataset and get rid of an outlier which is the maximum or the minimum value between the originals
df_correct = df[df['column name'] != df['column name'].max()]
df_correct = df[df['column name'] != df['column name'].min()]

# Identify the INDEX of the max or the min of a sample
max_index = df['column name'].idxmax() #min_index = df['column name'].idxmin()
# Now max_index is an index, so you can insert and use this variable when, for instance, you have to insert an index within a function

# Create a new column in your datafrate which contains the shifted version of your dataset
df['lag1'] = df['column name'].shift(1)       #this is just an example of lag1 shifting, you can create it for the shift you need

# Concatenate two DataFrames
df = pd.concat([df1, df2], ignore_index=True)
```

Code 1.9: Pandas and DataFrame management

1.1.4. Graphical representation: matplotlib to support stastical analysis

Matplotlib is a powerful library for creating static, animated, and interactive data visualizations in Python. To use Matplotlib, we first need to import it.

Below you will find codes for creating various types of charts and representations. The initial codes are intended to depict lines (distributions or time series).

MAKE SURE YOU HAVE IMPORTED THE NECESSARY LIBRARIES BEFORE USING THEM!!

The first blocks of code are used to visually display the results of hypothesis tests. In any case, it is possible to extract portions of these codes and use them for different purposes. CAUTION: it is not possible to copy and paste without first ensuring that you have correctly defined (with the same name) the variables:

- alpha
- test statistic
- degrees of freedom (if present)
- critical values that define the rejection region

Tailor and adapt your codes as best as you can.

Basic Plot Types

- **Line Plots**: Useful for showing trends, level shift, sationary or nonstationary meandering. oscillating and increasing variation processes over time. Created with **plt.plot()**.

  Line plots are very useful to create or display time series. To be more general let's say that they are appropriate when your objective is plotting the sequence of data that you have in your dataset or a set of data created from a specific distribution .

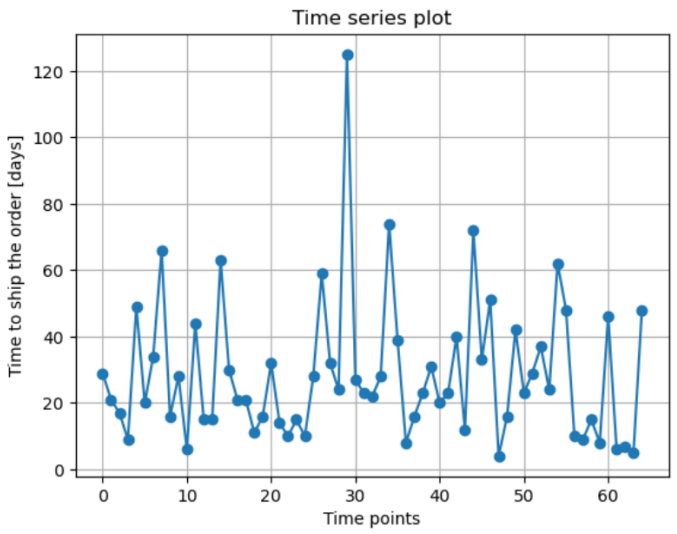


Figure 1.1: Time series plot from a pandas dataframe

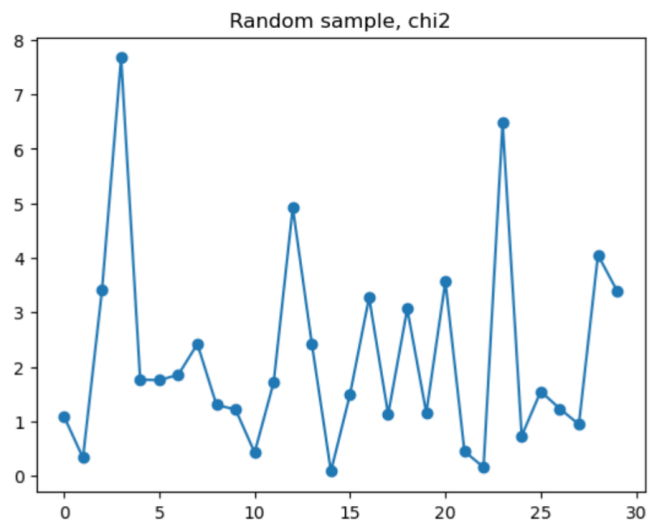


Figure 1.2: example of Chi2 random generated dataset

```
# plotting one sequence of data from dataframe called "df"
plt.plot(df['column name'], 'o-')
plt.xlabel('insert x label')
plt.ylabel('insert y label')
plt.title('insert title')
plt.grid()
plt.show()

# create and plot a random sequence of data from specific distributions
# Normally distributed data with given mean and std
mean = #define the mean
std = #define the std
n = #define the number of data generated
data_rand_norm = np.random.normal(loc=mean, scale=std, size=n) plt.plot(data_rand_norm, 'o-')
plt.title('Random sample, normal') plt.show()
plt.show()

# Chi2 distributed data
dof = #define the number of degrees of freedom
n = #define the number of data generated
data_rand_chi2 = np.random.chisquare(df=dof, size=n)
plt.plot(data_rand_chi2, 'o-')
plt.title('Random sample, chi2')
plt.show()

# T standard distributed random data
data_rand_t= np.random.standard_t(df=2, size=n)
plt.plot(data_rand_t, 'o-')
plt.title('Random sample, t-student')
plt.show()
```

Code 1.10: Dataframe (time series) plot and random data generation from specific distributions

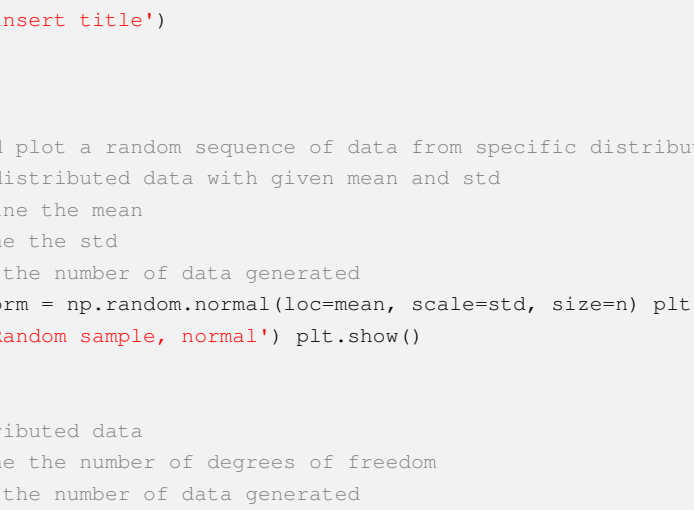


Figure 1.3: Time series original vs fitted data

Figure 1.4: Time series of "n" batches overlapped on one single plot

```
# Plotting two sequence of data
# Use this block of code when you compare original vs fitted data
plt.plot(df['column name'], 'o-', label='Original data')
plt.xlabel('x label') #e.g.: index
plt.ylabel('column name') #column of original data
plt.plot(model.fittedvalues, 's--', color='red', label='Fitted values', alpha=0.5)
plt.legend()
```

```
plt.grid()
plt.show()

# Plot the data time series as "n" separate batches
# Batch defininition and preparation must be carried out before (go to the batched process section to see how)
# We need to extract the data for each batch; change the color, the line, and the marker; assign a label for each batch
plt.plot(df['Diameter'][df['Batch'] == 1], 'o:b', label = 'Batch 1')
plt.plot(df['Diameter'][df['Batch'] == 2], 's:r', label = 'Batch 2')
plt.plot(df['Diameter'][df['Batch'] == 3], 'D:g', label = 'Batch 3')
plt.plot(df['Diameter'][df['Batch'] == 4], '^:m', label = 'Batch 4')
#remember to change 'Diameter' with the right column name of your dataframe df!!!
plt.xlabel('Index')
plt.ylabel('Diameter')
plt.legend()
plt.title('Time series plot of Diameter')
plt.grid()
plt.show()
```

Code 1.11: "Fitted vs original data plot" and plotting of "n" time series within one graph (BATCHED PROCESS)

- **Histograms**: Ideal for visualizing the distribution of numerical data.

  KDE stands for Kernel Density Estimate.

  KDE is a method for estimating the probability density function of a continuous variable.

  With **kde=True**, sns.histplot not only draws a histogram but also overlays the KDE curve, which smooths the data and shows a continuous distribution.

  This is useful for seeing the "shape" of the data distribution in addition to the bin frequency of the histogram.

  You can tune your histogram as **sns.histplot(df[col_name], kde=True, bins=num_bins)**.

  By deciding the number of bins you might achieve better representations.

  In general **the appropriate number of bins is given by the square root of number of the number of observation**, which correspond to the length of the dataframe.

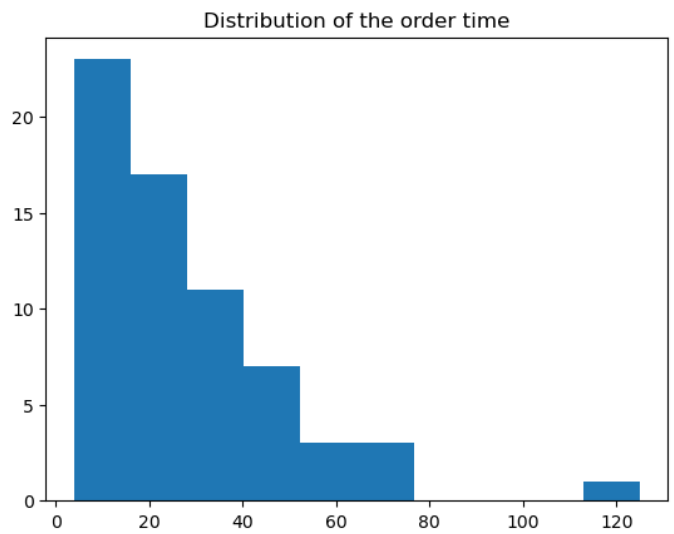


Figure 1.5: Simplest type of histogram plot

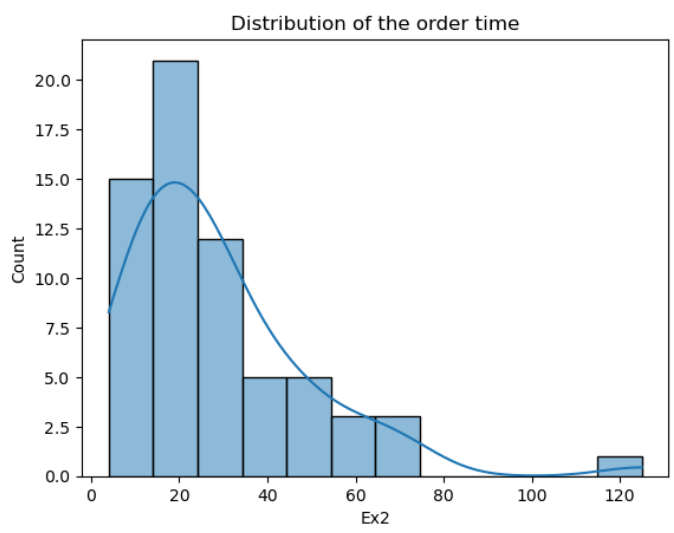


Figure 1.6: Histograms plot with KDE

```
# Plot the simplest type of histogram
plt.hist(df)
plt.title('Histogram')
plt.show()

# Plot KDE (Kernel density estimation) on histograms through seaborn
# Import the necessary library
Import seaborn as sns
# Plot the histogram with KDE
sns.histplot(df['col_name'], kde=True)
# Set the title
plt.title('Distribution of the data')   # + col_name reference a string in the title
# Show the graph
plt.show()

#Keep in mind the way to tune your histograms:
num_bins = int(np.sqrt(len(df)))    #otherwise input a value
sns.histplot(data_part[col], kde=True, bins=num_bins)
```

Code 1.12: Histograms code for both one-column or either multivariate dataset

- **Q-Q plot or probability plot**

  QQ plots (also known as probabibility plot) is a graphical representation to check how much your dataset tend to be normal: **the closer to the straight line, the more your dataset is normally distributed**

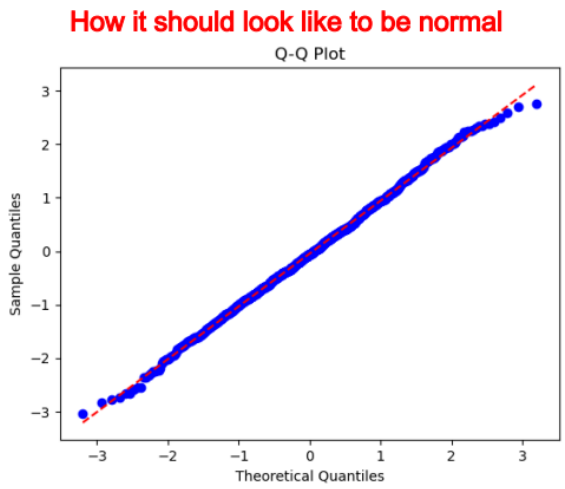


Figure 1.7: QQ plot in case of normally distributed data

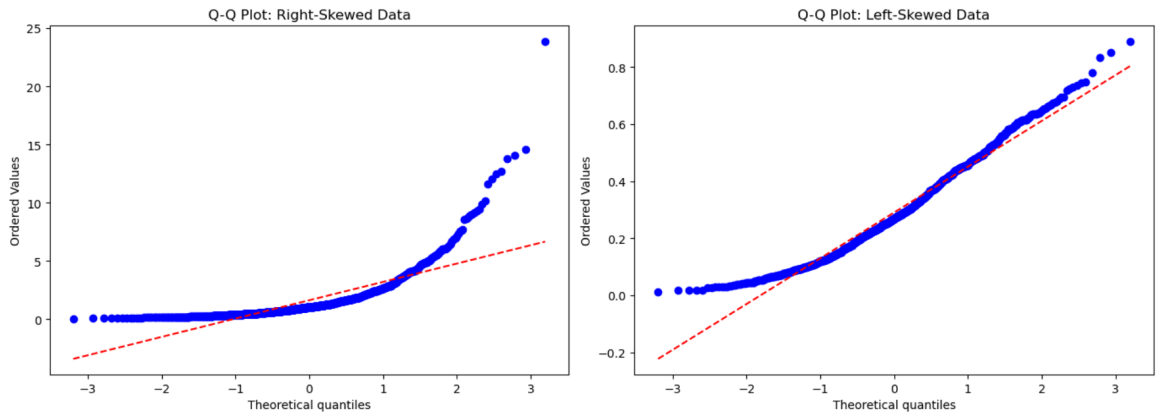


Figure 1.8: QQ plot in case of skewed distributions

```
# Plot the qqplot for a single column dataframe called "df"
stats.probplot(df['column_name'], dist="norm", plot=plt) #alternatively you may input the index of the column instead of the name
plt.show()

# Add Shapiro Wilk test result to speed up your analysis
_,pvalue_SW = stats.shapiro(df['column name'])
print('pvalue of the Shapiro Wilk test is: ', pvalue_SW)

# Plot the QQplot for a multivariate dataframe called "df"
for col in df.columns[first column index:last column index]:
    stats.probplot(df[col], dist="norm", plot=plt)
    plt.title(f'QQ plot for {col}')
    plt.show()

# Plot the QQplot for a multivariate dataframe called "df" if data are numerical. This version is more parametric and elegant
for column in df.columns:
    # Check if the data under the column are numerical
    if df[column].dtype.kind in 'bifc':  # b=bool, i=int, f=float, c=complex
        # Crea un qq plot per la colonna corrente
        stats.probplot(df[column], dist="norm", plot=plt)
        plt.title(f'QQ plot for {column}')
        plt.show()
```

Code 1.13: Q-Q plot code for both one-column or either multivariate dataset

- **Scatter plots**: there are many application for scatter plots. We will go through each of them but always take care to pick up the right code.

  The first application of scatter plots is to see the correlation between two variables or autocorrelation

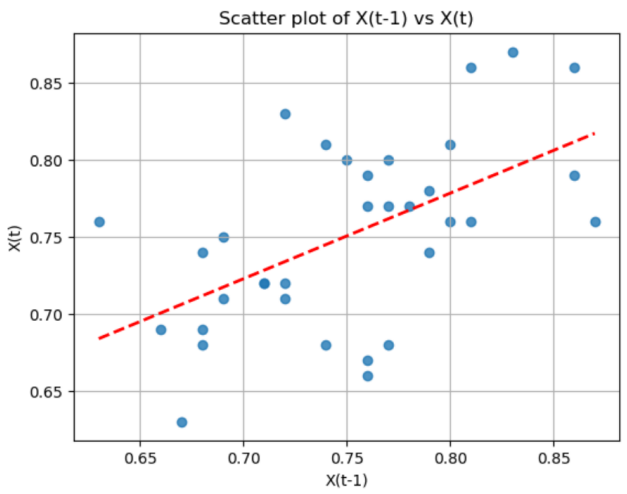


Figure 1.9: Scatter plot to check on autocorrelation (+ trend line)

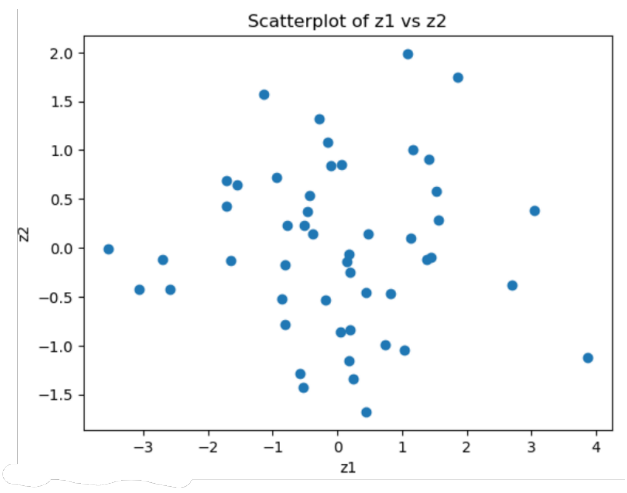


Figure 1.10: Scatter plot to check on correlation between two variables

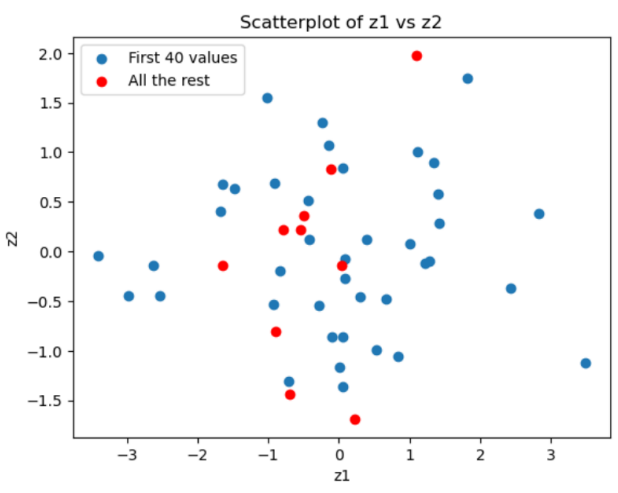


Figure 1.11: PCA: Scatter plot PCs correlation (+ cross validation)

This plot is one of our tools to show the autocorrelation of a variable: **if the scattered points drop close to the straight line it means there is autocorrelation. If the line is positively sloped, then autocorrelation is positive. It is negative otherwise.**

```
# Create scatterplot with regression line using seaborn (remember to import it)
import seaborn as sns
# If the scatter plot scope is showing autocorrelation you have to prepare your dataset first
# df['lag1'] = df['column name'].shift(1)       #this is just an example of lag1 shifting, you can create it for the shift you need
sns.regplot(x='column name', y='column name', data = df, fit_reg=True, ci=None, line_kws={'color': 'red', 'lw': 2, 'ls': '--'})
#here you find an example for autocorrelation at lag1 but always input the right name you've chosen for the columns
#sns.regplot(x='lag1', y='Ex4', data=data, fit_reg=True, ci=None, line_kws={'color': 'red', 'lw': 2, 'ls': '--'})

# When the scatter is ready, tailor your graph:
plt.title('Scatter plot of X(t-1) vs X(t)')
plt.xlabel('X(t-1)')
```

```
plt.ylabel('X(t)')
plt.title('Scatter plot of X(t-1) vs X(t)')
plt.grid()
```

Code 1.14: Scatter plot for (auto)correlation

```
# This is the simplest scatter plot you might think of; assume df is your dataframe
plt.plot(df['column name'], df['column name'], 'o', color='blue', label='Original data')
plt.title('Scatter plot of X(t) vs X(t-1)')
plt.xlabel('X(t-1)')
plt.ylabel('X(t)')
plt.show()

# What if I have to plot a subset of my dataframe??
# assume you want the scatter from row "a" to row "b" of your dataset; you already input column name so do not specify the column!!
plt.plot(df['column name'][a:b], df['column name'][a:b], 'o', color='blue', label='Original data')

# Now suppose you are doing cross validation (on PCA for example); the code is on PCA, tailor it for your problem
# Suppose you divided the original dataset in training and test set and you have trained your the model
# Then you want to see the scatter of the output (of the model) divided in training and test set, differentiated by the colors
# Z1 vs Z2 scatter plot WITH CROSS VALIDATION; scores_df is the dataframe in which we saved the scores
plt.scatter(scores_df['z1'][:40], scores_df['z2'][:40], label='First 40 values')
# Add the rest of the values to the scatterplot with a different color
plt.scatter(scores_df['z1'][40:], scores_df['z2'][40:], color='r', label='All the rest')
plt.xlabel('z1')
plt.ylabel('z2')
plt.title('Scatterplot of z1 vs z2')
plt.legend()
plt.show()
```

Code 1.15: Scatter plot and data subset scatter plot

Now suppose that you're process is divided in batches. Imagine that your data are stacked (piled) in just one column but in reality they comes from a sequence of batches.

	Diameter	
0	2.92	batch 1
1	2.90	batch 2
2	4.07	batch 3
3	3.09	batch 4

Figure 1.12: Batch process stacked data

	BATCH1	BATCH2	BATCH3	BATCH4
Day1	2.92	2.9	4.07	3.09
Day2	3.63	2.87	4.4	1.9
Day3	2.61	2.3	4.31	3.26
Day4	3.76	3.9	4.44	2.84
Day5	3.41	3.25	4.88	3.39
Day6	2.26	3.27	4.45	2.62
Day7	2.65	3.64	4.1	2.38

Figure 1.13: Batch process unstacked data

The CSV dataframe might not be already prepared to be used. It is likely that you start from a stacked bunch of data collected in sequence from different batches.

Then it is up to you to decide how to prepare the dataset for the analysis: you might add a new column called "batch" in which you carefully stack the index of the lot (conceptually as in the left picture), or you can decide to transpose one every "n" data where "n" is the number of batches. This second way allows you to create a column per each data

CHECK THE PARAGRAPH ON BATCHED PROCESS to know how to prepare the dataset, here we propose just an example. More detailed explanation are given in this file.

Well, imagine that you want a **scatter plot where on the X axis you put the batch, while on the Y axis you plot the values assumed by the parts beloning to that specific batch.**

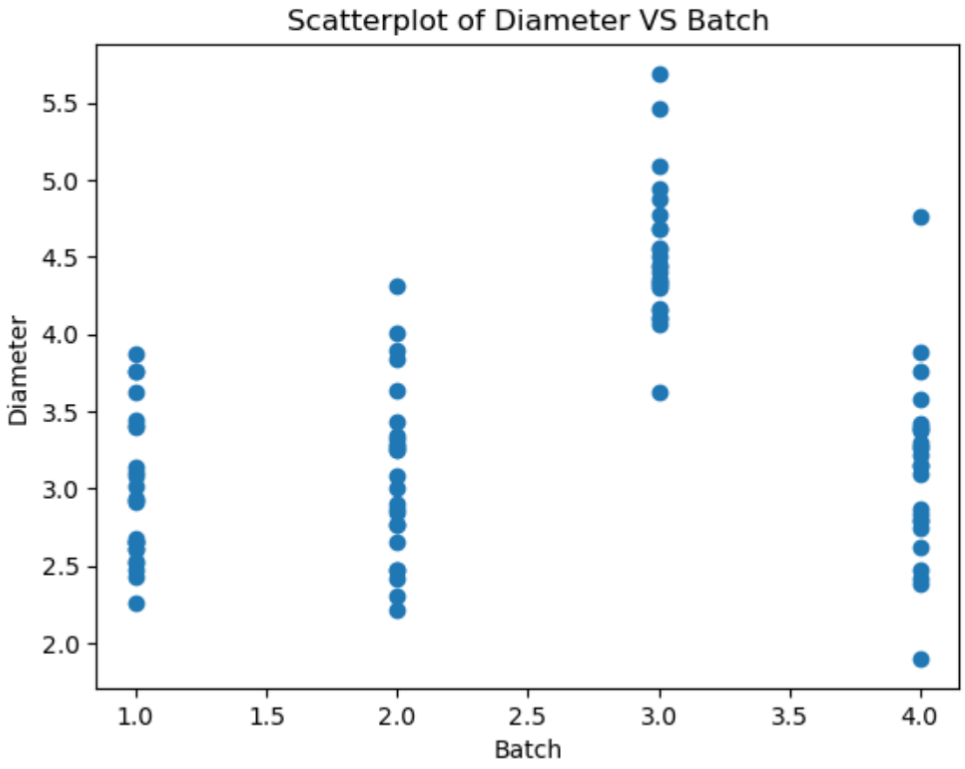


Figure 1.14: BATCH PROCESS: Piled points scatter plot sorted by batches index

Here there is the code to do that:

```
# first of all you must create a new column your in dataframe to create a link between the batches and their belonging observations
# from now on imagine to call this new column as 'Batch' and the dataframe 'df' as usual
#Diameter was the column name of the original data, just take it as an example and change it with the right name
plt.scatter(df['Batch'], df['column name']) # insert the column name of your original data
plt.xlabel('Batch')
plt.ylabel('Diameter')
plt.title('Scatterplot of Diameter VS Batch')
plt.show()
```

Code 1.16: BATCH PROCESS: Piled points scatter plot sorted by batches index

Scatter plots are very useful for univariate dataset with n $>$ 1 columns so that: **the dataset represents one quality feature; each rows represent a sampled batch; each column represent the position of the specific object within the batches**.

This situation is very typical for data which will be exploited to design control charts (n $>$ 1): this is not properly a scatter, indeed we won't use plt.scatter, but it is simply a plot where obs1, obs2, ... obs "n" within a batch are displayed for each of the "m" batches sampled.

To better explain: on the X axis you have the index of the batch, so the domain will be discrete from 0 to "m - 1". For each index "n" points are represented (pile made of "n" points) where, as mentioned, "n" is the size of the batch.

This "n" points has their own color in order to differentiate the position of the observation within a batch.

If you still don't get it, imagine you sample once out of 6 hours (4 points/day) and you do it for 25 days. So you have 25 batches made of 4 observations.

	x1	x2	x3	x4	x5
0	0.473	0.405	0.213	3.187	0.572
1	0.430	2.623	1.415	0.915	2.933
2	0.148	1.938	1.057	2.019	1.256
3	5.209	0.211	1.047	0.492	0.388
4	0.308	0.536	0.570	2.951	1.741

Figure 1.15: Typical dataframe for CC with n $>$ 1

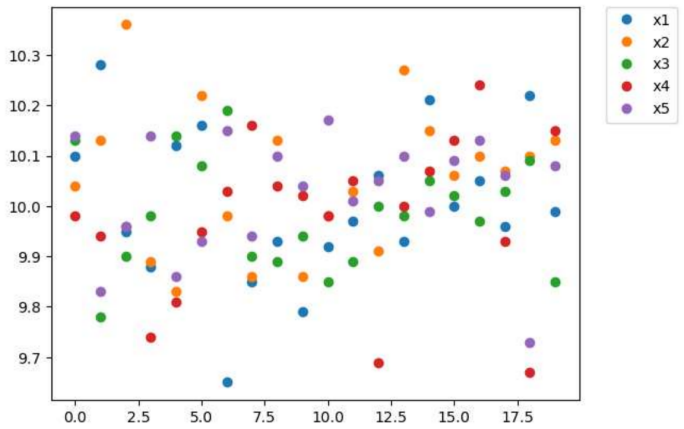


Figure 1.16: BATCH PROCESS: scattered data sorted by the position index within the batches

```
# Make a scatter plot of all the columns against the index of the rows
plt.plot(df['x1'], linestyle='none', marker='o', label = 'x1')
plt.plot(df['x2'], linestyle='none', marker='o', label = 'x2')
# repeat the code ...
plt.plot(df['last column name'], linestyle='none', marker='o', label = 'x5')
# place the legend outside the plot
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```

Code 1.17: BATCH PROCESS: scattered data sorted by the position index within the batches

One more application of scatter plot is the **Scatter Matrix** Scatter matrix allow to check the covariance in multivariate dataset. The following pictures shows one case of uncorrelated variables (PCs of a principal components analysis) and another one in which we clearly see the correlation among variables.

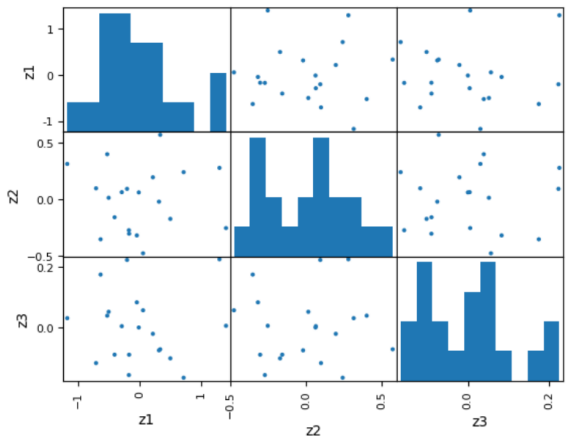


Figure 1.17: Scatter matrix for covariance (uncorrelated variables)

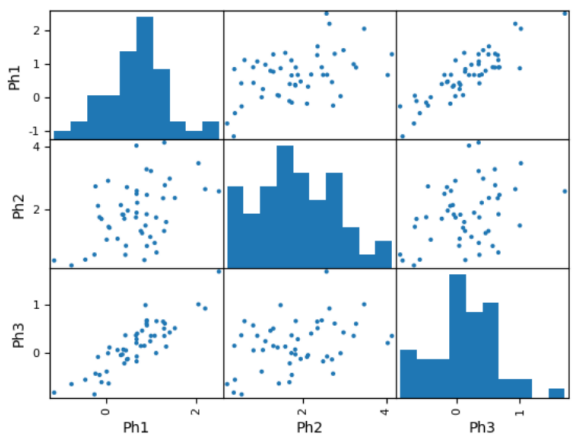


Figure 1.18: Scatter matrix for covariance (correlated variables)

```
# Create a correlation plot of the dataset "df"
# change the name of the dataset (e.g: scores_df) if needed
pd.plotting.scatter_matrix(df, alpha = 1) #leave alpha as it is
plt.show()
```

Code 1.18: BATCH PROCESS: scattered data sorted by the position index within the batches

- **Box Plots**: Box plots, also known as box-and-whisker plots, are a type of graphical display for describing the distribution of a dataset based on a five-number summary: minimum, first quartile (Q1), median (second quartile, Q2), third quartile (Q3), and maximum.

  First quartile: This marks the 25th percentile of the data set. It is the middle value between the smallest number and the median of the dataset. It represents where one-quarter of the data falls below this point.

  Median (Second Quartile, Q2): The median is the middle value of the dataset and is shown by a line across the box. If the number of observations is odd, it's the middle number; if even, it is the average of the two middle numbers. The median divides the dataset into two halves.

  Third Quartile (Q3): The 75th percentile, this is the middle value between the median and the highest value of the dataset. Three-quarters of the data falls below this point.

  Whiskers: These lines extend from the quartiles to the minimum or maximum values, giving a sense of the range of the data. The length of the whiskers can vary depending on the method used but often extends to 1.5 times the interquartile range (IQR, which is Q3 - Q1) from the quartiles. Data points beyond this are considered outliers and often depicted as individual points.

  Outliers: These are data points that lie outside the range defined by the whiskers. They are usually considered unusual points and are plotted as individual dots.

  Box plots are particularly useful for indicating whether a distribution is skewed and whether there are potential unusual observations (outliers) in the data set.

  Boxplots cannot be plotted untile you drop the data with the function .dropna() applied as in the code

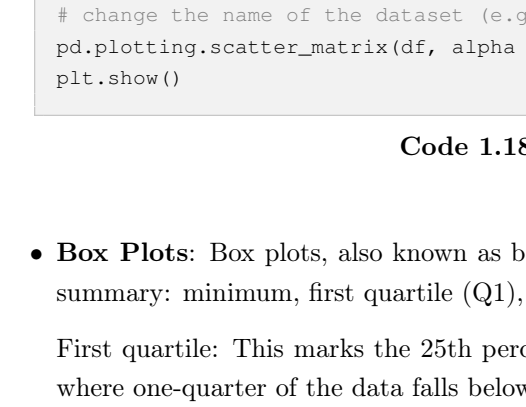


Figure 1.19: 2 Boxplots in 1 plot

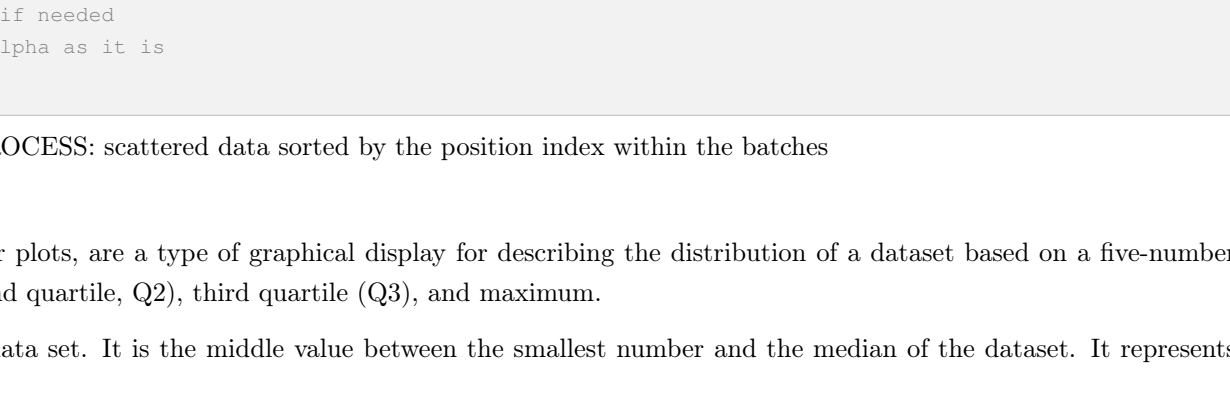


Figure 1.20: 2 Boxplots in 2 subplots

```
# Show one boxplot
plt.boxplot(df['column name'].dropna())
plt.title('Boxplot insert title')
plt.show()

# Show two boxplot in one plot
plt.boxplot([df['column name'].dropna(), df['column name'].dropna()], labels=['title 1', 'title 2'])
plt.show()

# Show two boxplot in two subplots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))
ax[0].boxplot(df['column name'].dropna())
ax[0].set_title('title 1')
ax[1].boxplot(df['column name'].dropna())
```

```
ax[1].set_title('title 2')
plt.tight_layout()
plt.show()
```

Code 1.19: Box plot

- **How to create subplots and examples** Subplots in Python are a great way to create multiple plots within a single figure. This is especially useful for comparing multiple datasets or visualizing different aspects of the same dataset.

```
# This is for a row of images
fig, ax = plt.subplots(nrows= 1, ncols= insert it, figsize = (15, 5)) #you might set the size...
ax[0].plot(define your plot)
ax[0].set_title('insert title')
ax[1].plot(define your plot)
ax[1].set_title('insert title')
...
ax[insert].plot(define your last plot)
ax[insert].set_title('insert title')
plt.tight_layout() #... or you might avoid overlapping of graphs
plt.show()

# This is for a matrix of images
fig, ax = plt.subplots(nrows= insert it, ncols= insert it, figsize = (15, 5)) #you might set the size...
ax[0,0].plot(define your plot) #indice riga e poi indice colonna
ax[0,0].set_title('insert title')
ax[0,1].plot(define your plot)
ax[0,1].set_title('insert title')
...
ax[insert, insert].plot(define your last plot)
ax[insert, insert].set_title('insert title')
plt.tight_layout() #... or you might avoid overlapping of graphs
plt.show()
```

Code 1.20: How to create SUBPLOTS

- **Autocorrelation measures the linear relationship between an observation and its previous observations at different lags**.

  Partial Autocorrelation measures the direct linear relationship between an observation and its previous observations at a specific lag, excluding the contributions from intermediate lags.

  It answers the question of how much of the correlation between observations at "lag k" is due to the direct relationship between them, and not due to the correlations at shorter lags.

  EXAMPLE:

  Suppose in the ACF plot, there are two significant spikes (violations) at lag 1 and lag 2, both outside the 95% confidence interval. This indicates that there is a significant correlation at both lag 1 and lag 2. Suppose in the PACF plot, there is only one significant spike at lag 1, with no significant spike at lag 2.

  Observation on ACF: The significant correlation at lag 2 in the ACF is likely due to the persistence of the strong autocorrelation at lag 1. This means that the correlation at lag 2 is, at least in part, an indirect effect of the strong autocorrelation at lag 1.

  Observation on PACF: The significant correlation only at lag 1 in the PACF indicates that the true linear dependency is primarily between an observation and its immediate previous observation (lag 1). The lack of a significant spike at lag 2 in the PACF suggests that there is no direct correlation between the observations at lag 2 once the influence of the lag 1 correlation is accounted for.

  To sum up the result: The PACF removes the indirect effects of the intermediate lags when calculating the correlation for a specific lag. Therefore, if the PACF shows significance only at lag 1, it implies that the correlation seen at lag 2 in the ACF is due to the propagation of the lag 1 correlation and not a direct linear dependency.

  This is typical of autoregressive processes of order 1 or AR(1)

  The image can better explain this concepts. Even though ACF at lag2 does not look that significant, this was an AR(1) and the graphs perfectly fit what we have been talking about in the example. By looking to the functions it's clear that they agree each other on autocorrelation at lag1, but you see that at lag2 there is no autocorrelation for the PACF while the the ACF has a value which is pretty close to the bounderies of the confidence interval.

  Check the chapter on autoregressive models to deep dive in this concepts

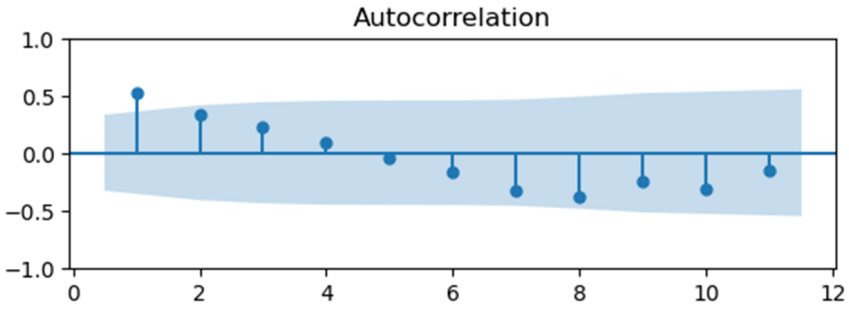


Figure 1.21: Autocorrelation Function

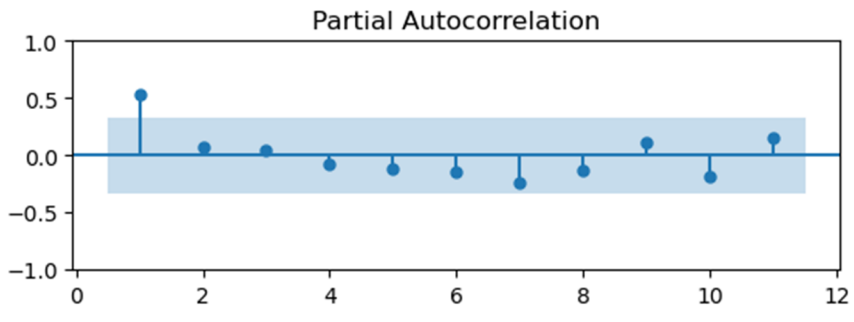


Figure 1.22: Partial Autocorrelation Function

```
#ACF and PACF
# Plot the acf and pacf using the statsmodels library
import statsmodels.graphics.tsaplots as sgt

fig, ax = plt.subplots(2, 1)
sgt.plot_acf(df['column name'], lags = int(len(df['column name'])/3), zero=False, ax=ax[0])
fig.subplots_adjust(hspace=0.5)
sgt.plot_pacf(df['column name'], lags = int(len(df['column name'])/3), zero=False, ax=ax[1], method = 'ywm')
plt.show()
```

Code 1.21: ACF and PACF PLOTS

Here we provide the values for this two functions stored respectively in arrays called acf_values and pacf_value

```
# import the functions from the libraries
from statsmodels.tsa.stattools import acf, pacf

# autocorrelation function (nlags = int(np.sqrt(n)))
#[acf_values, lbq, _] = acf(df['column name'], nlags = int(np.sqrt(n)), qstat=True, fft = False)

# autocorrelation function (nlags = int(n/3))
[acf_values, lbq, _] = acf(df['column name'], nlags = int(n/3) , qstat=True, fft = False)

# partial autocorrelation function
pacf_value = pacf(df['columna name'], nlags = int(len(data)/3))
```

Code 1.22: ACF and PACF functions' values stored in two vectors

1.2. Data modeling and preparation: Box Cox normality transformation + gapping algorithm + batching algorithm

1.2.1. Box Cox transformation

Box Cox transformation is useful when we need to transform data to normal data (normally distributed). Box Cox is effective when all the data are strictly higher than zero, so INSPECT YOUR DATASET BEFORE ITS APPLICATION. Here we provide different codes to approach this transformation in different ways.

Remind:

$$y = (x^{\lambda} - 1)/\lambda, \text{ for } \lambda \neq 0$$

$$y = ln(x), \text{ for } \lambda = 0$$

Figure 1.23: Box Cox transformation

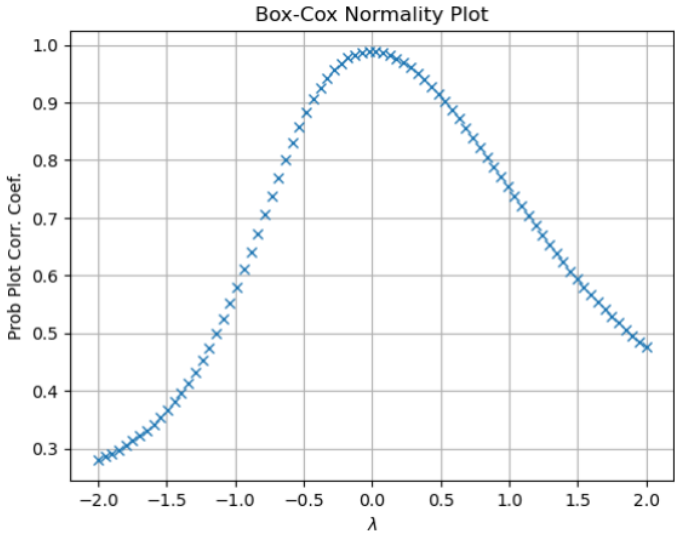


Figure 1.24: Box Cox normality plot

- The first code automatically applies Box Cox, runs a Shapiro Wilk test on the output dataframe and creates a QQ plot
- The second code can be used when Lambda is very close to zero. We can inspect the so called "Box-Cox normality plot", which shows the optimal lambda to be input in the transformation in order to better perform it. This value is represented by the abscissa lambda linked to the maximum of the represented function. If the spike is very close to zero, then you can run once again the transformation. This time you won't ask the code to give you back a lambda value, on the contrary you will input the value of lambda that probably would be zero (for what has been just explained).

```
# Eventually prepare your dataset in case of negative or null values
# Box-Cox transformation
[data_norm, lmbda]=stats.boxcox(df['column name'])
print('Lambda = %.3f' % lmbda)
#Box-Cox normality plot
fig = plt.figure()
ax = fig.add_subplot(111)
stats.boxcox_normplot(df['column name'], -2, 2, plot=ax)
# add grid
ax.grid(True)
plt.show()
# Plot the histogram with KDE
sns.histplot(data_norm, kde=True)
# Set the title
plt.title('Distribution of your data aftr Box Cox')
plt.show()
#Shapiro Wilk test
_, pval_SW = stats.shapiro(data_norm)
print('pvalue shapiro wilk: ', pval_SW)
if pval_SW > 0.05:
    print('Data after Box Cox are normally distributed')
else:
    print('Data after Box Cox are not normally distributed')
# Plot the qqplot for a single column dataframe called "df"
stats.probplot(data_norm, dist="norm", plot=plt) #alternatively you may input the index of the column instead of the name
plt.show()

# IF LAMBDA IS PRETTY DIFFERENT FROM ZERO SKIP THIS PART, otherwise attach this piece of code after plt.show()
# Input lambda = 0 for Box-Cox transformation and return the transformed data
data_norm = stats.boxcox(df['column name'], lmbda=0) #you can always change the value for lamda
# NOW YOU SHOULD PERFORM ONCE AGAIN THE NORMALITY TEST, so copy and paste here what goes from histogram to QQplot creation
```

Code 1.23: Box Cox tranformation

REMEMBER THAT THE ANTI-TRANSFORMATION FORMULA IS SIMPLY GIVEN BY ISOLATING THE VARIABLE X AS A FUNCTION OF Y.

Here the code for the anti-transformation

```
# if y = (x**lambda - 1)/lambda , then x = (y*lambda + 1)**(1/lambda)
```

Code 1.24: Box Cox anti-tranformation

1.2.2. Data gapping algorithm

Data gapping is useful when you have a rich but autocorrelated dataset.

You can keep 1 out of "gap_size" values and discard the other observations with the aim to break the autocorrelation pattern.

To make an example, if gap size is 3 you will retain: obs 0, obs 3, obs 6 ... meaning that you discard: obs 1, obs 2, obs 4, obs 6 and so on.

So be carefull because "gap_size" is not how many observation are discared between two retained observation! Actually the number of obs discared between two ritained obs is given by "gap_size - 1"

Here we show which observations are retained depending on gap size to help visualize it

```
# This code gives you a numpy ARRAY
gap_size = 3 input the gapping intervals
gap_num= int(len(data)/gap_size)
gap_data= np.zeros((gap_num))
for i in range (gap_num):
     gap_data[i]=data['Ex3'][i*gap_size]

# Instead, the following built-in function gives you a SERIES
gap_data = df['column name'][::gap_size]
```

Code 1.25: Box Cox anti-tranformation

1.2.3. Data batching

Batching has the same scope of gapping, but you have to define a batch size and then compute the average within a batch.

Each of these averages will represent an observation in your dataset.

The second provide code is crucial also for batched processes.

It is designed to add a new column in your dataframe so that each observations has its own link to the belonging batch

Then the function groupby.mean() will act exactly

```
# This code gives you a numpy ARRAY
batch_size = 6
batch_num = int(len(df['column name'])/batch_size)
j=0
batch_data= np.zeros((batch_num))
for i in range (batch_num):
    batch_data[i]=np.sum(df['column name'][j:j+5])/batch_size
    j=j+batch_size

# This code prepare the batches on the original dataframe and then use df.groupby('column name').mean() to averaging
# Create a new column in the dataframe with the corresponding batch number
df['Batch'] = np.repeat(np.arange(1, batch_num+1), batch_size) #watch out df in your original datafra and df['Batch'] is creating a new column called
    'Batch' in it
# Store the batch means in a new dataframe
batch_data = df.groupby('Batch').mean()
```

Code 1.26: Box Cox anti-tranformation

	GAP_SIZE				
OBS	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0
1	1	/	/	/	/
2	2	2	/	/	/
3	3	/	3	/	/
4	4	4	/	4	/
5	5	/	/	/	5
6	6	6	6	/	/
7	7	/	/	/	/
8	8	8	/	8	/
9	9	/	9	/	/
10	10	10	/	/	10
11	11	/	/	/	/
12	12	12	12	12	/
13	13	/	/	/	/
14	14	14	/	/	/
15	15	/	15	/	15
16	16	16	/	16	/
17	17	/	/	/	/
18	18	18	18	/	/
19	19	/	/	/	/
20	20	20	/	20	20

Figure 1.25: Gapping size effect

Observation number	Observation value	Batch_size 1	Batch_size 2	Batch_size 3	Batch_size 4	Batch_size 5	Batch_size 6
0	0.2562	1	1	1	1	1	1
1	0.6609	2	1	1	1	1	1
2	0.0711	3	2	1	1	1	1
3	0.4984	4	2	2	1	1	1
4	0.2407	5	3	2	2	1	1
5	0.4703	6	3	2	2	2	1
6	0.9232	7	4	3	2	2	2
7	0.7025	8	4	3	2	2	2
8	0.1221	9	5	3	3	2	2
9	0.0023	10	5	4	3	2	2
10	0.5677	11	6	4	3	3	2
11	0.0625	12	6	4	3	3	2
12	0.6572	13	7	5	4	3	3
13	0.9651	14	7	5	4	3	3
14	0.2232	15	8	5	4	3	3
15	0.1268	16	8	6	4	4	3
16	0.3614	17	9	6	5	4	3
17	0.1037	18	9	6	5	4	3
18	0.2850	19	10	7	5	4	4
19	0.1184	20	10	7	5	4	4
20	0.7485	21	11	7	6	5	4
21	0.3529	22	11	8	6	5	4
22	0.1966	23	12	8	6	5	4
23	0.0162	24	12	8	6	5	4
24	0.9957	25	13	9	7	5	5

Figure 1.26: Batch size effect

2 | Inferencial statistic

2.1. Hypothesis tests in presence of one sample (on one population)

The hypothesis testing procedure and the main type of tests are shown in the following pictures

General procedure:

1. From the problem context, identify the parameter of interest.
2. State the null hypothesis, H_0.
3. Specify an appropriate alternative hypothesis, H_1.
4. Choose a significance level α.
5. State an appropriate test statistic.
6. State the rejection region for the statistic.
7. Compute any necessary sample quantities, substitute these into the equation for the test statistic, and compute that value.
8. Decide whether or not H_0 should be rejected and report that in the problem context.

Figure 2.1: Hypothesis testing procedure

Some important tests:

- One sample tests:
  - Test for mean (known variance): one-sample z-test
  - Test for mean (unknown variance): one-sample t-test
  - Test for variance: chi-squared test (variance)
- Two samples tests
  - Test for mean difference (known var): two-sample z-test
  - Test for mean difference (unknown var): two-sample t-test
  - Test for mean of paired data (unknown var): paired t-test
  - Test for equality of variances: F-test (variances)

Figure 2.2: Types of tests introduced

There are two different type of errors when running a test of hypothesis

Decision	H_0 Is True	H_0 Is False
Fail to reject H_0	no error	type II error
Reject H_0	type I error	no error

$$\alpha = P(type\ I\ error) = P(reject\ H_0 | H_0 \text{ is true})$$

'Probability of rejecting a good product'

Also known as: ***significance level***

$$\beta = P(type\ II\ error) = P(fail\ to\ reject\ H_0 | H_0 \text{ is false})$$

'Probability of accepting a nonconforming product'

Figure 2.3: Error types (conceptually)

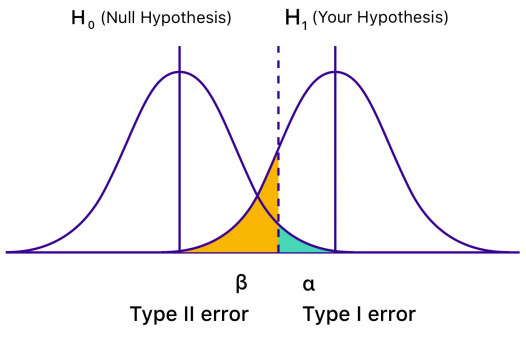


Figure 2.4: Error types (visually)

2.1.1. Z test on the mean: mean is unkown, variance is known

Assumptions

- X1, X2, ..., Xn is a random sample of size n from a population
- Population is normal
- **The variance of the population is known**

Testing Hypotheses on the Mean, Variance Known

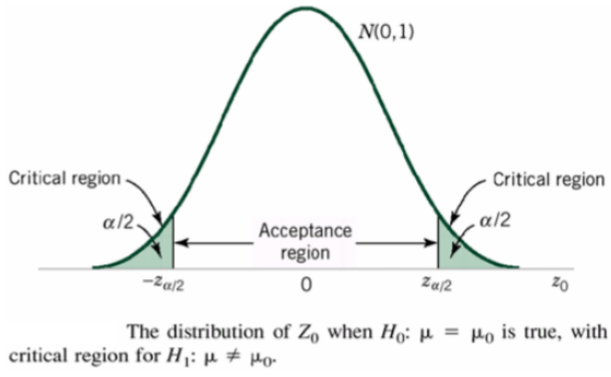
Null hypothesis: $H_0: \mu = \mu_0$

Test statistic: $Z_0 = \dfrac{\bar{X} - \mu_0}{\sigma/\sqrt{n}}$

Value of the test statistic when H_0 is true

Alternative Hypotheses	Rejection Criterion
$H_1: \mu \neq \mu_0$	$z_0 > z_{\alpha/2}$ or $z_0 < -z_{\alpha/2}$
$H_1: \mu > \mu_0$	$z_0 > z_\alpha$
$H_1: \mu < \mu_0$	$z_0 < -z_\alpha$

Figure 2.5: Z-test 1 sample structure



The distribution of Z_0 when $H_0: \mu = \mu_0$ is true, with critical region for $H_1: \mu \neq \mu_0$.

Reminder about notation:
$z_{\alpha/2}$ is the upper percentile of the distribution, but inverse cumulative distribution allows to compute the lower percentile

Figure 2.6: Z test acceptance and rejection region

REJECTION CRITERIA USING PERCENTILES Accept the NULL hypothesis Ho if and only if Zo is in the acceptance region. If Zo drops externally to the acceptance region then reject Ho and accept the alternative hypothesis H1.

P-value

The ***P-value*** is the smallest level of significance that would lead to rejection of the null hypothesis H_0

$$P = \begin{cases} 2[1 - \Phi(|z_0|)] & \text{for a two-tailed test: } H_0\text{: } \mu = \mu_0, \quad H_1\text{: } \mu \neq \mu_0 \\ 1 - \Phi(z_0) & \text{for an upper-tailed test: } H_0\text{: } \mu = \mu_0, \quad H_1\text{: } \mu > \mu_0 \\ \Phi(z_0) & \text{for a lower-tailed test: } H_0\text{: } \mu = \mu_0, \quad H_1\text{: } \mu < \mu_0 \end{cases}$$

Figure 2.7: Pvalue concept and computation for Z tests

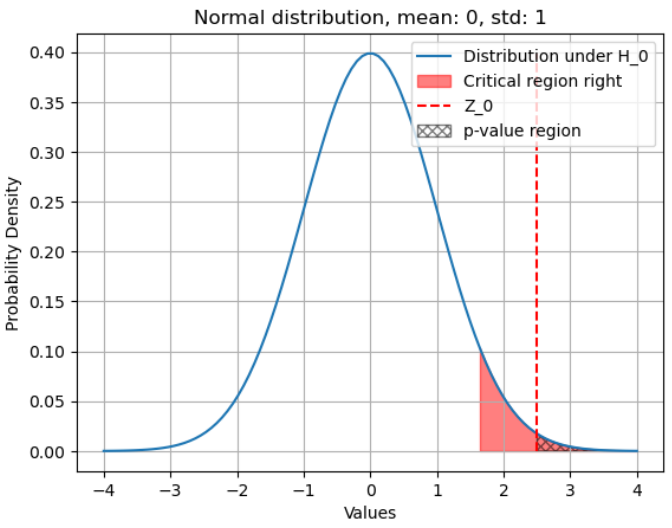


Figure 2.8: Pvalue region visual representation

REJECTION CRITERIA USING PVALUE

Accept the NULL hypothesis Ho if and only if Pvalue > alpha. If Pvalue < alpha reject Ho and accept the alternative hypotesis H1.

```
alpha = insert alpha
mu = insert mean                              #if std mu = 0
sigma = insert std                            #if std sigma = 1
Z_0 = insert statistic test value
# Compute z for alpha/2 (left) e 1-alpha/2 (right)
Z_alpha_left = stats.norm.ppf(alpha / 2, mu, sigma)
Z_alpha_right = stats.norm.ppf(1 - alpha / 2, mu, sigma)
#plotting a normal distribution
x = np.linspace(-4, 4, 100)            #if std (-4, 4, 100)
y = stats.norm.pdf(x, mu, sigma)
plt.title("Normal distribution, mean: %d, std: %d" % (mu, sigma))
plt.xlabel("Values")
plt.ylabel("Probability Density")
plt.grid(True)
plt.plot(x, y, label='Distribution under H_0')

#Fill the critical region
x_fill_left = np.linspace(-4, Z_alpha_left, 100)
y_fill_left = stats.norm.pdf(x_fill_left, mu, sigma)
plt.fill_between(x_fill_left, y_fill_left, color='red', alpha=0.5, label='Critical region left')
# Add text to the plot with the critical Z value and centering the text
plt.text(Z_alpha, 0.1, r'$Z_{%.3f} = {%.3f}$' % (1 - alpha, Z_alpha), fontsize=10) #watch out I wrote Z in capital letter!
x_fill_right = np.linspace(Z_alpha_right, 4, 100)
y_fill_right = stats.norm.pdf(x_fill_right, mu, sigma)
plt.fill_between(x_fill_right, y_fill_right, color='red', alpha=0.5, label='Critical region right')

# Plot the test statistic
plt.vlines(Z_0, 0, np.max(y), color='red', linestyles='dashed', label='Z_0')
# Plot the p-value region with a pattern
x_fill = np.linspace(Z_0, 4, 100)   #if Z_0 is negative modify the fill (-4, Z_0, 100)
y_fill = stats.norm.pdf(x_fill, 0, 1)
plt.fill_between(x_fill, y_fill, facecolor='none', alpha=0.5, hatch='xxxx', label='p-value region')
plt.legend()
plt.show()
```

Code 2.1: Gussian plots and representations for hypothesis testing

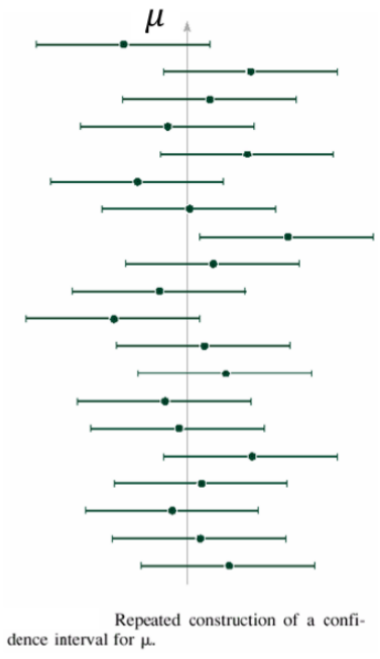
CONFIDENCE INTERVAL (ON THE TRUE MEAN)

There is a strict link between test of hypothesis and confidence intervals. A confidence interval an range of values that differs if it is computed on a different sample of same size and coming from the same population. The reason is that it depends on the sample mean of the sample extracted from a population. Furthermore it depends on the level of significance alpha, on the size of the sample extracted and on the standard deviation of the population, assumed to be known so far. MEANING OF CONFIDENCE INTERVALS: given a confidence level CL% = 1 - alpha%, it is the interval which, if built on M different samples of same size from the same population, CL% * M of them will contain the true value of the parameter (this parameter can be the mean or variance of a population, of models (think to CI in linear models), and so on and so forth).

The $100(1-\alpha)\%$ confidence interval on the population mean μ is given by:

$$\bar{X} - \frac{Z_{\alpha/2}\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + \frac{Z_{\alpha/2}\sigma}{\sqrt{n}}$$

Figure 2.9: Confidence interval Z test 1 sample



Repeated construction of a confidence interval for μ.

Figure 2.10: Pvalue region visual representation

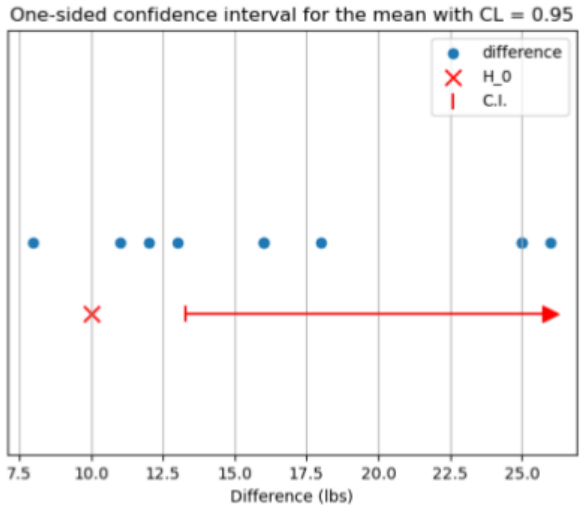


Figure 2.11: Example of rejection criteria using confidence interval

REJECTION CRITERIA USING CONFIDENCE INTERVALS

Accept the NULL hypothesis Ho if mu0 (hypothesized mean) drops within the confidence interval, otherwise reject Ho and accept H1.

THE REJECTION CRITERIA ARE THE SAME FOR EACH TYPE OF HYPOTHESIS TEST

2nd TYPE ERROR, POWER OF THE TEST AND OC CURVE

If you are quantitatively able to compute the 2nd type error, it is possible to compute the power of test, which simply represent the complementar of beta: power = 1 - beta. The power is the probability to rejecting Ho when Ho is actually false

Here it is provided the computation of the power of the test for a **two sided Z test**.

On the Type II error

The probability of a Type II error for the **two-sided alternative hypothesis on the mean, variance known**, is:

$$\beta = \Phi\left(z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma}\right) - \Phi\left(-z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma}\right)$$

$\delta = \mu_1 - \mu_0$

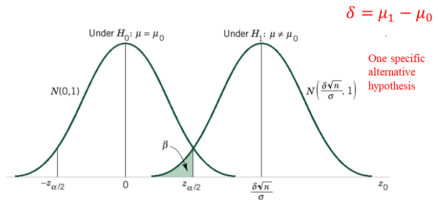


Figure 2.12: Scatter matrix for covariance (uncorrelated variables)

Proof:

$$-z_{\alpha/2} \leq \frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}} \leq z_{\alpha/2}$$

Under $H_0 : \bar{X} \sim N(\mu_0, \sigma^2/n)$

Under $H_1 : \bar{X} \sim N(\mu_1 = \mu_0 + \delta, \sigma^2/n)$ (*)

$$\beta = \Pr\left(\mu_0 - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \bar{X} \leq \mu_0 + z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \mid (*)\right)$$

$$= \Pr\left(\frac{\left(\mu_0 - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right) - \mu_0 - \delta}{\frac{\sigma}{\sqrt{n}}} \leq \frac{\bar{X} - \mu_1}{\frac{\sigma}{\sqrt{n}}} \leq \frac{\left(\mu_0 + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right) - \mu_0 - \delta}{\frac{\sigma}{\sqrt{n}}}\right) =$$

$$= \Pr\left(-z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma} \leq Z \leq z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma}\right) = \Phi\left(z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma}\right) - \Phi\left(-z_{\alpha/2} - \frac{\delta\sqrt{n}}{\sigma}\right)$$

Figure 2.13: 2nd type error computation (use it for the power of the curve)

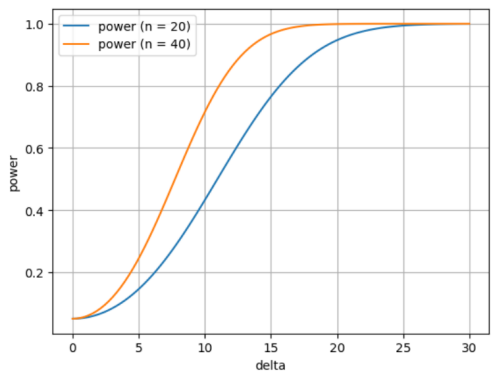


Figure 2.14: Operating characteristic curve (OC CURVE)

Once you get beta, the scope might be to plot the OPERATING CHARACTERISTIC CURVE (or power curve)

Let's see the code to compute beta and plot the OC curve.

```
# Define Z_alpha or Z_alpha2 depending on the type of test and the parameters to compute beta
Z_alpha2 = stats.norm.ppf(1-alpha/2)
Delta = np.linspace(0, 30, 100) #is a vector that measures the deviation
n = len(df) #or just a number which defines the sample size
#compute the power as 1 - beta,
beta = stats.norm.cdf(Z_alpha2 - Delta*np.sqrt(n)/std) - stats.norm.cdf(-Z_alpha2 -Delta*np.sqrt(n)/std)
power = 1 - stats.norm.cdf(Z_alpha2 - Delta*np.sqrt(n)/std) + stats.norm.cdf(-Z_alpha2 -Delta*np.sqrt(n)/std)
print('the second type error probability beta is: ', beta)
print('the power of the test is: ', power)

# if you have previously defined a different sample size "n" and computed the related power
# you can add a second OC curve in the same plot in order to compare them.
# to do that, add one more equal plt.plot with the propery instances declared
# Plot the OC curve
plt.plot(Delta, power, label = "power (n = insert no. of observation)")
plt.xlabel("delta")
plt.ylabel("power")
plt.grid(True)
plt.legend()
plt.show()
```

Code 2.2: Power of the test and OC curve

2.1.2. T test on the mean: mean and variance are unkown

Assumptions

- X1, X2, ..., Xn is a random sample of size n from a population
- Population is normal (or central limit theorem applies)
- **The variance of the population is unknown**

T-student distribution shapes symmetrically respect to the "y axis". Its tails are usually wider compared to the normal distribution and holds a lower height. As the number of **degrees of freedom** (shortened as **DoF** and in the code **dof**) increase, the shape of the t student tend to become a standard normal distribution.

Under those assumptions, the quantity:

$$T = \frac{\bar{X} - \mu}{S/\sqrt{n}}$$

follows a ***student-t distribution*** with $n - 1$ degrees of freedom

S=sample standard deviation

Figure 2.15: T test statistic

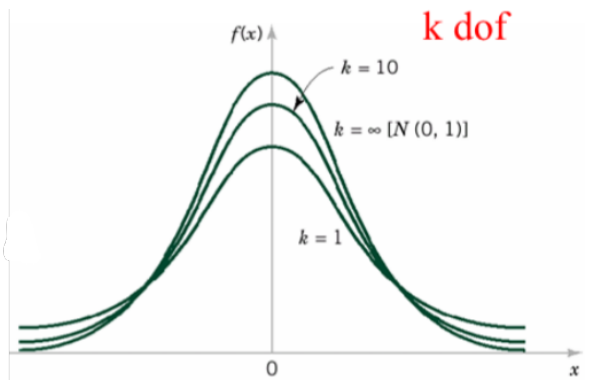


Figure 2.16: Probability density function t distribution

Testing Hypotheses on the Mean of a Normal Distribution, Variance Unknown

Null hypothesis: H_0: $\mu = \mu_0$

Test statistic: $T_0 = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$

Alternative Hypotheses	Rejection Criterion
H_1: $\mu \neq \mu_0$	$t_0 > t_{\alpha/2,n-1}$ or $t_0 < -t_{\alpha/2,n-1}$
H_1: $\mu > \mu_0$	$t_0 > t_{\alpha,n-1}$
H_1: $\mu < \mu_0$	$t_0 < -t_{\alpha,n-1}$

Figure 2.17: T test 1 sample structure

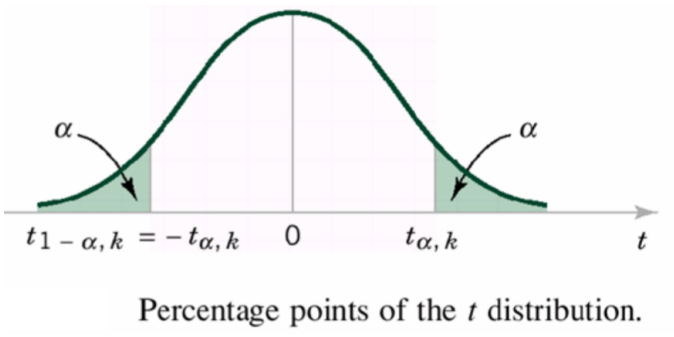


Figure 2.18: T test acceptance and rejection region

REJECTION CRITERIA USING PERCENTILES

Accept the NULL hypothesis Ho if and only if the test statistic "to" is in the acceptance region. If "to" drops externally to the acceptance region, then reject Ho and accept the alternative hypothesis H1

P-value

The *P-value* is the smallest level of significance that would lead to the rejection of the null hypothesis H_0.

$$P = \begin{cases} 2\left[1 - T(|t_0|)\right] & \text{for a two-tailed test: } H_0 : \mu = \mu_0, \quad H_1 : \mu \neq \mu_0 \\ 1 - T(t_0) & \text{for an upper-tailed test: } H_0 : \mu = \mu_0, \quad H_1 : \mu > \mu_0 \\ T(t_0) & \text{for a lower-tailed test: } H_0 : \mu = \mu_0, \quad H_1 : \mu < \mu_0 \end{cases}$$

Figure 2.19: Pvalue concept and computation for T tests)

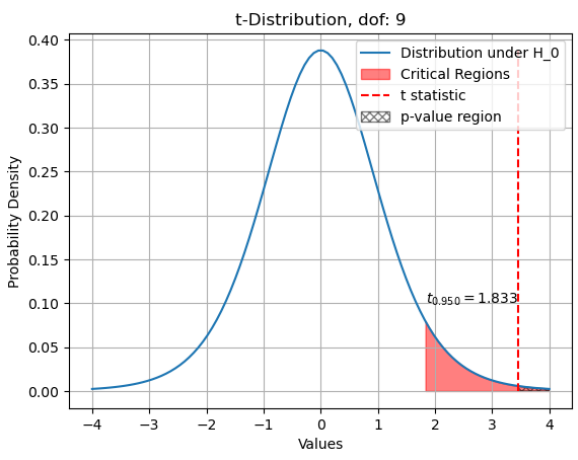


Figure 2.20: Pvalue region visual representation)

REJECTION CRITERIA USING PVALUE

Accept the NULL hypothesis Ho if and only if Pvalue > alpha. If Pvalue < alpha reject Ho and accept the alternative hypotesis H1.

```
dof = insert dof first  # Insert the degrees of freedom for the t-distribution
t0 = insert t statistic test  # Insert the test statistic value
alpha = insert alpha  # Insert the significance level

#ONE SIDED T TEST (UPPER AND LOWER EXPLANATIONS)
# Plot the cumulative probability
x = np.linspace(-4, 4, 100)  # Adjust range as necessary
plt.plot(x, stats.t.pdf(x, dof), label='Distribution under H_0')

# Adding Title, Labels and Grid
plt.title("t-Distribution, dof: %d" % dof)
plt.xlabel("Values")
plt.ylabel("Probability Density")
```

```
plt.grid(True)

# Filling the Probability Area FOR THE ONE SIDED (UPPER BOUNDED) Ttest
t_alpha = stats.t.ppf(1 - alpha, dof)
x_fill = np.linspace(t_alpha, np.max(x), 100)
y_fill = stats.t.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5, label='Critical Region')
# Add text to the plot with the critical t value and centering the text
plt.text(t_alpha, 0.1, r'$t_{%.3f} = {%.3f}$' % (1 - alpha, t_alpha), fontsize=10)

# Filling the Probability Area FOR THE ONE SIDED (LOWER BOUNDED) Ttest
t_alpha = stats.t.ppf(alpha, dof)
x_fill = np.linspace(np.min(x), t_alpha, 100)
y_fill = stats.t.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5, label='Critical Region')
# Add text to the plot with the t value and centering the text
plt.text(t_alpha, 0.1, r'$t_{%.3f} = {%.3f}$' % (alpha, t_alpha), fontsize=10)

# Filling the Probability Area for the two sided Ttest
t_1 = stats.t.ppf(alpha/2, dof)
t_2 = stats.t.ppf(1-alpha/2, dof)
x_fill = np.linspace(np.min(x), t_1, 100)
y_fill = stats.t.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5)
x_fill = np.linspace(t_2, np.max(x), 100)
y_fill = stats.t.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5, label='Critical Regions')
# Add text to the plot with the t values and centering the text
plt.text(t_1, 0.1, r'$t_{%.3f} = {%.3f}$' % (alpha/2, t_1), fontsize=10)
plt.text(t_2, 0.1, r'$t_{%.3f} = {%.3f}$' % (1-alpha/2, t_2), fontsize=10)

# Plot the test statistic t0
plt.vlines(t0, 0, np.max(stats.t.pdf(x, df1)), color='r', linestyle='--', label='t statistic')

# Plot p-value region
x_fill = np.linspace(t0, 4, 100)    #if t0 is negative modify the fill (-4, t0, 100)
y_fill = stats.t.pdf(x_fill, df1)
plt.fill_between(x_fill, y_fill, facecolor='none', alpha=0.5, hatch='xxxx', label='p-value region')

# Showing Plot
plt.legend()
plt.show()
```

Code 2.3: T-student distribution plot and representations for hypothesis testing

CONFIDENCE INTERVAL (ON THE TRUE MEAN) The following is the application in hypothesis testing in order to visualize the confidence interval

The $100(1-\alpha)\%$ confidence interval on the population mean μ (when the variance is unknown) is given by:

$$\bar{X} - t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}} \leq \mu \leq \bar{X} + t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}}$$

Figure 2.21: Confidence intervals for t student

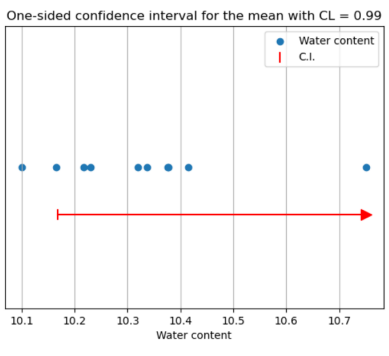


Figure 2.22: One-sided confidence interval for t student

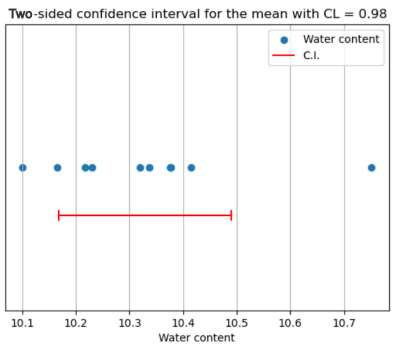


Figure 2.23: Two-sided confidence interval on t student

```
# Visualize a one sided confidence interval on a dot plot of the dataframe "df"
# CL = insert confidence level
# mu0 = insert null hypothesis' value
# CI_lower = df['column name'].mean() - t_alpha * df['column name'].std() / np.sqrt(n)

plt.title('One-sided confidence interval for the mean with CL = %.2f' % CL)
# plot of the dataframe's points
plt.scatter(df['column name'], np.zeros(n), label='insert a label for the scattered points')
# plot the hypothesized value "mu0" of the null hypothesis H0
plt.scatter(mu0, -0.01, label='H_0', color='r', marker='x', s=100)
# plot the confidence interval
plt.scatter(CI_lower, -0.01, label='C.I.', color='r', marker='|', s=100)
plt.plot([CI_lower, np.max(df['column name'])], [-0.01, -0.01], color='r')
plt.scatter(np.max(df['column name']), -0.01, color='r', marker='>', s=100)
# Add labels and legend
plt.ylim(-0.03, 0.03)
plt.xlabel('column name')
plt.yticks([])
plt.legend()
plt.grid()
plt.show()
```

```
# Visualize the two sided confidence interval on a dotted plot
# Before starting create a vector with just the confidence interval boundaries, for example:
# n = len(df['column name'])
# dof = n-1
# mu0 = insert null hypothesis' value
# t_alpha2 = stats.t.ppf(1-alpha/2, dof)
# CI_b = [df['column name'].mean() - t_alpha2 * df['column name'].std() / np.sqrt(n),
#         df['column name'].mean() + t_alpha2 * df['column name'].std() / np.sqrt(n)]
# print('The two sided confidence interval is: [%.3f, %.3f]' % (CI_b[0], CI_b[1]))

#now you can proceed referring to this vector
plt.title('Two-sided confidence interval for the mean with CL = %.2f' % CL)
plt.scatter(df['column name'], np.zeros(n), label='column name') # Original dataframe and column names
# plot the hypothesized mean "mu0" of the null hypothesis H0
plt.scatter(mu0, -0.01, label='H_0', color='r', marker='x', s=100)
# Plot the confidence interval
plt.scatter([CI_b[0], CI_b[1]], [-0.01, -0.01], color='r', marker='|', s=100)  # Both lower and upper bounds
plt.plot([CI_b[0], CI_b[1]], [-0.01, -0.01], color='r', label='C.I.')  # Connect the bounds with a red line
plt.ylim(-0.03, 0.03)
plt.xlabel('column name')
plt.yticks([])
plt.legend()
plt.grid()
plt.show()
```

Code 2.4: Scatter plot for confidence intervals

It is possible to automatically run a 1 sample ttest exploiting the function "stats.ttest_1samp"

```
# Perform the t-test on the difference using the stats.ttest_1samp function
t0_stats, p_value_t0_stats = stats.ttest_1samp(df['column name'], popmean = 0, alternative='greater')
# instead of 'greater' insert 'less' if you need to run a lower bounded t-test
# instead of 'greater' insert 'two-sided' if you need to run a two-sided t-test
print('t-statistic from stats.ttest_1samp: %.3f' % t0_stats)
print('p-value from stats.ttest_1samp: %.3f' % p_value_t0_stats)
```

Code 2.5: CODE TO AUTOMATICALLY RUN A 1 SAMPLE T-TEST

This code can be adapted to display different confidence intervals, but remember that:

- You usually create this plot when the confidence interval is on a mean
- this representations usually fit for all the test on the mean: every type of Ztest and Ttest (PairedTtest included)

REJECTION CRITERIA USING CONFIDENCE INTERVALS

Accept the NULL hypothesis Ho if mu0 (hypothesized mean) drops within the confidence interval, otherwise reject Ho and accept H1.

2.1.3. Chi2 test on the variance and standard deviation

Assumptions

- X1, X2, ..., Xn is a random sample of size n from a population
- Population is normal (or central limit theorem applies)
- **The mean of the population is unknown**

Chi-square distribution shapes asymmetrically and assumes only positive values. As the number of **degrees of freedom** (shortened as **DoF** and in the code **dof**) increase, the shape of the Chi2 distribution tend to become a standard normal distribution.

Under those assumptions, the quantity:

$$X^2 = \frac{(n-1)S^2}{\sigma^2}$$

follows a ***chi-squared*** (χ^2) ***distribution*** with $n-1$ degrees of freedom

Figure 2.24: Chi2 test statistic

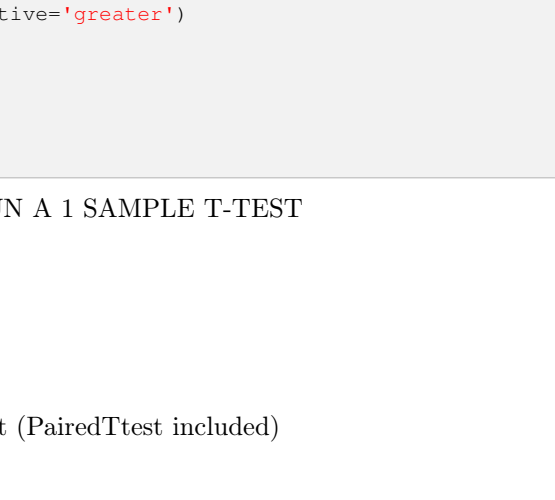


Figure 2.25: Probability density function Chi2 distribution

- Non-Negativity: The chi-square statistic is derived from the sum of squared deviations of observed and expected frequencies. This sum is always non-negative, leading to a distribution that starts at zero and extends positively.
- Right-Skewed: The chi-square distribution is right-skewed, meaning it has a long tail on the right side.

Testing Hypotheses on the Variance of a Normal Distribution

Null hypothesis: $H_0: \sigma^2 = \sigma_0^2$

Test statistic: $\chi_0^2 = \frac{(n-1)S^2}{\sigma_0^2}$

Alternative Hypotheses	Rejection Criterion
$H_1: \sigma^2 \neq \sigma_0^2$	$\chi_0^2 > \chi^2_{\alpha/2,n-1}$ or $\chi_0^2 < \chi^2_{1-\alpha/2,n-1}$
$H_1: \sigma^2 > \sigma_0^2$	$\chi_0^2 > \chi^2_{\alpha,n-1}$
$H_1: \sigma^2 < \sigma_0^2$	$\chi_0^2 < \chi^2_{1-\alpha,n-1}$

Figure 2.26: Chi2 test structure

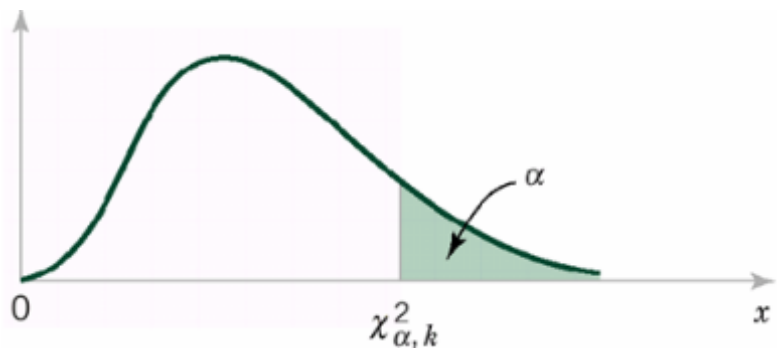


Figure 2.27: Chi2 test acceptance and rejection region

REJECTION CRITERIA USING PERCENTILES

Accept the NULL hypothesis Ho if and only if the test statistic is in the acceptance region. If "Chi2_0" drops externally to the acceptance region, then reject Ho and accept the alternative hypothesis H1

NOMENCLATURE AND COMPUTATION OF THE CRITICAL VALUES Here we want to underline that the way to call the upper and lower percentiles does not reflect the way in which they are computed: they are actually the opposite. To make it clear a two sided Chi2 test and its critical rejection values are provided visually on the right.

The critical values of the chi-square distribution are denoted as χ^2_α and $\chi^2_{1-\alpha}$:

- χ^2_α is computed as the inverse of the cumulative distribution function (CDF) using the probability $1-\alpha$.
- $\chi^2_{1-\alpha}$ is computed as the inverse of the cumulative distribution function (CDF) using the probability α.

This naming convention might seem counterintuitive because χ^2_α corresponds to the upper percentile (right tail) of the distribution, while $\chi^2_{1-\alpha}$ corresponds to the lower percentile (left tail). This is due to the non-symmetric shape of the chi-square distribution.

Figure 2.28: Chi2 critical values nomenclature and computation

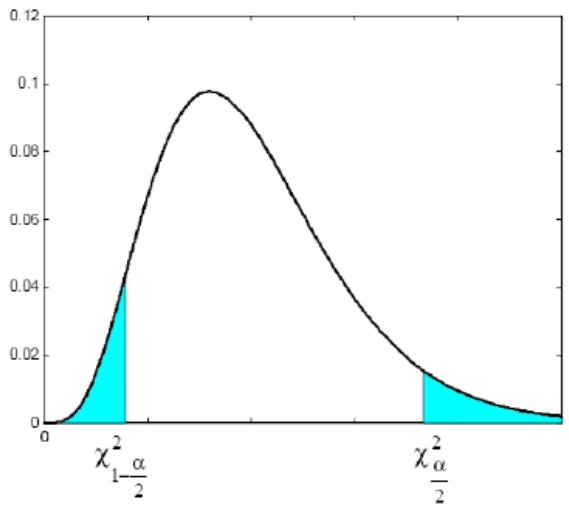


Figure 2.29: Chi2 nomenclature visually

Due to the skewness of the distribution it is not possible (does not make sense) to compute the pvalue on the two sided test. Whereas it might be useful to compute it when the test is limited by just one critical value (one sided test)

P-value

The *P-value* is the smallest level of significance that would lead to the rejection of the null hypothesis H_0.

$$P = \begin{cases} 1 - \chi^2(\chi_0^2) & \text{for an upper-tailed test: } H_0: \sigma^2 = \sigma_0^2, \quad H_1: \sigma^2 > \sigma_0^2 \\ \chi^2(\chi_0^2) & \text{for a lower-tailed test: } H_0: \sigma^2 = \sigma_0^2, \quad H_1: \sigma^2 < \sigma_0^2 \end{cases}$$

In these formulas:

- χ_0^2 is the calculated value of the chi-square statistic.
- χ^2 represents the cumulative distribution function of the chi-square distribution with k degrees of freedom, where k is the number of degrees of freedom.

Figure 2.30: Chi2 pvalue

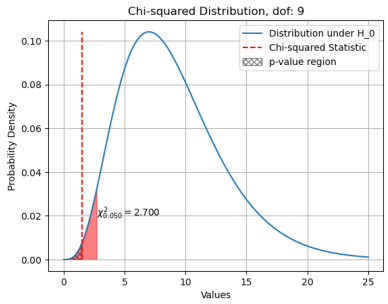


Figure 2.31: Chi2 test lower bounded

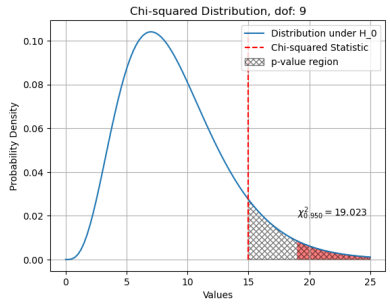


Figure 2.32: Chi2 test upper bounded

REJECTION CRITERIA USING PVALUE

Accept the NULL hypothesis Ho if and only if Pvalue $>$ alpha. If Pvalue $<$ alpha reject Ho and accept the alternative hypotesis H1.

Here we provide the code for running a Chi2 test and represent the graphs shown right above

```
# Code to run a Chi2 test
sigma2_0 = insert value of the null hypotesis
n = len(df)
dof = n-1  # Insert the degrees of freedom for the chi-squared distribution
CL = 0.95 #Insert the confidence level
alpha = 1 - CL  # Insert the significance level
S2 = df['column name'].var() #sample variance

chi2_statistic = (dof*S2)/sigma2_0  # compute the test statistic value
print('the test statistic is: ', chi2_statistic)
chi2_1 = stats.chi2.ppf(alpha/2, dof)
print('the rejection region spans in [0, %.5f]' % chi2_1) #lower bounded

chi2_2 = stats.chi2.ppf(1-alpha/2, dof)
print('the rejection region spans from %.5f onwards' % chi2_2) #upper bounded

print('the acceptance region spans within [%.5f, %.5f]' % (chi2_1 , chi2_2)) #two sided

# Two sided confidence intervals on the variance
CI_UB = (dof*S2)/chi2_1
CI_LB = (dof*S2)/chi2_2
CI_b = [CI_LB, CI_UB]
print('the confidence interval for the variance span within [%.3f,%.3f] ' % (CI_b[0], CI_b[1]))

#-----------------------------------------------------------------------

# PLOT THE CHI SQUARED
# Define the range for the chi-squared distribution
x = np.linspace(0, 25, 100)  # Adjust as necessary
plt.plot(x, stats.chi2.pdf(x, dof), label='Distribution under H_0')
# Adding Title, Labels, and Grid
plt.title("Chi-squared Distribution, dof: %d" % dof)
plt.xlabel("Values")
plt.ylabel("Probability Density")
plt.grid(True)

# LOWER BOUNDED CHI2 TEST CRITICAL REGION
x_fill_lower = np.linspace(0, chi2_1, 100)
y_fill_lower = stats.chi2.pdf(x_fill_lower, dof)
plt.fill_between(x_fill_lower, y_fill_lower, color='red', alpha=0.5)
# Adding text annotations for critical values
plt.text(chi2_1, 0.02, r'$\chi^2_{%.3f} = {%.3f}$' % (alpha, chi2_1), fontsize=10)
# Plot the test statistic chi2_statistic
plt.vlines(chi2_statistic, 0, np.max(stats.chi2.pdf(x, dof)), color='r', linestyle='--', label='Chi-squared Statistic')
# Plot p-value region
x_fill = np.linspace(0, chi2_statistic, 100)
y_fill = stats.chi2.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, facecolor='none', alpha=0.5, hatch='xxxx', label='p-value region')
# Showing Plot
plt.legend()
plt.show()

# UPPER BOUNDED CHI2 TEST CRITICAL REGION
x_fill_upper = np.linspace(chi2_2, np.max(x), 100)
y_fill_upper = stats.chi2.pdf(x_fill_upper, dof)
plt.fill_between(x_fill_upper, y_fill_upper, color='red', alpha=0.5)
# Adding text annotations for critical values
plt.text(chi2_2, 0.02, r'$\chi^2_{%.3f} = {%.3f}$' % (1-alpha, chi2_2), fontsize=10) #change 0.02 for the height of annotation
# Plot the test statistic chi2_statistic
plt.vlines(chi2_statistic, 0, np.max(stats.chi2.pdf(x, dof)), color='r', linestyle='--', label='Chi-squared Statistic')
# Plot p-value region
x_fill = np.linspace(chi2_statistic, np.max(x), 100)
y_fill = stats.chi2.pdf(x_fill, dof)
plt.fill_between(x_fill, y_fill, facecolor='none', alpha=0.5, hatch='xxxx', label='p-value region')
# Showing Plot
plt.legend(loc='upper right')
plt.show()

#For two sided representation stop at the critical region cause pvalue on two sided chi2 test does not make any sense
```

Code 2.6: Chi2 test code, CI code and rejection region representation

CONFIDENCE INTERVAL (ON THE TRUE VARIANCE)

Chi2 confidence intervals cannot be represented with the usual scatter plot as they were computed for the mean, and this one is a CI on the variance:

The $100(1-\alpha)\%$ confidence interval on the population variance σ is given by:

$$\frac{(n-1)S^2}{\chi^2_{\frac{\alpha}{2},n-1}} \leq \sigma^2 \leq \frac{(n-1)S^2}{\chi^2_{1-\frac{\alpha}{2},n-1}}$$

Remind: the chi-squared distribution is not symmetric

Figure 2.33: Confidence intervals for variance with Chi2)

For a confidence interval around the variance, as derived from the Chi-squared test, the nature of the interval is different. The Chi-squared distribution is not symmetric, and the interval calculated reflects the range within which the true variance is expected to lie, rather than the mean.

Using the same scatter plot method to visualize a confidence interval for the variance would not be appropriate for several reasons:

1. **Nature of Data Points**: Scatter plots represent individual data points and their distribution. The confidence interval for the mean shows the central tendency and uncertainty around it. However, a CI for variance relates to the spread of the data, not a specific central value.
2. **Visual Misrepresentation**: Plotting a CI for variance below the scatter plot data points might visually mislead the viewer to think it pertains to central tendency or individual data points, similar to how a mean CI is interpreted.
3. **Different Interpretations**: The CI for the mean focuses on where the average value lies, whereas the CI for variance concerns the data's spread. These represent fundamentally different aspects of the dataset.

CONFIDENCE INTERVAL (ON THE TRUE STANDARD DEVIATION)

$$\sqrt{\frac{(n-1)S^2}{\chi^2_{\frac{\alpha}{2},n-1}}} \leq \sigma \leq \sqrt{\frac{(n-1)S^2}{\chi^2_{1-\frac{\alpha}{2},n-1}}}$$

Figure 2.34: Confidence intervals for standard deviation with Chi2)

BE CAREFUL !!

- **Lower Bound:** For the lower bound of the CI, we must divide by the upper critical percentile $\chi^2_{\frac{\alpha}{2},n-1}$
- **Upper Bound:** For the upper bound of the CI, we must divide by the lower critical percentile $\chi^2_{1-\frac{\alpha}{2},n-1}$

Figure 2.35: Critical values to be place in Chi2 confidence intervals)

REJECTION CRITERIA USING CONFIDENCE INTERVALS

Accept the NULL hypothesis Ho if the hypothesized variance or standard deviation drops within the confidence interval, otherwise reject Ho and accept H1.

2.2. Hypothesis tests in presence of two samples (on two populations)

This types of test are designed and particularly useful when we need to check if the mean or the variances between population are equal. It is also possible to make a test on the distance between the mean of two populations.

Tests on equality of the means deal with the difference between the two sampled distributions.

This even allows us to state possible null hypothesis Ho to test, in case means would be different, how far they are from each other: if you state "Ho: mu0 = 0" you are testing the equality of the means, whereas if you pop an other value for mu0 you are assessing if there is statistical evidence to state that the two populations means are that value distant from each other.

2.2.1. Z test on the difference in means: variances are known

Assumptions

- X11, X12, ..., X1n is a random sample of size n1 from population 1
- X21, X22, ..., X2n is a random sample of size n2 from population 2
- The two populations are independent
- Both the populations are normal (or central limit theorem applies)
- **The variances of the populations are known**

Under those assumptions, the quantity:

$$Z = \frac{\overline{X_1} - \overline{X_2} - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

Follows a standard normal distribution, N(0,1)

Figure 2.36: Z statistic 2 samples test

Testing Hypotheses on the Difference in Means, Variances Known

Null hypothesis: H_0: $\mu_1 - \mu_2 = \Delta_0$

Test statistic: $Z_0 = \dfrac{\overline{X}_1 - \overline{X}_2 - \Delta_0}{\sqrt{\dfrac{\sigma_1^2}{n_1} + \dfrac{\sigma_2^2}{n_2}}}$

Alternative Hypotheses	Rejection Criterion
H_1: $\mu_1 - \mu_2 \neq \Delta_0$	$z_0 > z_{\alpha/2}$ or $z_0 < -z_{\alpha/2}$
H_1: $\mu_1 - \mu_2 > \Delta_0$	$z_0 > z_\alpha$
H_1: $\mu_1 - \mu_2 < \Delta_0$	$z_0 < -z_\alpha$

Figure 2.37: Z test 2 samples structure

The $100(1-\alpha)\%$ confidence interval on the difference between the population means (known variances) is given by:

$$\overline{X_1} - \overline{X_2} - Z_{\frac{\alpha}{2}}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} \leq \mu_1 - \mu_2 \leq \overline{X_1} - \overline{X_2} + Z_{\alpha/2}\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

Figure 2.38: Confidence interval Z-test 2 samples

2.2.2. T test of the difference in means: variances are unknown but EQUAL

Assumptions

- X11, X12, ..., X1n is a random sample of size n1 from population 1
- X21, X22, ..., X2n is a random sample of size n2 from population 2
- The two populations are independent
- Both the populations are normal (or central limit theorem applies)
- **The variances of the populations are unknown but EQUAL**

Under those assumptions, the quantity:

$$T = \frac{\overline{X_1} - \overline{X_2} - (\mu_1 - \mu_2)}{S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

Follows a student-t distribution with $n_1 + n_2 - 2$ degrees of fredom.
Pooled variance:

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}$$

Figure 2.39: T statistic 2 samples test equal variances

Testing Hypotheses on the Difference in Means of Two Normal Distributions, Variances Unknown and Equal[1]

Null hypothesis: $H_0: \mu_1 - \mu_2 = \Delta_0$

Test statistic: $T_0 = \dfrac{\overline{X}_1 - \overline{X}_2 - \Delta_0}{S_p\sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}}$ (5-9)

Alternative Hypothesis	Rejection Criterion
$H_1: \mu_1 - \mu_2 \neq \Delta_0$	$t_0 > t_{\alpha/2,n_1+n_2-2}$ or $t_0 < -t_{\alpha/2,n_1+n_2-2}$
$H_1: \mu_1 - \mu_2 > \Delta_0$	$t_0 > t_{\alpha,n_1+n_2-2}$
$H_1: \mu_1 - \mu_2 < \Delta_0$	$t_0 < -t_{\alpha,n_1+n_2-2}$

Figure 2.40: T test 2 samples structure

The $100(1-\alpha)\%$ confidence interval on the difference between the population means (unknown variances) is given by:

Two-Sample T-Test with Equal Variances CONFIDENCE INTERVAL

$$\bar{X}_1 - \bar{X}_2 \pm t_{\alpha/2,df}\sqrt{S_p^2\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}$$

Figure 2.41: Confidence interval T-test 2 samples equal variances

BE CAREFUL: here the **dof = n1 + n2 - 2**

2.2.3. T test of the difference in means (Welch's T-test): variances are unkown but DIFFERENT

Assumptions

- X11, X12, ..., X1n is a random sample of size n1 from population 1
- X21, X22, ..., X2n is a random sample of size n2 from population 2
- The two populations are independend
- Both the populations are normal (or central limit theorem applies)
- **The variances of the populations are unknown and different**

The only thing different from the previous test is how to compute the t stastic, but the structure of the test is always the same:

Under those assumptions, the quantity:

$$T = \frac{\overline{X_1} - \overline{X_2} - (\mu_1 - \mu_2)}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}} \qquad \nu = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{(S_1^2/n_1)^2}{n_1 - 1} + \frac{(S_2^2/n_2)^2}{n_2 - 1}}$$

Follows a student-t distribution with ν degrees of fredom

Figure 2.42: T statistic 2 samples test equal variances

Testing Hypotheses on the Difference in Means of Two Normal Distributions, Variances Unknown and Equal[1]

Null hypothesis: $H_0: \mu_1 - \mu_2 = \Delta_0$

Test statistic: $T_0 = \dfrac{\overline{X}_1 - \overline{X}_2 - \Delta_0}{S_p\sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}}$ (5-9)

Alternative Hypothesis	Rejection Criterion
$H_1: \mu_1 - \mu_2 \neq \Delta_0$	$t_0 > t_{\alpha/2,n_1+n_2-2}$ or $t_0 < -t_{\alpha/2,n_1+n_2-2}$
$H_1: \mu_1 - \mu_2 > \Delta_0$	$t_0 > t_{\alpha,n_1+n_2-2}$
$H_1: \mu_1 - \mu_2 < \Delta_0$	$t_0 < -t_{\alpha,n_1+n_2-2}$

Figure 2.43: T test 2 samples structure

The $100(1-\alpha)\%$ confidence interval on the difference between the population means (unknown variances) is given by:

Two-Sample T-Test with Unequal Variances (Welch's T-Test) **CONFIDENCE INTERVAL**

$$\bar{X}_1 - \bar{X}_2 \pm t_{\alpha/2,df}\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}$$

$$df \approx \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{\left(\frac{S_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{S_2^2}{n_2}\right)^2}{n_2 - 1}}$$

Figure 2.44: Confidence interval T-test 2 samples different variances

```
#if the samples are kept in two dataframe df1 and df2 run it as it is
t0, p_value_t0 = stats.ttest_ind(df1, df2, equal_var=True) #state False if variances were not equal
print('t-test: t0 = %.3f' % t0)
print('p-value for t-test: %.3f' % p_value_t0)
```

Code 2.7: CODE TO AUTOMATICALLY RUN A 2 SAMPLE T-TEST (BOTH WITH EQUAL OR NON EQUAL VARIANCES

2.2.4. Paired T-test for the mean of delta vector "D" = Dj = X1j-X2j : variances of the differences between pairs in unkown

Assumptions

- X11, X12, ..., X1n is a random sample of size n from population 1
- X21, X22, ..., X2n is a random sample of size n from population 2
- The differences between pairs **Dj = X1j - X2j** are normal (or central limit theorem applies). So you have to check if the new population, given from the differences, is normal or not;
- **The variance of the differences are unknown**

NOTE: HERE THE TWO SAMPLE HAVE THE SAME SIZE

ATTENTION: THIS TEST USUALLY FIT FOR BEFORE AND AFTER PROCESS SAMPLES (e.g.: weight before and after a diet program, yield load before and after a treatment, and so on.)

Under those assumptions, the quantity:

$$T = \frac{\overline{D} - \mu_D}{S_D/\sqrt{n}}$$

Follows a student-t distribution with $n-1$ degrees of freedom, where: $D_j = X_{1j} - X_{2j} \sim N(\mu_D, \sigma_D^2)$
(NOTICE: we are assuming differences to be random normal var.)

Figure 2.45: T statistic 2 samples test equal variances

The Paired *t*-Test

Null hypothesis: $H_0: \mu_D = \Delta_0$

Test statistic: $T_0 = \dfrac{\overline{D} - \Delta_0}{S_D/\sqrt{n}}$ (5-16)

Alternative Hypothesis	Rejection Region
$H_1: \mu_D \neq \Delta_0$	$t_0 > t_{\alpha/2,n-1}$ or $t_0 < -t_{\alpha/2,n-1}$
$H_1: \mu_D > \Delta_0$	$t_0 > t_{\alpha,n-1}$
$H_1: \mu_D < \Delta_0$	$t_0 < -t_{\alpha,n-1}$

Figure 2.46: T test 2 samples structure

The $100(1-\alpha)\%$ confidence interval on the difference between the population means is given by:

$$\bar{D} - t_{\frac{\alpha}{2},n-1} S_D/\sqrt{n} \leq \mu_D \leq \bar{D} + t_{\frac{\alpha}{2},n-1} S_D/\sqrt{n}$$

S_D is lower than the pooled standard deviation: paired test is more precise (smaller c.i.)

Figure 2.47: Confidence interval Paired T-test

```
# FOR A PAIRED T-TEST USE ONE OF THE FOLLOWING CODE (better the 2nd)
#THIS FUNCTIONS WORKS IF AND ONLY IF THE TWO SAMPLES HAVE THE SAME SIZE

# Perform a paired t-test using the stats.ttest_rel function
# THIS FUNCTION CHECK THE EQUALITY OF THE MEANS SO mu0 DOES NOT HAVE TO BE STATED IN THE FUNCTION. IT IS ASSUMED TO BE 0 BY SETTINGS.
t0_stats_trel, p_value_t0_stats_trel = stats.ttest_rel(data['before'], data['after'], alternative='greater')
print('t-statistic from stats.ttest_rel: %.3f' % t0_stats_trel)
print('p-value from stats.ttest_rel: %.3f' % p_value_t0_stats_trel)

# Perform the t-test on the difference using the stats.ttest_1samp function (this is just a 1sample ttest on the vector of differences, nothing more)
# THIS FUNCTION ALLOW TO SET A PROPER VALUE FOR Ho EVEN DIFFERENT FROM ZERO, THANKS TO THE INSTANCE "POPMEAN"
t0_stats, p_value_t0_stats = stats.ttest_1samp(df['column name'], popmean = 0, alternative='greater')
# instead of 'greater' insert 'less' if you need to run a lower bounded t-test
# instead of 'greater' insert 'two-sided' if you need to run a two-sided t-test
print('t-statistic from stats.ttest_1samp: %.3f' % t0_stats)
print('p-value from stats.ttest_1samp: %.3f' % p_value_t0_stats)
#------------------------------------------------------------------------
```

Code 2.8: CODE TO AUTOMATICALLY RUN A PAIRED T-TEST

2.2.5. F-test for equality of variances: variances are unkown

A brief note on the Fisher F distribution
Let W and Y be **independent chi-squared random variables** with u and v degrees of freedom, respectively. Then, the ratio:

$$F = \frac{W/u}{Y/v}$$

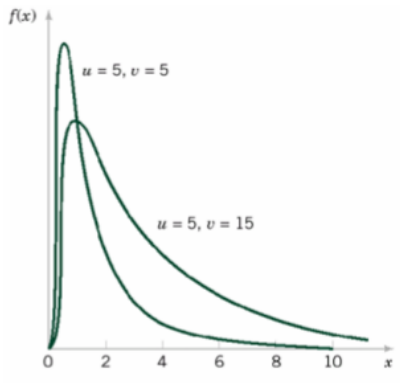
Follows an ***F* distribution** with u and v degrees of freedom ($F_{u,v}$).
Its probability density function is:

$$f(x) = \frac{\Gamma\left(\frac{u+v}{2}\right)\left(\frac{u}{v}\right)^{u/2} x^{(u/2)-1}}{\Gamma\left(\frac{u}{2}\right)\Gamma\left(\frac{v}{2}\right)\left[\left(\frac{u}{v}\right)x+1\right]^{(u+v)/2}}, \quad 0 < x < \infty$$

Remind: $X^2 = \frac{(n-1)S^2}{\sigma^2}$ follows a ***chi-squared*** (χ^2) ***distribution*** with $n-1$ degrees of freedom, thus: $S^2 \sim [\sigma^2/(n-1)]\chi^2(n-1)$

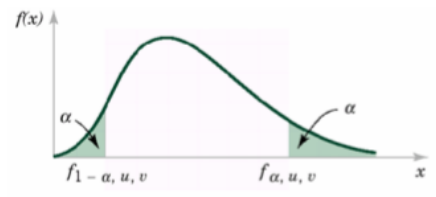
Figure 2.48: F distribution

A brief note on the Fisher F distribution



Probability density functions of two *F* distributions.

$$f_{1-\alpha,u,v} = \frac{1}{f_{\alpha,v,u}} \quad !$$



Upper and lower percentage points of the *F* distribution.

Figure 2.49: F distribution representation

Under those assumptions, the quantity:

$$F = \frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2}$$

Follows an F distribution with $n_1 - 1$ and $n_2 - 1$ degrees of freedom.

Figure 2.50: F test statistic for equality of variances

Testing Hypotheses on the Equality of Variances of Two Normal Distributions

Null hypothesis: $H_0: \sigma_1^2 = \sigma_2^2$

Test statistic: $F_0 = \frac{S_1^2}{S_2^2}$ (5-21)

Alternative Hypotheses	Rejection Criterion
$H_1: \sigma_1^2 \neq \sigma_2^2$	$f_0 > f_{\alpha/2,n_1-1,n_2-1}$ or $f_0 < f_{1-\alpha/2,n_1-1,n_2-1}$
$H_1: \sigma_1^2 > \sigma_2^2$	$f_0 > f_{\alpha,n_1-1,n_2-1}$
$H_1: \sigma_1^2 < \sigma_2^2$	$f_0 < f_{1-\alpha,n_1-1,n_2-1}$

Figure 2.51: F-test structure

```
#Code to run the test (TWO SIDED)
dof1 = insert dof first
dof2 = insert dof second
F0 = insert F statistic test
F_1 = stats.f.ppf(alpha/2, dof1, dof2)
F_2 = stats.f.ppf(1-alpha/2, dof1, dof2)
Print('acceptance region is: [%.5f, %.5f], whereas your Fstatistic is: %.5f' % (F_1, F_2, F0))
```

```
if F0 < F_2 and Fstat > F_1:
     print('Accept Ho')
else:
     print('Reject Ho')

# Confidence interval given two dataframe df1 and df2 for the populations
# You can modify it so that the data are picked from different columns of a single dataframe as df['column name'].var()
#With confidence intervals
CI_b = [(df1.var()/df2.var())*stats.f.ppf(alpha/2, dof2, dof1), (df1.var()/df2.var())*stats.f.ppf(1-alpha/2, dof2, dof1)]
print ('the confidence interval for the ratio of variances is: [%.5f, %.5f]' % (CI_b[0], CI_b[1]))

# plot the cumulative probability
x = np.linspace(0, 4, 100)
plt.plot(x, stats.f.pdf(x, dof1, dof2), label='Distribution under H_0')
# Adding Title, Labels and Grid
plt.title("F Distribution, dof2: %d, dof1: %d" % (dof2, dof1))
plt.xlabel("Values")
plt.ylabel("Probability Density")
plt.grid(True)

# Filling the Probability Area (TWO SIDED)
x_fill = np.linspace(0, F_1, 100)
y_fill = stats.f.pdf(x_fill, df1, df2)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5)
x_fill = np.linspace(F_2, np.max(x), 100)
y_fill = stats.f.pdf(x_fill, dof1, dof2)
plt.fill_between(x_fill, y_fill, color='red', alpha=0.5, label='Critical Regions')
# Add text to the plot with the f values and centering the text
plt.text(F_1, 0.1, r'$F_{%.3f} = {%.3f}$' % (alpha/2, F_1), fontsize=10)
plt.text(F_2, 0.1, r'$F_{%.3f} = {%.3f}$' % (1-alpha/2, F_2), fontsize=10)
# Plot the test statistic F0
plt.vlines(F0, 0, np.max(stats.f.pdf(x, df1, df2)), color='r', linestyle='--', label = 'F statistic')

# PVALUE REGION IS ONLY FOR ONE SIDED F-TEST
# Plot p-value region
# x_fill = np.linspace(F0, 4, 100)  #if nedeed modify the fill: (0, F0, 100)
# y_fill = stats.f.pdf(x_fill, dof1, dof2)
# plt.fill_between(x_fill, y_fill, facecolor='none', alpha=0.5, hatch='xxxx', label='p-value region')

# Showing Plot
plt.legend()
plt.show()
```

Code 2.9: F-TEST CODE + VISUAL REPRESENTATION

Confidence interval for variances ratio

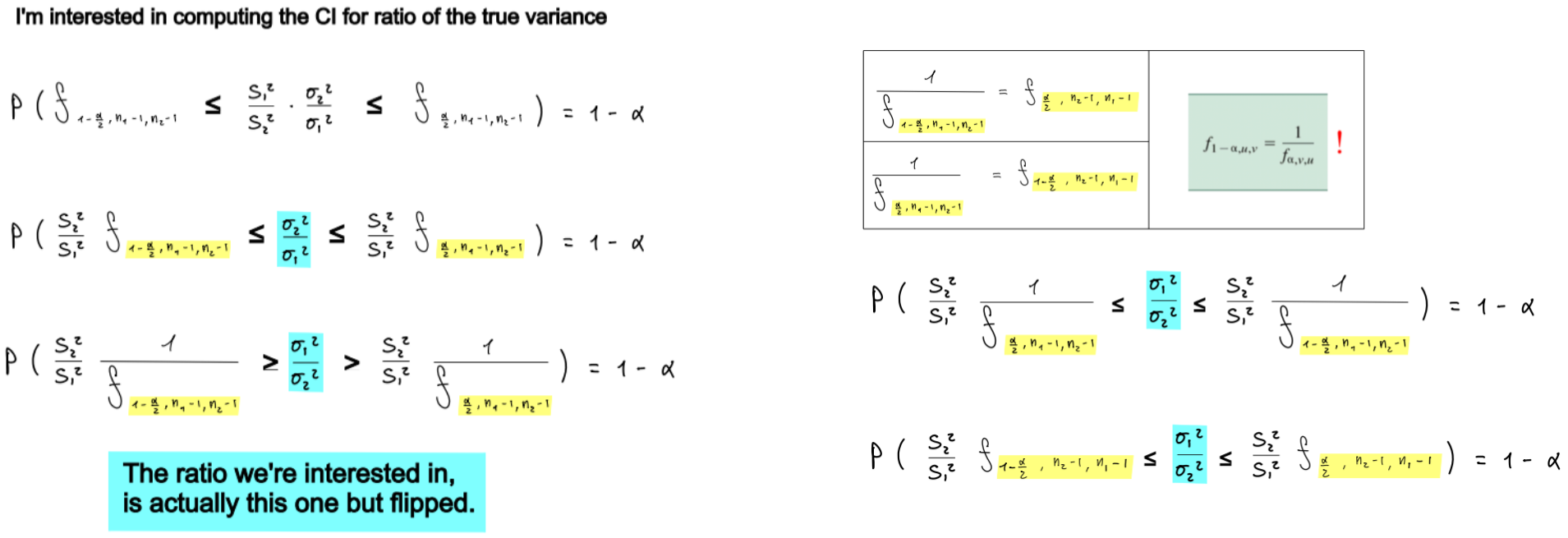


Figure 2.52: Steps to get the confidence interval

The $100(1-\alpha)\%$ confidence interval on the difference between the population means is given by:

$$\frac{S_1^2}{S_2^2} f_{1-\alpha/2,n_2-1,n_1-1} \leq \frac{\sigma_1^2}{\sigma_2^2} \leq \frac{S_1^2}{S_2^2} f_{\alpha/2,n_2-1,n_1-1}$$

Figure 2.53: Confidence interval F test on variances ratio

2nd type error F-test

The Type II error is the probability of accepting the null hypothesis when it is false.

$$\beta = Pr(\text{accept } H_0 \text{ when } H_1 \text{ is true})$$

Let's expand the formula for the F-test:

$$\beta = Pr\left(F_{1-\alpha/2,n_1-1,n_2-1} \leq \frac{s_1^2}{s_2^2} \leq F_{\alpha/2,n_1-1,n_2-1} \mid \frac{\sigma_1^2}{\sigma_2^2} = \delta \neq 1\right)$$

If we multiply all the terms by σ_2^2/σ_1^2 we get:

$$\frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2} \sim F_{n_1-1,n_2-1}$$

If we substitute σ_2^2/σ_1^2 with the ratio we want to test, we get:

$$\beta = Pr\left(\frac{F_{1-\alpha/2,n_1-1,n_2-1}}{1.5} \leq \frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2} \leq \frac{F_{\alpha/2,n_1-1,n_2-1}}{1.5}\right)$$

Rearranging the terms we get:

$\rightarrow \delta$

$$\beta = Pr\left(\frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2} \leq \frac{F_{\alpha/2,n_1-1,n_2-1}}{1.5}\right) - Pr\left(\frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2} \leq \frac{F_{1-\alpha/2,n_1-1,n_2-1}}{1.5}\right)$$

Figure 2.54: 2nd type error F-test

```
ratio = 1.5 #variances ratio under H1
beta = stats.f.cdf(stats.f.ppf(1-alpha/2, dof1, dof2)/ratio, dof1, dof2) - stats.f.cdf(stats.f.ppf(alpha/2, dof1, dof2)/ratio, dof1, dof2)
print('the 2nd type error with delta %.3f for this test is: %.3f' % (ratio, beta))
power = 1 - beta
print('so the power of the test is: %.3f' % power)

#Operating characteristic curve
delta = np.linspace(0, 10, 100) #is a vector that measures the deviation
beta = stats.f.cdf(stats.f.ppf(1-alpha/2, dof1, dof2)/delta, dof1, dof2) - stats.f.cdf(stats.f.ppf(alpha/2, dof1, dof2)/delta, dof1, dof2)
power = 1 - beta
plt.plot(delta, power, label = "power of the test")
plt.xlabel("delta")
plt.ylabel("power")
plt.grid(True)
plt.legend()
plt.show()
```

Code 2.10: F-test 2nd type error (and trial of OC curve)

To conclude this part, always remember that for all the types of test seen so far the rejection or acceptance rules are always the same

REJECTION CRITERIA USING PERCENTILES

Accept the NULL hypothesis Ho if and only if the test statistic is in the acceptance region. If test statistic drops externally to the acceptance region, then reject Ho and accept the alternative hypothesis H1

REJECTION CRITERIA USING PVALUE

Accept the NULL hypothesis Ho if and only if Pvalue $>$ alpha. If Pvalue $<$ alpha reject Ho and accept the alternative hypotesis H1.

REJECTION CRITERIA USING CONFIDENCE INTERVALS

Accept the NULL hypothesis Ho if mu0 (hypothesized mean) drops within the confidence interval, otherwise reject Ho and accept H1.

2.3. Test of hypothesis for assumption compliance

2.3.1. Runs test

The Runs test is a test which try to quantitatively assess the randomness of the data. The null hypothesis value is given by the expected value of runs in that conditions (given "n" and "m"). It is always a two sided test, so be careful on how you compute the pvalue. It can do it manually or automatically, the codes are given below:

The expected number of runs, Y, is given by the formula:

$$E(Y) = \frac{2m(n-m)}{n} + 1$$

n is the number of observations

m is the number of +

Standard deviation of Y:

$$\sqrt{V(Y)} = \sqrt{\frac{2m(n-m)[2m(n-m)-n]}{n^2(n-1)}}$$

Normal approximation of a Poisson distribution:

$$Y \sim N(E(Y), V(Y))$$

Confidence interval:

$$E(Y) \pm z_{\alpha/2}\sqrt{V(Y)}$$

Test statistic under Ho
(follows a std normal)

$$z = \frac{R - E(R)}{\sigma_R}$$

R is the actual number of runs observed on the sample series

You can solve it as a normal Z test once you have been able to compute R and E(R)

- If the observed number of runs R falls within the confidence interval, do not reject the null hypothesis of randomness (there is not enough evidence to say that the observations are not random).
- If R falls outside the confidence interval, reject the null hypothesis, suggesting that the observations are not random.

Figure 2.55: Runs test structure)

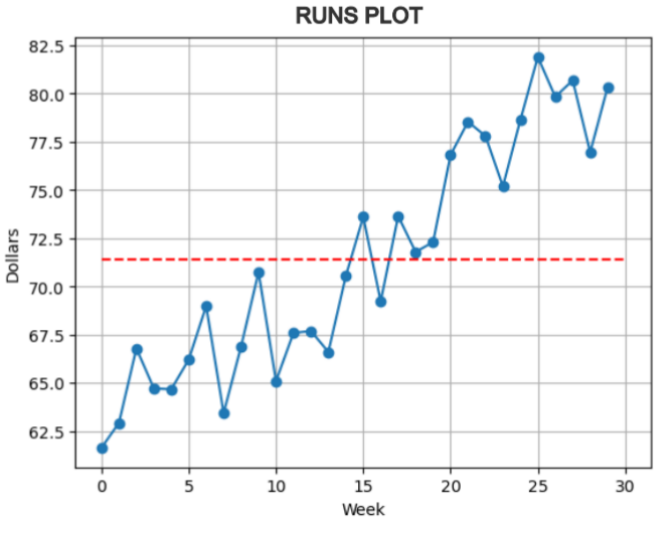


Figure 2.56: Runs test plot

```
# Plot the mean as well as the data
mean = df.mean()
n=len(df)
plt.figure(figsize=(15,5)) #eventually set the size
plt.plot(df['column name'], 'o-')
plt.hlines(mean, 0, n, colors='r', linestyles='dashed')
plt.xlabel('insert x label')
plt.ylabel('insert y label')
plt.title('Runs plot')
plt.grid()
plt.show()

# Get the number of points above the mean
mean = df.mean()
n = len(df)
m = np.sum(df > mean).values[0]
print('Number of points above the mean, m = %d' % m)
# Compute the number of runs
new_series = np.array(df - mean).flatten() # Count how many times the sign changes
runs = (np.sum(np.diff(np.sign(new_series)) != 0) + 1)
print('Number of runs runs = %d' % runs) #number of runs
#Expected number of runs
exp_runs= 2*m*(n-m)/n +1
print('Expected number of runs = %f' % exp_runs)
# Standard deviation of the number of runs
std_runs = np.sqrt((2*m*(n-m)*(2*m*(n-m)-n)/((n**2)*(n-1))))
print('Standard deviation of runs = %.03f' % std_runs)
#95% confidence interval
conf_int= stats.norm.interval(0.95, loc=exp_runs, scale=std_runs)
print('Confidence interval: (%.3f, %.3f)' % (conf_int[0], conf_int[1]))
if runs > conf_int[0] and runs < conf_int[1]:
     print("The series is considerable as random")
else:
     print("The series is not random")

# alternatively you can use critical region on the standard normal
alpha = 0.05
Z0 = (runs-exp_runs)/std_runs
print('the test statistic is: ', Z0)
Zalpha2 = stats.norm.ppf(1-alpha/2)
print('the acceptance region spans between [%.5f, %.5f]' % (-Zalpha2, Zalpha2))
if Z0 > -Zalpha2 and runs < Zalpha2:
     print("The series is considerable as random")
else:
     print("the series is not random")

# or as it is common to do, return the pvalue to get the randomness response
Z0 = (runs-exp_runs)/std_runs
pvalue_RT = 2*(1-stats.norm.cdf(abs(Z0)))
print('pvalue of the runs test is: ', pvalue_RT)
if pvalue_RT > alpha:
     print("The series is considerable as random")
else:
     print("the series is not random")
```

Code 2.11: Runs test codes for both plot and test

```
# Import the necessary libraries for the runs test
from statsmodels.sandbox.stats.runs import runstest_1samp
stat_runs, pval_runs = runstest_1samp(df['column name'], correction=False)
print('Runs test statistic = {:.3f}'.format(stat_runs))
print('Runs test p-value = {:.3f}'.format(pval_runs))
if pval_runs > 0.05:
     print("The series is considerable as random")
else:
     print("the series is not random")
```

Code 2.12: Code for automatic Runs Test

2.3.2. Barlett test

Barlett test is applied to check if there is autocorrelation at a specific lag. We want to estimate "rk" so the autocorrelation coefficient at lag "k".

The esimator is an unbiased estimator and it is called $\hat{\rho}_k$.

Note that when $\hat{\rho}_k$ is represented for different "k", the sample autocorrelation function shows up

- the null hypothesis Ho state assume no autocorrelation at lagk
- under Ho the data "Xt" are iid (independent and identically distributed)
- so under Ho, rk follows a normal distribution with mu = mu0 = 0 and variance given by the inverse of the sample size
- The test is two sided

Bartlett showed that for a random process (iid), the sample autocorrelation coefficient r_k approximately follows a normal distribution with: $E(r_k)=0$ and $Var(r_k)=1/n$ $\forall k$

A useful test with $H_0 : \rho_k = 0$ vs $H_1 : \rho_k \neq 0$

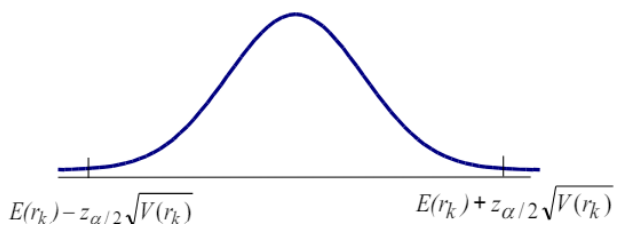
It is based on the rejection region: $|r_k| > \frac{z_{\alpha/2}}{\sqrt{n}}$

Rejection region:

$$r_k > 0 : r_k > \frac{z_{\alpha/2}}{\sqrt{n}}$$
$$r_k < 0 : r_k < -\frac{z_{\alpha/2}}{\sqrt{n}}$$
$$\Rightarrow |r_k| > \frac{z_{\alpha/2}}{\sqrt{n}}$$

If H_0 is true:

$r_k \sim N(0,1/n)$



Value that is often used

$Z_{\alpha/2} = 2 \rightarrow \alpha = 0.0456 \approx 0.05$

Figure 2.57: Barlett test for autocorrelation at a specific lag "k"

When conducting multiple analyses on the same dependent variable, the chance of committing a Type I error increases, thus increasing the likelihood of coming about a significant result by pure chance.
To correct for this, or protect from Type I error, a **Bonferroni correction** is conducted.

Assume we have N hypothesis tests (i=1,...,N)
- Each test has its own probability to reject H_{0i} when it is true - α_i
- Family-wise "first type" error: α'

The probability of **rejecting at least one** null hypothesis when **they are all true**

$\alpha' \leq \sum_{i=1,...,N} \alpha_i$ Bonferroni inequality

For **independent** tests, it can be shown that $1-\alpha' = \prod_{i=1,..N}(1-\alpha_i)$

If we set the same α for all the tests $\alpha_i = \alpha \quad \forall i = 1,...,N \Rightarrow \alpha' = 1-(1-\alpha)^N$

$\alpha = 1-(1-\alpha')^{1/N}$

We can build intervals to "constrain" the family error rate α (the overall first type error)

Choose the nominal family error rate alpha α'_{nom}

For each of the N tests to be performed (using the same set of data) choose

$$\alpha_i = \frac{\alpha'_{nom}}{N} \qquad \forall i = 1,...,N$$

You'll have an overall first type error $\alpha' \leq \sum_{i=1,...,N} \alpha_i = \alpha'_{nom}$

Figure 2.58: Bonferroni inequality

-Bartlett test (one lag):

$$|r_k| > \frac{z_{\alpha/2}}{\sqrt{n}}$$

-Bartlett test for *L different* lags (k=1,...,L): $\alpha_i = \frac{\alpha'_{nom}}{L}$

$$|r_k| > \frac{z_{\alpha'_{nom}/(2L)}}{\sqrt{n}} \qquad \forall k = 1,...,L$$

Figure 2.59: Barlett test stastic comparison with one or more lags (Bonferroni correction is applied)

```
# Import acf function from the libraries
from statsmodels.tsa.stattools import acf
n = len(df['column name'])
# autocorrelation function (nlags = int(np.sqrt(n)))
[acf_values, lbq, _] = acf(df['column name'], nlags = int(np.sqrt(n)), qstat=True, fft = False)
# autocorrelation function (nlags = int(n/3))
#[acf_values, lbq, _] = acf(df['columns name'], nlags = int(n/3) , qstat=True, fft = False)
#Bartlett's test
alpha = 0.05 #remember Bonferroni correction if you test more than one lag
lag_test = 1 #insert the lag you want to test
rk = acf_values[lag_test]
z_alpha2 = stats.norm.ppf(1-alpha/2)
print('rk = %f' % rk)
print('Rejection region starts at %f' % (z_alpha2/np.sqrt(n)))
if abs(rk)>z_alpha2/np.sqrt(n):
     print('The null hypothesis is rejected')
else:
     print('The null hypothesis is accepted')
```

Code 2.13: Code for AUTOCORRELATION FUNCTION and BARLETT TEST

2.3.3. Ljung Box Pierce or LBQ test

The LBQ test carry out the same job of the Barlett test computed on more lags.

BE CAREFULL: the degrees of freedom of the this chi2 is exactly the number of lags input

Moreover, note that it is a unilateral upper bounded test

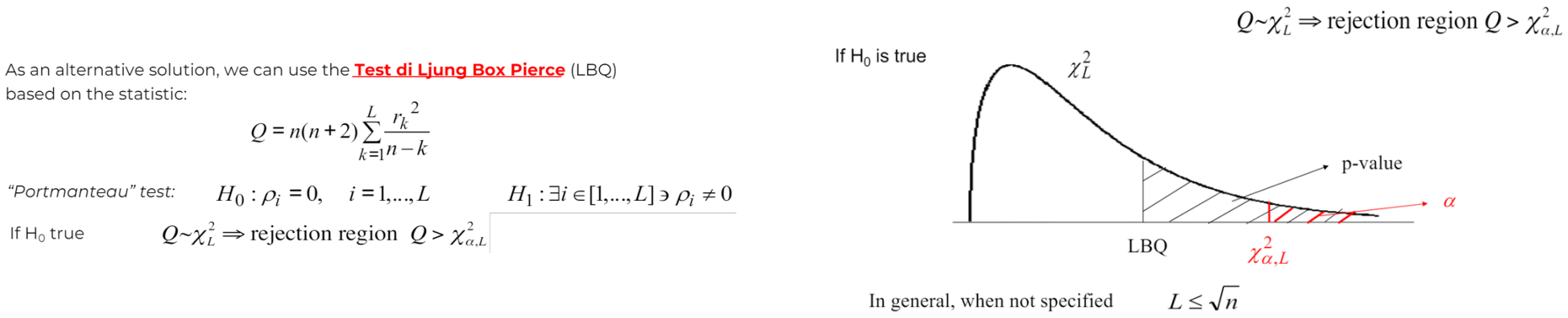


Figure 2.60: LBQ test

```
# Import acf function from the libraries
from statsmodels.tsa.stattools import acf
n = len(df['column name']) #DO NOT FORGET TO STATE IT
alpha = 0.05 #(do not change it)
# autocorrelation function (nlags = int(np.sqrt(n)))
[acf_values, lbq, _] = acf(df['column name'], nlags = int(np.sqrt(n)), qstat=True, fft = False)
# autocorrelation function (nlags = int(n/3))
#[acf_values, lbq, _] = acf(data, nlags = int(n/3) , qstat=True, fft = False)
# Generally speaking: how many lags? Rule of thumb: L<sqrt(n) but the exercicer sometimes uses n/3
lag_test = int(np.floor(np.sqrt(len(df['column name'])))) #int(np.floor()) round the outcome at the lower integer
Q0_LBQ = lbq[lag_test-1]
print('Q0_LBQ = %f' % Q0_LBQ)
# Rejection region for chi square distribution
dof = lag_test
chi2_alfa= stats.chi2.ppf(1-alpha,dof)
print('Rejection region starts at %f' % chi2_alfa)
if Q0_LBQ>chi2_alfa:
     print('The null hypothesis is rejected')
else:
     print('The null hypothesis is accepted')
# Compute the p-value for the LBQ test
pval = 1 - stats.chi2.cdf(Q0_LBQ, lag_test)
print('p-value = %f' % pval)
```

Code 2.14: Code for AUTOCORRELATION FUNCTION and LBQ test

```
#LBQ test for autocorrelation
from statsmodels.stats.diagnostic import acorr_ljungbox
lag_test = int(np.floor(np.sqrt(len(df['column name']))))
lbq_test = acorr_ljungbox(df['column name'], lags=[lag_test], return_df=True)
print('LBQ test statistic at lag %d = %f' % (lag_test, lbq_test.loc[lag_test,'lb_stat']))
print('LBQ test p-value at lag %d = %f' % (lag_test, lbq_test.loc[lag_test,'lb_pvalue']))
```

Code 2.15: Code for AUTOMATIC LBQ test

3 | Time series modeling: linear models and ARIMA

Distributional models differs from linear models for several reasons.

Distributional models

- These models are most useful when the process is stationary, meaning that its statistical properties (such as mean and variance) do not change over time.
- The goal is to understand the distribution of the data points. This involves looking at the probability distribution and making inferences based on this distribution.
- If the process is stationary you can use them to predict future observations
- If the process is stationary you can use them to create confidence intervals on parameters such as mean, variances and so on
- Suitable for processes where the primary interest is in the distributional properties rather than temporal dependencies. These are often used in quality control, reliability engineering, and risk management.

Linear models

- Time series models are required when the process has a drift or non-stationarity, meaning that its statistical properties change over time.
- The goal is to understand and model the temporal dependencies in the data. This involves identifying patterns such as trends, seasonal effects, and autocorrelations. Examples include simple time based correlation models, ARIMA (AutoRegressive Integrated Moving Average), Exponential Smoothing and so on
- The output includes forecasts and predictions of future data points, taking into account the temporal structure of the data.

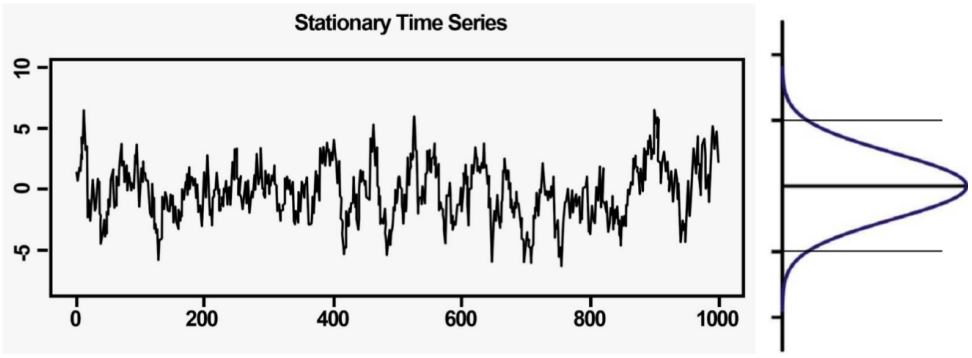


Figure 3.1: Example on distributional model application)

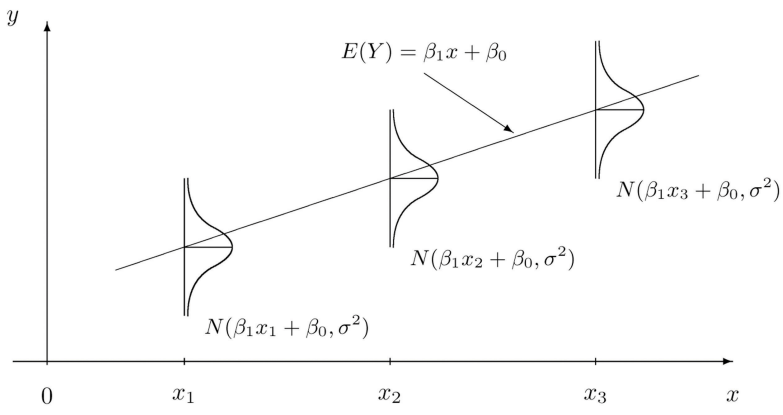


Figure 3.2: Example on linear model application

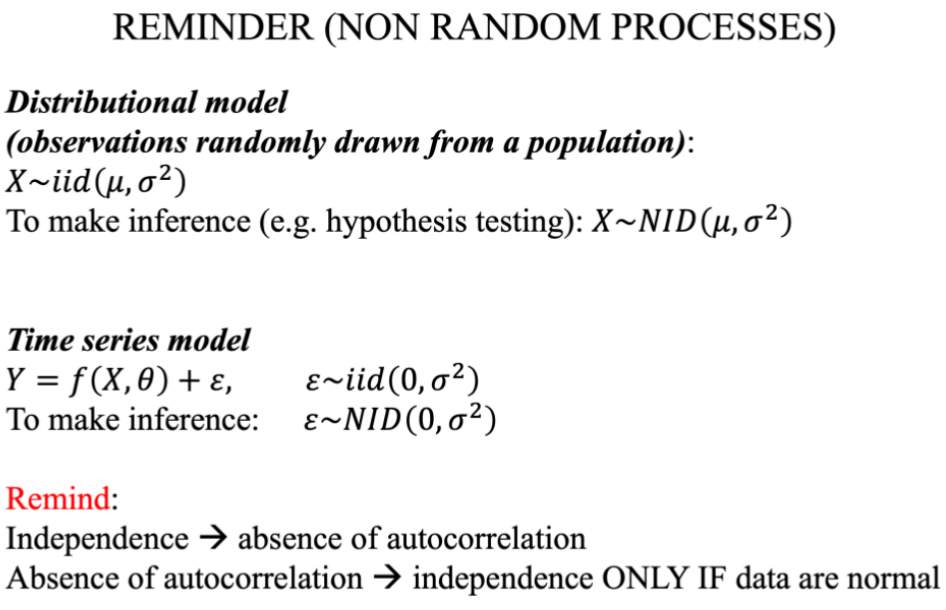


Figure 3.3: Reminder (non random processes)

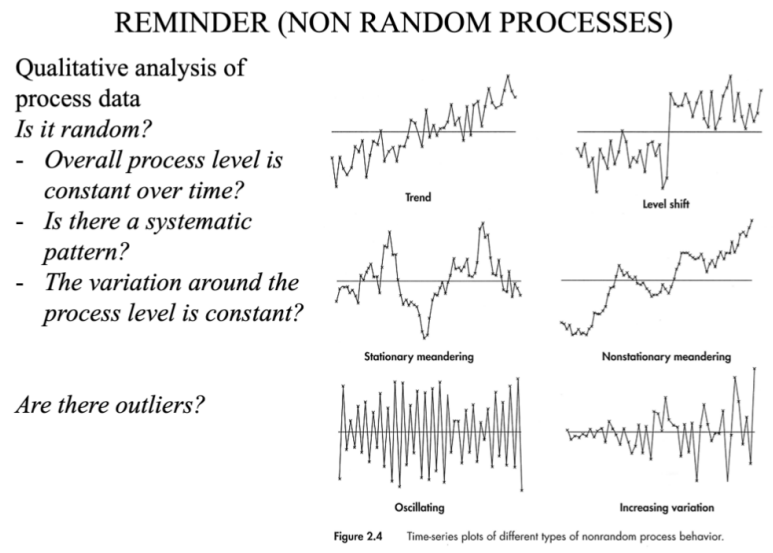


Figure 3.4: Main types of non random processes

3.1. How to create and assess linear models

Linear regression models are a fundamental statistical method for modeling relationships between a dependent variable and one or more independent variables. The regressors can be one, few or severals depending on the scope.

We can look for the model/coefficients to minimize the Sum of Squared Errors (Minimum Mean Squared Error – MSE – approach).

$$Y_t = \mu_t + \varepsilon_t \tag{3.1}$$

From this model, we take new data y_t for $t = 1, \ldots, n$.

Structured data means that the sampling process is structured: take the same time interval between two data points.

Reality

$$\text{SSE} = \sum_{t=1}^{n}(y_t - \mu_t)^2 = \sum_{t=1}^{n}\varepsilon_t^2 \tag{3.2}$$

- **Observed data:** y_t
- **Unknown mean (deterministic):** μ_t
- **Sum of the squared errors (or noises):** $\sum_{t=1}^{n}\varepsilon_t^2$

Esteem

$$\hat{\text{SSE}} = \sum_{t=1}^{n}(y_t - \hat{\mu}_t)^2 = \sum_{t=1}^{n}\hat{\varepsilon}_t{}^2 \tag{3.3}$$

- $\hat{\mu}_t$ is the estimated mean.
- $\hat{\varepsilon}_t$ are the estimated residuals.

Why we estimate the SSE:

The Sum of Squared Errors (SSE) quantifies the discrepancy between the observed data and the model's predictions. In practice, the true mean μ_t is often unknown and must be estimated from the data. By minimizing the SSE, we find the best-fitting model that reduces the error between the observed values and the values predicted by the model. This estimation process helps in understanding the underlying data structure and in making predictions on new data points.

For more detailed theory on linear models go through the QDA lectures' files. NOTE: in the following notation "k" is the number of regressors in the equation (so the number of independent variables) while "p" is the number of coefficients (included the the intercept, so the constant not linked to any regressor)

Short reminder about regression

$$\boldsymbol{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim iid\left(\mathbf{0}, \sigma^2\mathbf{I}\right) \ ^*$$

$\boldsymbol{y}$ is $nx1$
$\mathbf{X}$ is nxp (p = number of parameters), e.g.: $y = \beta_0 + \beta_1 x + \varepsilon \rightarrow$ p=2
$\boldsymbol{\beta}$ is $px1$
$\boldsymbol{\varepsilon}$ is $nx1$

$$\mathbf{X} = \begin{bmatrix} \mathbf{1} & \boldsymbol{x_{11}} & \cdots & \boldsymbol{x_{1k}} \\ \cdots & \cdots & \cdots & \cdots \\ \mathbf{1} & \boldsymbol{x_{n1}} & \cdots & \boldsymbol{x_{nk}} \end{bmatrix} \qquad p = k + 1$$

$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y}$ Estimated parameters $\quad \mathrm{E}(\widehat{\boldsymbol{\beta}}) = \boldsymbol{\beta}$, $\mathrm{V}(\widehat{\boldsymbol{\beta}}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$

$\widehat{\boldsymbol{y}} = \mathbf{X}\widehat{\boldsymbol{\beta}}$ Estimated response

$\widehat{\sigma^2} = MS_E$ Estimated variance

$\boldsymbol{e} = \boldsymbol{y} - \widehat{\boldsymbol{y}}$ Residuals

* But to make inference we need normality assumption (NID)

Figure 3.5: Short recap on linear models)

```
REGRESSION EQUATION
-------------------
Ex4 =  + 0.334 const + 0.555 lag1

COEFFICIENTS
------------
 Term    Coef  SE Coef  T-Value  P-Value
const 0.3343   0.1108   3.0171   0.0050
 lag1 0.5549   0.1471   3.7734   0.0007

MODEL SUMMARY
-------------
     S   R-sq  R-sq(adj)
0.0509 0.3079     0.2863

ANALYSIS OF VARIANCE
--------------------
    Source   DF  Adj SS  Adj MS  F-Value  P-Value
Regression  1.0  0.0369  0.0369  14.2383   0.0007
     const  1.0  0.0236  0.0236   9.1029   0.0050
      lag1  1.0  0.0369  0.0369  14.2383   0.0007
     Error 32.0  0.0829  0.0026      NaN      NaN
     Total 33.0  0.1198     NaN      NaN      NaN
```

Figure 3.6: Output sm.OLS(y,x).fit()

In linear regression, the total variability in the dependent variable y can be decomposed into the variability explained by the regression model and the unexplained variability. This is known as the partition of variance.

Partition of the Variance

$$SS_{TOT} = SS_{REG} + SS_E$$

Degrees of Freedom

$$(n-1) = (p-1) + (n-p)$$

Definition of the Sum of Squares

$$SS_{TOT} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

$$SS_{REG} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

$$SS_E = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Here:

- SS_{TOT} is the total sum of squares, representing the total variability in the observed data.
- SS_{REG} is the regression sum of squares, representing the variability explained by the regression model.
- SS_E is the error sum of squares, representing the unexplained variability.

Applying ANOVA in Linear Regression

In the context of linear regression, the total variability in the dependent variable y can be decomposed into two parts: the variability explained by the regression model (regression sum of squares, SS_{REG}) and the unexplained variability (error sum of squares, SS_E). This is known as the partition of variance.

Analysis of Variance (ANOVA) is used in linear regression to compare the variability explained by the model to the unexplained variability. The ANOVA table provides F-tests to determine whether the overall regression model is statistically significant.

The F-statistic in ANOVA is calculated as the ratio of the mean regression sum of squares to the mean error sum of squares, and it tests the null hypothesis that all regression coefficients are equal to zero against the alternative that at least one is not. Here we provide the code for the regression model and residuals checks First of

all you need to import the necessary library statsmodel.api and the qda package for the summary of the model. You need to prepare the column(s) in which you store the regressor(s). This column can be "time" (expressed by the indexes of the observations), can be a lagged version of the original observations (in case of ARIMA) or it can even be a boolean dummy variable, used as reggressor in case of special causes.

When you have only one regressor (column) you can store it in a vector called "x", while the observations are stored in another vector called "y", for simplicity.

DO NOT FORGET TO ADD THE CONSTANT IN YOUR MODEL. You can avoid it if you are 100% sure it won't be significant. "y" will be a series while "x" will be a dataframe with both the columns of the constant and the regressor(s)

Then you are ready to build up your fitting model.

```
# suppose you have already built the regressor vector "x" and the vector of the esteems (so the fitted values) "y"
# you can always play on subset of your original data by setting a = first obs index, and b = last obs index
# if you don't need a subset avoid the brackets, if you need to fit from "a" onwards, then erase b: [a:]

# import libraries and package
import statsmodels.api as sm
import qda

# add a constant on your x vector
x = sm.add_constant(df['regressor column name'][a:b])
# in case "x" was made of more than one regressor
x = sm.add_constant(x)

# run your model
model = sm.OLS(y, x).fit()

# show the results
qda.summary(model)

# RESIDUALS CHECKS (graphs + Shapiro Wilk test)
fig, axs = plt.subplots(2, 2, figsize = (10, 8))
fig.suptitle('Residual Plots')
axs[0,0].set_title('Normal probability plot')
stats.probplot(model.resid, dist="norm", plot=axs[0,0])
axs[0,1].set_title('Versus Fits')
axs[0,1].scatter(model.fittedvalues, model.resid)
fig.subplots_adjust(hspace=0.5)
axs[1,0].set_title('Histogram')
axs[1,0].hist(model.resid)
axs[1,1].set_title('Time series plot')
axs[1,1].plot(np.arange(1, len(model.resid)+1), model.resid, 'o-')

_, pval_SW_res = stats.shapiro(model.resid)
print('Shapiro-Wilk test p-value on the residuals = %.3f' % pval_SW_res)

# RESIDUALS CHECKS (randomnsess + ACF/PACF)
#RANDOMNESS OF RESIDUALS
from statsmodels.sandbox.stats.runs import runstest_1samp
_, pval_runs_res = runstest_1samp(model.resid, correction=False)
print('Runs test p-value on the residuals = {:.3f}'.format(pval_runs_res))
#THE SIZE TO BE USED IN RESIDUALS CHECK IS: len(df) - shift
fig, ax = plt.subplots(2, 1)
sgt.plot_acf(model.resid, lags = int(len(model.resid)/3), zero=False, ax=ax[0])
fig.subplots_adjust(hspace=0.5)
sgt.plot_pacf(model.resid, lags = int(len(model.resid)/3), zero=False, ax=ax[1],method = 'ywm')
plt.show()
```

Code 3.1: AUTOMATIC ORDINARY LEAST SQUARE LINEAR MODEL FITTING + RESIDUALS CHECK

Even though it has been highlighted also in the code, I want to remark that

This is particularly crucial when Barlett and/or LBQ tests are applied on residuals autocorrelation. Moreover, the ACF/PACF plots requires "n" as well, so also there but the right number of residuals!**the more you shift the dataset, the less is the number of observation on which you fit, so less is the number of residuals. Whenever you need to check the residuals assumption use the proper size "n". It will be given by the lenght of the original dataset minus the largest shift carried out.**

This is particularly crucial when Barlett and/or LBQ tests are applied on residuals autocorrelation. Moreover, the ACF/PACF plots requires "n" as well, so also there but the right number of residuals!

3.1.1. Test of hypothesis on the significance of the simple linear regression coefficients

Test for individual coefficients

- $H_0 : \beta_i = 0$
- $H_1 : \beta_i \neq 0$

Test statistic: $T_0 = \frac{\hat{\beta}_i}{se(\hat{\beta}_i)} = \frac{\hat{\beta}_i}{\sqrt{\hat{\sigma}^2 C_i}}$

Where C_i is the i-th diagonal element of the $(\mathbf{X}^T\mathbf{X})^{-1}$ matrix.

If H_0 is true: $T_0 \sim t_{n-p}$

Reject H_0 if: $|T_0| > t_{\alpha/2,n-p}$

The t-statistic and the corresponding p-value is displayed in the **COEFFICIENTS** table.

Figure 3.7: Test of hypothesis on the significance of a regressor)

Test for significance of regression

- $H_0 : \beta_1 = \beta_2 = \beta_k = 0$
- $H_1 : \exists \beta_i \neq 0$

If H_0 is true:

- $\frac{SS_{REG}}{\sigma^2} \sim \chi^2_{p-1}$
- $\frac{SS_E}{\sigma^2} \sim \chi^2_{n-p}$

then:

$$F_0 = \frac{SS_{REG}/(p-1)}{SS_E/(n-p)} = \frac{MS_{REG}}{MS_E} \sim F_{p-1,n-p}$$

Reject H_0 if: $F_0 > F_{\alpha,p-1,n-p}$

The F-statistic and the corresponding p-value is displayed in the **ANALYSIS OF VARIANCE** table.

Figure 3.8: Test of hypothesis on the whole regression model significance

Once you now if your coefficient is significant or not you can create a confidence interval for it. It is possible to code it or do it automatically with the proper function

CONFIDENCE INTERVAL FOR INDIVIDUAL COEFFICIENT

$$t_0 = \frac{\hat{\beta}_i - \beta_i}{se(\hat{\beta}_i)} \sim t_{n-p} \quad \text{Student - t with } n\text{-}p \text{ dof}$$

$$-t_{\alpha/2,n-p} \leq \frac{\hat{\beta}_i - \beta_i}{se(\hat{\beta}_i)} \leq t_{\alpha/2,n-p}$$

$$\hat{\beta}_i - t_{\alpha/2,n-p} se(\hat{\beta}_i) \leq \beta_i \leq \hat{\beta}_i + t_{\alpha/2,n-p} se(\hat{\beta}_i)$$

Figure 3.9: Confidence interval on coefficients

```
# script for manual confidence interval on a regressor coefficient
# beta1 if you test the first regressor coefficient, otherwise modify it to let it make sense
# the regressor name is the same name of the column in the "x" dataframe
beta1 = model.params['regressor name'] # regressor of which you want to test the coefficient
print('The estimated coefficient beta1 is %.3f' % beta1)
se_beta1 = model.bse['lag1']
print('The standard error of the estimated coefficient beta1 is %.3f' % se_beta1)
alpha = 0.05
n = len(df)
t_alpha2 = stats.t.ppf(1-alpha/2, n-2)
CI_beta1 = [beta1 - t_alpha2*se_beta1, beta1 + t_alpha2*se_beta1]
print('The confidence interval for beta1 is [%.3f, %.3f]' % (CI_beta1[0], CI_beta1[1]))

# script for automatic confidence interval on a regressor coefficient
# beta1 if you test the first regressor coefficient, otherwise modify it to let it make sense
# the regressor name is the same name of the column in the "x" dataframe
CI_beta1 = model.conf_int(alpha=0.05).loc['regressor name'] # regressor name of which you want to test the coefficient
print('The confidence interval for beta1 is [%.3f, %.3f]' % (CI_beta1[0], CI_beta1
```

Code 3.2: Confidence interval on a regressor coefficient

3.1.2. Confidence and prediction intervals respectively for the mean and single next process outcome

Confidence intervals

$L \leq \theta \leq U$ such that $P(L \leq \theta \leq U) = 1 - \alpha$

is called $100(1-\alpha)\%$ confidence interval for the (unknown) parameter θ

Interpretation: if, in repeated random samplings, a large number of such intervals is constructed, $100(1-\alpha)\%$ of them will contain the true value of θ.

Prediction intervals

$L \leq Y \leq U$ such that $P(L \leq Y \leq U) = 1 - \alpha$

is called $100(1-\alpha)\%$ prediction interval for the future process outcome Y

Interpretation: if, in repeated random samplings, a large number of such intervals is constructed, $100(1-\alpha)\%$ of them will contain the future outcome Y.

Figure 3.10: Confidence and prediction intervals definition)

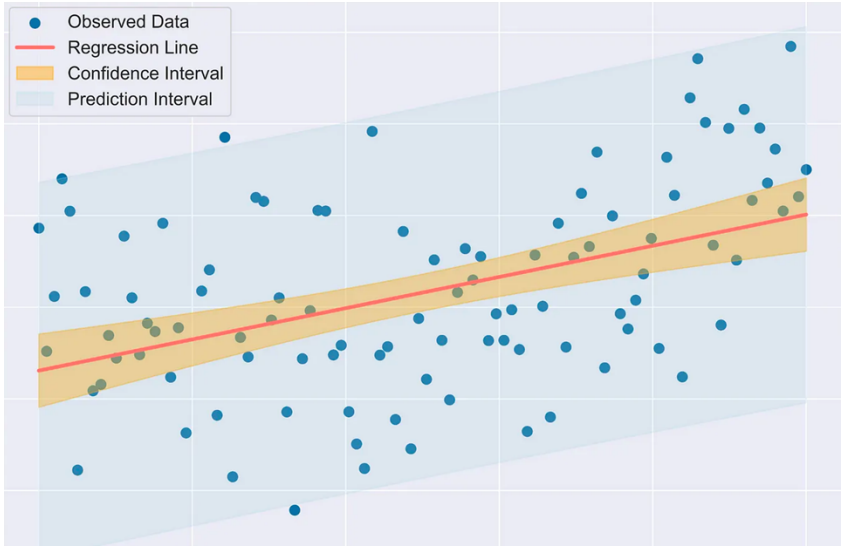


Figure 3.11: Confidence and prediction intervals on a linear model

Confidence interval (mean response)

$$\hat{Y} \pm t_{\alpha/2}, n-p \sqrt{\hat{\sigma}_\varepsilon^2 \left[\frac{1}{n} + \frac{(X^* - \bar{X})^2}{(n-1)S_X^2}\right]}$$

Prediction interval

$$\hat{Y} \pm t_{\alpha/2}, n-p \sqrt{\hat{\sigma}_\varepsilon^2 \left[1 + \frac{1}{n} + \frac{(X^* - \bar{X})^2}{(n-1)S_X^2}\right]}$$

$\hat{Y}$: new process outcome computed with the regression equation on the last observation

$S_Y^2 = \hat{\sigma}_\varepsilon^2 = MS_E$: is the mean square error, so the variance of the residuals

$\bar{X}$: sample mean of the regressor column overtime

X^* : this is the last observation, which is the last value in your ORIGINAL dataframe

S_X^2 : variance of the regressor vector

Figure 3.12: Confidence and prediction intervals on the next observation

```
last_obs = df['column name'].iloc[-1] #pick up the index or the value of your last observation
print('last observation was: ',last_obs) #eventually change the prints to make it more meaningful

# next observation prediction
# model.predict([constant, last value first regressor]
Yhat = model.predict([1,last_obs]) #do not change 1, as it is linked with the constant: beta0*1
print('the next observation would be: ', Yhat)

# compute parameters for confidence and prediction intervals
p = len(model.model.exog_names)
n = len(df)
print('the number of observation is: %d, while the number of coefficients is: %d' % (n,p))
alpha = 0.05
MSE = np.var(model.resid, ddof=p)
print('mean square error: ',MSE)
# sample mean and sample variance of the first regressor vector (NOT THE ORIGINAL VARIABLE!!!)
Xbar = df['regressor column name'].mean()
# alternatively Xbar = x.iloc[0:,1].mean()
print('Sample mean of the regressor vector: ', Xbar)
S2_x = df['regressor column name'].var()
# alternatively S2_x = x.iloc[0:,1].var()
print('Sample variance of the regressor vector: ', S2_x)
# compute the critical value of the two sided t-test
t_alpha2 = stats.t.ppf(1-alpha/2, n-p)
print('t alpha/2 = ', t_alpha2)

# Compute the confidence interval
CI = [Yhat - t_alpha2*np.sqrt(MSE*(1/n + ((last_obs - Xbar)**2)/((n-1)*S2_x))), Yhat + t_alpha2*np.sqrt(MSE*(1/n + ((last_obs -
    Xbar)**2)/((n-1)*S2_x)))]
print('The confidence interval for the mean response is [%.3f, %.3f]' % (CI[0], CI[1]))
# Compute the prediction interval
PI = [Yhat - t_alpha2*np.sqrt(MSE*(1 + 1/n + ((last_obs - Xbar)**2)/((n-1)*S2_x))), Yhat + t_alpha2*np.sqrt(MSE*(1 + 1/n + ((last_obs -
    Xbar)**2)/((n-1)*S2_x)))]
print('The prediction interval for the next value is [%.5f, %.5f]' % (PI[0], PI[1]))
```

Code 3.3: Prediction next observation + confidence interval + prediction interval on it

That was the manual computation of the confidence and prediction interval for an autoregressive model of order 1. First of all it is possible to use the automatic command: model.get_prediction().summary_frame(alpha=0.05) do that. Secondly, it is possible to create prediction intervals also for more complicated model (more regressors) by giving as input an array (or a DataFrame) with the values that should be insert in the equation model.

```
# New data point for which we want to predict
# Replace these values with actual future values
new_data = pd.DataFrame({
    'const': [1],  # intercept term, if not significant erase it
    'predictor1': [0.8],
    'predictor2': [0.3],
    'predictor3': [0.5]
})

# Get the prediction
prediction = model.get_prediction(new_data)
prediction_summary = prediction.summary_frame(alpha=0.05)

# Extract predicted value and confidence intervals
predicted_value = prediction_summary['mean'][0]
mean_ci_lower = prediction_summary['mean_ci_lower'][0]
mean_ci_upper = prediction_summary['mean_ci_upper'][0]
obs_ci_lower = prediction_summary['obs_ci_lower'][0]
obs_ci_upper = prediction_summary['obs_ci_upper'][0]

print(f"Predicted next value: {predicted_value}")
print(f"95% confidence interval for the mean prediction: [{mean_ci_lower}, {mean_ci_upper}]")
print(f"95% prediction interval: [{obs_ci_lower}, {obs_ci_upper}]")
```

Code 3.4: Automatic prediction next observation + CI + PI in case of more predictors models

When working with an unstationary dataset, we first need to differentiate the data to achieve stationarity. Once we have differentiated the dataset, we fit our model on this transformed data. To make predictions for new observations, we use the fitted model to forecast the next value of the differenced series. This involves providing the last observed differenced value and any relevant predictors (such as lagged values and dummy variables) as input to the model. The `get_prediction()` method can then be used to obtain the predicted differenced value and the associated confidence and prediction intervals.

After obtaining the predicted differenced value and intervals, we must transform these predictions back to the original scale of the time series. This is done by adding the predicted differenced value to the last actual observation from the original series. Similarly, we adjust the confidence and prediction intervals by adding the last actual value to the lower and upper bounds of these intervals. This final step ensures that the predictions and intervals are correctly interpreted in the context of the original unstationary data.

The following is an example coming from a Simulation exam where the fitted model was an ARIMA(1,1,0) with dummy variable It is provided to help you with the predictions in case of DIFFERENTIATION: UNSTATIONARY AR

```
# Use the last actual value for making predictions
last_actual_value = df['column of the original variable'].iloc[-1]
# You should have created a column called "diff1" in your original dataset to reach stationarity
last_lag1_value = df['diff1'].iloc[-1]

# Prepare the input for prediction
next_dummy_value = 0  # assuming the dummy for the next value is 0
input_data = np.array([last_lag1_value, next_dummy_value]).reshape(1, -1)

# Compute the prediction
prediction_df = model.get_prediction(input_data).summary_frame(alpha=0.05)

# Get the predicted differenced value and its intervals
predicted_diff = prediction_df['mean'][0]
mean_ci_lower = prediction_df['mean_ci_lower'][0]
mean_ci_upper = prediction_df['mean_ci_upper'][0]
obs_ci_lower = prediction_df['obs_ci_lower'][0]
obs_ci_upper = prediction_df['obs_ci_upper'][0]

# Transform back to the original scale
predicted_value = last_actual_value + predicted_diff
mean_ci_lower_original = last_actual_value + mean_ci_lower
mean_ci_upper_original = last_actual_value + mean_ci_upper
obs_ci_lower_original = last_actual_value + obs_ci_lower
obs_ci_upper_original = last_actual_value + obs_ci_upper
# The computations right above allows you to jump from the prediction of the next difference, to the true next prediction
print(f"Predicted next value: {predicted_value}")
print(f"95% confidence interval for the mean prediction: [{mean_ci_lower_original}, {mean_ci_upper_original}]")
print(f"95% prediction interval: [{obs_ci_lower_original}, {obs_ci_upper_original}]")
```

Code 3.5: Automatic prediction next observation + CI + PI in case of ARIMA(1,1,0)

Follow this example

```
# Load and prepare the new data
new_data = pd.read_csv('oxygen_phase2.csv')
# Create the lag1 variable for the new data
```

```
new_data['lag1'] = new_data['val'].shift(1)
# Create the dummy variable for the new data and set them properly to fit your needs (even all zero)
new_data['dummy'] = np.zeros(len(new_data))
# Ensure the new_data has the same columns as the training data
new_data = new_data.iloc[1:, :] # drop the first row with NaN in lag1
new_data.head()

# Select the features
X_new = new_data[['layer', 'lag1', 'dummy']]
# Add constant term to the predictors if the model has it
X_new = sm.add_constant(X_new)
# Make predictions
predictions = model.predict(X_new)
new_data['predictions'] = predictions
# Compute the residuals
new_data['residuals'] = new_data['val'] - new_data['predictions']
# Display the residuals
print(new_data[['val', 'predictions', 'residuals']])
```

Code 3.6: How to get a vector of future prediction and residuals computation with new data

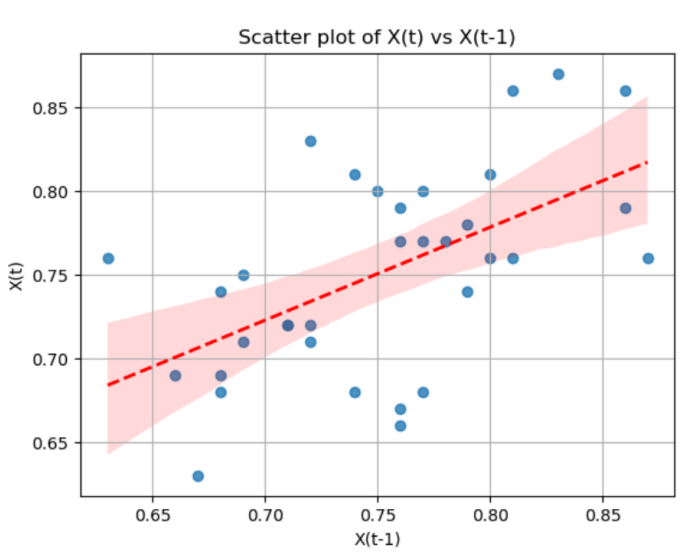


Figure 3.13: Time series fitting: scatter plot with confidence interval for next mean process outcome

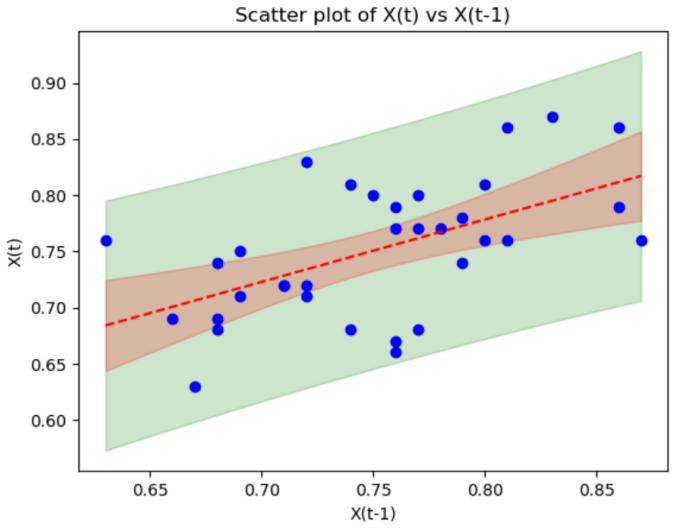


Figure 3.14: Time series fitting: scatter plot with confidence and prediction intervals for the next mean and single process outcome

```
# Scatter plot + trend line + confidence interval (NOTE: CI is expressed in percentage, so do not put 0.95!)
sns.regplot(x='column name', y='column name', data=df, fit_reg=True, ci=95, line_kws={'color': 'red', 'lw': 2, 'ls': '--'}) #modify CI if needed
plt.title('Scatter plot of X(t) vs X(t-1)')
plt.xlabel('X(t-1)')
plt.ylabel('X(t)')
plt.grid()

# Scatter plot + trend line + confidence interval + prediction interval
# get the range of values for the regressor;
# I have not written "column name" instead of lag1" just to let you note that you have to insert the same column name, but input the right column name
    you have chosen.
x_range = np.linspace(df['lag1'].min(), df['lag1'].max(), 100)
# add a constant to the regressor
x_range = sm.add_constant(x_range)
# get the prediction interval for each value of the regressor
prediction_df = model.get_prediction(x_range).summary_frame(alpha=0.05) #CHANGE ALPHA IF REQUESTED
# plot the data and the intervals. Remeber if the process was autocorrelated put x='lag1', y='column of original data'
sns.regplot(x='column name', y='column name', data=df, fit_reg=True, ci=insert the value, line_kws={'color': 'red', 'lw': 2, 'ls': '--'})
plt.fill_between(x_range[:,1], prediction_df['obs_ci_lower'], prediction_df['obs_ci_upper'], color='green', alpha=0.2)
plt.title('insert title')
plt.xlabel('insert lable')
plt.ylabel('insert lable')
plt.show()

# alternatively this is how to "manually" compute both confidence and prediction interval in 1 plot
x_range = np.linspace(df['regressor column name'].min(), df['same regressor column name'].max(), 100)
# add a constant to the regressor
x_range = sm.add_constant(x_range)
# get the prediction interval for each value of the regressor
prediction_df = model.get_prediction(x_range).summary_frame(alpha=0.05)
# plot the scatter separately from the trend line
plt.plot(df['regressor column'], df['orginal variable column name'], 'o', color='blue', label='Original data')
plt.plot(x_range[:,1], prediction_df['mean'], '--', color='red', label='Fitted values')
#plot confidence and prediction interval.
plt.fill_between(x_range[:,1], prediction_df['obs_ci_lower'], prediction_df['obs_ci_upper'], color='green', alpha=0.2)
plt.fill_between(x_range[:,1], prediction_df['mean_ci_lower'], prediction_df['mean_ci_upper'], color='red', alpha=0.2)
plt.title('insert title')
plt.xlabel('insert lable')
plt.ylabel('insert lable')
plt.show()
```

Code 3.7: CONFIDENCE AND PREDICTION INTERVAL plotted on a time series model

3.1.3. STEPWISE REGRESSION APPROACH

Stepwise regression is a method used in statistical modeling to select a subset of predictor variables for a regression model. It is particularly useful when you have a large number of potential predictors and you want to find a more parsimonious model that still explains the data well. The goal is to find the most significant variables while eliminating those that do not contribute meaningfully to the model.

Therefore apply this method instead of the simple "ordinary least square" when you need to chose between more predictors. As mentioned, they can be of several types such as: year (or time in general), lagged versions of the original data, dummy variables and so on.

The following image shows an example of time series which was both autocorrelated (lag1 and lag4), and correlated with time (trend behaviour). Moreover in the right pictures is shown how our model fit the original data.

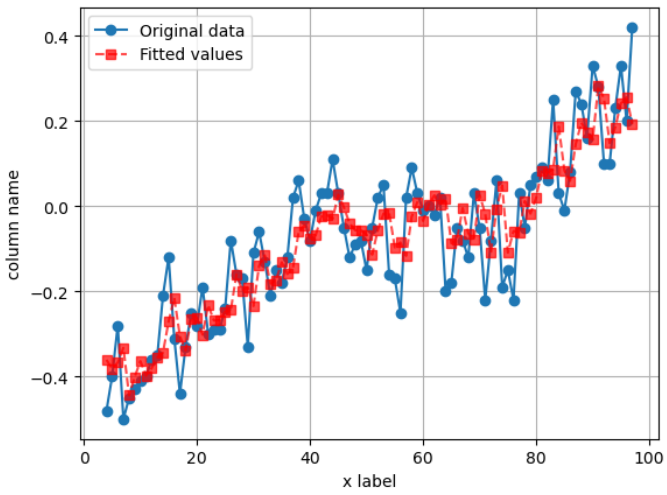


Figure 3.15: Original vs Step-wise regression model fits

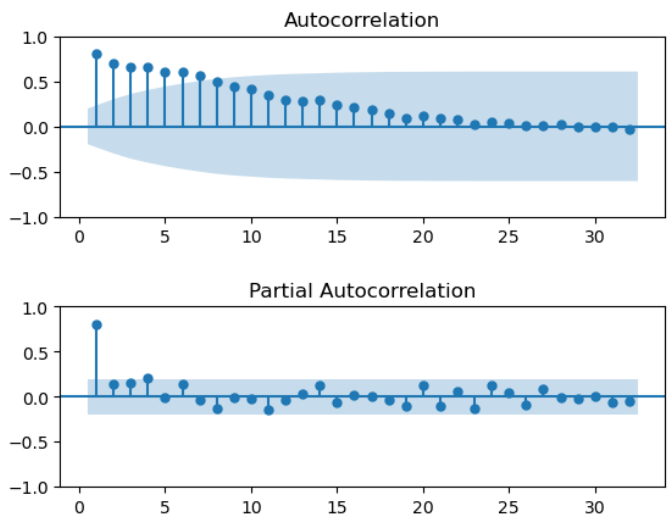


Figure 3.16: Example of ACF + PACF of a model solved by stepwise regession

Note: by looking at ACF and PACF, plus considering the instationary of the series, we can say for sure there is an autocorrelation pattern within the data. Now there are two way to go: try to fit a simple AR(1) + trend or, as we will introduce with ARIMA, differentiate the original dataset (due to linear decay which highlights an unstable pattern) and then proceed with further analysis. Well there is not always only one single model which can fit your data, sometimes it's crucial to make some trials. To conclude, this series was correctly modeled by including as regressors the years, lag1 and lag4 version of the original data, so it's up to the model designer to decide which path its analysis has to undertake.

Here the dataframe handling gets slightly harder, but let's go step by step. First of all, let's make an introduction on how step-wise regression method is structured.

STEP #1

1. Specify an **Alpha-to-Enter** ($\alpha_E = 0.15$) significance level.
2. Specify an **Alpha-to-Remove** ($\alpha_R = 0.15$) significance level

STEP #2

1. Fit each of the one-predictor models, that is, regress y on x_1, regress y on x_2, … regress y on x_{p-1}.
2. The first predictor put in the stepwise model is the predictor that has the **smallest *t*-test *P*-value** (below $\alpha_E = 0.15$).
3. If no P-value < 0.15, stop.

STEP #3

1. Suppose x_1 was the "best" predictor.
2. Fit each of the two-predictor models with x_1 in the model, that is, regress y on (x_1, x_2), regress y on (x_1, x_3), …, and y on (x_1, x_{p-1}).
3. The second predictor put in stepwise model is the predictor that has the **smallest *t*-test *P*-value** (below $\alpha_E = 0.15$).
4. If no P-value < 0.15, stop.

STEP #4

1. Suppose x_2 was the "best" second predictor.
2. Step back and check P-value for $\beta_1 = 0$. If the P-value for $\beta_1 = 0$ has become not significant (above $\alpha_R = 0.15$), remove x_1 from the stepwise model.

STEP #5

1. Suppose both x_1 and x_2 made it into the two-predictor stepwise model.
2. Fit each of the three-predictor models with x_1 and x_2 in the model, that is, regress y on (x_1, x_2, x_3), regress y on (x_1, x_2, x_4), …, and regress y on (x_1, x_2, x_{p-1}).

STEP #6

1. The third predictor put in stepwise model is the predictor that has the **smallest *t*-test *P*-value** (below $\alpha_E = 0.15$).
2. If no P-value < 0.15, stop.
3. Step back and check P-values for $\beta_1 = 0$ and $\beta_2 = 0$. If either P-value has become not significant (above $\alpha_R = 0.15$), remove the predictor from the stepwise model.

STEP #7

- The procedure is stopped when adding an additional predictor does not yield a ***t*-test *P*-value** below $\alpha_E = 0.15$.

Figure 3.17: Step-wise regression approach

Let's enter the code now. It will be more fragmented due to some needed explanations. Main steps:

- **Store in your original dataframe the column with the new regressors.**

As mentioned, they can be a shifted version of the original data, numbers indicating time (day, months, year...) or a boolean vector composed by 0,1 where 1 is linked to the observation recognized to be influenced by special causes.

- Create the dataframe composed by the predictors + constant (column of 1) and you will call it "x".

  NOTE: if you shifted the dataset of "n" then the rows of "x" should start from the "n-th" observation, otherwise you will have "nan" values within your predictors dataframe.

- Create the vector "y" in which there will be the original data.

  NOTE: "y" must have the same number of rows of "x", meaning that if you shifted your data of "n" then you need to place values in "y" from the "n-th" observation onwards

	index	Ex5	year	lag1	lag4
0	0	-0.14	1900	nan	nan
1	1	-0.2	1901	-0.14	nan
2	2	-0.33	1902	-0.2	nan
3	3	-0.46	1903	-0.33	nan
4	4	-0.48	1904	-0.46	-0.14
5	5	-0.4	1905	-0.48	-0.2
6	6	-0.28	1906	-0.4	-0.33
7	7	-0.5	1907	-0.28	-0.46
8	8	-0.45	1908	-0.5	-0.48
9	9	-0.43	1909	-0.45	-0.4
10	10	-0.41	1910	-0.43	-0.28

Figure 3.18: Shifted DataFrame and preparation

	index	Ex5
0	4	-0.48
1	5	-0.4
2	6	-0.28
3	7	-0.5
4	8	-0.45
5	9	-0.43
6	10	-0.41
7	11	-0.4
8	12	-0.36
9	13	-0.35
10	14	-0.21

Figure 3.19: y: original observations series prepared for step-wise regression

	index	year	lag1	lag4
0	4	1904	-0.46	-0.14
1	5	1905	-0.48	-0.2
2	6	1906	-0.4	-0.33
3	7	1907	-0.28	-0.46
4	8	1908	-0.5	-0.48
5	9	1909	-0.45	-0.4
6	10	1910	-0.43	-0.28
7	11	1911	-0.41	-0.5
8	12	1912	-0.4	-0.45
9	13	1913	-0.36	-0.43
10	14	1914	-0.35	-0.41

Figure 3.20: x: regressors DataFrame prepared for step-wise regression

- import the needed libraries run the stepwise regression. Once stepwise is obtained, store the model results. In the code we will store it as "results = model.model_fit". For this reason checks on residual, plots and mostly of the previous script are okay, but we won't refer anymore to our model as, for instance, "model.fittedvalues" or "model.resid" because our model fits are stored in "results" now. So we will use "results.fittedvalues", "results.resid" and so on.
- When you check the residuals be aware about the size "n" you input because if you have shifted the dataframe to fit an AR process, it will be the original size minus the number of shifts
- If residuals are NID the job is done.

  On the contrary, if there is still some autocorrelation in the residuals (quantitatively checked with the proper tests) it means that your model has not been able to capture it all. Therefore a new model must be created and the first trial will be carried out by adding that lagged version of the data into your predictors dataframe. It's likely that the inclusion of the regressive patterns remained unexplained might lead to NID residuals.

This first block is intended to be a hint on how to manage the preparation of your dataframes for the regression. They are examples taken from exercises which shows how to deal with different types of predictors.

THIS CODES PROVIDE SOME EXAMPLE OF HOW TO PREPARE DUMMY VARIABLES BEFORE INTRODUCING THEM IN THE "x" VECTOR FOR THE REGRESSION MODEL

```
# code to insert a column with time values on the original dataframe
df['year'] = np.arange(1900, 1998) #last value is excluded, 1998 won't appear

# code to insert a column with the shifted version of the original dataframe
df['lag1'] = df['Ex5'].shift(1)

# DUMMY VARIABLES PREPARETION

# dummy for occasional days (or year, or either other units of time)
df['dummy'] = np.zeros(len(df))
df['dummy'][a:b] = 1        #set the proper range of observation for the occasional days, years... and so on

# dummy for batched processes: use the functions np.tile and np.repeat shown at the beginning of the document
# len(df)/num batches = num days
df['Batch']=np.tile(np.arange(1,5),int(len(df)/4))
# len(df)/num batches = num days, but then +1 is needed because np.arange leave out the last value!
df['Day']=np.repeat(np.arange(1,int(len(df)/4+1)), 4)
```

Code 3.8: Original dataframe preparation for stepwise regression

Prepare the dataframe "x" and the series "y" using .iloc[a:b, c:d] to respectively define the range of rows and columns from which the data composing the two has to be picked. Keep in mind that the range of rows must be set equal for "x" and "y".

```
# Create a StepwiseRegression object using the qda library
```

```
import qda
stepwise = qda.StepwiseRegression(add_constant = True, direction = 'both', alpha_to_enter = 0.15, alpha_to_remove = 0.15)
# Fit the model
model = stepwise.fit(y, x)
```

Code 3.9: Step-wise object creation and model fitting

Since python notebook cannot show long outputs it is better to divide this codes

```
results = model.model_fit
qda.summary(results)
```

Code 3.10: Model result storing and displaying

The model is fitted, the results are shown, let's proceed with the checks. FIRST OF ALL I SUGGEST TO PLOT A MODEL VS FITS PLOT, shown in the matplotlib initial section.

NOTE: change "model.fittedvalues" in "results.fittedvalues"

```
# RESIDUALS CHECKS (graphs + Shapiro Wilk test)
fig, axs = plt.subplots(2, 2, figsize = (10, 8))
fig.suptitle('Residual Plots')
axs[0,0].set_title('Normal probability plot')
stats.probplot(results.resid, dist="norm", plot=axs[0,0])
axs[0,1].set_title('Versus Fits')
axs[0,1].scatter(results.fittedvalues, results.resid)
fig.subplots_adjust(hspace=0.5)
axs[1,0].set_title('Histogram')
axs[1,0].hist(results.resid)
axs[1,1].set_title('Time series plot')
axs[1,1].plot(np.arange(1, len(results.resid)+1), results.resid, 'o-')
_, pval_SW_res = stats.shapiro(results.resid)
print('Shapiro-Wilk test p-value on the residuals = %.3f' % pval_SW_res)

# RESIDUALS CHECKS (randomnsess + ACF/PACF)
#RANDOMNESS OF FESIDUALS
_, pval_runs_res = runstest_1samp(results.resid, correction=False)
print('Runs test p-value on the residuals = {:.3f}'.format(pval_runs_res))

#WATCH OUT: IF YOU FITTED AN AUTOREGRESSIVE MODEL THE NUMBER OF RESIDUALS ON WHICH YOU ARE TESTING IS NOT THE ORIGINAL ONE ANYMORE
#THE SIZE TO BE USED IN RESIDUALS CHECK IS: len(df) - shift
fig, ax = plt.subplots(2, 1)
sgt.plot_acf(results.resid, lags = int(len(df)/3), zero=False, ax=ax[0])
fig.subplots_adjust(hspace=0.5)
sgt.plot_pacf(results.resid, lags = int(len(df)/3), zero=False, ax=ax[1],method = 'ywm')
plt.show()

#if Barlett or LBQ tests are required remember to properly set the size of the residuals dataframe
```

Code 3.11: Residuals plots and tests

As mentioned, in the case you see some autocorrelation in the residuals add as predictor that lagged version of the data and repeat the process. If you solve it, it may be consistent to test autocorrelation at that lag also in the new residuals to figure out whether the model is able to capture all the hidden patterns.

3.1.4. Confidence and prediction intervals on the step-wise regression output

Confidence intervals on a step-wise regression model do not differ that much from the ones explained above (go to subsection 3.1.2). The previous code was proper for single predictor models. The following script will help you to:

- **Sort in the correct order the predictors for the next observation: note that the function "stepwise.fit(y, x)" mix the order in which the predictors are stored respect the ones you defined through the sequence of columns in your dataframe "df". This code checks the true order inspecting the model stored.**
- **Create the vector of regressors: to forecast the next observation it is necessary to input a vector that saves into its scores the last value assumed by the predictors (or further back in time if there are k times lagged predictors with k $>$ 1).**

  Be careful: if there are shifted data predictors, the right observation be picked up by the method .iloc[-k] where "k" is entity of the shift. Moreover, the constant value [1] must be set there if and only if the constant was significant and given as output by the results.param.

  Finally, be aware that some variables SUCH AS THE TIME need to be placed as input incremented by 1 respect the last observation. For example, if your dataframe ends at year 1997, the next observation involves a predictor value of 1998.
- **Numerically compute the confidence and the prediction interval for the next observation**

```
# first check the order in which predictors are placed
print(results_2.params)
# CREATE A DATAFRAME which contains the values to be inserted in the step-wise model equation.
# follow the same structure and input your data
regressor_vector = pd.DataFrame({'const': [1], 'lag1': [df['Ex5'].iloc[-1]], 'year': [1998], 'lag4': df['Ex5'].iloc[-4]})
print(regressor_vector)
prediction = results_2.predict(regressor_vector)
```

```
print('The next observation prediction is: ', prediction[0]) #leave it as it is
# Compute the fit, confidence intervals and prediction intervals
prediction_summary = results_2.get_prediction(regressor_vector).summary_frame(alpha=0.05)
print(prediction_summary)
```

Code 3.12: Next observation, confidence and prediction interval on step-wise output

3.2. ARIMA

REMINDER

Autoregressive models: AR(p)

$$X_t = \xi + \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \varepsilon_t$$

- ACF "geometrically decays"
- PACF indicates the order p

Moving average models: MA(q)

$$X_t = \mu - \theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} - \ldots - \theta_q \varepsilon_{t-q} + \varepsilon_t$$

$$\tilde{X}_t = X_t - \mu = -\theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} - \ldots - \theta_q \varepsilon_{t-q} + \varepsilon_t$$

- PACF "geometrically decays"
- ACF indicates the order q

Quality Engineering

REMINDER

ARMA(p,q): both AR(p) and MA(q) terms are present

- It resembles an AR(p) after q lags
- Parsimony: low order models are preferred (easier to deal with)
- Model identification is often a *trial and error* problem

Homogeneous nonstationary ARMA (p,q) = ARIMA(p,d,q)

- ACF with slow (e.g. linear) decay, not a "geometrical decay"

Figure 3.21: ARIMA models introduction

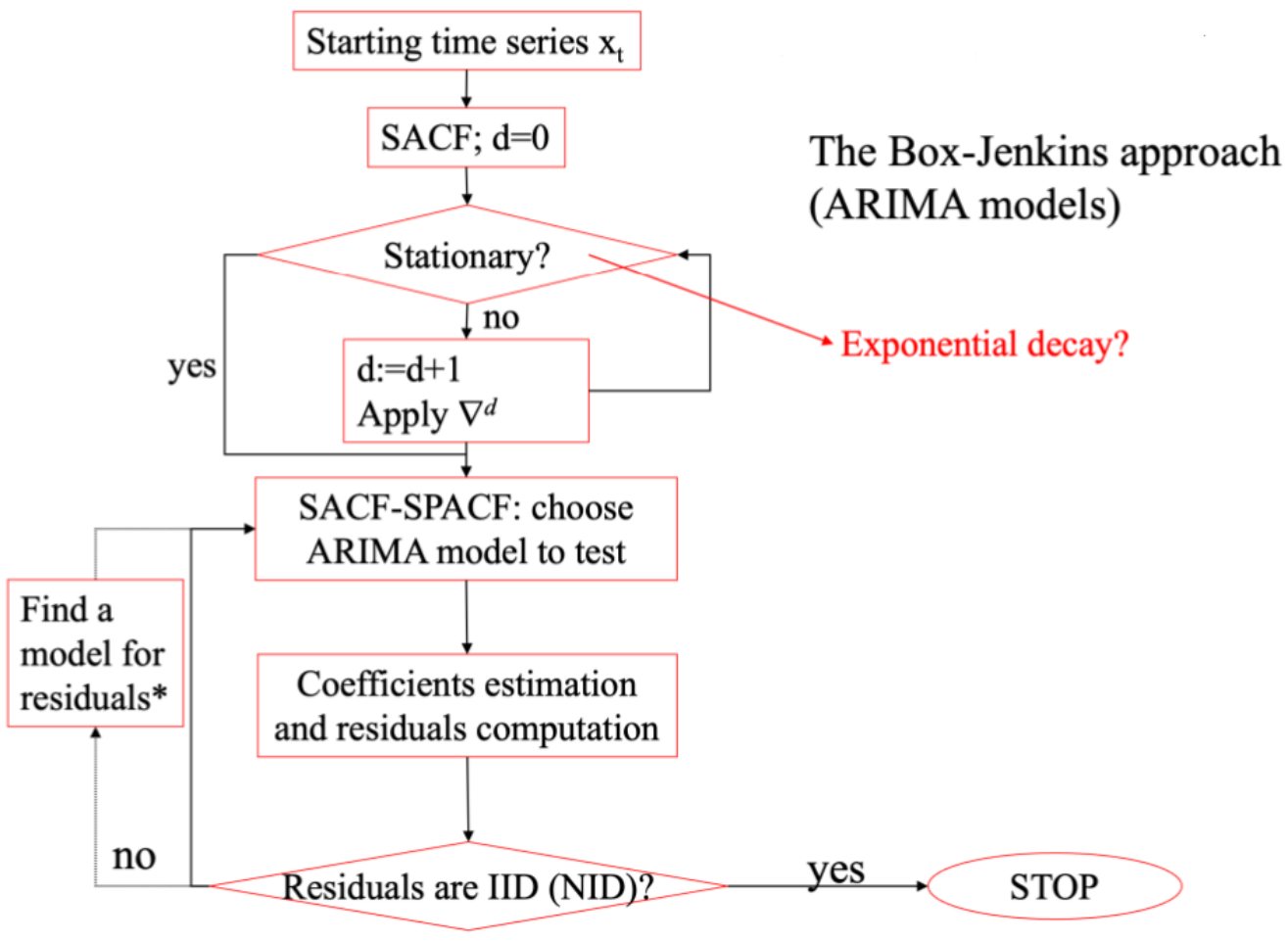


Figure 3.22: Steps to fit an ARIMA: Box-Jenkis approach

In few words when the series looks unstable it can be due to a trend or it can be a consequence of an unstationary autoregressive process. The best way to go is to check if the ACF shows a linear decay and there is any violation in the PACF. In that case it might be an autoregressive process of order one with autocorrelation coefficient (coefficient multiplying the lag1 predictor) equal to 1.

The first thing to do is to differentiate your dataset once and see if the differentiated time series is stationary or not. If it is not stationary, we need to differentiate twice, or more times, until the series becomes stationary (generally once may be enough, maximum two, otherwise start think that the unstationarity might not be caused by that).

Then if stationarity is reached, you have to make the same test on residuals that we usually do after model fitting. The reason is simply that if the model was an ARIMA(0,1,0), your differentiated dataset is exactly your series of residuals by definition (go check the theory).

If those residuals do not respect the assumption you should check their ACF/PACF looking for pattern remained unexplained. Therefore, according to the rule to recognize AR and MA we should go on by fitting the proper shown pattern.

Let's see the codes:

```
# Differentiate your dataset first
df['diff1'] = df['column name'].diff(1) #change the number if it is not the first iteration

# Plot your differentiated dataset
plt.plot(df['diff1'][1:], 'o-')
plt.xlabel('Index')
plt.ylabel('DIFF 1') #change the number if it is not the first iteration
plt.title('Time series plot of DIFF 1')
plt.grid()
plt.show()
#look if the output is stationary or not, otherwise re-iterate

# if stationarity is reached at the first iteration
# RANDOMNESS DIFFERENCING VECTOR
_, pval_runs = runstest_1samp(df['diff1'][1:], correction=False)
print('Runs test p-value = {:.3f}'.format(pval_runs))

# ACF/PACF DIFFERENCING VECTOR
fig, ax = plt.subplots(2, 1)
sgt.plot_acf(df['diff1'][1:], lags = int((len(df)-1)/3), zero=False, ax=ax[0])
fig.subplots_adjust(hspace=0.5)
sgt.plot_pacf(df['diff1'][1:], lags = int((len(df)-1)/3), zero=False, ax=ax[1], method = 'ywm')
plt.show()

# use Barlett or LBQ to verify the autocorrelation
# if no more autocorrelation run the code for automatic ARIMA with p=0, d=1, q=1
# if there is still autocorrelation check if you can fit an ARIMA(p,1,q) with the proper p and q
```

Code 3.13: ARIMA in case of non stationarity

ONCE STATIONARITY IS REACHED USE THIS CODE TO RUN AN AUTOMATIC ARIMA MODEL FIT

This is a parametric code which allows you to try different models just by changing the value of "p", "d" and "q"

```
p = insert p
d = insert d
q = insert q
#calculate an ARIMA model: import the necessary library
import qda
# fit the model ARIMA with constant term (place "False" in case it is not significant)
model = qda.ARIMA(df['column name'], order=(p,d,q), add_constant=True) #input the original data
qda.ARIMAsummary(model)
```

Code 3.14: ARIMA in case of non stationarity

```
#extract the residuals (remember the first "a" of them are nan if you shifted your dataset of "a")
residuals = model.resid[np.max((p,d,q)):]
fits = model.fittedvalues[np.max((p,d,q)):]

# Perform the Shapiro-Wilk test + residuals plots
_, pval_SW = stats.shapiro(residuals)
print('Shapiro-Wilk test p-value = %.3f' % pval_SW)
fig, axs = plt.subplots(2, 2)
fig.suptitle('Residual Plots')
stats.probplot(residuals, dist="norm", plot=axs[0,0])
axs[0,0].set_title('Normal probability plot')
axs[0,1].scatter(fits, residuals)
axs[0,1].set_title('Versus Fits')
fig.subplots_adjust(hspace=0.5)
axs[1,0].hist(residuals)
axs[1,0].set_title('Histogram')
axs[1,1].plot(np.arange(1, len(residuals)+1), residuals, 'o-')
plt.show()
# Residuals randomness check
_, pval_runs = runstest_1samp(residuals, correction=False)
print('Runs test p-value = {:.3f}'.format(pval_runs))
# ACF/PACF on residuals
fig, ax = plt.subplots(2, 1)
sgt.plot_acf(residuals, lags = int(len(residuals)/3), zero=False, ax=ax[0])
fig.subplots_adjust(hspace=0.5)
sgt.plot_pacf(residuals, lags = int(len(residuals)/3), zero=False, ax=ax[1], method = 'ywm')
plt.show()
```

Code 3.15: residuals check

Try different p,d,q until your model's residuals are NID (normal and idependent and indentically distributed).

```
plt.plot(df['column name'][np.max((p,d,q)):], 'o-', label='Original data')
plt.xlabel('index') #e.g.: index
plt.ylabel('column name') #column of original data
plt.plot(fits, 's--', color='red', label='Fitted values', alpha=0.5)
plt.legend()
plt.grid()
plt.show()
```

Code 3.16: ARIMA original data vs fits

4 | PCA

PCA is a linear transformation, hence it does not require any type of assumptions. As a rule of thumb, the performance of the dimensional reduction model increases when data are normal and random (no pattern, including autocorrelation within the variables vectors. On the contrary, PCA is effective when there is correlation between the variables

Principal Component Analysis (PCA) is a statistical technique used to simplify a dataset by reducing its number of dimensions while preserving as much variability as possible. This is particularly useful in fields like data visualization, noise reduction, and feature extraction for machine learning and other statistical analyses.

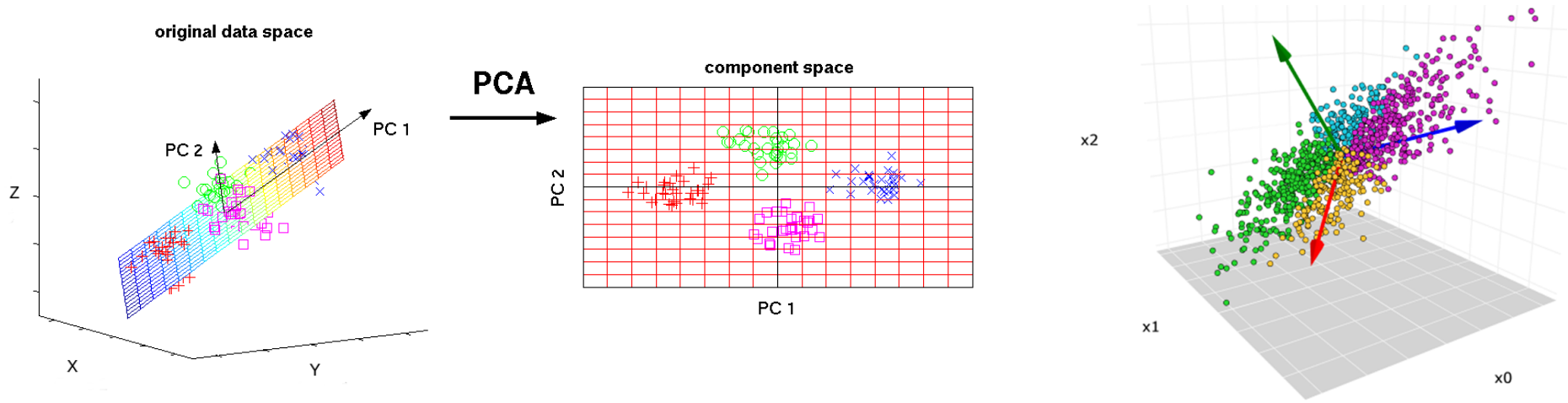


Figure 4.1: Principal component analysis purposes

Principal components are new, uncorrelated variables formed by linear combinations of the original variables. The first principal component captures the maximum variance in the dataset, and each subsequent component captures the next highest variance under the constraint that it is orthogonal (uncorrelated) to the previous components.

Eigenvalues indicate the magnitude of variance captured by each eigenvector (principal component). Higher eigenvalues correspond to components that capture more of the dataset's variability.

The choice of whether to compute eigenvalues and eigenvectors on the covariance matrix or the correlation matrix in Principal Component Analysis (PCA) depends on the nature and scale of your data.

- **Covariance matrix The covariance matrix is used when the variables are on similar scales or have the same units of measurement. This approach preserves the original variance of each variable, making it suitable when the scale of measurement is meaningful and consistent across variables. The covariance matrix captures the total variance within the dataset directly, which can be important when the absolute magnitudes of the variables are significant**
- **Correlation matrix The correlation matrix is used when the variables have different units or vastly different scales. This method standardizes the data, transforming all variables to have a mean of zero and a standard deviation of one. By doing this, it ensures that each variable contributes equally to the analysis, avoiding domination by variables with larger variances. The correlation matrix is particularly useful when dealing with data where the relative variability between variables is more important than their absolute values**

Variance-Covariance Matrix

$$V(\boldsymbol{x}) = E[(\boldsymbol{x} - E(\boldsymbol{x}))(\boldsymbol{x} - E(\boldsymbol{x}))'] = \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1p} \\ \sigma_{12} & \sigma_2^2 & \cdots & \sigma_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ \sigma_{1p} & \sigma_{2p} & \cdots & \sigma_p^2 \end{bmatrix}$$

pxp px1 1xp

$$\sigma_{ij} = E[(x_i - \mu_i)(x_j - \mu_j)] = \mathrm{cov}(x_i, x_j)$$

Figure 4.2: Variance-Covariance matrix

correlation

$$\rho_{ij} = \frac{\mathrm{cov}(x_i x_j)}{\sqrt{V(x_i)V(x_j)}} \qquad \boldsymbol{P} = \begin{bmatrix} 1 & \rho_{12} & \cdots & \rho_{1p} \\ \rho_{12} & 1 & \cdots & \rho_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ \rho_{1p} & \rho_{2p} & \cdots & 1 \end{bmatrix}$$

Correlation matrix

Figure 4.3: Correlation matrix

Loadings are the coefficients or weights that quantify the contribution of each original variable to a principal component. They indicate how much each variable influences a principal component and can be visualized in loading plots, where vectors show the magnitude and direction of each variable's contribution to the components.

Eigenvectors in PCA are the directions in the original variable space along which the data varies the most. When you compute the covariance matrix of the data, the eigenvectors of this matrix represent the principal components, which are new orthogonal axes in the transformed coordinate system. Each eigenvector points in the direction of maximum variance in the data.

Understanding the relationship between loadings and eigenvectors is essential for interpreting PCA results. The loadings help identify which original variables are most influential in each principal component, aiding in data interpretation, dimensionality reduction, and feature selection.

By linking eigenvectors (directions of maximum variance) with loadings (contributions of original variables), we gain insights into the structure and patterns within the data, facilitating more effective data analysis and decision-making.

Scores are the coordinates of the original data points in the new principal component space. Each score represents a projection of the data onto the principal components, effectively transforming the original dataset into a new set of coordinates that emphasize the most significant variations

- Dataset X is a $n \times p$ matrix, where n is the number of observations, and p is the number of variables.
- The sample variance-covariance matrix S is a $p \times p$ matrix.
- The sample correlation matrix R is a $p \times p$ matrix.

Eigendecomposition of S:

- Eigenvalues is a $p \times 1$ vector, $\lambda_1, \lambda_2, \ldots, \lambda_p$ (explained variance).
- Eigenvectors is a $p \times p$ matrix, $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_p}$ (loadings).

$$\mathbf{u_1} = [u_{11}, u_{12}, \ldots, u_{1p}]^T$$
$$\mathbf{u_2} = [u_{21}, u_{22}, \ldots, u_{2p}]^T$$
$$\vdots$$
$$\mathbf{u_p} = [u_{p1}, u_{p2}, \ldots, u_{pp}]^T$$

Figure 4.4: Variance-Covariance matrix

Projection of data onto the space spanned by the PCs (scores) ($n \times p$ matrix):

$$\mathbf{z_1} = [z_{11}, z_{21}, \ldots, z_{n1}]^T = (\mathbf{X} - \bar{\mathbf{X}})\mathbf{u_1}$$
$$\mathbf{z_2} = [z_{12}, z_{22}, \ldots, z_{n2}]^T = (\mathbf{X} - \bar{\mathbf{X}})\mathbf{u_2}$$
$$\vdots$$
$$\mathbf{z_p} = [z_{1p}, z_{2p}, \ldots, z_{np}]^T = (\mathbf{X} - \bar{\mathbf{X}})\mathbf{u_p}$$

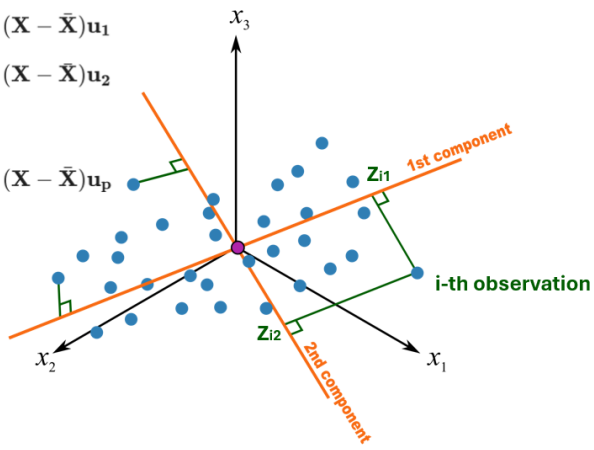


Figure 4.5: Correlation matrix

THE FIRST STEP NOT MENTIONED IS A BRIEF PREPARATION OF OUR DATA BY SUBTRACTING TO EACH SINGLE VARIABLE THEIR RESPECTIVE MEAN.

In order to compute the principal component analysis, the first step involve the motion of the ORIGINAL BASE's ORIGIN TO THE SAMPLE MEAN OF EACH VARIABLE. In the following picture p = 2 for simplicity. The original base was represented by thee black axes. Then this same base is moved to a new center that is the sample mean of the two variables values. The latter is shown with green axes. Finally, the principal components are shown in red

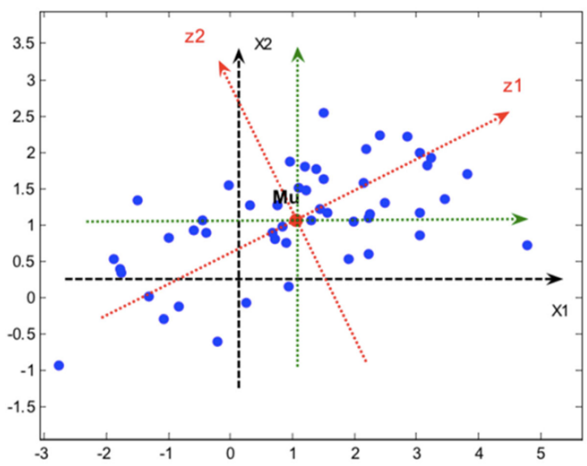


Figure 4.6: Orthogonal bases changes in PCA

Curious fact about PCA and linear models

While PCA and linear regression have different theoretical bases and optimize different criteria, their results converge in the case of two variables when finding the direction of maximum variance (PCA) or the best-fit line (regression). This alignment shows that under certain conditions, these methods are not only comparable but can lead to the same analytical conclusion.

- **PCA Objective: PCA finds the direction (line) that captures the maximum variance of the data, aiming to reduce dimensionality while preserving as much information as possible.**
- **Linear Regression Objective: Linear regression finds the line that minimizes the sum of squared residuals (errors) between the observed data points and the predicted values.**

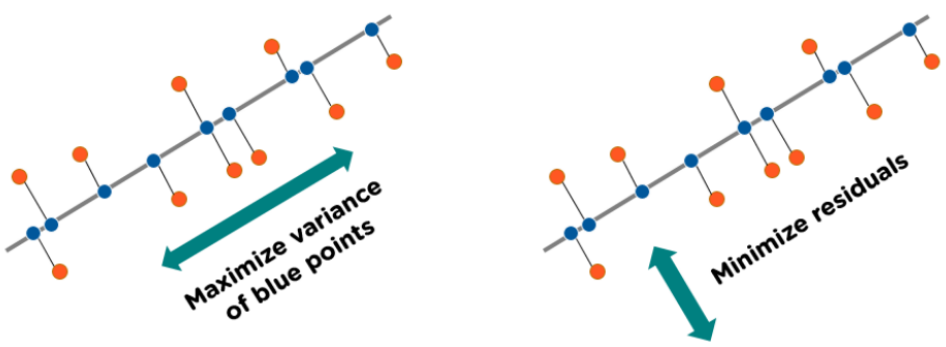


Figure 4.7: PCA vs regressions in case of p = 2 variables

4.1. Principal components analysis on python

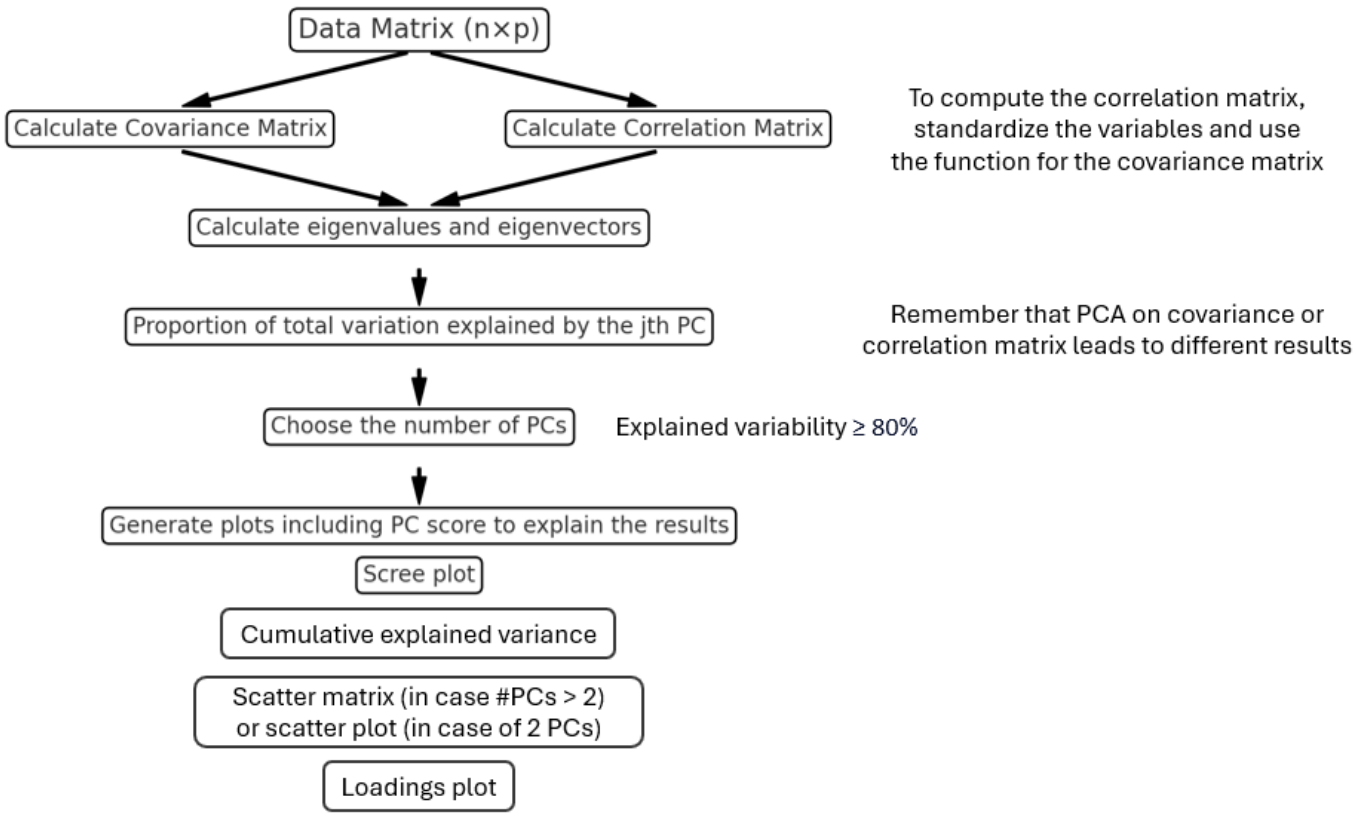


Figure 4.8: PCA steps

Even though 80% of explained variance is considered as a good result, it is better to reach 85%-90%. There is a kind of trade-off between pca performance and dimensional reduction which should be optimized and tuned by this decision

4.1.1. PCA with covariance matrix

```
# Create a scatter matrix (covariance plot) of the dataset
pd.plotting.scatter_matrix(df, alpha = 1)
plt.show()

# Create the variance covariance matrix using pandas
cov_matrix = df.cov()
print(cov_matrix)
```

Code 4.1: Checks and computation before PCA (covariance matrix)

```
# import the libraries for PCA
from sklearn.decomposition import PCA
# Create the PCA object
pca = PCA()
# Fit the PCA object to the data
pca.fit(df)
# Print the eigenvalues
print("Eigenvalues \n", pca.explained_variance_)
# Print the eigenvectors
print("\nEigenvectors \n", pca.components_)
# Print the explained variance ratio
print("\nExplained variance ratio \n", pca.explained_variance_ratio_)
# Print the cumulative explained variance ratio
print("\nCumulative explained variance ratio \n", np.cumsum(pca.explained_variance_ratio_))
```

Code 4.2: PCA (covariance) fitting

```
# Plot the eigenvalues (scree plot)
plt.plot(pca.explained_variance_, 'o-')
plt.xlabel('Component number')
plt.ylabel('Eigenvalue')
plt.title('Scree plot')
plt.show()
# Plot the cumulative explained variance
plt.plot(np.cumsum(pca.explained_variance_ratio_), 'o-')
# add a bar chart to the plot
plt.bar(range(0, len(pca.explained_variance_ratio_)), pca.explained_variance_ratio_, width = 0.5, alpha=0.5, align='center')
plt.xlabel('Component number')
plt.ylabel('Cumulative explained variance')
plt.title('Cumulative explained variance')
plt.show()

# Alternatively, create a figure with two horizontal subplots
fig, axs = plt.subplots(1, 2, figsize=(20, 8))
# Plot the eigenvalues (scree plot) in the first subplot
axs[0].plot(pca.explained_variance_, 'o-')
axs[0].set_xlabel('Component number')
```

```
axs[0].set_ylabel('Eigenvalue')
axs[0].set_title('Scree plot')
# Plot the cumulative explained variance in the second subplot
axs[1].plot(np.cumsum(pca.explained_variance_ratio_), 'o-')
# Add a bar chart to the plot
axs[1].bar(range(0, len(pca.explained_variance_ratio_)), pca.explained_variance_ratio_, width=0.5, alpha=0.5, align='center')
axs[1].set_xlabel('Component number')
axs[1].set_ylabel('Cumulative explained variance')
axs[1].set_title('Cumulative explained variance')
# Adjust layout for better spacing
plt.tight_layout()
# Show the plot
plt.show()
```

Code 4.3: Scree plot + cumulative explained variance

```
# Compute the scores (i.e. all the principal components, n x 3)
scores = pca.transform(df)
# create a dataframe with the scores
scores_df = pd.DataFrame(scores, columns = ['z1', 'z2', 'z3']) #add or reduce the number of PCs
# Print the first rows of the scores dataframe
scores_df.head()
```

Code 4.4: PCA (covariance) scores computation

```
# Plot the scores in a scatter matrix if #PCs > 2
pd.plotting.scatter_matrix(scores_df, alpha = 1)
plt.show()

# Plot the scores in a scatter plot if #PCs = 2
plt.plot(scores_df['z1'], scores_df['z2'], 'o', color='blue', label='Original data')
plt.title('Scatter plot of Z1 vs Z2')
plt.xlabel('Z1')
plt.ylabel('Z2')
plt.show()

# Plot the loadings
fig, ax = plt.subplots(1, 3, figsize = (15, 5))
ax[0].plot(pca.components_[0], 'o-')
ax[0].set_title('Loading 1')
ax[1].plot(pca.components_[1], 'o-')
ax[1].set_title('Loading 2')
ax[2].plot(pca.components_[2], 'o-')
ax[2].set_title('Loading 3')
plt.show()
```

Code 4.5: Checks post PCA: scores scatter and loadings plot

4.1.2. PCA with correlation matrix

```
# Create a scatter matrix (covariance plot) of the dataset
pd.plotting.scatter_matrix(df, alpha = 1)
plt.show()

# Compute the corr. matrix by standardizing the variables and applying the variance-covariance matrix function: .cov()
# Standardize the data by subtracting the mean and dividing by the standard deviation
df_std = (df - df.mean()) / df.std()
df_std.describe()
# Put the matrix in another cell or Visual code notebook won't be able to show all the results
cov_matrix_std = df_std.cov()
print(cov_matrix_std)

# Alternatively you can use this, BUT IN ANY CASE YOU MUST STANDARDIZE THE VARIABLE SINCE THE PCA WILL BE FITTED ON THEM
corr_matrix = df.corr()
print(corr_matrix)
```

Code 4.6: Checks and computation before PCA (covariance matrix)

```
# Apply the PCA on the correlation matrix instead of the covariance matrix
from sklearn.decomposition import PCA
# Create the PCA object
pca_corr = PCA()
# Fit the PCA object to the data
pca_corr.fit(df_std)
# Print the eigenvalues
print("Eigenvalues from STANDARDIZED data \n", pca_corr.explained_variance_)
# Print the eigenvectors
print("Eigenvectors from STANDARDIZED data \n", pca_corr.components_)
# Print the explained variance ratio
print("Explained variance ratio from STANDARDIZED data \n", pca_corr.explained_variance_ratio_)
# Print the cumulative explained variance ratio
print("Cumulative explained variance ratio from STANDARDIZED data \n", np.cumsum(pca_corr.explained_variance_ratio_))
```

Code 4.7: PCA (correlation) fitting

```
# Plot the eigenvalues (scree plot)
plt.plot(pca_corr.explained_variance_, 'o-')
plt.xlabel('Component number')
plt.ylabel('Eigenvalue')
plt.title('Scree plot')
plt.show()

# Plot the cumulative explained variance
plt.plot(np.cumsum(pca_corr.explained_variance_ratio_), 'o-')
# add a bar chart to the plot
plt.bar(range(0, len(pca_corr.explained_variance_ratio_)), pca_corr.explained_variance_ratio_, width = 0.5, alpha=0.5, align='center')
plt.xlabel('Component number')
plt.ylabel('Cumulative explained variance')
plt.title('Cumulative explained variance')
plt.show()

# if you want to merge them in two sided suplots:
# Create a figure with two horizontal subplots
fig, axs = plt.subplots(1, 2, figsize=(20, 8))
# Plot the eigenvalues (scree plot) in the first subplot
axs[0].plot(pca_corr.explained_variance_, 'o-')
axs[0].set_xlabel('Component number')
axs[0].set_ylabel('Eigenvalue')
axs[0].set_title('Scree plot')
# Plot the cumulative explained variance in the second subplot
axs[1].plot(np.cumsum(pca_corr.explained_variance_ratio_), 'o-')
# Add a bar chart to the plot
axs[1].bar(range(0, len(pca_corr.explained_variance_ratio_)), pca_corr.explained_variance_ratio_, width=0.5, alpha=0.5, align='center')
axs[1].set_xlabel('Component number')
axs[1].set_ylabel('Cumulative explained variance')
axs[1].set_title('Cumulative explained variance')
# Adjust layout for better spacing
plt.tight_layout()
# Show the plot
plt.show()
```

Code 4.8: Scree plot + cumulative explained variance

```
# Compute the scores
scores = pca_corr.transform(df_std)
# create a dataframe with the scores
scores_df = pd.DataFrame(scores, columns = ['z1', 'z2', 'z3']) #add or reduce the number of PCs
# Print the first rows of the scores dataframe
scores_df.head()
```

Code 4.9: PCA (correlation) scores computation

```
# Plot the scores in a scatter matrix if #PCs > 2
pd.plotting.scatter_matrix(scores_df, alpha = 1)
plt.show()

# Plot the scores in a scatter plot if #PCs = 2
plt.plot(scores_df['z1'], scores_df['z2'], 'o', color='blue', label='Original data')
plt.title('Scatter plot of Z1 vs Z2')
plt.xlabel('Z1')
plt.ylabel('Z2')
plt.show()

# Plot the loadings
fig, ax = plt.subplots(1, 3, figsize = (15, 5))
ax[0].plot(pca_corr.components_[0], 'o-')
ax[0].set_title('Loading 1')
ax[1].plot(pca_corr.components_[1], 'o-')
ax[1].set_title('Loading 2')
ax[2].plot(pca_corr.components_[2], 'o-')
ax[2].set_title('Loading 3')
plt.show()
```

Code 4.10: Checks post PCA: PCs scatter and loadings plot

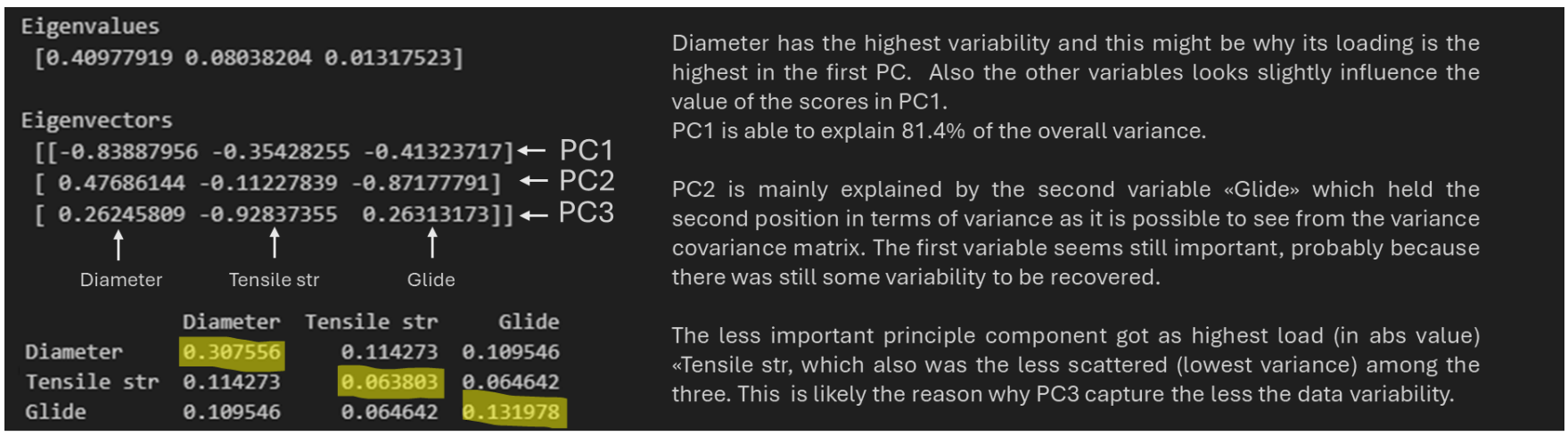
4.1.3. Main comments on loadings plots

The principal compontents' loadings should be explained one by one and we may carry out a final comparison (brief sum up) according to the obtained results

How to interpret and comment on the loadings:

- **Highlight which are the most influent variables in the scores computation (so the ones that hold the highest loading) and put the focus on whether they increase or decrease the scores values. Make comments on non significant variables.**
- **Underline if there is contrast between two or more variables.**
- **If possible link the result to the variance-covariance matrix and the obtained eigenvalues.**

  If there are large differences between the variances of the elements of, then those variables whose variances are largest will tend to dominate the first few PCs.



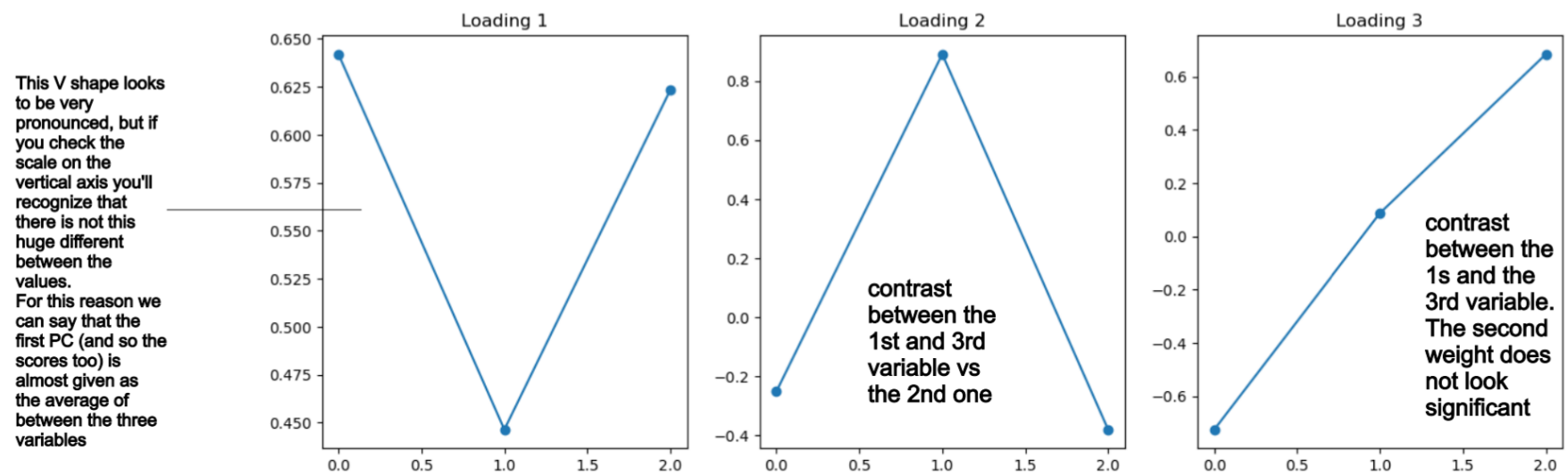
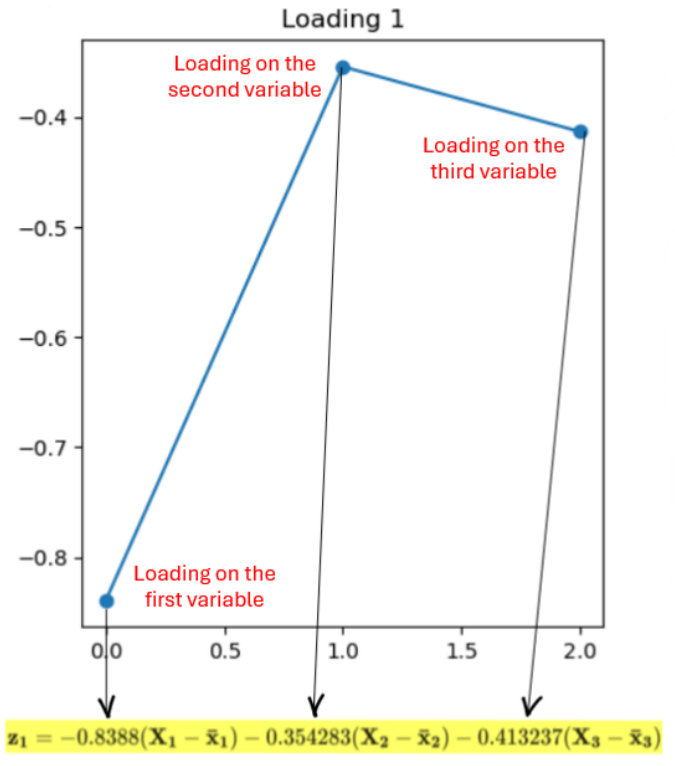


Figure 4.9: Example of loading plot + comments



Remember that:

- Loading are placed in the same order in which the variables are stored in the original dataframe;
- Each eigenvalue is referred to the variance explained by a principal component, while the eigenvector values weights (and so are linked) to the variables (quality features)
- It's crucial to look at the scale of the loadings, the closer to zero, the less the influence and viceversa;
- Do not forget how one score is computed (take a look on the equality highlighted in yellow);

Figure 4.10: Additional remarks on the loadings

4.2. Data reconstruction

4.2.1. How to reconstruct the data with k < P principal components

How to get the principal components

$$\mathbf{u}_1 = [u_{11}, u_{21}, \ldots, u_{p1}]^T$$
$$\mathbf{u}_2 = [u_{12}, u_{22}, \ldots, u_{p2}]^T$$
$$\mathbf{u}_3 = [u_{13}, u_{23}, \ldots, u_{p3}]^T$$

most compact formulation

$\mathbf{U}$ is the matrix of eigenvectors:

$$\mathbf{U} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_p \end{bmatrix}$$

$$\mathbf{Z} = \mathbf{X} \cdot \mathbf{U}$$

[nxp] [nxp] [pxp]

"p" principal components of size "n"

1. First Principal Component:

$$\mathbf{Z}_1 = \mathbf{X} \cdot \mathbf{u}_1 = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{p1} \end{bmatrix} = \begin{bmatrix} Z_{11} \\ Z_{21} \\ \vdots \\ Z_{n1} \end{bmatrix}$$

2. Second Principal Component:

$$\mathbf{Z}_2 = \mathbf{X} \cdot \mathbf{u}_2 = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix} \cdot \begin{bmatrix} u_{12} \\ u_{22} \\ \vdots \\ u_{p2} \end{bmatrix} = \begin{bmatrix} Z_{12} \\ Z_{22} \\ \vdots \\ Z_{n2} \end{bmatrix}$$

3. General Case for k-th Principal Component:

$$\mathbf{Z}_k = \mathbf{X} \cdot \mathbf{u}_k = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix} \cdot \begin{bmatrix} u_{1k} \\ u_{2k} \\ \vdots \\ u_{pk} \end{bmatrix} = \begin{bmatrix} Z_{1k} \\ Z_{2k} \\ \vdots \\ Z_{nk} \end{bmatrix}$$

If you carry out the matrix multiplication you'll see that behind each score there is a linear combination of the observation "j", under the p variables, with the loadings so the value of the eigenvector

Figure 4.11: How to get the principal components

1. **Select the first k principal components**: Let's denote the scores of the first k principal components as $\mathbf{Z}_k$ and the corresponding eigenvectors as $\mathbf{U}_k$.

2. **Matrix of Principal Component Scores**: The matrix $\mathbf{Z}_k$ of dimensions $[n \times k]$ contains the scores for the first k principal components.

$$\mathbf{Z}_k = \begin{bmatrix} Z_{11} & Z_{12} & \cdots & Z_{1k} \\ Z_{21} & Z_{22} & \cdots & Z_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ Z_{n1} & Z_{n2} & \cdots & Z_{nk} \end{bmatrix}$$

3. **Matrix of Selected Eigenvectors**: The matrix $\mathbf{U}_k$ of dimensions $[p \times k]$ contains the first k eigenvectors.

$$\mathbf{U}_k = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1k} \\ u_{21} & u_{22} & \cdots & u_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ u_{p1} & u_{p2} & \cdots & u_{pk} \end{bmatrix}$$

4. **Reconstruction of the Original Variables**: The reconstruction $\hat{\mathbf{X}}_k$ using the first k principal components is obtained by multiplying $\mathbf{Z}_k$ by the transpose of $\mathbf{U}_k$:

$$\hat{\mathbf{X}}_k = \mathbf{Z}_k \cdot \mathbf{U}_k^T$$

5. **Detailed Formula**:

- Let $\hat{\mathbf{X}}_k$ be the reconstructed data matrix of dimensions $[n \times p]$.
- $\mathbf{Z}_k$ is $[n \times k]$.
- $\mathbf{U}_k^T$ is $[k \times p]$.

So the reconstructed data matrix $\hat{\mathbf{X}}_k$ is:

$$\hat{\mathbf{X}}_k = \begin{bmatrix} Z_{11} & Z_{12} & \cdots & Z_{1k} \\ Z_{21} & Z_{22} & \cdots & Z_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ Z_{n1} & Z_{n2} & \cdots & Z_{nk} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{21} & \cdots & u_{p1} \\ u_{12} & u_{22} & \cdots & u_{p2} \\ \vdots & \vdots & \ddots & \vdots \\ u_{1k} & u_{2k} & \cdots & u_{pk} \end{bmatrix}$$

This results in:

$$\hat{\mathbf{X}}_k = \begin{bmatrix} \hat{X}_{11} & \hat{X}_{12} & \cdots & \hat{X}_{1p} \\ \hat{X}_{21} & \hat{X}_{22} & \cdots & \hat{X}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{X}_{n1} & \hat{X}_{n2} & \cdots & \hat{X}_{np} \end{bmatrix}$$

Where each $\hat{X}_{ij}$ is the reconstructed value of the original variable.

Figure 4.12: How to reconstruct data based on k < P principal components

4.2.2. How to reconstruct the data with python

```
# Compute the reconstructed data using the first two principal components (modify the code to fit your specific case)
reconstructed_df = scores_df[['z1', 'z2']].dot(pca.components_[0:2, :])    #for more PCs add other "z" and change the "2"

# Compare the original data with the reconstructed data
print("Original data\n", df.head())
print("\nReconstructed (ORIGINAL) data\n", reconstructed_df.head())

#In case you have to prove that keeping all the PCs we can come back to original data use this line insted of the first one
reconstructed_df_std = scores_df.dot(pca_corr.components_)
```

Code 4.11: Data reconstruction with "k2 of the "p" PCs on python (covariance)

```
# Compute the reconstructed data_std using the first two principal components (modify the code to fit your specific case)
reconstructed_df_std = scores_df[['z1', 'z2']].dot(pca_corr.components_[0:2, :])    #for more PCs add other "z" and change the "2"
mean = df.mean()
std = df.std()
# Now use the mean and standard deviation to compute the reconstructed data
reconstructed_df = reconstructed_df_std.dot(np.diag(std)) + np.asarray(mean)
# Compare the original data with the reconstructed data
print("Original data\n", df.head())
print("\nReconstructed (ORIGINAL) data\n", reconstructed_df.head())

#In case you have to prove that keeping all the PCs we can come back to original data use this line insted of the first one
reconstructed_df_std = scores_df.dot(pca_corr.components_)
```

Code 4.12: Data reconstruction with "k2 of the "p" PCs on python (correlation)

5 | SPC iid

Statistical process control for iid variables can follows two different approaches:

- **Shewhart's approach which also need the quality feature to meet the normality assumption**
- **Probabilistic control charts**

5.1. Shewhart's approach based SPC

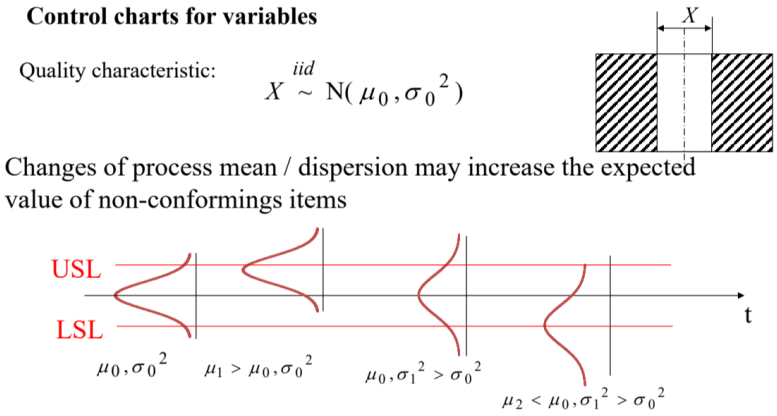


Figure 5.1: SPC based on Shewhart's approach and assumptions

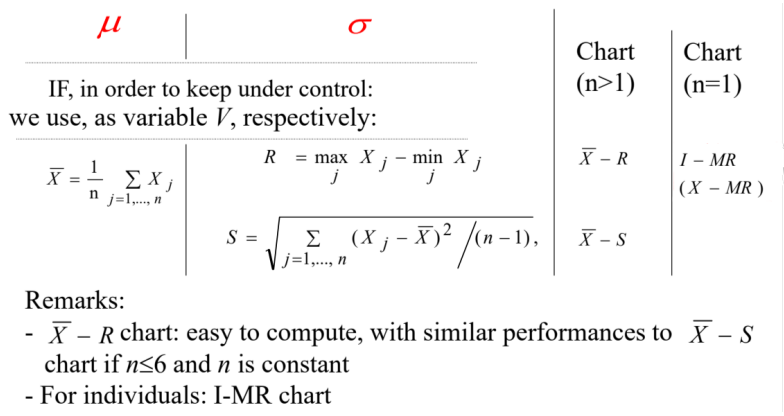


Figure 5.2: Types of control charts based on Shewhart's approach

5.1.1. Xbar-R control charts with known parameters

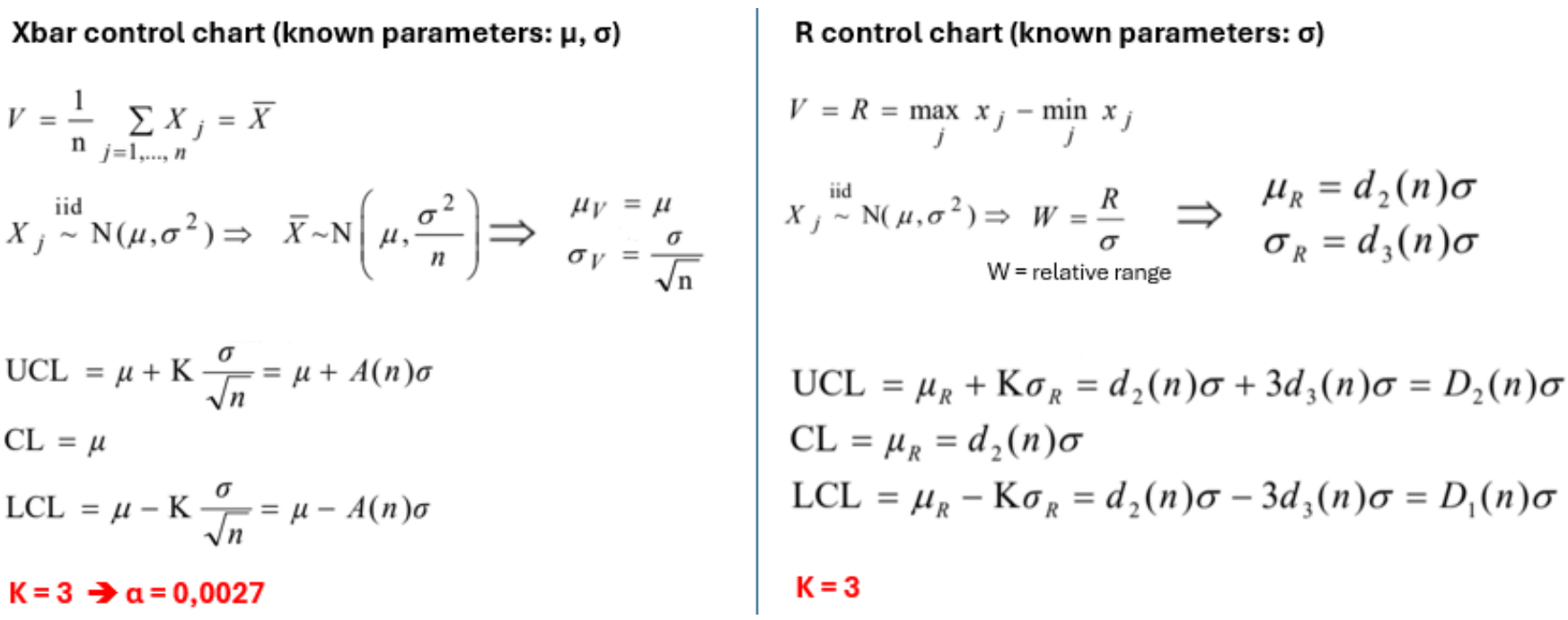


Figure 5.3: Xbar-R control charts with known parameters

5.1.2. Xbar-R control charts with unknown parameters

If process mean and variance are unknown - Phase 1 of control charting:

1. **Pick up m (m=20-25) samples of size n from the process under stable (a regime) conditions;**
2. **Estimate unknown parameters and design the control chart;**
3. **Plot the control chart (retrospective usage or Phase 1 or design phase) – control limits vs. data used to estimate those limits;**
4. **If an out-of-control is signalled, look for assignable causes:**

   If assignable cause is found, remove the observation and go back to step 2;

   If no assignable cause is found (Alwan): if observation is far beyond the limits it will strongly influence the limit computation itself (and assumption checking) – it may be cautious to remove the observation anyway;
5. **Assumption checking: assumption: the reason to check the assumption at the end is that maybe I will be able to get rid of unusual data, recognized as OOC (out of control) by the charts, which can lead me to fail the assumptions.**

Xbar control chart (Unknown parameters)

$$V = \frac{1}{\mathrm{n}} \sum_{j=1,\ldots,n} X_j = \overline{X}$$

$$X_j \overset{\text{iid}}{\sim} \mathrm{N}(\mu,\sigma^2) \Rightarrow \quad \overline{X} \sim \mathrm{N}\left(\mu, \frac{\sigma^2}{n}\right) \Rightarrow \quad \begin{matrix} \mu_V = \mu \\ \sigma_V = \frac{\sigma}{\sqrt{\mathrm{n}}} \end{matrix}$$

Parameters esteem (if and only if iid assumptions hold)

$$\hat{\mu} = \overline{\overline{x}} = \frac{\sum_{i=1,\ldots,m} \overline{x}_i}{\mathrm{m}} \qquad \hat{\sigma} = \frac{\overline{R}}{d_2(n)}$$

$$\mathrm{UCL} = \hat{\mu} + \mathrm{K}\frac{\hat{\sigma}}{\sqrt{n}} = \overline{\overline{x}} + 3\frac{1}{d_2\sqrt{n}}\overline{R} = \overline{\overline{x}} + A_2(n)\overline{R}$$

$$\mathrm{CL} = \hat{\mu} = \overline{\overline{x}}$$

$$\mathrm{LCL} = \hat{\mu} - \mathrm{K}\frac{\hat{\sigma}}{\sqrt{n}} = \overline{\overline{x}} - 3\frac{1}{d_2\sqrt{n}}\overline{R} = \overline{\overline{x}} - A_2(n)\overline{R}$$

K = 3 ➔ α = 0,0027

R control chart (known parameters: σ)

$$V = R = \max_j x_j - \min_j x_j$$

$$X_j \overset{\text{iid}}{\sim} \mathrm{N}(\mu,\sigma^2) \Rightarrow \quad W = \frac{R}{\sigma} \quad \Rightarrow \quad \begin{matrix} \mu_R = d_2(n)\sigma \\ \sigma_R = d_3(n)\sigma \end{matrix}$$

W = relative range

Parameters esteem (if and only if iid assumptions hold)

$$E(W) = d_2(n) = \frac{E(R)}{\sigma} \Rightarrow \sigma = \frac{E(R)}{d_2(n)} \Rightarrow \hat{\sigma} = \frac{\overline{R}}{d_2(n)}$$

$$\mathrm{UCL} = d_2(n)\hat{\sigma} + 3d_3(n)\hat{\sigma} = \overline{R} + 3\frac{d_3(n)}{d_2(n)}\overline{R} = D_4(n)\overline{R}$$

$$\mathrm{CL} = d_2(n)\hat{\sigma} = \overline{R}$$

$$\mathrm{LCL} = d_2(n)\hat{\sigma} - 3d_3(n)\hat{\sigma} = \overline{R} - 3\frac{d_3(n)}{d_2(n)}\overline{R} = D_3(n)\overline{R}$$

K = 3

Figure 5.4: Xbar-R control charts with unknown parameters

Pay attention:

- **Make R chart before Xbar chart:**

  Both the charts (Xbar and R) need the estimation of the process standard deviation. On the contrary, the grand mean appears only in Xbar control chart. In case of problem with a strange sample with huge range, it can be detected by retrospectively project the control limits of the range control chart. Then I might find an assignable cause which would lead me to throw that sample.

  We would like to prevent that the Xbar control chart is affected by a wrong estimation of process std.

- **Recall the phase 1: more than 1 or 2 iterations are uncommon (pay attention to the assumptions)**

  When control limits are projected back, in case of OOC with assignable causes that sample should be taken out from the design. Then it's necessary to redesign the charts (mean and standard deviation changes). Once design phase is over control limits are projected back again. This is the loop mentioned and more than 2 iterations are rare, there might be problems with not met assumptions.

- **The confidence interval limits (LCL and UCL) in the $\bar{X}$ control chart represent the recursive application of confidence intervals on the mean over time t. This assumes that the sample variance is an accurate estimate of the population variance.**

  The standard deviation is estimated using:

  $$\hat{\sigma} = \frac{\bar{R}}{d_2(n)}$$

  The confidence interval for the mean is given by:

  $$\bar{X} \pm Z_{\alpha/2} \cdot \frac{\hat{\sigma}}{\sqrt{n}}$$

When control limits are computed using the factors THE CHARTS ARE BUILT ON K = 3, corresponding to alpha = 0,0027. In case you need a different alpha you cannot use the factors anymore and you must built your control limits from scratch computing K as "K = stats.norm.ppf(1-alpha/2)".

You might need a different alpha, for instance, when you are creating more than one couple of control charts for your process DUE TO THE BONFERRONI EFFECT (the same explained in the "Barlett test" section").

The following code is basically divided in 4 parts:

1. **Preparation of the DataFrame**
2. **Control limits computation (k=3)**
3. **Alternative control limits computation (K different from 3, thus alpha different from 0.0027)**
4. **Test on OOC presence**
5. **Charts plotting**

```
# Make a copy of your original dataframe
df_XR = df.copy()
# Compute the sample mean and the range row by row (Note: axis = 1 allows you to operate on rows rather than columns)
df_XR['sample mean'] = df_XR.mean(axis=1)
df_XR['sample range'] = df_XR.max(axis=1) - df.min(axis=1)
# Display the result
df_XR.head(10)
# Compute and print Xbarbar and R_mean
Xbarbar = df_XR['sample mean'].mean()
R_mean = df_XR['sample range'].mean()
print('Mean of the sample mean: ', Xbarbar)
print('Mean of the range: ', R_mean)

# Control limits with K = 3 (hence alpha = 0.0027)
n = #define the sample size
A2 = qda.constants.getA2(n)
D4 = qda.constants.getD4(n)
D3 = qda.constants.getD3(n)
# Compute the control limits
df_XR['UCL Xbarchart'] = Xbarbar + A2*R_mean
df_XR['CL Xbarchart'] = Xbarbar
df_XR['LCL Xbarchart'] = Xbarbar - A2*R_mean
df_XR['UCL Rchart'] = D4*R_mean
df_XR['CL Rchart'] = R_mean
df_XR['LCL Rchart'] = D3*R_mean
df_XR.head()

# Alternatively control limits with K different from 3 (other alpha value)
n = #define the sample size
d2 = qda.constants.getd2(n)
d3 = qda.constants.getd3(n)
std = R_mean/d2
alpha = #define alpha
k = stats.norm.ppf(1-alpha/2)
df_XR['UCL Xbarchart'] = Xbarbar + k*(std/np.sqrt(n))
df_XR['CL Xbarchart'] = Xbarbar
df_XR['LCL Xbarchart'] = Xbarbar - k*(std/np.sqrt(n))
df_XR['UCL Rchart'] = R_mean + k*(d3/d2)*R_mean
df_XR['CL Rchart'] = R_mean
df_XR['LCL Rchart'] = R_mean - k*(d3/d2)*R_mean
df_XR.head()

# Carry out the test for OOC detection
df_XR['Xbar_TEST1'] = np.where((df_XR['sample mean'] > df_XR['UCL Xbarchart']) |
                (df_XR['sample mean'] < df_XR['LCL Xbarchart']), df_XR['sample mean'], np.nan)
df_XR['R_TEST1'] = np.where((df_XR['sample range'] > df_XR['UCL Rchart']) |
                (df_XR['sample range'] < df_XR['LCL Rchart']), df_XR['sample range'], np.nan)
df_XR.head()

# Plot the R chart
plt.title('R chart')
plt.plot(df_XR['sample range'], color='b', linestyle='--', marker='o')
plt.plot(df_XR['UCL Rchart'], color='r')
plt.plot(df_XR['CL Rchart'], color='g')
plt.plot(df_XR['LCL Rchart'], color='r')
plt.ylabel('Sample range')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_XR)+.5, df_XR['UCL Rchart'].iloc[0], 'UCL = {:.3f}'.format(df_XR['UCL Rchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['CL Rchart'].iloc[0], 'CL = {:.3f}'.format(df_XR['CL Rchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['LCL Rchart'].iloc[0], 'LCL = {:.3f}'.format(df_XR['LCL Rchart'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_XR['R_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()

# Plot the Xbar chart
plt.title('Xbar chart')
plt.plot(df_XR['sample mean'], color='b', linestyle='--', marker='o')
plt.plot(df_XR['UCL Xbarchart'], color='r')
plt.plot(df_XR['CL Xbarchart'], color='g')
plt.plot(df_XR['LCL Xbarchart'], color='r')
plt.ylabel('Sample mean')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_XR)+.5, df_XR['UCL Xbarchart'].iloc[0], 'UCL = {:.3f}'.format(df_XR['UCL Xbarchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['CL Xbarchart'].iloc[0], 'CL = {:.3f}'.format(df_XR['CL Xbarchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['LCL Xbarchart'].iloc[0], 'LCL = {:.3f}'.format(df_XR['LCL Xbarchart'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_XR['Xbar_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 5.1: Xbar-R control charts manual computation

It is provided an automatic command for control chart creation. The first one takes as input k = 3 by default.

The alternative allows to specify the value of alpha for the control charts

```
# Create the control chart with the original data
df_XR = qda.ControlCharts.XbarR(df)

# in case of known parameters or other specifics
# e.g.: k_alpha = stats.norm.ppf(1-alpha/2)
# qda.ControlCharts.chartname(DataFrame name,mean = #inser mean, sigma = #insert true std, K = k_alpha, plotit = insert True or False)
```

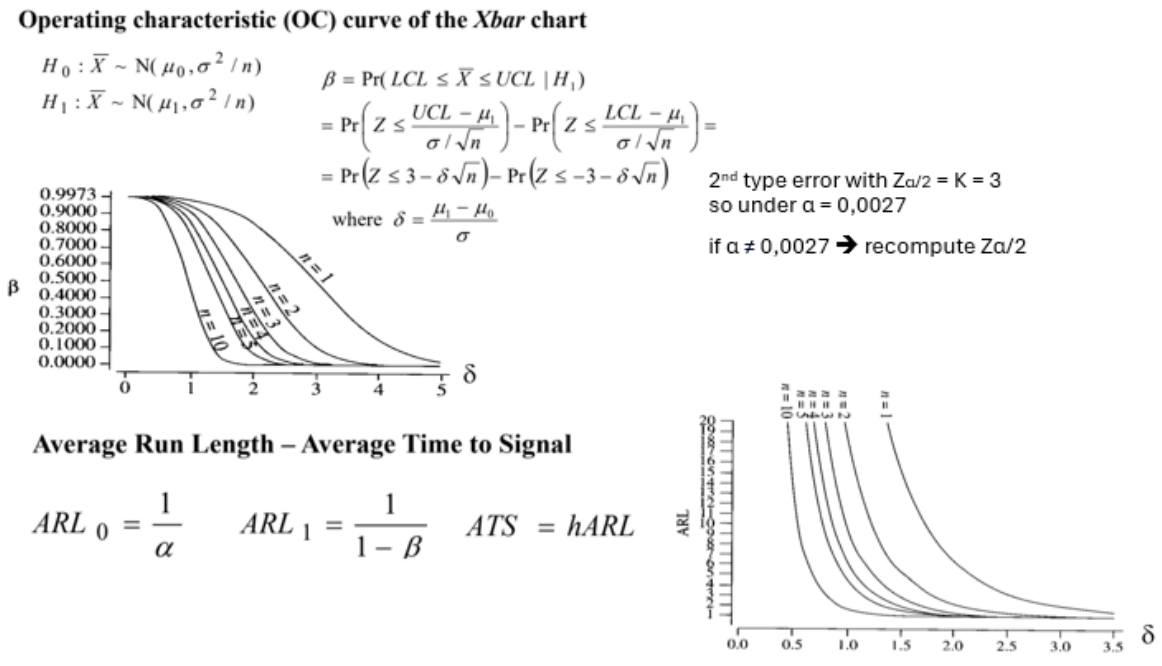
Code 5.2: Xbar-R control charts automatic computation

```
# Find the index of the I_TEST1 column different from NaN
OOC_idx = np.where(df_XR['Xbar_TEST1'].notnull())[0]
# Print the index of the OOC points
print('The index of the OOC point is: {}'.format(OOC_idx))
# Find the index of the I_TEST1 column different from NaN
OOC_idx = np.where(df_XR['R_TEST1'].notnull())[0]
# Print the index of the OOC points
print('The index of the OOC point is: {}'.format(OOC_idx))
```

Code 5.3: How to get OOC's index in Xbar-R

IF and only IF an assignable cause is linked to the OOC we have to go through the phase1 loop: exclude that observation from the control chart design and check if other out of controls show up. According to python rules it's suggested to make COPY of your ORIGINAL DATA. On this new dataset, make that whole observation line "nan" (not a number). Suppose "df2" is the copy of "df", then use "df2.iloc[row,:] = np.nan"). Finally run the automatic code for Xbar-R chart on this new dataset to be faster.

5.1.3. OC curve for Xbar and R chart + ARL + ATS



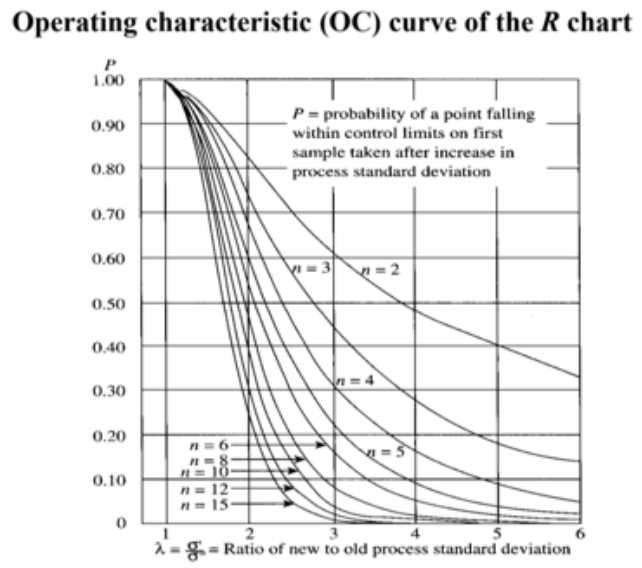


Figure 5.6: R chart: OC curve (only shape)

Figure 5.5: Xbar chart: OC curve + ARLo + ARL + ATS

```
# Define a range of values for beta
delta = np.linspace(0, 4, 100)
# Compute the corresponding beta values
beta = stats.norm.cdf(3 - delta*np.sqrt(n)) - stats.norm.cdf(-3 - delta*np.sqrt(n))

# Plot the beta values
plt.plot(delta, beta)
plt.xlabel('Delta')
plt.ylabel('Beta')
plt.title('Operating characteristic curve')
plt.show()

# Compute ARL using the previous values of beta
ARL = 1/(1-beta)

# Plot the ARL values
plt.plot(delta, ARL)
plt.xlabel('Delta')
plt.ylabel('ARL')
plt.title('Average run length')
plt.show()
```

Code 5.4: OC and ARL curves for Xbar control chart

5.1.4. Xbar-S charts (n constant)

Xbar-S are charts with better performances when the sample size "n" is higher than 6. This happens because the accuracy of the process variability esteemed is higher with S, rather than R, when "n" increases.

Xbar chart in Xbar-S is different from Xbar chart in Xbar-R when parameters are unknown. This is due to the different estimation of the process standard deviation.

On the contrary, if the parameters are already known the Xbar CC structure will be the same as previously.

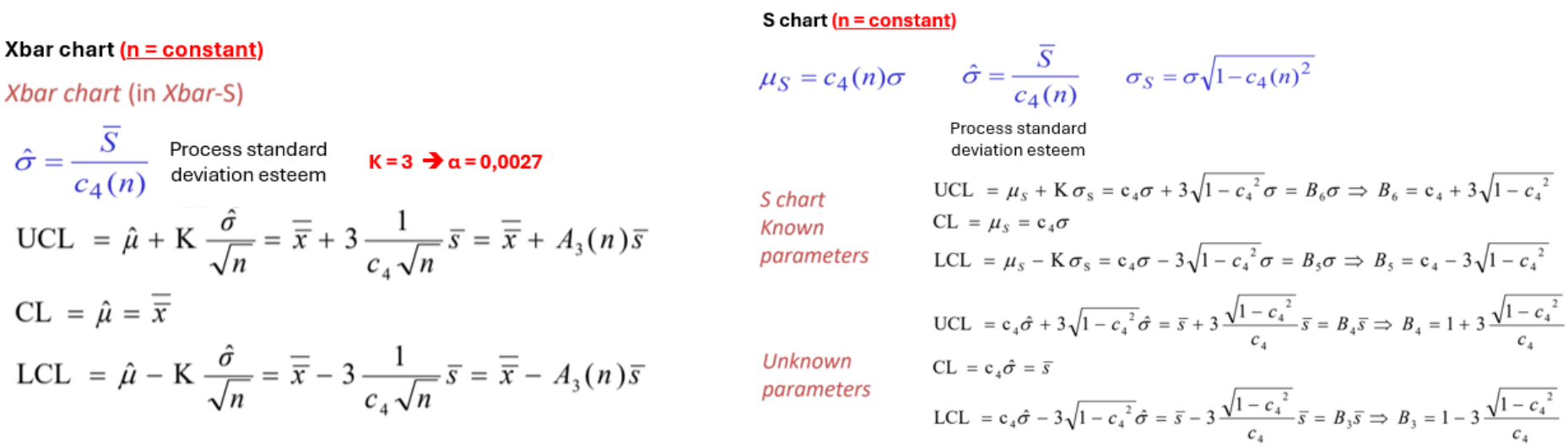


Figure 5.7: Xbar-S control chart with n = constant

5.1.5. Preparation of the DataFrame in case of normality violation for n = const > 2

In case normality assumption is not met we can proceed in this way: once data are stacked we can use Box Cox to recover the normality. If the transformation is effective, you will have a dataset of stacked normal data.

Then, you have to recover the original shape of your data (meaning to unstuck them).

Once the original shape is restored, the result won't be a DataFrame. For this reason a new DataFrame has to be defined and has to host our normal unstacked dataset.

Once ready, it is possible to create control charts directly on this new dataframe.

If the original observations were individuals (no batch: n = 2), obviously stacking data is not required since they are already in the right position to apply Box Cox.

```
# Make a copy of the data
df_XS = df.copy()
# Add a column with the mean of the rows
df_XS['sample mean'] = df_XS.mean(axis=1)
# Add a column with the standard deviation of the rows
df_XS['sample std'] = df_XS.std(axis=1)
# Inspect the dataset
df_XS.head()

Xbarbar = df_XS['sample mean'].mean()
S_mean = df_XS['sample std'].mean()
n = 5
k = 3
# k = stats.norm.ppf(1-alpha/2) once defined alpha
# no command to automatically get A3, B3, B4
A3 = k * 1 / (qda.constants.getc4(n) * np.sqrt(n))
B3 = np.maximum(1 - k * (np.sqrt(1-qda.constants.getc4(n)**2)) / (qda.constants.getc4(n)), 0)
B4 = 1 + k * (np.sqrt(1-qda.constants.getc4(n)**2)) / (qda.constants.getc4(n))


# Now we can compute the CL, UCL and LCL for Xbar and S
df_XS['UCL Xbarchart'] = Xbarbar + A3 * S_mean
df_XS['CL Xbarchart'] = Xbarbar
df_XS['LCL Xbarchart'] = Xbarbar - A3 * S_mean
df_XS['UCL Schart'] = B4 * S_mean
df_XS['CL Schart'] = S_mean
df_XS['LCL Schart'] = B3 * S_mean
# Inspect the dataset
df_XS.head()

df_XS['Xbar_TEST1'] = np.where((df_XS['sample mean'] > df_XS['UCL Xbarchart']) |
                (df_XS['sample mean'] < df_XS['LCL Xbarchart']), df_XS['sample mean'], np.nan)
df_XS['S_TEST1'] = np.where((df_XS['sample std'] > df_XS['UCL Schart']) |
                (df_XS['sample std'] < df_XS['LCL Schart']), df_XS['sample std'], np.nan)
df_XS.head()

# Plot the Xbar chart
plt.title('Xbar chart')
plt.plot(df_XS['sample mean'], color='b', linestyle='--', marker='o')
```

```
plt.plot(df_XS['UCL Xbarchart'], color='r')
plt.plot(df_XS['CL Xbarchart'], color='g')
plt.plot(df_XS['LCL Xbarchart'], color='r')
plt.ylabel('Sample mean')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_XS)+.5, df_XS['UCL Xbarchart'].iloc[0], 'UCL = {:.3f}'.format(df_XS['UCL Xbarchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XS)+.5, df_XS['CL Xbarchart'].iloc[0], 'CL = {:.3f}'.format(df_XS['CL Xbarchart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XS)+.5, df_XS['LCL Xbarchart'].iloc[0], 'LCL = {:.3f}'.format(df_XS['LCL Xbarchart'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_XS['Xbar_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()

# Plot the S chart
plt.title('S chart')
plt.plot(df_XS['sample std'], color='b', linestyle='--', marker='o')
plt.plot(df_XS['UCL Schart'], color='r')
plt.plot(df_XS['CL Schart'], color='g')
plt.plot(df_XS['LCL Schart'], color='r')
plt.ylabel('Sample S')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_XS)+.5, df_XS['UCL Schart'].iloc[0], 'UCL = {:.3f}'.format(df_XS['UCL Schart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XS)+.5, df_XS['CL Schart'].iloc[0], 'CL = {:.3f}'.format(df_XS['CL Schart'].iloc[0]), verticalalignment='center')
plt.text(len(df_XS)+.5, df_XS['LCL Schart'].iloc[0], 'LCL = {:.3f}'.format(df_XS['LCL Schart'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_XS['S_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 5.5: Xbar-S control charts manual computation

```
# X-bar and S charts
df_XS = qda.ControlCharts.XbarS(df)

# in case of known parameters or other specifics
# qda.ControlCharts.chartname(DataFrame name,mean = #inser mean, sigma = #insert true std, K = insert K, plotit = insert True or False)
```

Code 5.6: Xbar-S control charts automatic computation

```
# suppose you have alreade stacked your data, tested the normality, applied the transformation
# suppose the output of BoxCox is the array data_norm_unstack
data_norm_unstack = data_norm_stack.reshape(df.shape)   #leave the original dataframe "df"
# when you stack and the unstuck the data you get to arrays so you need to convert this array into a dataframe
data_norm_unstack = pd.DataFrame(data_norm_unstack, columns = df.columns)   #leave the original dataframe "df"
# The DataFrame is ready, apply the automatic code for control charts to so how data behave
```

Code 5.7: DataFrame preparation in case of normality violation

5.1.6. Xbar-S control charts (n variable)

$$\overline{\overline{x}} = \frac{\sum_{i=1}^{m} n_i \overline{x}_i}{\sum_{i=1}^{m} n_i} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}}{\sum_{i=1}^{m} n_i} \qquad s_p = \sqrt{\frac{\sum_{i=1}^{m}(n_i - 1)s_i^{\,2}}{\sum_{i=1}^{m}(n_i - 1)}}$$

$$d = \sum_{i=1}^{m} n_i - m + 1$$

$$\Rightarrow \hat{\sigma} = \frac{s_p}{c_4(d)}$$

$\overline{X}$ Chart

$$UCL_i = \overline{\overline{x}} + \frac{3}{\sqrt{n_i}}\hat{\sigma}$$

$$CL_i = \overline{\overline{x}}$$

$$LCL_i = \overline{\overline{x}} - \frac{3}{\sqrt{n_i}}\hat{\sigma}$$

S Chart
or, analogously

$$UCL_i = B_6(n_i)\hat{\sigma}$$

$$CL_i = c_4(n_i)\hat{\sigma}$$

$$LCL_i = B_5(n_i)\hat{\sigma}$$

$$UCL_i = \frac{c_4(n_i)}{c_4(d)} B_4(n_i) s_p$$

$$CL_i = \frac{c_4(n_i)}{c_4(d)} s_p$$

$$LCL_i = \frac{c_4(n_i)}{c_4(d)} B_3(n_i) s_p$$

Figure 5.8: Xbar-S control charts with "n" variable

This chart is not different from the previous one.

The only thing that changes is the way we esteem the standard deviation of the process. At the numerator we have Sp, which represent the weighted average of the observation over the batches (with "n" as weight).

```python
# Function to calculate sample variance and sample size for each row
def calc_variance_and_size(row):
valid_values = row.dropna()
sample_variance = valid_values.var(ddof=1) if len(valid_values) > 1 else np.nan
sample_size = len(valid_values)
return pd.Series([sample_variance, sample_size], index=['Sample variance', 'Sample size'])
# Apply the function to each row in the DataFrame
df[['Sample variance', 'Sample size']] = df.apply(calc_variance_and_size, axis=1)
df.head(25)

df['Sample mean'] = df.mean(axis=1)
Xbarbar = (df['Sample size'] * df['Sample mean']).sum() / df['Sample size'].sum()

numerator_Sp = ((df['Sample size'] - 1) * df['Sample variance']).sum()
denominator_Sp = (df['Sample size'] - 1).sum()
Sp = np.sqrt(numerator_Sp / denominator_Sp)

m = len(df)
d = df['Sample size'].sum() - m + 1

std = Sp/qda.constants.getc4(d)

print("Xbarbar:", Xbarbar)
print("Sp:", Sp)
print("d:", d)
print("Process standard deviation esteem: ", std)
```

Code 5.8: DataFrame preparation in case of normality violation

5.1.7. X and I-MR control charts for individuals

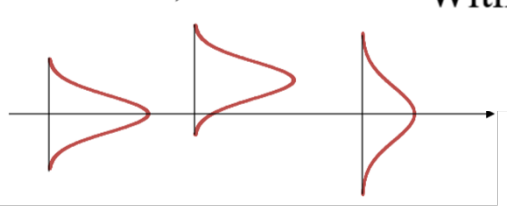
Control charts for individuals (*X* or *I control chart*)

Process with low throughput

Chemical processes

Overall company performance indicators: turnover, amount of provisions, customer satisfaction, …

$$X \overset{iid}{\sim} \mathrm{N}(\mu, \sigma^2) \quad (x_1, x_2, \ldots, x_n)$$



X Control chart ($V = X$)

With known parameters

$$\mathrm{UCL} = \mu_V + \mathrm{K}\sigma_V$$
$$\mathrm{CL} = \mu_V$$
$$\mathrm{LCL} = \mu_V - \mathrm{K}\sigma_V$$

With unknown parameters

$$\mathrm{UCL} = \mu + \mathrm{K}\sigma$$
$$\mathrm{CL} = \mu$$
$$\mathrm{LCL} = \mu - \mathrm{K}\sigma$$

$$\hat{\mu} = \bar{x}$$
$$\hat{\sigma} = \overline{MR} \,/\, d_2(2)$$

$$MR_i = | x_i - x_{i-1} | \quad \mathrm{i} = 2, \ldots, n$$
$$\overline{MR} = \frac{1}{n-1} \sum_{i=2,\ldots,n} MR_i$$

I Control Chart:

$$d_2(2) = 1.128$$
$$\bar{x} \pm 3\left(\frac{\overline{MR}}{d_2}\right) = \bar{x} \pm 2.66\,\overline{MR}$$

MR Control chart

$$UCL = D_4(2)\ \overline{MR} = (3.267)\ \overline{MR}$$
$$\mathrm{CL} = \overline{MR}$$
$$LCL = D_3(2)\ \overline{MR} = (0)\ \overline{MR} = 0$$

Figure 5.9: X and I-MR control charts for individuals

```
# Compute the moving ranges using the diff function
df['MR'] = df['column name'].diff().abs()
# Print out descriptive statistics of MR and time
df.describe()

n = 2
d2 = qda.constants.getd2(n)
D4 = qda.constants.getD4(n)
# make a copy of the data
df_IMR = df.copy()
# change the name of the column time to I
df_IMR.rename(columns={'column name':'I'}, inplace=True)
# Print the first 5 rows of the new dataframe
df_IMR.head()

# Create columns for the upper and lower control limits
df_IMR['I_UCL'] = df_IMR['I'].mean() + (3*df_IMR['MR'].mean()/d2)
df_IMR['I_CL'] = df_IMR['I'].mean()
df_IMR['I_LCL'] = df_IMR['I'].mean() - (3*df_IMR['MR'].mean()/d2)
df_IMR['MR_UCL'] = D4 * df_IMR['MR'].mean()
df_IMR['MR_CL'] = df_IMR['MR'].mean()
df_IMR['MR_LCL'] = 0
# Print the first 5 rows of the new dataframe
df_IMR.head()

# Define columns for possible violations of the control limits
df_IMR['I_TEST1'] = np.where((df_IMR['I'] > df_IMR['I_UCL']) |
                             (df_IMR['I'] < df_IMR['I_LCL']), df_IMR['I'], np.nan)
df_IMR['MR_TEST1'] = np.where((df_IMR['MR'] > df_IMR['MR_UCL']) |
                             (df_IMR['MR'] < df_IMR['MR_LCL']), df_IMR['MR'], np.nan)
# Print the first 5 rows of the new dataframe
df_IMR.head()

# Plot the I chart
plt.title('I chart')
plt.plot(df_IMR['I'], color='b', linestyle='--', marker='o')
plt.plot(df_IMR['I'], color='b', linestyle='--', marker='o')
plt.plot(df_IMR['I_UCL'], color='r')
plt.plot(df_IMR['I_CL'], color='g')
plt.plot(df_IMR['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_IMR)+.5, df_IMR['I_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_IMR['I_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_IMR)+.5, df_IMR['I_CL'].iloc[0], 'CL = {:.2f}'.format(df_IMR['I_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_IMR)+.5, df_IMR['I_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_IMR['I_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_IMR['I_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()

# Plot the MR chart
```

```
plt.title('MR chart')
plt.plot(df_IMR['MR'], color='b', linestyle='--', marker='o')
plt.plot(df_IMR['MR_UCL'], color='r')
plt.plot(df_IMR['MR_CL'], color='g')
plt.plot(df_IMR['MR_LCL'], color='r')
plt.ylabel('Moving Range')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_IMR)+.5, df_IMR['MR_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_IMR['MR_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_IMR)+.5, df_IMR['MR_CL'].iloc[0], 'CL = {:.2f}'.format(df_IMR['MR_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_IMR)+.5, df_IMR['MR_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_IMR['MR_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_IMR['MR_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 5.9: I-MR control charts manual computation

```
df_IMR = qda.ControlCharts.IMR(df, 'column name')

# tune the chart
# df_IMR = qda.ControlCharts.IMR(df, 'column name', K = insert K, plotit = True or False)
```

Code 5.10: I-MR control charts automatic computation

```
# Find the index of the I_TEST1 column different from NaN
OOC_idx = np.where(df_IMR['I_TEST1'].notnull())[0]
# Print the index of the OOC points
print('The index of the OOC point is: {}'.format(OOC_idx))
# Find the index of the MR_TEST1 column different from NaN
OOC_idx = np.where(df_IMR['MR_TEST1'].notnull())[0]
# Print the index of the OOC points
print('The index of the OOC point is: {}'.format(OOC_idx))
```

Code 5.11: How to get OOC's index in I-MR

IF and only IF an assignable cause is linked to the OOC we have to go through the phase1 loop: exclude that observation from the control chart design and check if other out of controls show up. According to python rules it's suggested to make COPY of your ORIGINAL DATA. On this new dataset, make that observation "nan" (not a number). Suppose "df2" is the copy of "df", then use "df2.iloc[row,0] = np.nan"). Finally run the automatic code for I-MR chart on this new dataset to be faster.

Factors for Constructing Variables Control Charts

Observations in Sample, n	Chart for Averages: Factors for Control Limits A		A_3	Chart for Averages: Factors for Center Line c_4	$1/c_4$	Chart for Standard Deviations: Factors for Control Limits B_3	B_4	B_5	B_6	Factors for Center Line d_2	$1/d_2$	Chart for Ranges: Factors for Control Limits d_3	D_1	D_2	D_3	D_4
2	2.121	1.880	2.659	0.7979	1.2533	0	3.267	0	2.606	1.128	0.8865	0.853	0	3.686	0	3.267
3	1.732	1.023	1.954	0.8862	1.1284	0	2.568	0	2.276	1.693	0.5907	0.888	0	4.358	0	2.574
4	1.500	0.729	1.628	0.9213	1.0854	0	2.266	0	2.088	2.059	0.4857	0.880	0	4.698	0	2.282
5	1.342	0.577	1.427	0.9400	1.0638	0	2.089	0	1.964	2.326	0.4299	0.864	0	4.918	0	2.114
6	1.225	0.483	1.287	0.9515	1.0510	0.030	1.970	0.029	1.874	2.534	0.3946	0.848	0	5.078	0	2.004
7	1.134	0.419	1.182	0.9594	1.0423	0.118	1.882	0.113	1.806	2.704	0.3698	0.833	0.204	5.204	0.076	1.924
8	1.061	0.373	1.099	0.9650	1.0363	0.185	1.815	0.179	1.751	2.847	0.3512	0.820	0.388	5.306	0.136	1.864
9	1.000	0.337	1.032	0.9693	1.0317	0.239	1.761	0.232	1.707	2.970	0.3367	0.808	0.547	5.393	0.184	1.816
10	0.949	0.308	0.975	0.9727	1.0281	0.284	1.716	0.276	1.669	3.078	0.3249	0.797	0.687	5.469	0.223	1.777
11	0.905	0.285	0.927	0.9754	1.0252	0.321	1.679	0.313	1.637	3.173	0.3152	0.787	0.811	5.535	0.256	1.744
12	0.866	0.266	0.886	0.9776	1.0229	0.354	1.646	0.346	1.610	3.258	0.3069	0.778	0.922	5.594	0.283	1.717
13	0.832	0.249	0.850	0.9794	1.0210	0.382	1.618	0.374	1.585	3.336	0.2998	0.770	1.025	5.647	0.307	1.693
14	0.802	0.235	0.817	0.9810	1.0194	0.406	1.594	0.399	1.563	3.407	0.2935	0.763	1.118	5.696	0.328	1.672
15	0.775	0.223	0.789	0.9823	1.0180	0.428	1.572	0.421	1.544	3.472	0.2880	0.756	1.203	5.741	0.347	1.653
16	0.750	0.212	0.763	0.9835	1.0168	0.448	1.552	0.440	1.526	3.532	0.2831	0.750	1.282	5.782	0.363	1.637
17	0.728	0.203	0.739	0.9845	1.0157	0.466	1.534	0.458	1.511	3.588	0.2787	0.744	1.356	5.820	0.378	1.622
18	0.707	0.194	0.718	0.9854	1.0148	0.482	1.518	0.475	1.496	3.640	0.2747	0.739	1.424	5.856	0.391	1.608
19	0.688	0.187	0.698	0.9862	1.0140	0.497	1.503	0.490	1.483	3.689	0.2711	0.734	1.487	5.891	0.403	1.597
20	0.671	0.180	0.680	0.9869	1.0133	0.510	1.490	0.504	1.470	3.735	0.2677	0.729	1.549	5.921	0.415	1.585
21	0.655	0.173	0.663	0.9876	1.0126	0.523	1.477	0.516	1.459	3.778	0.2647	0.724	1.605	5.951	0.425	1.575
22	0.640	0.167	0.647	0.9882	1.0119	0.534	1.466	0.528	1.448	3.819	0.2618	0.720	1.659	5.979	0.434	1.566
23	0.626	0.162	0.633	0.9887	1.0114	0.545	1.455	0.539	1.438	3.858	0.2592	0.716	1.710	6.006	0.443	1.557
24	0.612	0.157	0.619	0.9892	1.0109	0.555	1.445	0.549	1.429	3.895	0.2567	0.712	1.759	6.031	0.451	1.548
25	0.600	0.153	0.606	0.9896	1.0105	0.565	1.435	0.559	1.420	3.931	0.2544	0.708	1.806	6.056	0.459	1.541

Per $n \geq 25$: $A = \frac{3}{\sqrt{n}}$, $A_3 = \frac{3}{c_4\sqrt{n}}$, $c_4 = \frac{4(n-1)}{4n-3}$, $B_3 = 1 - \frac{3}{c_4\sqrt{2(n-1)}}$, $B_4 = 1 + \frac{3}{c_4\sqrt{2(n-1)}}$, $B_5 = c_4 - \frac{3}{\sqrt{2(n-1)}}$, $B_6 = c_4 + \frac{3}{\sqrt{2(n-1)}}$.

Figure 5.10: Table of factors for control charts in SPC

5.2. Probabilistic control chart

Traditional control charts use control limits based on fixed statistical properties, typically assuming a normal distribution of process data. Examples include X-bar, R, S, and I-MR charts, where control limits are often set at ±3 standard deviations from the mean. This approach assumes that the process data fits a normal distribution. As mentioned, if the original data are not normally distributed then a normality transformation is needed if we want to apply traditional control charts.

Normality transformation on a statistic (such as the MR) and application of a traditional control chart (usually I-chart).

Sometimes, the transformation approach involves transforming statistics derived from the original data, such as Moving Range (MR). In this case is the moving range to be transformed and fit a normal distribution and not the data from where it comes from. Once the transformation is carried out, usually an "I control chart" is applied on this transformed data. This approach change the perspective and considers the statistic as a new process variable on which traditional control chart can be fitted. To sum up, this method makes the data suitable for traditional control chart by normalizing the "data statistic", thus making the application of control limits statistically valid. However, this approach is not inherently probabilistic as it manipulates the data to fit a predefined model (normal distribution), rather than directly using the true distribution of the original data.

Pure probabilistic control chart.

Probabilistic control charts, on the other hand, set control limits based on the actual probability distribution of the process data without transforming it to fit a normal distribution. This method directly uses the true distribution of the process data (e.g., half-normal, Poisson) to determine control limits. These charts are more flexible and can accurately reflect the true variability of the process, leading to better detection of special cause variations.

Merging the ideas

When you transform a statistic, such as MR, into a normal distribution and then fit an I-chart, you are not creating a probabilistic control chart. Instead, you are applying traditional control chart methods to transformed data. A probabilistic control chart directly uses the original data's true distribution to set control limits without requiring such transformations. This distinction is crucial for understanding the difference between traditional control charts applied to transformed data and truly probabilistic control charts.

- $UCL = D_{1-\alpha/2}\frac{\overline{MR}}{d_2}$
- $LCL = D_{\alpha/2}\frac{\overline{MR}}{d_2}$

For $n = 2$ (Alwan, Appendix A):

- $D_{1-\alpha/2} = \sqrt{2}z_{alpha/4}$
- $D_{\alpha/2} = \sqrt{2}z_{1/2-alpha/4}$

Figure 5.11: Probabilistic control chart on Moving Range (half-normal approximation)

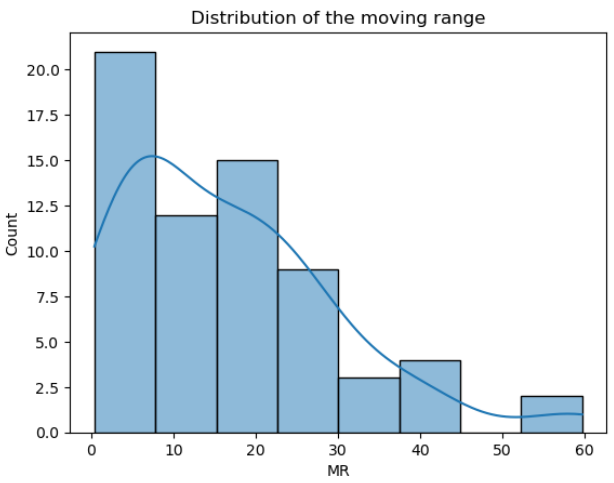


Figure 5.12: Example of MR distribution (note the half normal)

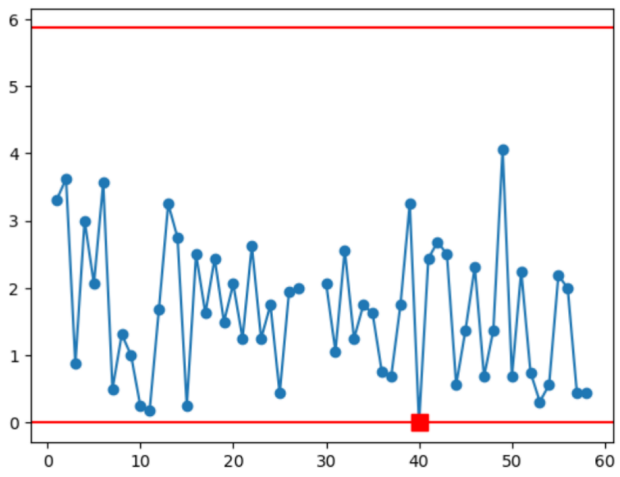


Figure 5.13: Probabilistic control chart outcome

We can consider what is shown above as a pure probabilistic control chart on the moving range.

Once your DataFrame "df" is ready to be exploited run the following code

```
alpha = #define alpha
K_alpha = stats.norm.ppf(1-alpha/2)
print('New K value = %.3f' % K_alpha)
# Compute the traditional I-MR table withouth displaying the charts
# It is useful just to get the MR faster
df_IMR = qda.ControlCharts.IMR(df, 'GM', K = K_alpha, plotit=False)
# Compute the control limits
D_UCL = np.sqrt(2) * stats.norm.ppf(1-alpha/4)
D_LCL = np.sqrt(2) * stats.norm.ppf(1 - (1/2 - alpha/4))
MR_UCL = D_UCL * df_IMR['MR'].mean()/qda.constants.getd2(2)
MR_LCL = D_LCL * df_IMR['MR'].mean()/qda.constants.getd2(2)
print('MR_UCL = %.4f' % MR_UCL)
print('MR_LCL = %.4f' % MR_LCL)
# Carry out the test for OOC
df_IMR['MR_TEST1'] = np.where((df_IMR['MR'] > MR_UCL) | (df_IMR['MR'] < MR_LCL), df_IMR['MR'], np.nan)
plt.plot(df_IMR['MR'], 'o-')
plt.axhline(MR_UCL, color = 'r')
plt.axhline(MR_LCL, color = 'r')
plt.plot(df_IMR['MR_TEST1'], linestyle='none', marker='s', color='r', markersize = 10)
plt.show()
```

Code 5.12: Moving range probabilistic control chart

As we said it is also possible to create traditional control chart with PROBABILISTIC CONTROL LIMITS. The procedure consist in:

- **Create the traditional I-MR control chart table on the original data (use the automatic command for I-MR creation. Use the command the command "plotit = False" to avoid the display of the traditional I-MR;**

- **Extract the moving range column from the traditional "I-MR C.C." and save it in a new DataFrame. This one will be the DataFrame on which we are going to compute the control chart;**
- **If there are values equal to 0, use transformation to the power of lambda = 0.4, according to the theoretical result shown in Alwan textbook. Otherwise use BoxCox;**
- **Check the normality of the transformed moving range;**
- **Rename the column from "MR" to "MR_transformed" otherwise the automatic control chart command would not work properly;**
- **Use the automatic command for I-MR charts creation on "MR_transformed", the objective is to use the I-chart on the normally distributed moving range; Use the commands "K = K_alpha" to tune your control limits and "plotit = false" to avoid the display of the charts. The reason is that we are only interested in the I-chart of the moving range statistic. It would not make any sense to do the MR chart on the moving range statistic (even if it was transformed)**
- **Display the I-chart manually**

```
# Exploit the automatic command to get the traditional I-MR table
# Change K and plotit parameters according to your needs if needed, but they do not influence the farthest steps
df_IMR = qda.ControlCharts.IMR(df, 'original data column name', K = K_alpha, plotit = False)

# Extract and save the column 'MR' from the table computed before
df_Icc_MR = pd.DataFrame(df_IMR['MR'])
df_Icc_MR.head(10)
```

```
def transform_data(df_Icc_MR):
    if all(df_Icc_MR['MR'][1:] > 0):
        # Apply Box-Cox transformation
        data_norm, lmbda = stats.boxcox(df_Icc_MR['MR'][1:])
        print('Box Cox transformation has been applied')
        print('Lambda = %.3f' % lmbda)

        # Box-Cox normality plot
        fig = plt.figure()
        ax = fig.add_subplot(111)
        stats.boxcox_normplot(df_Icc_MR['MR'][1:], -2, 2, plot=ax)
        ax.grid(True)
        plt.show()

        # Plot the histogram with KDE
        sns.histplot(data_norm, kde=True)
        plt.title('Distribution of your data after Box Cox')
        plt.show()

        # Shapiro Wilk test
        _, pval_SW = stats.shapiro(data_norm)
        print('pvalue shapiro wilk: ', pval_SW)
        if pval_SW > 0.05:
            print('Data after Box Cox are normally distributed')
        else:
            print('Data after Box Cox are not normally distributed')

        # Plot the qqplot
        stats.probplot(data_norm, dist="norm", plot=plt)
        plt.show()

        # Overwrite the transformed data in the original dataframe
        df_Icc_MR['MR'] = data_norm
    else:
        # Get the index of the value equal to 0 if any
        idx = df_Icc_MR['MR'][df_Icc_MR['MR'] == 0].index[0]

        # Change it to NaN
        df_Icc_MR['MR'].iloc[idx] = np.nan

        # Apply a normal transformation
        df_Icc_MR['MR'] = df_Icc_MR['MR'].transform(lambda x: ((x**0.4)))
        print('Alwan transformation to the power of lambda = 0.4 is applied')

        # Replace the NaN value with 0
        df_Icc_MR['MR'].iloc[idx] = 0

        # Perform normality checks and plots for Alwan transformation
        # Shapiro Wilk test
        _, pval_SW = stats.shapiro(df_Icc_MR['MR'].dropna())
        print('pvalue shapiro wilk: ', pval_SW)
        if pval_SW > 0.05:
            print('Data after Alwan transformation are normally distributed')
        else:
            print('Data after Alwan transformation are not normally distributed')

        # Plot the qqplot
        stats.probplot(df_Icc_MR['MR'].dropna(), dist="norm", plot=plt)
        plt.show()

        # Plot the histogram with KDE
        sns.histplot(df_Icc_MR['MR'].dropna(), kde=True)
```

```
        plt.title('Distribution of your data after Alwan transformation')
        plt.show()

    return df_Icc_MR


df_Icc_MR = transform_data(df_Icc_MR)

# Rename the MR column in MR_transformed to avoid wrong subrscriptions of the data
df_Icc_MR = df_Icc_MR.rename(columns = {'MR': 'MR_transformed'})
# Use the automatic command WITH THE SCOPE OF GETTING THE I-CHART ON THE TRANSFORMED MOVING RANGE
df_Icc_MR = qda.ControlCharts.IMR(df_Icc_MR, 'MR_transformed', K = K_alpha, plotit=False)

# Plot the I chart
plt.title('I chart for Moving Range stastic transformed by BoxCox')
plt.plot(df_Icc_MR['MR_transformed'], color='b', linestyle='--', marker='o')
plt.plot(df_Icc_MR['MR_transformed'], color='b', linestyle='--', marker='o')
plt.plot(df_Icc_MR['I_UCL'], color='r')
plt.plot(df_Icc_MR['I_CL'], color='g')
plt.plot(df_Icc_MR['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_Icc_MR)+.5, df_Icc_MR['I_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_Icc_MR['I_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_Icc_MR)+.5, df_Icc_MR['I_CL'].iloc[0], 'CL = {:.2f}'.format(df_Icc_MR['I_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_Icc_MR)+.5, df_Icc_MR['I_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_Icc_MR['I_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_Icc_MR['I_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 5.13: Normality transformation on moving range statistic + I-chart on the outcome

Probabilistic control chart and I-chart on the moving range statistic are usually designed when I-chart and MR-chart of your data do not agree each other in terms of OOC and there is no assignable cause for them

Note: we are going to introduce SPC non iid where usually regression models are created. We will see that there are "Special Causes control charts" that are I-MR control chart on the residuals of the model. It can happen this I-MR chart on residuals do not agree each other, so in that case it might be crucial to try a different approach. When it happens we should keep the I-chart on the residuals and compute a new control chart for the moving range of the model residuals. The methods are the ones described above

6 | SPC non iid

When the iid assumption is not met it is still possible to create control charts, but unfortunately we cannot apply Shewhart's method anymore.

There are several types of charts that can be performed.

6.1. Trend control charts and model-based control charts

Trend control charts are used when your data has a drift of the mean and there is correlation with time. The concept is pretty simple: fit the regression model and your fitted values will be the center line of your control chart.

The upper and lower control limits are simply computed with

A TREND control chart will be based on the residuals of the trend model.

$$UCL = \beta_0 + \beta_1 \cdot t + k\frac{\overline{MR}}{d_2(2)}$$

$$CL = \beta_0 + \beta_1 \cdot t$$

$$LCL = \beta_0 + \beta_1 \cdot t - k\frac{\overline{MR}}{d_2(2)}$$

Which MR should be used?

To be rigorous, we should use the MR of the residuals. Alwan states that the difference between the MR of the residuals and the MR of the original data is negligible, but let's use the MR of the residuals.

K = 3 ➔ α = 0,0027

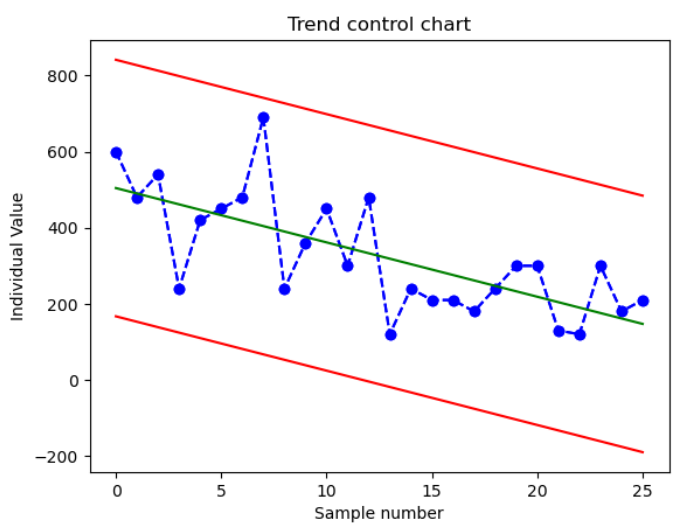


Figure 6.1: Trend control chart

Once you have fitted your regression model and fitted values + residuals are stored in "model", apply this code for the control chart creation

```
df_res = pd.DataFrame({'I': model.resid})
df_res['MR'] = df_res['I'].diff().abs()
df_res.describe()
n = 2
d2 = qda.constants.getd2(n)
df_trendchart = pd.DataFrame()
# The centerl line is made of fitted values, while the individuals displayed are the original data
df_trendchart['I'] = df['insert original data']
df_trendchart['I_CL'] = model.fittedvalues
df_trendchart['I_UCL'] = df_trendchart['I_CL'] + 3 * df_res['MR'].mean() / d2
df_trendchart['I_LCL'] = df_trendchart['I_CL'] - 3 * df_res['MR'].mean() / d2
df_trendchart['I_TEST1'] = np.where((df_trendchart['I'] > df_trendchart['I_UCL']) | (df_trendchart['I'] < df_trendchart['I_LCL']), df_trendchart['I'],
    np.nan)

# Plot the I chart
plt.title('Trend control chart')
plt.plot(df_trendchart['I'], color='b', linestyle='--', marker='o')
plt.plot(df_trendchart['I'], color='b', linestyle='--', marker='o')
plt.plot(df_trendchart['I_UCL'], color='r')
plt.plot(df_trendchart['I_CL'], color='g')
plt.plot(df_trendchart['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# highlight the points that violate the alarm rules
plt.plot(df_trendchart['I_TEST1'], linestyle='none', marker='s',
        color='r', markersize=10)
plt.show()
```

Code 6.1: Trend control chart plot on non-negative values and flattening control limits

Suppose that your quality feature is always non-negative (as it was for this example). It can happens, for instance, when you are measuring time (production time, service time, and so on...)

In that case it is important to correct the lower control limit by making it flat from the istant in which it overcome zero and becomes negative.

When we correct the lower control limit it becomes crucial to modify also the upper control limit.

If the upper control limit (UCL) were not changed to be flat where the lower control limit (LCL) is flat, the following issues could arise:

- **Inconsistent Control Limits: The distance between the UCL and LCL would vary after the LCL becomes flat, making the control limits less reliable and more difficult to interpret.**
- **Increased Type I Error Rate: With the LCL flat at zero and the UCL remaining sloped, the control limits would narrow over time. This could lead to more points falling outside the UCL, increasing the Type I error rate (false positives), where the process appears out of control even though it is in control.**
- **Misleading Process Control Signals: A narrower control limit due to the unchanged UCL would result in more frequent false alarms, suggesting unnecessary adjustments to the process and increasing variability.**

By adjusting the UCL to be flat, the control chart accurately reflects process variation and maintains the intended error rates.

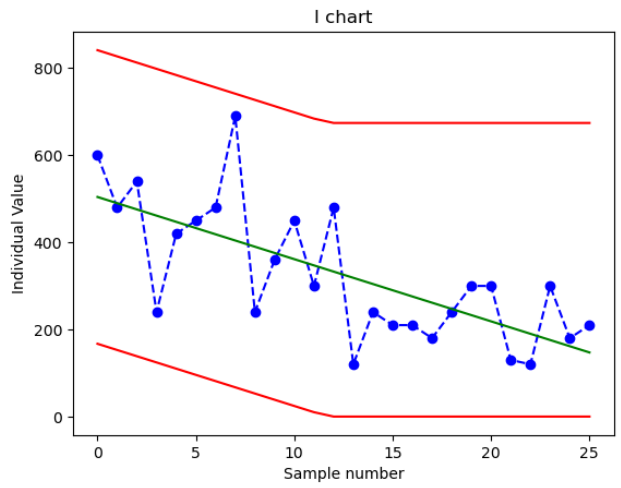


Figure 6.2: Trend control chart on non negative values

The following code is just for plotting, but the control chart creation have to be recovered from the previous script.

```
# Set the LCL to be 0 where it's less than 0
df_trendchart['I_LCL'] = np.where((df_trendchart['I_LCL'] < 0), 0, df_trendchart['I_LCL'])
# Find the first index where the LCL is 0
flat_index = df_trendchart.index[df_trendchart['I_LCL'] == 0][0]
# Calculate the distance between UCL and LCL before the flat point
constant_distance = df_trendchart['I_UCL'][flat_index - 1] - df_trendchart['I_LCL'][flat_index - 1]
# Set the UCL to be flat from the flat_index onward
df_trendchart.loc[flat_index:, 'I_UCL'] = df_trendchart['I_LCL'][flat_index:] + constant_distance
# Plot the I chart
plt.title('Trend control chart')
plt.plot(df_trendchart['I'], color='b', linestyle='--', marker='o')
plt.plot(df_trendchart['I_UCL'], color='r')
plt.plot(df_trendchart['I_CL'], color='g')
plt.plot(df_trendchart['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# Highlight the points that violate the alarm rules
plt.plot(df_trendchart['I_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 6.2: Trend control chart plot on non-negative values and flattening control limits

Trend control chart are just a specific type of monitoring system within the set of the "model-based control chart": it is possible to create this type of control chart on more complicated models. Once the model is created and fitted values + residuals are stored in "model" you can run the first code of this subsection on it. REMEMBER TO CHANGE THE NAME OF THE COLUMN OF YOUR ORIGINAL DATA AND DO NOT FORGET TO CHANGE THE TITLE.

Example of more complicated model-based control chart:

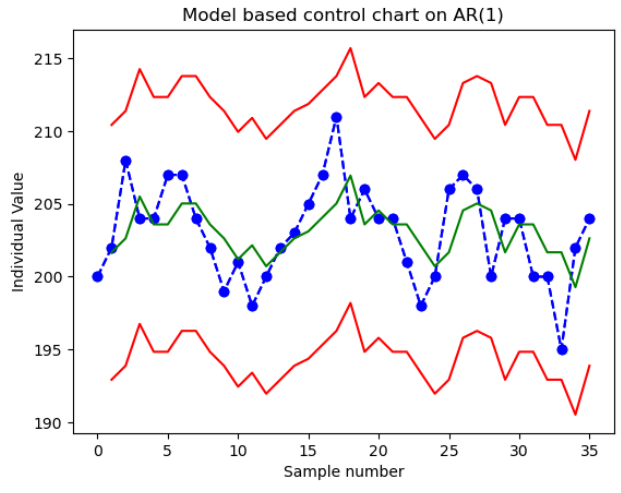


Figure 6.3: Model-based control chart

6.2. Fitted Value control chart and Special Cause control chart

The fitted value chart is just a plot of the fitted values vs original data.

The special cause control chart is just a I-MR control chart on the residuals after model regression fitting.

The starting point is exactly the same: Once you have fitted your regression model and fitted values + residuals are stored in "model", apply this code for the control chart creation

```
df_res = pd.DataFrame({'I': model.resid})
df_res['MR'] = df_res['I'].diff().abs()
df_res.describe()

n = 2
d2 = qda.constants.getd2(n)
D4 = qda.constants.getD4(n)
# Create columns for the upper and lower control limits
df_res['I_UCL'] = df_res['I'].mean() + (3*df_res['MR'].mean()/d2)
df_res['I_CL'] = df_res['I'].mean()
df_res['I_LCL'] = df_res['I'].mean() - (3*df_res['MR'].mean()/d2)
df_res['MR_UCL'] = D4 * df_res['MR'].mean()
df_res['MR_CL'] = df_res['MR'].mean()
df_res['MR_LCL'] = 0
# Print the first 5 rows of the new dataframe
df_res.head()

# Define columns for possible violations of the control limits
df_res['I_TEST1'] = np.where((df_res['I'] > df_res['I_UCL']) |
(df_res['I'] < df_res['I_LCL']), df_res['I'], np.nan)
df_res['MR_TEST1'] = np.where((df_res['MR'] > df_res['MR_UCL']) |
(df_res['MR'] < df_res['MR_LCL']), df_res['MR'], np.nan)
# Print the first 5 rows of the new dataframe
df_res.head()

# Plot the I chart
plt.title('I chart')
plt.plot(df_res['I'], color='b', linestyle='--', marker='o')
plt.plot(df_res['I'], color='b', linestyle='--', marker='o')
plt.plot(df_res['I_UCL'], color='r')
plt.plot(df_res['I_CL'], color='g')
plt.plot(df_res['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_res)+.5, df_res['I_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_res['I_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_res)+.5, df_res['I_CL'].iloc[0], 'CL = {:.2f}'.format(df_res['I_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_res)+.5, df_res['I_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_res['I_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_res['I_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()

plt.title('MR chart')
plt.plot(df_res['MR'], color='b', linestyle='--', marker='o')
plt.plot(df_res['MR_UCL'], color='r')
plt.plot(df_res['MR_CL'], color='g')
plt.plot(df_res['MR_LCL'], color='r')
plt.ylabel('Moving Range')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_res)+.5, df_res['MR_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_res['MR_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_res)+.5, df_res['MR_CL'].iloc[0], 'CL = {:.2f}'.format(df_res['MR_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_res)+.5, df_res['MR_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_res['MR_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_res['MR_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

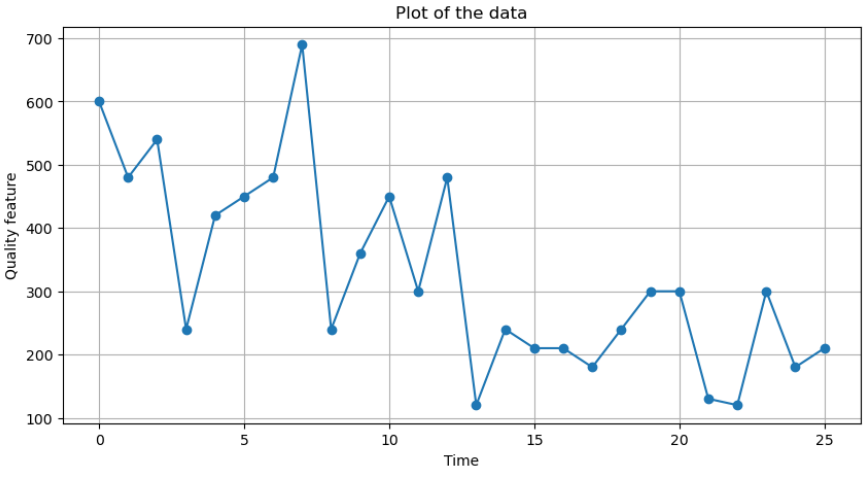
Code 6.3: Special Cause control chart manual computation

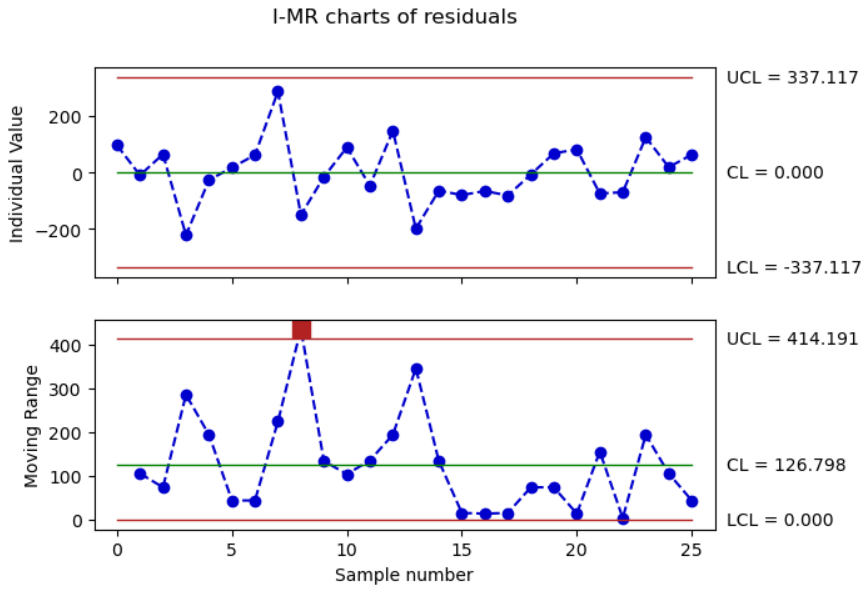
```
df_res = pd.DataFrame({'residuals': model.resid})
df_IMR = qda.ControlCharts.IMR(df_res, 'residuals')
```

Code 6.4: Special Cause control chart automatic computation

6.2.1. SCC charts: handling OOCs in case of assignable causes

Suppose now that one of your chart shows an out of control.





Suppose we can find an assignable cause for that out of control.

The way to proceed is to fit a model with a dummy variable. Dummy should be defined as follows:

- **dummy = 1 for the out of control observation index**
- **dummy = 0 for all the other indexes**

Once the model is ready and the residuals are NID it is possible to recompute the control charts.

The final outcome should be something like that:

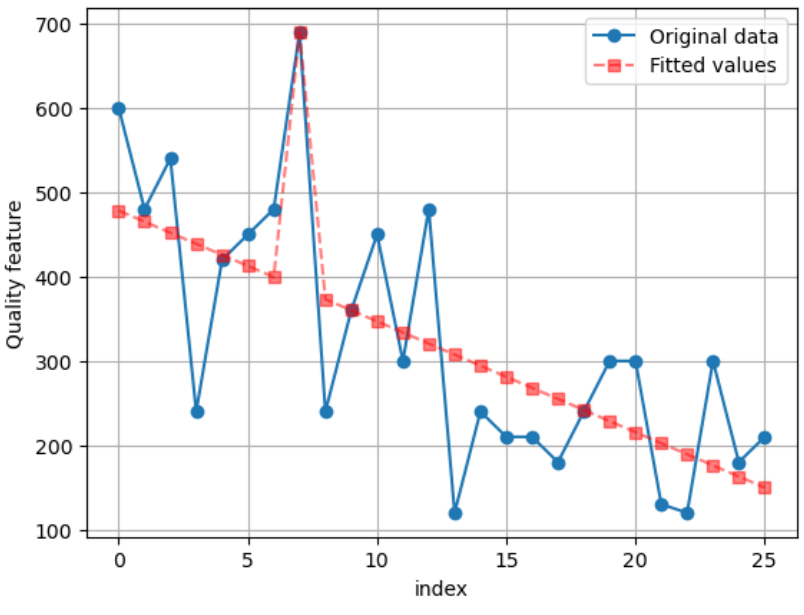


Figure 6.4: Fitted Value chart

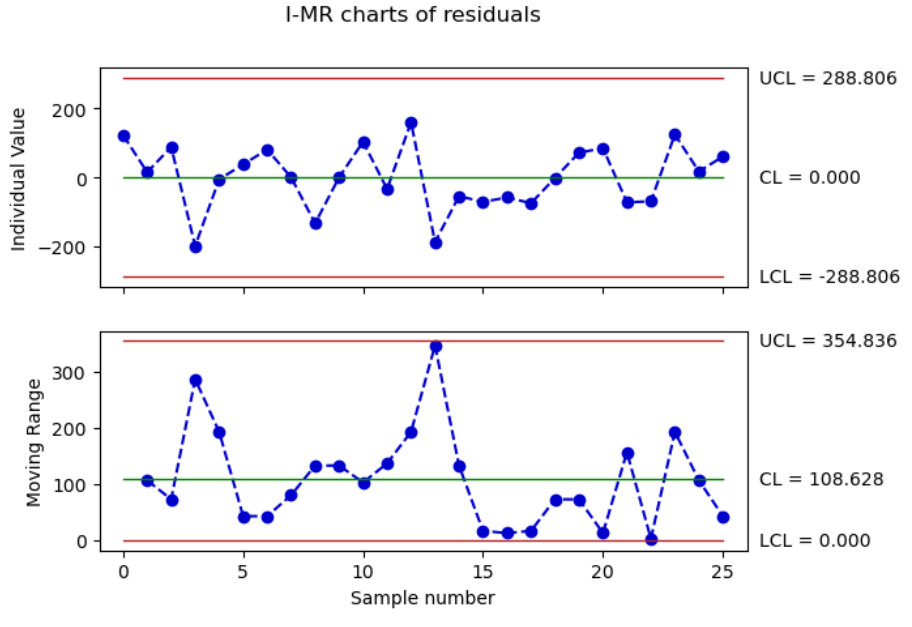


Figure 6.5: Special Cause control chart

6.2.2. SCC charts: handling OOCs in case of NO ASSIGNABLE CAUSES

"Special Causes control charts" are I-MR control chart on the residuals of the model. It can happen this I-MR chart on residuals do not agree each other in terms of OOCs and there are no assignable causes for that (or them). In that case it might be crucial to try a different approach. When it happens we should keep the I-chart on the residuals and compute a new control chart for the moving range of the model residuals.

To sum up, use the control chart introduced below when I-chart and MR-chart ON RESIDUALS do not agree each other, no assignable causes are found and the moving range results to be not normal with Shapiro Wilk test (keep in mind it has to be tested)

If one of the previous condition is missed, particularly if the charts actually agree each other, you have to classify those OOC as false alarms and the control chart design process gets over

This control chart might be of two types:

- **Probabilistic control chart for the residuals' moving range**
- **I-chart on the residuals' moving range once it is transformed by BoxCox (try this one first)**

Let's provide the code for the second one. Recover the first method from the "probabilistic control chart section".

```python
# If this code is applied is just because the previous one is applied. So let's resume from that
df_res = pd.DataFrame({'residuals': model.resid})
df_res['MR'] = df_res['residuals'].diff().abs()
df_IMR = qda.ControlCharts.IMR(df_res, 'residuals')
# We find out that there are OOC on the MR chart non linked to assignable causes, so first run the normality checks on df_res['MR']

# HERE THE CODE STARTS
def transform_data(df_res):
    if all(df_res['MR'][1:] > 0):
        # Apply Box-Cox transformation
        data_norm, lmbda = stats.boxcox(df_res['MR'][1:])
        print('Lambda = %.3f' % lmbda)
        print('Box Cox transformation has been applied')
        # Box-Cox normality plot
        fig = plt.figure()
        ax = fig.add_subplot(111)
        stats.boxcox_normplot(df_res['MR'][1:], -2, 2, plot=ax)
        ax.grid(True)
        plt.show()
        # Shapiro Wilk test
        _, pval_SW = stats.shapiro(data_norm)
        print('pvalue shapiro wilk: ', pval_SW)
        if pval_SW > 0.05:
            print('Data after Box Cox are normally distributed')
        else:
            print('Data after Box Cox are not normally distributed')
        # Plot the qqplot
        stats.probplot(data_norm, dist="norm", plot=plt)
        plt.show()
        # Plot the histogram with KDE
        sns.histplot(data_norm, kde=True)
        plt.title('Distribution of your data after Box Cox')
        plt.show()
        # Save the transformed data as a DataFrame
        df_SCC = pd.DataFrame(data_norm, columns=['MRres_BC'])
    else:
        # Get the index of the value equal to 0 if any
        idx = df_res['MR'][df_res['MR'] == 0].index[0]
        # Change it to NaN
        df_res['MR'].iloc[idx] = np.nan
        # Apply a normal transformation
        df_res['MR'] = df_res['MR'].transform(lambda x: ((x**0.4)))
        print('Alwan transformation to the power of lambda = 0.4 is applied')
        # Replace the NaN value with 0
        df_res['MR'].iloc[idx] = 0
        # Perform normality checks and plots for Alwan transformation
        # Shapiro Wilk test
        _, pval_SW = stats.shapiro(df_res['MR'].dropna())
        print('pvalue shapiro wilk: ', pval_SW)
        if pval_SW > 0.05:
            print('Data after Alwan transformation are normally distributed')
        else:
            print('Data after Alwan transformation are not normally distributed')
        # Plot the qqplot
        stats.probplot(df_res['MR'].dropna(), dist="norm", plot=plt)
        plt.show()
        # Plot the histogram with KDE
        sns.histplot(df_res['MR'], kde=True)
        plt.title('Distribution of your data after Alwan transformation')
        plt.show()
        # Save the transformed data as a DataFrame
        df_SCC = pd.DataFrame(df_res['MR'].dropna(), columns=['MRres_BC'])
    return df_SCC
# Apply the function
df_SCC = transform_data(df_res)

# Plot the I-chart
df_SCC = qda.ControlCharts.IMR(df_SCC, 'MRres_BC', plotit=False)
# Plot the I chart
plt.title('I chart')
plt.plot(df_SCC['MRres_BC'], color='b', linestyle='--', marker='o')
plt.plot(df_SCC['MRres_BC'], color='b', linestyle='--', marker='o')
plt.plot(df_SCC['I_UCL'], color='r')
plt.plot(df_SCC['I_CL'], color='g')
plt.plot(df_SCC['I_LCL'], color='r')
plt.ylabel('Individual Value')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_SCC)+.5, df_SCC['I_UCL'].iloc[0], 'UCL = {:.2f}'.format(df_SCC['I_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_SCC)+.5, df_SCC['I_CL'].iloc[0], 'CL = {:.2f}'.format(df_SCC['I_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_SCC)+.5, df_SCC['I_LCL'].iloc[0], 'LCL = {:.2f}'.format(df_SCC['I_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_SCC['I_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 6.5: Normality transformation on residuals' moving range statistic + I-chart on the outcome

6.3. Between group control charts for SPC non iid

Sometimes happens that we have batched processes but we do not know in which order data has been sampled within a batch. Being impossible to reconstruct the sampling series we cannot rely on the usual tools as ACF/PACF, Runs test and so on. Every test for checking assumptions which involves the quality engineer to know the sampling order cannot be accessible.

For this reason, the main randomness checking tool is represented by the scatter plot of the batches over time (Figure 1.16). Once data are stacked it is also possible to check the normality of the data.

Suppose an Xbar-R chart is applied and we get a control chart where the data are:

- **Hugging the center line**
- **Or creating an overdispersion within and also outside the chart**

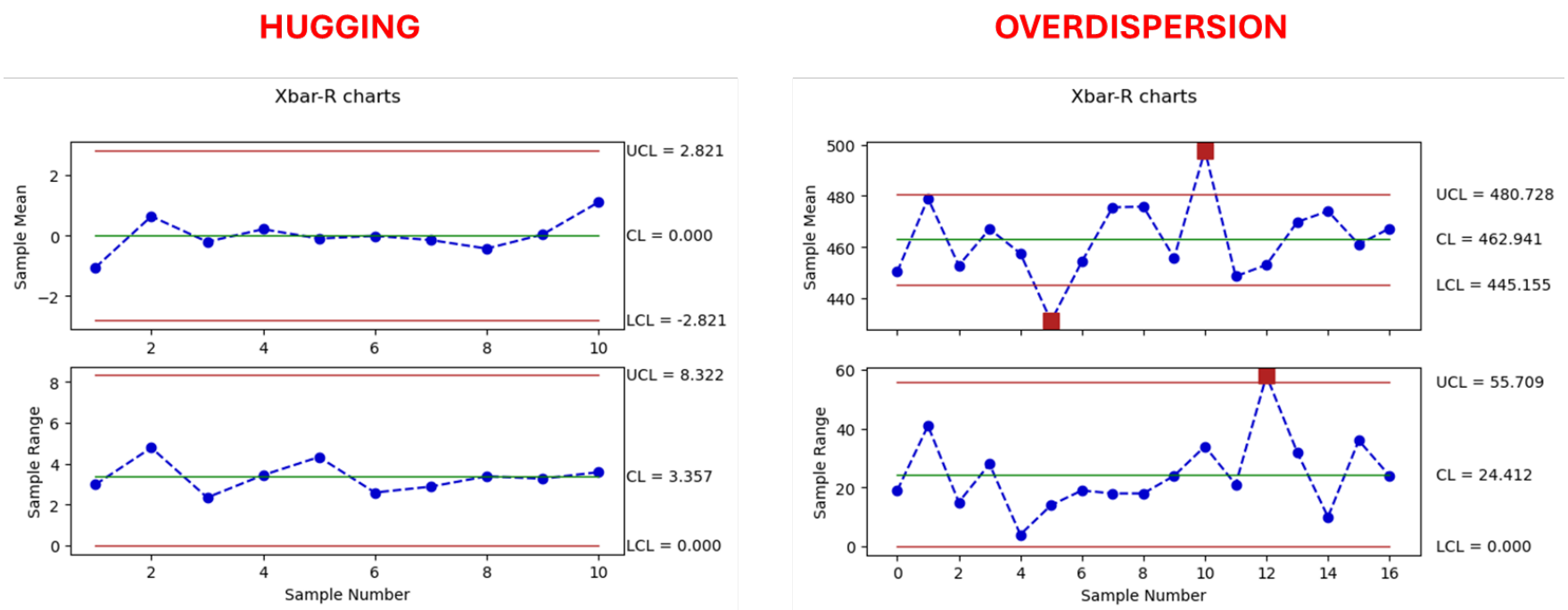


Figure 6.6: Model-based control chart

This happens when the iid assumtpions are not met. The reason is that Xbar-R control chart are able to capture the "within batches variability" but not the "among batch variability". As we know there might be thousand of reason why data are not random and independent from each other.

For instance, in the overdispersion case, data were autocorrelated (YES TRUE, WE CANNOT COMPUTE AUTOCORRELATION IF WE DON'T KNOW THE SAMPLING ORDER WITHIN BATCHES, but in reality it was a collection of data made for practicing and trials. It was initially given as hint that the order within the batches was the true order, then the exercise was repeated without this assumption). However, even if we don't know the sampling order within the batches we might also recognize we can try to make some guess:

Note:

- Two batches looks to be at the extremes
- Points of a same batch drops close each other
- AR(1) is the order of the autoregressive pattern behind the data (barely intuitable from this plot)

Essentially this data do not respect the "iid" assumptions, but if we are not able to apply the usual tools it is likely that we are not able to recognize this phenomena.

By recognizing hugging (or stratification) and overdispersion we can guess those violations.

Figure 6.7: Example of iid violation and overdispersion

Now let's leave aside the example and let's go through the solution.

What is the remedy in this two cases?

The method suggested is to design 3 control charts: "Between groups" control charts, i.e., charts that assume all the samples to be individual measurements

- **I-MR chart on the mean of the batches:**

  With the I chart we can get rid of the violation of the independece assumption within the sample. This chart exploit a batching solution to break the non randomness and the lack of independence within the data.

  The MR chart allows monitoring the between sample variability, providing insights into changes over time.

- **R (or S) chart on the original data: ensures that the variability within individual samples is under control.**

The primary advantage of I-MR charts in such scenarios is their robustness and ability to provide meaningful control even with minor deviations from ideal conditions.

```
# Xbar-R chart on original data that shows hugging or overdispersion
df_XR = qda.ControlCharts.XbarR(df, plotit=False) #if you need to plot this chart change False in True

# Here the code starts
# Create a new dataframe that stores the mean of all the samples
df_Xbar = pd.DataFrame(df_XR['sample_mean'])
# Build the IMR chart using this new dataframe
df_Xbar = qda.ControlCharts.IMR(df_Xbar, 'sample_mean')
# Plot the R chart as well
plt.title('R chart')
plt.plot(df_XR['sample_range'], color='b', linestyle='--', marker='o')
plt.plot(df_XR['R_UCL'], color='r')
plt.plot(df_XR['R_CL'], color='g')
plt.plot(df_XR['R_LCL'], color='r')
plt.ylabel('Sample range')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(df_XR)+.5, df_XR['R_UCL'].iloc[0], 'UCL = {:.3f}'.format(df_XR['R_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['R_CL'].iloc[0], 'CL = {:.3f}'.format(df_XR['R_CL'].iloc[0]), verticalalignment='center')
plt.text(len(df_XR)+.5, df_XR['R_LCL'].iloc[0], 'LCL = {:.3f}'.format(df_XR['R_LCL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_XR['R_TEST1'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 6.6: Normality transformation on residuals' moving range statistic + I-chart on the outcome

In case OOC are detected:

- **In case of assignable causes discard the observation and redesign the chart on the remaining set of data**
- **In case no assignable causes are addressable we need to lable the OOC as a false alarm**

For completeness of the previous example the final charts are shown below.

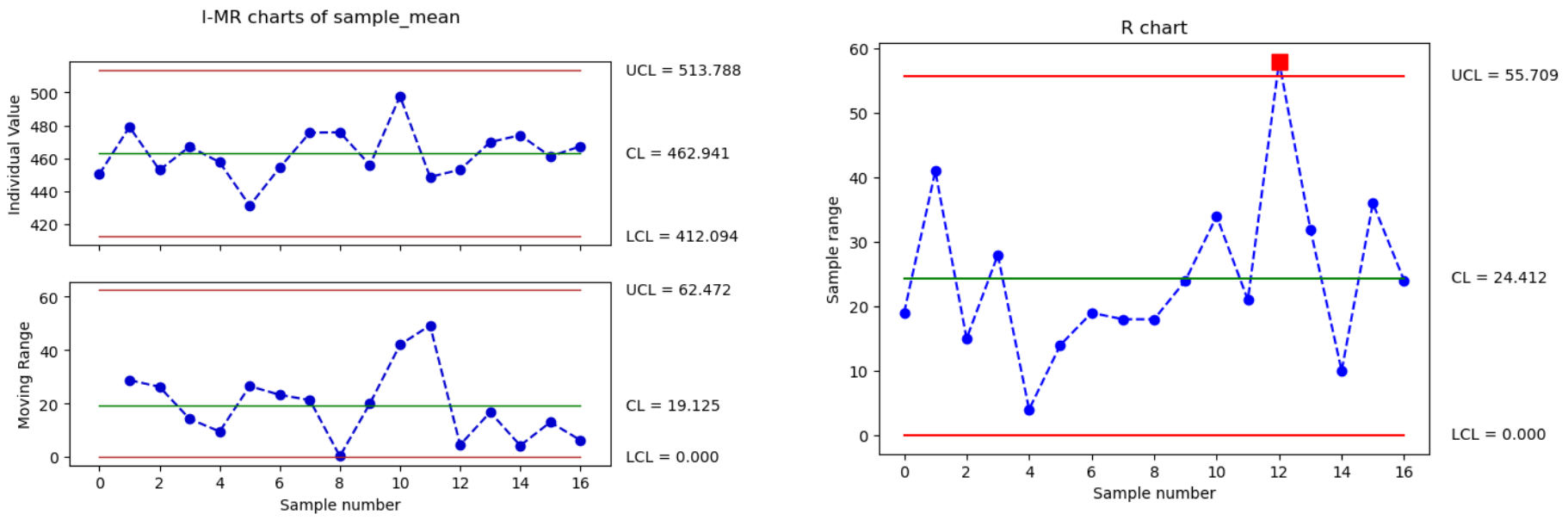


Figure 6.8: I-MR-R control charts for non iid process (n > 1)

Since no assignable causes were found for the OOC in the R-chart we need to consider it as a false alarm

7 | Multivariate control charts

Control Chart for the Mean: Chi2 and Hotelling's T^2 control charts

7.0.1. Multivariate control charts for batched process: n > 1

Hotelling's T^2 control chart is a multivariate control chart that monitors multiple interrelated variables simultaneously. It calculates a single statistic, T^2, which is a function of all the variables and their covariance matrix. The T^2 is distributed as a F distribution, but if the number of samples collected is high it becomes a Chi2 distribution.

$$T_i^2 = n\left(\bar{X}_i - \bar{X}\right)^T S^{-1}\left(\bar{X}_i - \bar{X}\right)$$
$$i = 1, \ldots, m$$

where p is the number of variables.

$$S = \begin{pmatrix} S_{11} & S_{12} & \cdots & S_{1p} \\ S_{21} & S_{22} & \cdots & S_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ S_{p1} & S_{p2} & \cdots & S_{pp} \end{pmatrix}$$

Explanation

- **T_i^2: The Hotelling's T^2 statistic for the i-th observation.**
- **n: The sample size.**
- **$\bar{X}_i$: The mean vector of the i-th subgroup or observation.**
- **$\bar{X}$: The overall mean vector of the process.**
- **S^{-1}: The inverse of the covariance matrix S of the process.**
- **$(\bar{X}_i - \bar{X})$: The difference between the i-th mean vector and the overall mean vector.**
- **$(\bar{X}_i - \bar{X})^T$: The transpose of the difference vector.**

This method takes into account the correlations between variables, making it more sensitive to shifts that affect multiple variables simultaneously. The T^2 statistic is plotted on a single control chart with its own control limits. By considering the covariance structure, Hotelling's T^2 chart can detect joint variations in the variables that single control charts might miss. For example, if there are shifts that only become apparent when considering the combined behavior of $\bar{X}_1$ and $\bar{X}_2$, the Hotelling's T^2 chart will identify these shifts effectively, while single control charts may not.

The T_i^2 statistic measures the multivariate distance of the i-th observation from the overall mean, accounting for the correlation between variables.

Covariance Matrix S

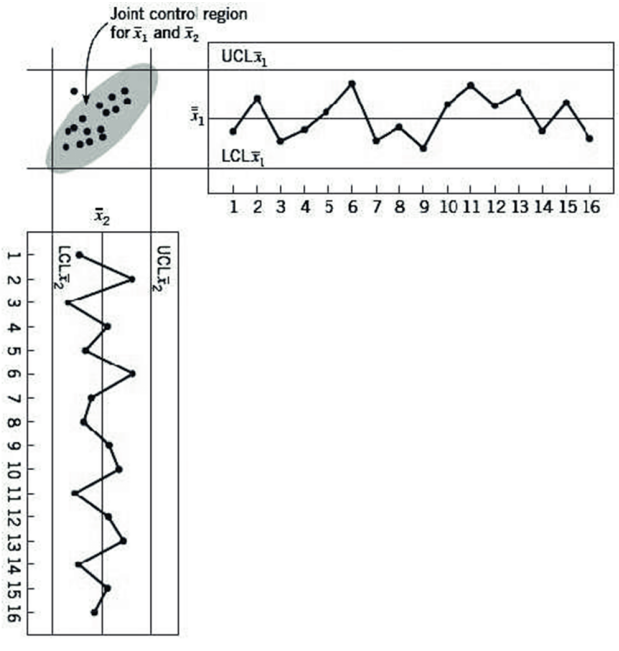
The covariance matrix S captures the variances and covariances of the variables in the process. It is given by:

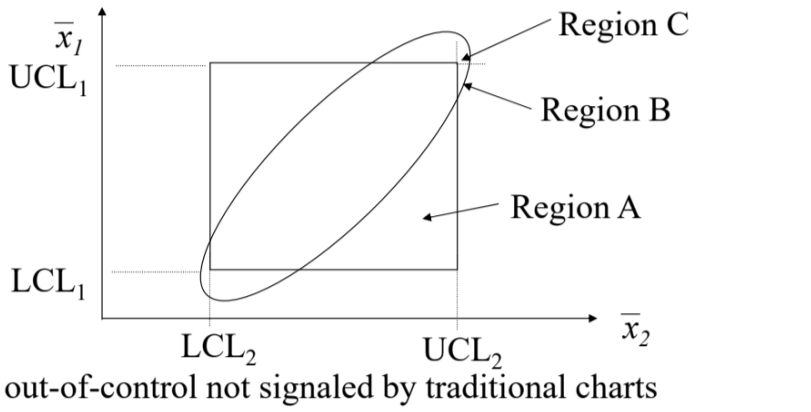
$$S = \begin{pmatrix} S_{11} & S_{12} & \cdots & S_{1p} \\ S_{21} & S_{22} & \cdots & S_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ S_{p1} & S_{p2} & \cdots & S_{pp} \end{pmatrix}$$

- **S_{ij} represents the covariance between the i-th and j-th variables.**
- **The diagonal elements S_{ii} represent the variances of the variables.**
- **The off-diagonal elements S_{ij} (where $i \neq j$) represent the covariances between pairs of variables.**

The covariance matrix S provides information about the variability of each variable and the relationship between pairs of variables in the process. It is used to normalize the differences between the observation vectors and the mean vector when calculating the Hotelling's T^2 statistic.

For sake of simplicity let's analyze the case with two variables:





Region A: out-of-control not signaled by traditional charts
Region B: false out-of-control signaled
Region C: 'twice' false out-of-control (because of $\bar{x}_1$ and $\bar{x}_2$) signaled

Figure 7.1: Multivariate SPC

As it is possible to note, the correlation structure can generate a defined shape in scatter of data.

- **It is a "slanting" ellipse in case the variables are correlated and have different variability**
- **It is an ellipse with axes parallel to Xbar1 and Xbar2 (so the axes of the reference system) if variables are NOT correlated, but they have differnt variance**
- **It is a circle (approximately) if variables are not correlated and have similar variability**

The following pictures introduce us to this multivariate control chart design.

If the parameters are known, the control chart created will be a Chi2 control chart.

If the parameters are unknown, the control chart created will be based on the Hotelling F-statistic, hence it is called Hotelling control chart or T^2-chart

χ^2 control chart

A statistical control scheme can be implemented directly into the plane $\bar{x}_1$ - $\bar{x}_2$ by using the control ellipse:

- But the information about the temporal sequence would be lost;
- It would be difficult to depict the control region for 3 variables, and even impossible for larger numbers of variables

A control chart can be designed to monitor the quantity χ_0^2 by using the control limit $\chi_\alpha^2(p)$. In the most general case:

Known parameters
$\boldsymbol{\mu}' = [\mu_1, \mu_2, \ldots, \mu_p]$
$\boldsymbol{\Sigma}$: true covariance matrix

$$\bar{\mathbf{x}} = \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_p \end{bmatrix}$$

$$\chi_0^2 = n(\bar{\mathbf{x}} - \boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\bar{\mathbf{x}} - \boldsymbol{\mu}) \quad \text{scalare}$$
(1xp, pxp, px1)

$$UCL = \chi_{\alpha,p}^2$$

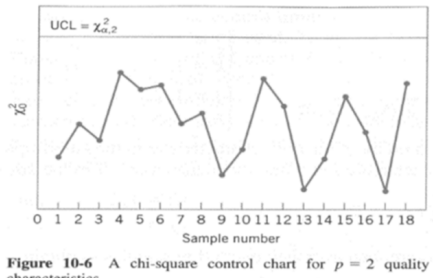


Figure 10-6 A chi-square control chart for $p = 2$ quality characteristics.

Figure 7.2: Chi2 control chart design for multivariate SPC (n > 1)

Assume μ and Σ are unknown.
We have "m" samples of size "n" to be used in design phase

Estimators (considering *m* samples): $\boldsymbol{\mu} \longrightarrow \bar{\bar{\mathbf{x}}}$, $\boldsymbol{\Sigma} \longrightarrow \mathbf{S}$

For each sample k (k=1,…,m):

$$\chi_{0k}^2 = n(\bar{\boldsymbol{x}}_k - \boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\bar{\boldsymbol{x}}_k - \boldsymbol{\mu}) \longrightarrow T_k^2 = n(\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}})' \boldsymbol{S}^{-1} (\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}})$$
(1xp, pxp, px1)

The T^2 statistic is referred to as Hotelling's statistic and it follows the F distribution (not the squared chi distribution), corrected by a constant

- Phase 1 (design phase)

$$T_k^2 = n(\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}})' \boldsymbol{S}^{-1} (\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}}) \sim c_1(m,n,p) F(p, m(n-1) - (p-1))$$

$$c_1(m,n,p) = \frac{p(n-1)(m-1)}{m(n-1) - (p-1)} \Rightarrow \begin{aligned} UCL &= c_1(m,n,p) F_\alpha(p, m(n-1) - (p-1)) \\ LCL &= 0 \end{aligned}$$

- Phase 2 (future observations):

Under the assumption that m* samples during the design phase

$$T_k^2 = n(\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}})' \boldsymbol{S}^{-1} (\bar{\boldsymbol{x}}_k - \bar{\bar{\boldsymbol{x}}}) \sim c_2(m^*,n,p) F(p, m^*(n-1) - (p-1))$$

$$c_2(m^*,n,p) = \frac{p(n-1)(m^*+1)}{m^*(n-1) - (p-1)} \Rightarrow \begin{aligned} UCL &= c_2(m^*,n,p) F_\alpha(p, m^*(n-1) - (p-1)) \\ LCL &= 0 \end{aligned}$$

It is possible to prove that, for large values of m:

$$c_1(m,n,p) F_\alpha(p, m(n-1) - (p-1)) \rightarrow \chi_\alpha^2(p)$$

Remind that:

$$\lim_{\nu_2 \to \infty} F(\nu_1, \nu_2) = \frac{\chi^2(\nu_1)}{\nu_1}$$

Figure 7.3: Hotelling control chart design for multivariate SPC (n > 1)

Let's go through the code steps:

Preparation of the dataframes and checks to decide the right control system

- **Create for each of the "p" variables in the original DataFrame a new dedicated DataFrame**
- **If for each "p", the sample size "n" > 1 and you don't know the sampling order, check the randomnsess through "p" dedicated scatter plots**
- **Check the marginal normality of your data by stacking them. The creation of a stacked version of your original dedicated dataframes is not needed, so just input in all your normality checks "df_variable_name.stack()" and it will work for both Shapiro Wilk test and QQ-plot.**
- **CHECK THE CORRELATION STRUCTURE THROUGH A SCATTER PLOT (if p = 2) OR A SCATTER MATRIX (if p $>$ 2) in order to identify the control chart to be used**

Hotelling's control chart creation

How to create the vector of sample means

- **Create a DataFrame called sample_mean**
- **Define the "p" sample mean columns and respectively populate it with the sample mean of each sample of size "n"**
- **Then compute the sample mean for each of those "p" columns. Note: this time the sample mean command is on columns and not on rows. Write down the code so that it store the resulting "p" Xbarbars in a vector called Xbarbar.**

How to create the sample variance-covariance Matrix:

- **Create a DataFrame of stacked data where the first column represent the sample and each column conrespond to a variable. In order to do that, we use a combination of the factions .transpose() and .melt()**
- **Once the DataFrame is ready, the objective is to create a covariance matrix between the variables for each sample. Combine the function .groupby('sample') and .cov()**
- **Compute the mean-covariance matrix, the sample mean among the covariance matrixes of the samples**

  BE CAREFULL: this operation will change the original order (to alphabetic order) in which the resulting matrix is provided. Unfortunately the values are not following this changes so we need to recover the original order.
- **Create the control chart**

```
# CREATE THE VECTOR OF Xbarbars
sample_mean = pd.DataFrame()
sample_mean['variable 1 name'] = variable_1_df.mean(axis=1)    #fit the code on you real df names
sample_mean['variable 2 name'] = variable_2_df.mean(axis=1)
# --- if other variables add more
# Calculate the grand mean
Xbarbar = sample_mean.mean()
print(Xbarbar)

# COMPUTE THE SAMPLE VARIANCE-COVARIANCE MATRIX
data_stack = pd.DataFrame()
data_stack[['sample', 'variable 1 name']] = variable_1_df.transpose().melt()
data_stack['variable 2 name'] = variable_2_df.transpose().melt()['value']
# --- if other varriable add more (strcture as the right above line of code)
data_stack.head()
# Compute the variance and covariance matrix of each group (sample)
cov_matrix = data_stack.groupby('sample').cov()
cov_matrix.head(8)
# Compute the mean covariance matrix
S = cov_matrix.groupby(level=1).mean()
# Change from alphabetic to original order
cols = S.columns.tolist()
S = S.reindex(columns=cols, index=cols)
print(S)

# Compute the inverse of the variance/covariance matrix
S_inv = np.linalg.inv(S)

# CREATE THE CONTROL CHART
p = insert p                # number of random variables
m = insert m                # number of samples
n = insert n                # number of replicates (sample size)
alpha = insert alpha        # significance level

# Create a new dataframe to store the T2 statistics
data_CC = sample_mean.copy()
data_CC['T2'] = np.nan
for i in range(m):
    data_CC['T2'].iloc[i] = n * (sample_mean.iloc[i] - Xbarbar).transpose().dot(S_inv).dot(sample_mean.iloc[i] - Xbarbar)
# Now we can add the UCL, CL and LCL to the dataframe
data_CC['T2_UCL'] = (p*(m-1)*(n-1))/(m*(n-1)-(p-1))*stats.f.ppf(1-alpha, p, m*n-m+1-p)
data_CC['T2_CL'] = data_CC['T2'].median()
data_CC['T2_LCL'] = 0
# Add one column to test if the sample is out of control
data_CC['T2_TEST'] = np.where((data_CC['T2'] > data_CC['T2_UCL']), data_CC['T2'], np.nan)
# Inspect the dataset
data_CC.head()
```

```
# Plot the T2 control chart
plt.title('T2 control chart')
plt.plot(data_CC['T2'], color='b', linestyle='--', marker='o')
plt.plot(data_CC['T2_UCL'], color='r')
plt.plot(data_CC['T2_CL'], color='g')
plt.plot(data_CC['T2_LCL'], color='r')
plt.ylabel('T2 statistic')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(data_CC)+.5, data_CC['T2_UCL'].iloc[0], 'UCL = {:.3f}'.format(data_CC['T2_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(data_CC)+.5, data_CC['T2_CL'].iloc[0], 'median = {:.3f}'.format(data_CC['T2_CL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(data_CC['T2_TEST'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 7.1: Hotelling's control chart design

```
# Copy and paste all the previous computations until the UCL and then go on with the following code

# Calculate the Hotelling T2 statistic for the i-th sample
index = insert sample index     #remember if sample "i", index(i) = i - 1
S_inv = np.linalg.inv(S)
T2 = n * (sample_mean.iloc[index]-Xbarbar).transpose().dot(S_inv).dot(sample_mean.iloc[index]-Xbarbar)
print('\nThe Hotelling T2 statistic for the sample number %d is: %.3f' % (index + 1, T2))
if T2 >= UCL:
    print('Sample %d is out of control' % (index+1))
else:
    print('Sample %d is in control' % (index+1))
```

Code 7.2: Hotelling's control chart: check if a single sample is in-control

```
p = insert p                # number of random variables
m = insert m                # number of samples
n = insert n                # number of replicates (sample size)
alpha = insert alpha        # significance level

c2 = (p*(n-1)*(m+1))/(m*(n-1)-(p-1))

UCL_new = c2*stats.f.ppf(1-alpha, p, (m*(n-1)-(p-1)))

print("New UCL = %.4f" % UCL_new)
```

Code 7.3: Future observation UCL control limit

7.0.2. Multivariate control charts for individuals: n = 1

Let's see what happens if we need to design multivariate control charts for individuals First, the case with known parameter is introduced.

Conditions

- **Sample Size**: $n = 1$
- **Parameters**: Both the vector of true means (μ) and the true covariance matrix (Σ) are known.

Hotelling's T^2 Statistic Calculation:

$$T^2 = (X - \mu)'\Sigma^{-1}(X - \mu)$$

where X is the observed vector, μ is the vector of true means, and Σ is the true covariance matrix.

Control Limits:

- The upper control limit (UCL) is based on the chi-squared distribution with p degrees of freedom (where p is the number of variables).
- Given a significance level α, the UCL is:

$$UCL = \chi^2_{1-\alpha,p}$$

- The lower control limit (LCL) is zero because T^2 cannot be negative. This is due to the fact that T^2 is a quadratic form, which always results in a non-negative value.

Figure 7.4: Chi2 control chart design for multivariate SPC (n = 1)

The center line in a control chart typically represents the expected value of the monitored statistic under normal conditions. However, using the median is sometimes preferred because it can provide a more robust measure of central tendency, particularly when the distribution is skewed. This perfectly fit what happens for the Chi2 control charts.

THE CENTER LINE OF THE CHI2 CONTROL CHARTS IS THE MEDIAN OF THE UNDERLINED DISTRIBUTION

Be careful: for what has been mentioned above, as the number of sample "m" gets higher the Hotelling's statistic tend to become a Chi2. For this reason we can adopt as a rule of thumb that the Chi2 approximation for the control limit starts to be a good approximation for m $>$ 100, even if parameters are unknown.

The following code is provided as an example of multivariate individual control chart with known parameters.

For sake of simplicity it is written on p = 2 variables.

```
# DEFINE ALL THE KNOWN PARAMETERS
mu_1 = 10
mu_2 = 20
std_1 = #insert the std of the first variable
std_2 = #insert the std of the second variable
m = len(df)
n = 1
p = 2
cov = #insert the covariance

# if corr is given instead of cov, compute the latter
covariance = corr * std_1 * std_2
var_1 = std_1**2
var_2 = std_2**2

# Create the mean vector
mu = pd.Series({'x1': mu_1, 'x2': mu_2})
# Create the Variance-Covariance matrix
S = pd.DataFrame([[var_1, covariance], [covariance, var_2]], columns=['x1', 'x2'], index=['x1', 'x2'])
print('The mean vector is: \n', mu)
print('\nThe sample covariance matrix S is:\n', S)
# Compute the inverse of the variance/covariance matrix
S_inv = np.linalg.inv(S)
```

```
# If present drop all the additional columns of the original dataframe imported from csv that are not variables values
# for instance, drop the sample column
df = df.drop('Sample', axis=1)
df.head()

# Calculate the upper control limit
UCL = stats.chi2.ppf(1 - alpha, df = p)
print("The UCL is: %.3f" % UCL)

# Copy and add an empty column to the dataframe to store the Chi2 statistic
data_CC = df.copy()
data_CC['Chi2'] = np.nan
for i in range(m):
    data_CC['Chi2'].iloc[i] = n * (df.iloc[i] - mu).transpose().dot(S_inv).dot(df.iloc[i] - mu)

# Now we can add the UCL, CL and LCL to the dataframe
data_CC['Chi2_UCL'] = UCL
data_CC['Chi2_CL'] = data_CC['Chi2'].median()
data_CC['Chi2_LCL'] = 0
# Add one column to test if the sample is out of control
data_CC['Chi2_TEST'] = np.where((data_CC['Chi2'] > data_CC['Chi2_UCL']), data_CC['Chi2'], np.nan)
# Inspect the dataset
data_CC.head()

# Plot the Chi2 control chart
plt.title('Chi2 control chart')
plt.plot(data_CC['Chi2'], color='b', linestyle='--', marker='o')
plt.plot(data_CC['Chi2_UCL'], color='r')
plt.plot(data_CC['Chi2_CL'], color='g')
plt.plot(data_CC['Chi2_LCL'], color='r')
plt.ylabel('Chi2 statistic')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(data_CC)+.5, data_CC['Chi2_UCL'].iloc[0], 'UCL = {:.3f}'.format(data_CC['Chi2_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(data_CC)+.5, data_CC['Chi2_CL'].iloc[0], 'median = {:.3f}'.format(data_CC['Chi2_CL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(data_CC['Chi2_TEST'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 7.4: Chi2 control chart design n = 1

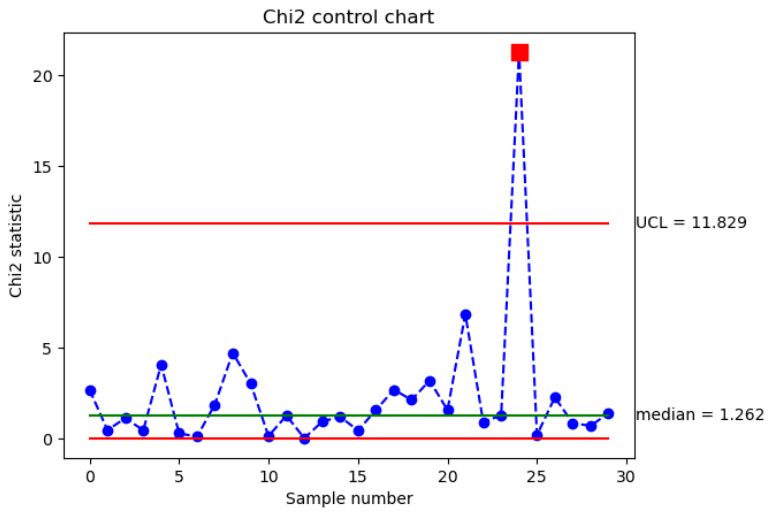


Figure 7.5: Example of Chi2 control chart plot for multivariate SPC (n = 1)

However, when parameters are unkwon things get slightly harder, therefore let's go through them before providing the sum up of the formulas.

Conditions

- **Sample Size:** $n = 1$
- **Parameters: Both the vector of true means (μ) and the true covariance matrix (Σ) are unknown and need to be estimated from the data.**

Statistic

Hotelling's T^2 Statistic Calculation:

$$T^2 = (X - \bar{X})' S^{-1} (X - \bar{X})$$

where X is the observed vector, $\bar{X}$ is the sample mean vector, and S is the sample covariance matrix.

Phase I Control Limits (Parameter Estimation Phase)

Estimate Parameters

- **Sample mean vector $\bar{X}$:**

$$\bar{X} = \frac{1}{m} \sum_{i=1}^{m} X_i$$

- **Sample covariance matrix S (two methods):**
  - **Long-period estimation:**

$$S = \frac{1}{m-1} \sum_{i=1}^{m} (X_i - \bar{X})(X_i - \bar{X})'$$

  - **Short-period estimation:**

$$S = \frac{1}{2(m-1)} \sum_{i=2}^{m} (X_i - X_{i-1})(X_i - X_{i-1})'$$

Control Limits Phase I limits:

$$\begin{cases} UCL = \frac{(m-1)^2}{m} \beta_{\alpha, p/2, (m-p-1)/2} \\ LCL = 0 \end{cases}$$

Phase II Control Limits for Future Observations

Use Phase I Estimates

Use $\bar{X}$ and S estimated from Phase I.

Control Limits

Phase II limits:

$$\begin{cases} UCL = \frac{p(m+1)(m-1)}{m^2 - mp} F_{\alpha, p, m-p} \\ LCL = 0 \end{cases}$$

Summary

For both Phase I and Phase II when parameters are unknown and need to be estimated:

- **Phase I involves estimating μ and Σ from the data and setting the control limits using either the Beta distribution or the F-distribution.**
- **Phase II uses the estimates from Phase I to monitor future observations, with control limits still based on either the F-distribution or the Beta distribution.**

Phase I limits:

$$\begin{cases} \text{UCL} = \dfrac{(m-1)^2}{m} \beta_{\alpha, p/2, (m-p-1)/2} \\ \text{LCL} = 0 \end{cases}$$

Phase II limits:

$$\begin{cases} \text{UCL} = \dfrac{p(m+1)(m-1)}{m^2 - mp} F_{\alpha, p, m-p} \\ \text{LCL} = 0 \end{cases}$$

Long term estimator for S

$$\mathbf{S}_1 = \frac{1}{m-1} \sum_{i=1}^{m} (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})'$$

Short term estimator for S

$$\mathbf{v}_i = \mathbf{x}_{i+1} - \mathbf{x}_i \qquad i = 1, 2, \ldots, m-1 \qquad \mathbf{V} = \begin{bmatrix} \mathbf{v}_1' \\ \mathbf{v}_2' \\ \vdots \\ \mathbf{v}_{m-1}' \end{bmatrix}$$

The resulting (*short range*) estimator is: $\mathbf{S}_2 = \frac{1}{2} \frac{\mathbf{V}'\mathbf{V}}{(m-1)}$

Figure 7.6: Hotelling's multivariate control chart for individuals (n = 1) with unknown parameters

Usually the short term estimator for S is more accurate in detecting shift so it should be preferred.

Let's go through the code for this type of chart

```
m = len(df)
n = 1
p = 2 # Insert a different p in case of more variables

# SHORT TERM ESTIMATOR FOR S
# Create the V matrix
V = df.diff().dropna()
# Calculate the short range estimator S2
S2 = 1/2 * V.transpose().dot(V) / (m-1)
# Display the short range estimator
print("The short range estimator is: \n", S2)

'''# LONG TERM ESTIMATOR FOR S
# Calculate the sample mean vector
mean_vector = df.mean()
# Calculate the deviations from the mean
deviations = df - mean_vector
# Calculate the long-period estimator S (sample covariance matrix)
S_long = (deviations.T @ deviations) / (m - 1)
# Display the long-period estimator
print("The long-period estimator is: \n", S_long)'''

# FROM NOW ON, THE CODE IS CONSIDERING THE SHORT TERM ESTIMATOR FOR S. IN OTHER CASES CHANGE EVERYWHERE S2 AND S2_INV IN S_long and S_long_inv
# Calculate the Xbar from the data
Xbar = df.mean()
S2_inv = np.linalg.inv(S2)

# Calculate the Hotelling T2 statistic
T2_values = []
for i in range(m):
    T2 = n * (df.iloc[i] - Xbar).transpose().dot(S2_inv).dot(df.iloc[i] - Xbar)
    T2_values.append(T2)
data_CC = pd.DataFrame({'T2': T2_values})

# Now we can add the UCL, CL and LCL to the dataframe
data_CC['T2_UCL'] = ((m-1)**2)/m*stats.beta.ppf(1 - alpha, p/2, (m-p-1)/2)
data_CC['T2_CL'] = data_CC['T2'].median()
data_CC['T2_LCL'] = 0
# Add one column to test if the sample is out of control
data_CC['T2_TEST'] = np.where((data_CC['T2'] > data_CC['T2_UCL']), data_CC['T2'], np.nan)
# Inspect the dataset
data_CC.head()

# Plot the T2 control chart
plt.title('T2 control chart')
plt.plot(data_CC['T2'], color='b', linestyle='--', marker='o')
plt.plot(data_CC['T2_UCL'], color='r')
plt.plot(data_CC['T2_CL'], color='g')
plt.plot(data_CC['T2_LCL'], color='r')
plt.ylabel('T2 statistic')
plt.xlabel('Sample number')
# add the values of the control limits on the right side of the plot
plt.text(len(data_CC)+.5, data_CC['T2_UCL'].iloc[0], 'UCL = {:.3f}'.format(data_CC['T2_UCL'].iloc[0]), verticalalignment='center')
plt.text(len(data_CC)+.5, data_CC['T2_CL'].iloc[0], 'median = {:.3f}'.format(data_CC['T2_CL'].iloc[0]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(data_CC['T2_TEST'], linestyle='none', marker='s', color='r', markersize=10)
plt.show()
```

Code 7.5: Hotelling's control chart design n = 1

```
# IN CASE YOU NEED THE CODE FOR FUTURE OBSERVATIONS
# It is likely that we have to recompute the T^2 statistic on the new observations and then...
# Do not only copy and paste, some preparation is needed before
# Calculate Phase II limits
UCL_phase2 = (p * (m + 1) * (m - 1)) / (m**2 - m * p) * stats.f.ppf(1 - alpha, p, m - p)
data_CC['T2_UCL_phase2'] = UCL_phase2
data_CC['T2_LCL_phase2'] = 0

# Plot the Phase II control chart
plt.figure(figsize=(10, 6))
plt.title('Hotelling $T^2$ Control Chart (Phase II)')
plt.plot(data_CC['T2'], color='b', linestyle='--', marker='o', label='$T^2$ Statistic')
plt.axhline(y=UCL_phase2, color='r', linestyle='-', label='UCL (Phase II)')
plt.axhline(y=0, color='r', linestyle='-')
plt.ylabel('$T^2$ Statistic')
plt.xlabel('Sample Number')
plt.text(len(data_CC) + 0.5, UCL_phase2, 'UCL = {:.3f}'.format(UCL_phase2), verticalalignment='center')
plt.legend()
plt.show()
```

Code 7.6: Hotelling's control chart design n = 1 (phase2)

7.1. Univariate control charts for multivariate processes

Designing Control Charts for Low Correlation Variables

When dealing with multivariate data where the variables have low or no correlation among each other, the design method involves creating separate univariate control charts for each variable. This approach simplifies the process and makes it easier to interpret individual variables' behavior. However, it introduces the need for Bonferroni correction to control the overall Type I error rate.

Univariate Control Charts

For each variable, we create a standard univariate control chart. The control limits for these charts are calculated as follows:

$$\begin{cases} UCL_i = \mu_i + Z_{\alpha/2}\frac{\sigma_i}{\sqrt{n}} \\ CL_i = \mu_i \\ LCL_i = \mu_i - Z_{\alpha/2}\frac{\sigma_i}{\sqrt{n}} \end{cases}$$

where:

- μ_i **is the mean of the** i**-th variable.**
- σ_i **is the standard deviation of the** i**-th variable.**
- $Z_{\alpha/2}$ **is the critical value from the standard normal distribution corresponding to the desired significance level** $\alpha/2$**.**
- n **is the sample size (in this case,** $n = 1$**).**

The Bonferroni Effect

When multiple univariate control charts are used simultaneously, the probability of a false alarm (Type I error) increases. This is known as the Bonferroni effect. To address this, we apply the Bonferroni correction, which adjusts the significance level for each individual test to ensure that the overall (family-wise) Type I error rate remains at the desired level.

Bonferroni Correction

The Bonferroni correction is used to control the overall significance level (α_{FAM}**) for a family of tests. It ensures that the probability of at least one false alarm across all tests does not exceed the desired significance level. The correction is applied as follows:**

- **Overall Significance Level (**α_{FAM}**):**

  $$\alpha_{FAM} \leq p \cdot \alpha$$

  where p **is the number of individual tests (control charts).**

- **Significance Level for Individual Tests (**α**): To achieve an overall significance level of** α_{FAM}**, the significance level for each individual test is:**

  $$\alpha = \frac{\alpha_{FAM}}{p}$$

Example

Consider we want an average run length (ARL) of 500 when the process is in control. The overall significance level α_{FAM} **is derived from the ARL as follows:**

$$ARL(0) = \frac{1}{\alpha_{FAM}} \geq 500 \implies \alpha_{FAM} \leq \frac{1}{500}$$

Using Bonferroni's inequality for p **tests:**

$$\alpha_{FAM} \leq p \cdot \alpha \implies \alpha = \frac{1}{p \cdot 500}$$

For $p = 3$ **variables:**

$$\alpha = \frac{1}{3 \cdot 500} = 6.67 \times 10^{-4}$$

Summary

In multivariate process control with low or no correlation among variables:

- **Design separate univariate control charts for each variable.**
- **Apply the Bonferroni correction to adjust the significance level for each chart to maintain an overall desired Type I error rate.**
- **Calculate the control limits for each univariate chart using the adjusted significance level.**

8 | Small shift control charts

8.1. Application field and main differences respect to Shewhart's approach

Small-shift control charts, such as Cumulative Sum (CUSUM) and Exponentially Weighted Moving Average (EWMA) charts, are specialized tools used in quality control to detect minor and persistent shifts in a process mean. These charts are particularly important in industries where precision is critical, and even slight deviations can result in significant defects or wastage. Examples include semiconductor manufacturing, pharmaceuticals, and aerospace engineering, where product specifications are extremely tight, and small shifts can lead to substantial quality issues.

Unlike traditional Shewhart control charts, which are designed to detect larger shifts in the process mean, CUSUM and EWMA charts accumulate information from all past data points, making them more sensitive to small changes. Shewhart charts, such as X-bar, R, and S charts, only utilize the information from the most recent sample to make decisions, which can make them less responsive to small, gradual shifts. This characteristic means that Shewhart charts are more suitable for identifying abrupt and significant deviations from the process mean.

The CUSUM chart works by accumulating the deviations of each sample point from a target value, effectively summing up small shifts over time until they become significant enough to signal a potential issue. This cumulative approach allows it to detect small shifts more quickly than Shewhart charts. The EWMA chart, on the other hand, assigns exponentially decreasing weights to past data points, giving more importance to recent observations while still considering the historical data. This weighting scheme helps EWMA charts to detect shifts more promptly than Shewhart charts, especially when the shifts are small and persistent.

Small-shift control charts are required when the detection of minute changes in the process is crucial for maintaining high-quality standards. They are particularly effective in environments where process stability and consistency are paramount, and the cost of defects due to minor shifts is high. Additionally, these charts are robust against non-normality and can be used effectively even when the data distribution deviates from normality, which enhances their applicability in various industrial scenarios.

EWMA Control Chart

Assumptions in Terms of Normality and Independence

For both EWMA charts ($n > 1$ and $n = 1$):

- **Normality: The EWMA chart does not strictly require the data to be normally distributed due to the Central Limit Theorem, especially as the sample size increases. However, normality is often assumed for the initial design of the control limits.**
- **Independence: The observations should be independent of each other. Autocorrelated data can affect the performance of the EWMA chart, leading to increased false alarms or missed shifts.**

8.1.1. EWMA Control Chart for $n > 1$

Design of EWMA Control Chart for $n > 1$

Step 1: Initialization

- **Starting Point: The EWMA chart starts with an initial estimate of the process mean, typically the target mean (μ_0) or the grand average of past data.**

$$z_0 = \mu_0$$

Step 2: EWMA Statistic Calculation

- **EWMA Statistic (z_t): For each subgroup mean $\bar{x}_t$, the EWMA statistic is calculated using:**

$$z_t = \lambda \bar{x}_t + (1 - \lambda) z_{t-1}$$

Here, $\bar{x}_t$ is the mean of the current subgroup, λ is the weighting factor ($0 < \lambda$ 1), and z_{t-1} is the EWMA statistic from the previous time period.

- **Weighting Factor (λ): The choice of λ determines how quickly past data points are discounted. Smaller λ values give more weight to past observations, while larger λ values make the chart more responsive to recent changes. Typical values of λ range from 0.05 to 0.25, with 0.2 being common.**

Step 3: Control Limits Calculation

- **Control Limits: The control limits for the EWMA chart are computed using:**

$$UCL_t = \mu_0 + L \cdot \sigma_z$$

$$CL = \mu_0$$

$$LCL_t = \mu_0 - L \cdot \sigma_z$$

where L is the control limit multiplier (usually 3), and σ_z is the standard deviation of the EWMA statistic, calculated as:

$$\sigma_z = \sigma \sqrt{\frac{\lambda}{2-\lambda}(1-(1-\lambda)^{2t})}$$

σ is the standard deviation of the process, and t is the number of subgroups.

- Steady State: In steady state (as t approaches infinity), the standard deviation simplifies to:

$$\sigma_z = \sigma \sqrt{\frac{\lambda}{2-\lambda}}$$

Step 4: Interpretation

- Monitoring: The process is monitored by plotting the z_t values and comparing them to the control limits. If z_t falls outside the control limits, it signals a potential shift in the process mean, indicating that the process may be out of control.

Example Calculation for $n > 1$

Suppose we have a process with the following parameters:

- Target mean (μ_0) = 10
- Process standard deviation (σ) = 1
- Weighting factor (λ) = 0.2
- Control limit multiplier (L) = 3
- Calculate Initial EWMA Statistic: $z_0 = \mu_0 = 10$
- Calculate Subsequent EWMA Statistic: For each subgroup mean $\bar{x}_t$:

$$z_t = 0.2\bar{x}_t + 0.8z_{t-1}$$

- Calculate Control Limits:

$$\sigma_z = 1 \cdot \sqrt{\frac{0.2}{2-0.2}} = 1 \cdot \sqrt{0.111} = 0.333$$

$$UCL_t = 10 + 3 \cdot 0.333 = 10.999$$

$$LCL_t = 10 - 3 \cdot 0.333 = 9.001$$

These limits and the z_t statistics are then plotted on the EWMA control chart to monitor the process.

```
# DA RIVEDERE PERCHE' QUA DF ERA DIRRETTAMENTE UN DATAFRAME CON LE MEDIE DEI SAMPLES.
n = 5
sigma = 1
lambda_ = 0.2
sigma_xbar = sigma/np.sqrt(n)
xbarbar = df['column name'].mean()
df_EWMA = df.copy()
df_EWMA['a_t'] = lambda_/(2-lambda_) * (1 - (1-lambda_)**(2*np.arange(1, len(df_EWMA)+1)))
col_name = 'column name'        #insert the column name of the original data within "df"
for i in range(len(df_EWMA)):
    if i == 0:
        df_EWMA.loc[i, 'z'] = lambda_*df_EWMA.loc[i, col_name] + (1-lambda_)*xbarbar
    else:
        df_EWMA.loc[i, 'z'] = lambda_*df_EWMA.loc[i, col_name] + (1-lambda_)*df_EWMA.loc[i-1, 'z']
df_EWMA['UCL'] = xbarbar + 3*sigma_xbar*np.sqrt(df_EWMA['a_t'])
df_EWMA['CL'] = xbarbar
df_EWMA['LCL'] = xbarbar - 3*sigma_xbar*np.sqrt(df_EWMA['a_t'])

df_EWMA['z_TEST1'] = np.where((df_EWMA['z'] > df_EWMA['UCL']) | (df_EWMA['z'] < df_EWMA['LCL']), df_EWMA['z'], np.nan)

# Plot the control limits
plt.plot(df_EWMA['UCL'], color='firebrick', linewidth=1)
plt.plot(df_EWMA['CL'], color='g', linewidth=1)
plt.plot(df_EWMA['LCL'], color='firebrick', linewidth=1)
# Plot the chart
plt.title('EWMA chart of %s (lambda=%.2f)' % (col_name, lambda_))
plt.plot(df_EWMA['z'], color='b', linestyle='-', marker='o')
# add the values of the control limits on the right side of the plot
plt.text(len(df_EWMA)+.5, df_EWMA['UCL'].iloc[-1], 'UCL = {:.3f}'.format(df_EWMA['UCL'].iloc[-1]), verticalalignment='center')
plt.text(len(df_EWMA)+.5, df_EWMA['CL'].iloc[-1], 'CL = {:.3f}'.format(df_EWMA['CL'].iloc[-1]), verticalalignment='center')
plt.text(len(df_EWMA)+.5, df_EWMA['LCL'].iloc[-1], 'LCL = {:.3f}'.format(df_EWMA['LCL'].iloc[-1]), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_EWMA['z_TEST1'], linestyle='none', marker='s', color='firebrick', markersize=10)
plt.xlim(-1, len(df_EWMA))
plt.show()
```

Code 8.1: EWMA control chart (n > 1)

```
n = 5
sigma = 1
mu = 10.75
lambda_ = 0.2
sigma_xbar = sigma/np.sqrt(n)
```

```
# df is the dataframe containing the sample mean of each sample
df_EWMA = qda.ControlCharts.EWMA(df, 'EXE4', lambda_, mu, sigma_xbar)
```

Code 8.2: Automatic EWMA control chart (n > 1)

8.1.2. EWMA Control Chart for Individual Observations ($n = 1$)

Design of EWMA Control Chart for $n = 1$

Step 1: Initialization

- **Starting Point: The EWMA chart starts with an initial estimate of the process mean, typically the target mean (μ_0) or the grand average of past data.**

$$z_0 = \mu_0$$

Step 2: EWMA Statistic Calculation

- **EWMA Statistic (z_t): For each individual observation x_t, the EWMA statistic is calculated using:**

$$z_t = \lambda x_t + (1 - \lambda) z_{t-1}$$

Here, x_t is the current observation, λ is the weighting factor ($0 < \lambda$ 1), and z_{t-1} is the EWMA statistic from the previous time period.

- **Weighting Factor (λ): The choice of λ determines how quickly past data points are discounted. Smaller λ values give more weight to past observations, while larger λ values make the chart more responsive to recent changes. Typical values of λ range from 0.05 to 0.25, with 0.2 being common.**

Step 3: Control Limits Calculation

- **Control Limits: The control limits for the EWMA chart are computed using:**

$$UCL_t = \mu_0 + L \cdot \sigma_z$$

$$CL = \mu_0$$

$$LCL_t = \mu_0 - L \cdot \sigma_z$$

where L is the control limit multiplier (usually 3), and σ_z is the standard deviation of the EWMA statistic, calculated as:

$$\sigma_z = \sigma \sqrt{\frac{\lambda}{2 - \lambda} (1 - (1 - \lambda)^{2t})}$$

σ is the standard deviation of the process, and t is the number of individual observations.

- **Steady State: In steady state (as t approaches infinity), the standard deviation simplifies to:**

$$\sigma_z = \sigma \sqrt{\frac{\lambda}{2 - \lambda}}$$

Step 4: Interpretation

- **Monitoring: The process is monitored by plotting the z_t values and comparing them to the control limits. If z_t falls outside the control limits, it signals a potential shift in the process mean, indicating that the process may be out of control.**

Example Calculation for $n = 1$

Suppose we have a process with the following parameters:

- **Target mean (μ_0) = 10**
- **Process standard deviation (σ) = 1**
- **Weighting factor (λ) = 0.2**
- **Control limit multiplier (L) = 3**
- **Calculate Initial EWMA Statistic:** $z_0 = \mu_0 = 10$
- **Calculate Subsequent EWMA Statistic: For each individual observation x_t:**

$$z_t = 0.2 x_t + 0.8 z_{t-1}$$

- **Calculate Control Limits:**

$$\sigma_z = 1 \cdot \sqrt{\frac{0.2}{2 - 0.2}} = 1 \cdot \sqrt{0.111} = 0.333$$

$$UCL_t = 10 + 3 \cdot 0.333 = 10.999$$

$$LCL_t = 10 - 3 \cdot 0.333 = 9.001$$

These limits and the z_t statistics are then plotted on the EWMA control chart to monitor the process.

CUSUM Control Chart

Assumptions in Terms of Normality and Independence

For both CUSUM charts ($n > 1$ and $n = 1$):

- **Normality: The CUSUM chart also benefits from the Central Limit Theorem, and while normality is often assumed, it is not strictly necessary.**
- **Independence: Observations should be independent of each other. Autocorrelated data can affect the CUSUM chart's performance.**

CUSUM Control Chart for $n > 1$

Design of CUSUM Control Chart for $n > 1$

Step 1: Initialization

- **Starting Point: The CUSUM chart starts with initial cumulative sums set to zero.**

$$C_0^+ = 0, \quad C_0^- = 0$$

Step 2: CUSUM Statistic Calculation

- **CUSUM Statistic (C_t^+ and C_t^-): For each subgroup mean $\bar{x}_t$, the CUSUM statistics are calculated using:**

$$C_t^+ = \max\left(0, (\bar{x}_t - (\mu_0 + K)) + C_{t-1}^+\right)$$

$$C_t^- = \max\left(0, ((\mu_0 - K) - \bar{x}_t) + C_{t-1}^-\right)$$

Here, K is the reference value, typically chosen as half the shift to be detected.

Step 3: Control Limits Calculation

- **Control Limits: The control limit H is chosen based on the desired average run length (ARL) and is typically set to 5 times the standard deviation of the process mean. We use to the parameter which multiplies the standard deviation as "h"**

$$H = 5\sigma_{\bar{x}}$$

where $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$ and n is the subgroup size.

Step 4: Interpretation

- **Monitoring: The process is monitored by plotting the C_t^+ and C_t^- values. If either C_t^+ or C_t^- exceeds the control limit H, it signals a potential shift in the process mean, indicating that the process may be out of control.**

Example Calculation for $n > 1$

Suppose we have a process with the following parameters:

- **Target mean (μ_0) = 10**
- **Process standard deviation (σ) = 1**
- **Subgroup size (n) = 5**
- **Reference value (K) = 0.5**
- **Control limit multiplier (H) = 2.236**
- **Calculate Initial CUSUM Statistics: $C_0^+ = 0$, $C_0^- = 0$**
- **Calculate Subsequent CUSUM Statistics: For each subgroup mean $\bar{x}_t$:**

$$C_t^+ = \max\left(0, (\bar{x}_t - 10.5) + C_{t-1}^+\right)$$

$$C_t^- = \max\left(0, (9.5 - \bar{x}_t) + C_{t-1}^-\right)$$

- **Calculate Control Limits:**

$$H = 5 \cdot \frac{1}{\sqrt{5}} = 2.236$$

These limits and the C_t^+ and C_t^- statistics are then plotted on the CUSUM control chart to monitor the process.

```
# DA RIVEDERE PERCHE' QUA DF ERA DIRRETTAMENTE UN DATAFRAME CON LE MEDIE DEI SAMPLES.
n = 5
sigma = 1
h = 4
k = 0.5
sigma_xbar = sigma/np.sqrt(n)
xbarbar = df['EXE4'].mean()
df_CUSUM = df.copy()
H = h*sigma_xbar
K = k*sigma_xbar
df_CUSUM['Ci+'] = 0.0
df_CUSUM['Ci-'] = 0.0
col_name = 'EXE4'
for i in range(len(df_CUSUM)):
    if i == 0:
        df_CUSUM.loc[i, 'Ci+'] = max(0, df_CUSUM.loc[i, col_name] - (xbarbar + K))
        df_CUSUM.loc[i, 'Ci-'] = max(0, (xbarbar - K) - df_CUSUM.loc[i, col_name])
    else:
        df_CUSUM.loc[i, 'Ci+'] = max(0, df_CUSUM.loc[i, col_name] - (xbarbar + K) + df_CUSUM.loc[i-1, 'Ci+'])
        df_CUSUM.loc[i, 'Ci-'] = max(0, (xbarbar - K) - df_CUSUM.loc[i, col_name] + df_CUSUM.loc[i-1, 'Ci-'])

df_CUSUM['Ci+_TEST1'] = np.where((df_CUSUM['Ci+'] > H) | (df_CUSUM['Ci+'] < -H), df_CUSUM['Ci+'], np.nan)
df_CUSUM['Ci-_TEST1'] = np.where((df_CUSUM['Ci-'] > H) | (df_CUSUM['Ci-'] < -H), df_CUSUM['Ci-'], np.nan)

# Plot the control limits
```

```
plt.hlines(H, 0, len(df_CUSUM), color='firebrick', linewidth=1)
plt.hlines(0, 0, len(df_CUSUM), color='g', linewidth=1)
plt.hlines(-H, 0, len(df_CUSUM), color='firebrick', linewidth=1)
# Plot the chart
plt.title('CUSUM chart of %s (h=%.2f, k=%.2f)' % (col_name, h, k))
plt.plot(df_CUSUM['Ci+'], color='b', linestyle='-', marker='o')
plt.plot(-df_CUSUM['Ci-'], color='b', linestyle='-', marker='D')
# add the values of the control limits on the right side of the plot
plt.text(len(df_CUSUM)+.5, H, 'UCL = {:.3f}'.format(H), verticalalignment='center')
plt.text(len(df_CUSUM)+.5, 0, 'CL = {:.3f}'.format(0), verticalalignment='center')
plt.text(len(df_CUSUM)+.5, -H, 'LCL = {:.3f}'.format(-H), verticalalignment='center')
# highlight the points that violate the alarm rules
plt.plot(df_CUSUM['Ci+_TEST1'], linestyle='none', marker='s', color='firebrick', markersize=10)
plt.plot(-df_CUSUM['Ci-_TEST1'], linestyle='none', marker='s', color='firebrick', markersize=10)
plt.xlim(-1, len(df_CUSUM))
plt.show()
```

Code 8.3: CUSUM control chart (n > 1)

```
n = 5
sigma = 1
mu = 10.75
sigma_xbar = sigma/np.sqrt(n)
h = 4
k = 0.5
# df is the dataframe containing the sample mean of each sample
df_CUSUM = qda.ControlCharts.CUSUM(df, 'EXE4', params=(h,k), mean = mu, sigma_xbar = sigma_xbar)
```

Code 8.4: Automatic CUSUM control chart (n > 1)

CUSUM Control Chart for Individual Observations ($n = 1$)

Design of CUSUM Control Chart for $n = 1$

Step 1: Initialization

- **Starting Point: The CUSUM chart starts with initial cumulative sums set to zero.**

$$C_0^+ = 0, \quad C_0^- = 0$$

Step 2: CUSUM Statistic Calculation

- **CUSUM Statistic (C_t^+ and C_t^-): For each individual observation x_t, the CUSUM statistics are calculated using:**

$$C_t^+ = \max\left(0, (x_t - (\mu_0 + K)) + C_{t-1}^+\right)$$

$$C_t^- = \max\left(0, ((\mu_0 - K) - x_t) + C_{t-1}^-\right)$$

Here, K is the reference value, typically chosen as half the shift to be detected.

Step 3: Control Limits Calculation

- **Control Limits: The control limit H is chosen based on the desired average run length (ARL) and is typically set to 5 times the standard deviation of the process mean. We use to the parameter which multiplies the standard deviation as "h"**

$$H = 5\sigma$$

where σ is the standard deviation of the process.

Step 4: Interpretation

- **Monitoring: The process is monitored by plotting the C_t^+ and C_t^- values. If either C_t^+ or C_t^- exceeds the control limit H, it signals a potential shift in the process mean, indicating that the process may be out of control.**

Example Calculation for $n = 1$

Suppose we have a process with the following parameters:

- **Target mean (μ_0) = 10**
- **Process standard deviation (σ) = 1**
- **Reference value (K) = 0.5**
- **Control limit multiplier (H) = 5**
- **Calculate Initial CUSUM Statistics: $C_0^+ = 0$, $C_0^- = 0$**
- **Calculate Subsequent CUSUM Statistics: For each individual observation x_t:**

$$C_t^+ = \max\left(0, (x_t - 10.5) + C_{t-1}^+\right)$$

$$C_t^- = \max\left(0, (9.5 - x_t) + C_{t-1}^-\right)$$

- **Calculate Control Limits:**

$$H = 5 \cdot 1 = 5$$

These limits and the C_t^+ and C_t^- statistics are then plotted on the CUSUM control chart to monitor the process.

9 | Predefined algorithm to handle data

9.1. Outliers detection function

The "df" you see in the function definition of "def detect_outliers(df, k=1.5):" is a parameter, which acts like a placeholder or a local variable within that function.

This means you don't need to rename your DataFrame to df when you use this function; instead, you pass your DataFrame to the function as an argument, and it will be referred to as df inside the function.

```
def PrintGraphs(data):
    # Defining number of bins for histograms
    dim = int(np.round(np.sqrt(len(data))))
    region = data.iloc[0, 2]

    # Creating all the graphs
    for i in range(3, 12):
        variable_name = data.columns[i]
        fig, ax = plt.subplots(2, 2, figsize=(10, 10))

        #normality Shapiro Wilk test
        _ , pvalue=stats.shapiro(data[variable_name])
        print('Shapiro Wilk test pvalue for normality is:', pvalue)

        # Scatter plot
        ax[0, 0].plot(data.index, data.iloc[:, i], 'o')
        ax[0, 0].set_title(f'Scatter Plot of {variable_name} from {region}')

        # Histogram
        ax[0, 1].hist(data.iloc[:, i], bins=dim)
        ax[0, 1].set_title(f'Histogram of {variable_name} from {region}')

        # Probability Plot
        stats.probplot(data.iloc[:, i], dist="norm", plot=ax[1, 0])
        ax[1, 0].set_title(f'Probability Plot of {variable_name} from {region}')

        # Boxplot
        ax[1, 1].boxplot(data.iloc[:, i])
        ax[1, 1].set_title(f'Box Plot of {variable_name} from {region}')

        # Printing the graphs
        plt.tight_layout()
        plt.show()
```

Code 9.1: Shapiro Wilk test + Scatter plot + histogram + QQ plot + boxplot of each column of the dataframe (TO BE REVISED

```
def detect_outliers(df, k=1.5):
    # Calculate Q1, Q3 and IQR for each column
    Q1 = df.quantile(0.25)
    Q3 = df.quantile(0.75)
    IQR = Q3 - Q1
    # Define lower and upper bounds for outliers
    lower_bound = Q1 - k * IQR
    upper_bound = Q3 + k * IQR
    # Replace outliers with NaN
    df_out = df.where((df >= lower_bound) & (df <= upper_bound), np.nan)
return df_out

dataclean = detect_outliers(insert dataframe name)
print(dataclean)
```

Code 9.2: Outliers detection (NEED TO BE REVISED)

```
def ApplyNormality(data):
    for column in data.columns[:-2]: #Ask why -2
        values = data[column]

        # Shapiro-Wilk Test
        _, pvalueSW = stats.shapiro(values)

        # Checking if pvalue < 0.05
        if pvalueSW < p_value_threshold:
```

```
            # Box-Cox transformation requires positive values
            if np.any(values <= 0):
                values = values + abs(values.min()) + 1
            try:
                # Box-Cox Transformation
                transformed_values, _ = stats.boxcox(values)

                # Applying the transformation to the dataset
                _, p2 = stats.shapiro(transformed_values)

                if p2 > p_value_threshold:
                    data.loc[:, column] = transformed_values
                else:
                    print('Not normal')

            except ValueError:
                pass
    return data
```

Code 9.3: SHAPIRO WILK TEST => BOX-COX => SHAPIRO WILK TEST => RETURN ERROR (NEED TO BE REVISED)

```
    def ApplyPCA(data, region):
# Creating the PCA object and applying the PCA to the dataset
pca = PCA()
pca.fit(data.iloc[:, 3:])
print("\nEigenvalues\n", pca.explained_variance_)
print("\nEigenvectors\n", pca.components_)
print("\nExplained variance ratio\n", pca.explained_variance_ratio_)
print("\n", region, "\nCumulative explained variance ratio\n", np.cumsum(pca.explained_variance_ratio_))

# Plotting the eigenvalues
plt.plot(pca.explained_variance_, 'o-')
plt.xlabel('Component Number')
plt.ylabel('Eigenvalue')
plt.title('Eigenvalue Scree Plot of ' + region)
plt.show()

# Plotting the variance explained
plt.plot(np.cumsum(pca.explained_variance_ratio_), 'o-')
plt.bar(range(0, len(pca.explained_variance_ratio_)), pca.explained_variance_ratio_, width=0.5, alpha=0.5,
align='center')
plt.title('Explained Variance Plot of ' + region)
plt.show()

# Saving the scores in a dataset
scores = pca.transform(data.iloc[:, 3:])
scores_df = pd.DataFrame(scores, columns=["z1", "z2", "z3", "z4", "z5", "z6", "z7", "z8", "z9"])

# Finding the minimum number of components that are able to explain at least 85% of variability
loop = True
number_of_components = 1
i = 0
while loop:
    if np.cumsum(pca.explained_variance_ratio_)[i] < min_var_pca:
        i = i + 1
    else:
        number_of_components = i + 1
        loop = False

    # Saving the first n components that are able to explain at least 85% of variability
    relevant_df = scores_df.iloc[:, :number_of_components]

    # Plots of the loadings
    fig, ax = plt.subplots(nrows=len(relevant_df.columns), ncols=1, figsize=(10, 10))
    for i in range(len(relevant_df.columns)):
        ax[i].plot(pca.components_[i], "o-")
        ax[i].set_title('Loading ' + str(i + 1) + ' of ' + region)
    plt.tight_layout()
    plt.show()

    # Plots of the scores
    pd.plotting.scatter_matrix(relevant_df, alpha=1)
    plt.show()

    # Creating the final dataframe
    data.reset_index(drop=True, inplace=True)
    final_df = relevant_df.copy()
    final_df["Photo #"] = data.iloc[:, 0]
    final_df["Position"] = data["Position"]

    return final_df
```

Code 9.4: Apply PCA (NEED TO BE REVISED)

```
        # 3.3: Standardizing all variables.
for i in range(insert firts column index, insert last column index + 1):
```

```
    df.iloc[:, i] = ((df.iloc[:, i] - df.iloc[:, i].mean()) /
                                df.iloc[:, i].std())
```

Code 9.5: Dataframe standardization