Statistics and Computing

Thomas Haslwanter

An Introduction to Statistics with Python

With Applications in the Life Sciences

Springer

Statistics and Computing

Series editor
W.K. Härdle

More information about this series at http://www.springer.com/series/3022

Thomas Haslwanter

An Introduction to Statistics with Python

With Applications in the Life Sciences

Springer

Thomas Haslwanter
School of Applied Health and Social Sciences
University of Applied Sciences Upper Austria
Linz, Austria

Series Editor:
W.K. Härdle
C.A.S.E. Centre for Applied
Statistics and Economics
School of Business and Economics
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin
Germany

The Python code samples accompanying the book are available at www.quantlet.de. All Python programs and data sets can be found on GitHub: https://github.com/thomas-haslwanter/statsintro_python.git. Links to all material are available at http://www.springer.com/de/book/9783319283159.

ISSN 1431-8784 ISSN 2197-1706 (electronic)
Statistics and Computing
ISBN 978-3-319-28315-9 ISBN 978-3-319-28316-6 (eBook)
DOI 10.1007/978-3-319-28316-6

Library of Congress Control Number: 2016939946

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

To my two, three, and four-legged household companions: my wife Jean, Felix, and his sister Jessica.

Preface

In the data analysis for my own research work, I was often slowed down by two things: (1) I did not know enough statistics, and (2) the books available would provide a theoretical background, but no real practical help. The book you are holding in your hands (or on your tablet or laptop) is intended to be the book that will solve this very problem. It is designed to provide enough basic understanding so that you know what you are doing, and it should equip you with the tools you need. I believe that the *Python* solutions provided in this book for the most basic statistical problems address at least 90 % of the problems that most physicists, biologists, and medical doctors encounter in their work. So if you are the typical graduate student working on a degree, or a medical researcher analyzing the latest experiments, chances are that you will find the tools you require here—explanation and source-code included.

This is the reason I have focused on statistical basics and hypothesis tests in this book and refer only briefly to other statistical approaches. I am well aware that most of the tests presented in this book can also be carried out using statistical modeling. But in many cases, this is not the methodology used in many life science journals. Advanced statistical analysis goes beyond the scope of this book and—to be frank—exceeds my own knowledge of statistics.

My motivation for providing the solutions in *Python* is based on two considerations. One is that I would like them to be available to everyone. While commercial solutions like *Matlab*, *SPSS*, *Minitab*, etc., offer powerful tools, most can only use them legally in an academic setting. In contrast, *Python* is completely free ("as in free beer" is often heard in the *Python* community). The second reason is that *Python* is the most beautiful coding language that I have yet encountered; and around 2010 *Python* and its documentation matured to the point where one can use it without being a serious coder. Together, this book, *Python*, and the tools that the *Python* ecosystem offers today provide a beautiful, free package that covers all the statistics that most researchers will need in their lifetime.

For Whom This Book Is

This book assumes that:

- You have some basic programming experience: If you have done no programming previously, you may want to start out with *Python*, using some of the great links provided in the text. Starting programming *and* starting statistics may be a bit much all at once.
- You are not a statistics expert: If you have advanced statistics experience, the online help in *Python* and the *Python* packages may be sufficient to allow you to do most of your data analysis right away. This book may still help you to get started with Python. However, the book concentrates on the basic ideas of statistics and on hypothesis tests, and only the last part introduces linear regression modeling and Bayesian statistics.

This book is designed to give you all (or at least most of) the tools that you will need for statistical data analysis. I attempt to provide the background you need to understand what you are doing. I do not prove any theorems and do not apply mathematics unless necessary. For all tests, a working *Python* program is provided. In principle, you just have to define your problem, select the corresponding program, and adapt it to your needs. This should allow you to get going quickly, even if you have little *Python* experience. This is also the reason why I have not provided the software as one single *Python* package. I expect that you will have to tailor each program to your specific setup (data format, plot labels, return values, etc.).

This book is organized into three parts:

Part I gives an introduction to *Python*: how to set it up, simple programs to get started, and tips how to avoid some common mistakes. It also shows how to read data from different sources into *Python* and how to visualize statistical data.

Part II provides an introduction to statistical analysis. How to design a study, and how best to analyze data, probability distributions, and an overview of the most important hypothesis tests. Even though modern statistics is firmly based in statistical modeling, hypothesis tests still seem to dominate the life sciences. For each test a *Python* program is provided that shows how the test can be implemented.

Part III provides an introduction to statistical modeling and a look at advanced statistical analysis procedures. I have also included tests on discrete data in this section, such as logistic regression, as they utilize "generalized linear models" which I regard as advanced. The book ends with a presentation of the basic ideas of Bayesian statistics.

Additional Material

This book comes with many additional *Python* programs and sample data, which are available online. These programs include listings of the programs printed in the book, solutions to the examples given at the end of most chapters, and code samples

with a working example for each test presented in this book. They also include the code used to generate the pictures in this book, as well as the data used to run the programs.

The Python code samples accompanying the book are available at http://www.quantlet.de. All Python programs and data sets can be found on GitHub: https://github.com/thomas-haslwanter/statsintro_python.git. Links to all material are available at http://www.springer.com/de/book/9783319283159.

Acknowledgments

Python is built on the contributions from the user community, and some of the sections in this book are based on some of the excellent information available on the web. (Permission has been granted by the authors to reprint their contributions here.)

I especially want to thank the following people:

- Paul E. Johnson read the whole manuscript and provided invaluable feedback on the general structure of the book, as well as on statistical details.
- Connor Johnson wrote a very nice blog explaining the results of the statsmodels OLS command, which provided the basis for the section on *Statistical Models*.
- Cam Davidson Pilon wrote the excellent open source e-book *Probabilistic-Programming-and-Bayesian-Methods-for-Hackers*. From there I took the example of the Challenger disaster to demonstrate Bayesian statistics.
- Fabian Pedregosa's blog on ordinal logistic regression allowed me to include this topic, which otherwise would be admittedly beyond my own skills.

I also want to thank Carolyn Mayer for reading the manuscript and replacing colloquial expressions with professional English. And a special hug goes to my wife, who not only provided important suggestions for the structure of the book, but also helped with tips on how to teach programming, and provided support with all the tea-related aspects of the book.

If you have a suggestion or correction, please send an email to my work address `thomas.haslwanter@fh-linz.at`. If I make a change based on your feedback, I will add you to the list of contributors unless advised otherwise. If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not as easy to work with. Thanks!

Linz, Austria Thomas Haslwanter
December 2015

Contents

Acronyms

ANOVA	ANalysis Of VAriance
CDF	Cumulative distribution function
CI	Confidence interval
DF/DOF	Degrees of freedom
EOL	End of line
GLM	Generalized linear models
HTML	HyperText Markup Language
IDE	Integrated development environment
IQR	Inter quartile range
ISF	Inverse survival function
KDE	Kernel density estimation
MCMC	Markov chain Monte Carlo
NAN	Not a number
OLS	Ordinary least squares
PDF	Probability density function
PPF	Percentile point function
QQ-Plot	Quantile-quantile plot
ROC	Receiver operating characteristic
RVS	Random variate sample
SD	Standard deviation
SE/SEM	Standard error (of the mean)
SF	Survival function
SQL	Structured Query Language
SS	Sum of squares
Tukey HSD	Tukey honest significant difference test

Part I
Python and Statistics

The first part of the book presents an introduction to statistics based on *Python*. It is impossible to cover the whole language in 30 or 40 pages, so if you are a beginner, please see one of the excellent *Python* introductions available in the internet for details. Links are given below. This part is a kick-start for *Python*; it shows how to install *Python* under Windows, Linux, or MacOS, and goes step-by-step through documented programming examples. Tips are included to help avoid some of the problems frequently encountered while learning *Python*.

Because most of the data for statistical analysis are commonly obtained from text files, Excel files, or data preprocessed by Matlab, the second chapter presents simple ways to import these types of data into *Python*.

The last chapter of this part illustrates various ways of visualizing data in *Python*. Since the flexibility of *Python* for interactive data analysis has led to a certain complexity that can frustrate new *Python* programmers, the code samples presented in Chap. 3 for various types of interactive plots should help future Pythonistas avoid these problems.

Chapter 1
Why Statistics?

Statistics is the explanation of variance in the light of what remains unexplained.

Every day we are confronted with situations with uncertain outcomes, and must make decisions based on incomplete data: "Should I run for the bus? Which stock should I buy? Which man should I marry? Should I take this medication? Should I have my children vaccinated?" Some of these questions are beyond the realm of statistics ("Which person should I marry?"), because they involve too many unknown variables. But in many situations, statistics can help extract maximum knowledge from information given, and clearly spell out what we know and what we don't know. For example, it can turn a vague statement like "This medication may cause nausea," or "You could die if you don't take this medication" into a specific statement like "Three patients in one thousand experience nausea when taking this medication," or "If you don't take this medication, there is a 95 % chance that you will die."

Without statistics, the interpretation of data can quickly become massively flawed. Take, for example, the estimated number of German tanks produced during World War II, also known as the "German Tank Problem." The estimate of the number of German tanks produced per month from standard intelligence data was *1,550*; however, the statistical estimate based on the number of tanks observed was *327*, which was very close to the actual production number of *342* (http://en.wikipedia.org/wiki/German_tank_problem).

Similarly, using the wrong tests can also lead to erroneous results.

In general, statistics will help to

- Clarify the question.
- Identify the variable and the measure of that variable that will answer that question.
- Determine the required sample size.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_1

- Describe variation.
- Make quantitative statements about estimated parameters.
- Make predictions based on your data.

Reading the Book Statistics was originally invented—like so many other things—by the famous mathematician C.F. Gauss, who said about his own work, "Ich habe fleissig sein müssen; wer es gleichfalls ist, wird eben so weit kommen." ("I had to work hard; if you work hard as well, you, too, will be successful."). Just as reading a book about playing the piano won't turn you into a great pianist, simply reading this book will not teach you statistical data analysis. If you don't have your own data to analyze, you need to do the exercises included. Should you become frustrated or stuck, you can always check the sample Solutions provided at the end of the book.

Exercises Solutions to the exercises provided can be found at the end of the book. In my experience, very few people work through large numbers of examples on their own, so I have not included additional exercises in this book.

If the information here is not sufficient, additional material can be found in other statistical textbooks and on the web:

Books There are a number of good books on statistics. My favorite is Altman (1999): it does not dwell on computers and modeling, but gives an extremely useful introduction to the field, especially for life sciences and medical applications. Many formulations and examples in this manuscript have been taken from that book. A more modern book, which is more voluminous and, in my opinion, a bit harder to read, is Riffenburgh (2012). Kaplan (2009) provides a simple introduction to modern regression modeling. If you know your basic statistics, a very good introduction to Generalized Linear Models can be found in Dobson and Barnett (2008), which provides a sound, advanced treatment of statistical modeling.

WWW In the web, you will find very extensive information on statistics in English at

- http://www.statsref.com/
- http://www.vassarstats.net/
- http://www.biostathandbook.com/
- http://onlinestatbook.com/2/index.html
- http://www.itl.nist.gov/div898/handbook/index.htm

A good German web page on statistics and regulatory issues is http://www.reiter1.com/.

I hope to convince you that *Python* provides clear and flexible tools for most of the statistical problems that you will encounter, and that you will enjoy using it.

Chapter 2
Python

Python is a very popular open source programming language. At the time of writing, *codeeval* was rating *Python* "the most popular language" for the fourth year in a row (http://blog.codeeval.com/codeevalblog). There are three reasons why I have switched from other programming languages to *Python*:

1. It is the most elegant programming language that I know.
2. It is free.
3. It is powerful.

2.1 Getting Started

2.1.1 Conventions

In this book the following conventions will be used:

- Text that is to be typed in at the computer is written in Courier font, e.g., `plot(x,y)`.
- Optional text in command-line entries is expressed with square brackets and underscores, e.g., `[_InstallationDir_]\bin`. (I use the underscores in addition, as sometimes the square brackets will be used for commands.)
- Names referring to computer programs and applications are written in italics, e.g., *IPython*.
- I will also use italics when introducing new terms or expressions for the first time.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_2

Code samples are marked as follows:

Python code samples.

All the marked code samples are freely available, under http://www.quantlet.de.

Additional *Python* scripts (the listings of complete programs, as well as the *Python* code used to generate the figures) are available at *github*: https://github.com/thomas-haslwanter/statsintro_python.git, in the directory `ISP` (for "Introduction to Statistics with Python"). `ISP` contains the following subfolders:

Exercise_Solutions contains the solutions to the exercises which are presented at the end of most chapters.
Listings contains programs that are explicitly listed in this book.
Figures lists all the code used to generate the remaining figures in the book.
Code_Quantlets contains all the marked code samples, grouped by book-chapter.

Packages on *github* are called *repositories*, and can easily be copied to your computer: when git is installed on your computer, simply type

```
git clone [_RepositoryName_]
```

and the whole repository—code as well as data—will be "cloned" to your system. (See Sect. 2.4.4 for more information on *git*, *github* and code-versioning.)

2.1.2 Distributions and Packages

a) Python Packages for Statistics

The *Python* core distribution contains only the essential features of a general programming language. For example, it does not even contain a specialized module for working efficiently with vectors and matrices! These specialized modules are being developed by dedicated volunteers. The relationship of the most important *Python* packages for statistical applications is delineated in Fig. 2.1.

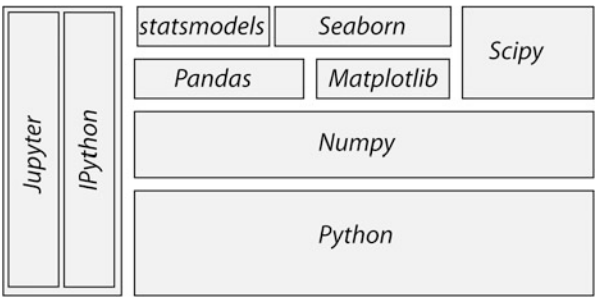


Fig. 2.1 The structure of the most important *Python* packages for statistical applications

To facilitate the use of *Python*, the so-called *Python distributions* collect matching versions of the most important packages, and I strongly recommend using one of these distributions when getting started. Otherwise one can easily become overwhelmed by the huge number of *Python* packages available. My favorite *Python* distributions are

- *WinPython* recommended for Windows users. At the time of writing, the latest version was 3.5.1.3 (newer versions also ok).
  https://winpython.github.io/
- *Anaconda* by Continuum. For Windows, Mac, and Linux. Can be used to install Python 2.x and 3.x, even simultaneously! The latest *Anaconda* version at time of writing was 4.0.0 (newer versions also ok).
  https://store.continuum.io/cshop/anaconda/

Neither of these two distributions requires administrator rights. I am presently using *WinPython*, which is free and customizable. *Anaconda* has become very popular recently, and is free for educational purposes.

Unless you have a specific requirement for 64-bit versions, you may want to install a 32-bit version of *Python*: it facilitates many activities that require compilation of module parts, e.g., for Bayesian statistics (PyMC), or when you want to speed up your programs with *Cython*. Since all the *Python* packages required for this course are now available for *Python* 3.x, I will use *Python* 3 for this book. However, all the scripts included should also work for *Python* 2.7. Make sure that you use a current version of *IPython/Jupyter* (4.x), since the *Jupyter Notebooks* provided with this book won't run on *IPython* 2.x.[1]

The programs included in this book have been tested with Python 2.7.10 and 3.5.1, under Windows and Linux, using the following package versions:

- *ipython 4.1.2* ... For interactive work.
- *numpy 1.11.0* ... For working with vectors and arrays.
- *scipy 0.17.1* ... All the essential scientific algorithms, including those for basic statistics.
- *matplotlib 1.5.1* ... The de-facto standard module for plotting and visualization.
- *pandas 0.18.0* ... Adds *DataFrames* (imagine powerful spreadsheets) to *Python*.
- *patsy 0.4.1* ... For working with statistical formulas.
- *statsmodels 0.8.0* ... For statistical modeling and advanced analysis.
- *seaborn 0.7.0* ... For visualization of statistical data.

In addition to these fairly general packages, some specialized packages have also been used in the examples accompanying this book:

- *xlrd 0.9.4* ... For reading and writing MS Excel files.
- *PyMC 2.3.6* ... For Bayesian statistics, including Markov chain Monte Carlo simulations.

[1] During the writing of this book, the former monolithic *IPython* was split into two separate projects: *Jupyter* is providing the front end (the notebook, the qtconsole, and the console), and *IPython* the computational kernel running the *Python* commands.

- *scikit-learn 0.17.1* . . . For machine learning.
- *scikits.bootstrap 0.3.2* . . . Provides bootstrap confidence interval algorithms for scipy.
- *lifelines 0.9.1.0* . . . Survival analysis in *Python*.
- *rpy2 2.7.4* . . . Provides a wrapper for *R*-functions in *Python*.

Most of these packages come either with the *WinPython* or *Anaconda* distributions, or can be installed easily using `pip` or `conda`. To get *PyMC* to run, you may need to install a C-compiler. On my *Windows* platform, I installed *Visual Studio 15*, and set the environment variable `SET VS90COMNTOOLS=%VS14COMNTOOLS%`.

To use *R*-function from within *Python*, you also have to install *R*. Like *Python*, *R* is available for free, and can be downloaded from the *Comprehensive R Archive Network*, the latest release at the time of writing being *R-3.3.0* (http://cran.r-project.org/).

b) PyPI: The Python Package Index

The Python Package Index (*PyPI*) (Currently at https://pypi.python.org/pypi, but about to migrate to https://pypi.io) is a repository of software for the *Python* programming language. It currently contains more than 80,000 packages!

Packages from *PyPI* can be installed easily, from the Windows command shell (`cmd`) or the Linux `terminal`, with

```
pip install [_package_]
```

To update a package, use

```
pip install [_package_] -U
```

To get a list of all the *Python* packages installed on your computer, type

```
pip list
```

Anaconda uses `conda`, a more powerful installation manager. But *pip* also works with *Anaconda*.

2.1.3 Installation of Python

a) Under Windows

Neither *WinPython* nor *Anaconda* require administrator rights for installation.

WinPython

In the following, I assume that `[_WinPythonDir_]` is the installation directory for *WinPython*.

Tip: Do NOT install *WinPython* into the *Windows* program directory (typically `C:\Program Files` or `C:\Program Files (x86)`), because this typically leads to permission problems during the execution of *WinPython*.

- Download WinPython from https://winpython.github.io/.
- Run the downloaded `.exe`-file, and install *WinPython* into the `[_WinPythonDir_]` of your choice.
- After the installation, make a change to your *Windows Environment*, by typing `Win -> env -> Edit environment variables for your account`:
  - Add `[_WinPythonDir_]\python-3.5.1;[_WinPythonDir_]\python-3.5.1\Scripts\;` to your `PATH`. (This makes *Python* and *ipython* accessible from the standard *Windows* command-line.)[2]
  - If you do have administrator rights, you should activate `[_WinPythonDir_]\WinPython Control Panel.exe -> Advanced -> Register Distribution`. (This associates `.py`-files with this *Python* distribution.)

Anaconda

- Download *Anaconda* from https://store.continuum.io/cshop/anaconda/.
- Follow the installation instructions from the webpage. During the installation, allow *Anaconda* to make the suggested modifications to your environment `PATH`.
- After the installation: in the *Anaconda Launcher*, click *update* (besides the Apps), in order to ensure that you are running the latest version.

Installing Additional Packages

Important Note: When I have had difficulties installing additional packages, I have been saved more than once by the pre-compiled packages from Christoph Gohlke, available under http://www.lfd.uci.edu/~gohlke/pythonlibs/: from there you can download the `[_xxx_x].whl` file for your current version of *Python*, and then install it simply with `pip install [_xxx_].whl`.

b) Under Linux

The following procedure worked on *Linux Mint 17.1*:

- Download *Anaconda* for *Python 3.5* (I used the 64 bit version, since I have a 64-bit *Linux Mint* Installation).

[2] In my current Windows 10 environment, I have to change the path directly by using the command "regedit" to modify the variable "HKEY_CURRENT_USER | Environment"

- Open `terminal`, and navigate to the location where you downloaded the file to.
- Install *Anaconda* with `bash Anaconda3-4.0.0-Linux-x86.sh`
- Update your Linux installation with `sudo apt-get update`

Notes

- You do NOT need root privileges to install *Anaconda*, if you select a user writable install location, such as `~/Anaconda`.
- After the self extraction is finished, you should add the *Anaconda* binary directory to your `PATH` environment variable.
- As all of *Anaconda* is contained in a single directory, uninstalling *Anaconda* is easy: you simply remove the entire install location directory.
- If any problems remain, Mac and Unix users should look up Johansson' installations tips: (https://github.com/jrjohansson/scientific-python-lectures).

c) Under Mac OS X

Downloading *Anaconda* for *Mac OS X* is simple. Just

- go to continuum.io/downloads
- choose the Mac installer (make sure you select the *Mac OS X Python 3.x Graphical Installer*), and follow the instructions listed beside this button.
- After the installation: in the *Anaconda Launcher*, click *update* (besides the Apps), in order to ensure that you are running the latest version.

After the installation the *Anaconda* icon should appear on the desktop. No admin password is required. This downloaded version of *Anaconda* includes the *Jupyter notebook*, *Jupyter qtconsole* and the IDE *Spyder*.

To see which packages (e.g., *numpy*, *scipy*, *matplotlib*, *pandas*, etc.) are featured in your installation look up the *Anaconda Package List* for your *Python* version. For example, the *Python*-installer may not include *seaborn*. To add an additional package, e.g., *seaborn*, open the `terminal`, and enter `pip install seaborn`.

2.1.4 Installation of R and rpy2

If you have not used *R* previously, you can safely skip this section. However, if you are already an avid *R* used, the following adjustments will allow you to also harness the power of *R* from within *Python*, using the package *rpy2*.

a) Under Windows

Also *R* does not require administrator rights for installation. You can download the latest version (at the time of writing *R 3.0.0*) from http://cran.r-project.org/, and install it into the `[_RDir_]` installation directory of your choice.

With WinPython

- After the installation of *R*, add the following two variables to your *Windows Environment*, by typing
  `Win -> env -> Edit environment variables for your account:`
  - `R_HOME=[_RDir_]\R-3.3.0`
  - `R_USER=[_YourLoginName_]`

  The first entry is required for *rpy2*. The last entry is not really necessary, just better style.

With Anaconda

Anaconda comes without *rpy2*. So after the installation of *Anaconda* and *R*, you should:

- Get *rpy2* from http://www.lfd.uci.edu/~gohlke/pythonlibs/: Christoph Gohlkes *Unofficial Windows Binaries for Python Extension Packages* are one of the mainstays of the Python community—Thanks a lot, Christoph!
- Open the *Anaconda command prompt*
- Install *rpy2* with `pip`. In my case, the command was
  `pip rpy2-2.6.0-cp35-none-win32.whl`

b) Under Linux

- After the installation of *Anaconda*, install *R* and *rpy2* with
  `conda install -c https://conda.binstar.org/r rpy2`

2.1.5 Personalizing IPython/Jupyter

When working on a new problem, I always start out with the *Jupyter qtconsole* (see Sect. 2.3). Once I have the individual steps working, I use the *IPython* command `%history` to get the sequence of commands I have used, and switch to an IDE (integrated development environment), typically *Wing* or *Spyder* (see below).

In the following, *[_mydir_]* has to be replaced with your home-directory (i.e., the directory that opens up when you run `cmd` in Windows, or `terminal` in Linux). And `[_myname_]` should be replaced by your name or your userID.

To start up *IPython* in a folder of your choice, and with personalized startup scripts, proceed as follows.

a) In Windows

- Type Win+R, and start a command shell with `cmd`
- In the newly created command shell, type `ipython`. (This will launch an *ipython* session, and create the directory `[_mydir_]\.ipython`).
- Add the Variable `IPYTHONDIR` to your environment (see above), and set it to `[_mydir_]\.ipython`. This directory contains the startup-commands for your *ipython*-sessions.
- Into the startup folder `[_mydir_].ipython\profile_default\startup` place a file with, e.g., the name `00_[_myname_].py`, containing the startup commands that you want to execute every time that you launch *ipython*. My personal startup file contains the following lines:

```
import pandas as pd
import os
os.chdir(r'C:\[_mydir_]')
```

  This will import *pandas*, and start you working in the directory of your choice.

  Note: since Windows uses `\` to separate directories, but `\` is also the escape character in strings, directory paths using a simple backslash have to be preceded by "r," indicating "raw strings".
- Generate a file "ipy.bat" in *mydir*, containing

```
jupyter qtconsole
```

To see all *Jupyter Notebooks* that come with this book, for example, do the following:

- Type Win+R, and start a command shell with `cmd`
- Run the commands

```
cd [_ipynb-dir_]
jupyter notebook
```

- Again, if you want, you can put this command sequence into a batch-file.

b) In Linux

- Start a Linux terminal with the command `terminal`
- In the newly created command shell, execute the following command

```
ipython
```

  (This generates a folder *.ipython*)

- Into the sub-folder `.ipython/profile_default/startup`, place a file with e.g., the name `00[_myname_].py`, containing the lines

```
import pandas as pd
import os
os.chdir([_mydir_])
```

- In your `.bashrc` file (which contains the startup commands for your shell-scripts), enter the lines

```
alias ipy='jupyter qtconsole'
IPYTHONDIR='~/.ipython'
```

- To see all *Jupyter Notebooks*, do the following:
  - Go to `[_mydir_]`
  - Create the file `ipynb.sh`, containing the lines

```
#!/bin/bash
cd [wherever_you_have_the_ipynb_files]
jupyter notebook
```

  - Make the file executable, with `chmod 755 ipynb.sh`

Now you can start "your" *IPython* by just typing `ipy`, and the *Jupyter Notebook* by typing `ipynb.sh`

c) In Mac OS X

- Start the *Terminal* either by manually opening *Spotlight* or the shortcut `CMD + SPACE` and entering `Terminal` and search for "Terminal."
- In *Terminal*, execute `ipython`, which will generate a folder under `[_mydir_]/.ipython`.
- Enter the command `pwd` into the *Terminal*. This lists `[_mydir_]`; copy this for later use.
- Now open *Anaconda* and launch an editor, e.g., *spyder-app* or `TextEdit`.[3] Create a file containing the command lines you regularly use when writing code (you can always open this file and edit it). For starters you can create a file with the following command lines:

```
import pandas as pd
import os
os.chdir('[_mydir_]/.ipython/profile_[_myname_]')
```

- The next steps are somewhat tricky. *Mac OS X* hides the folders that start with ".". So to access `.ipython` open `File -> Save as` \ .... Now open a *Finder* window, click the *Go* menu, select `Go to Folder` and enter

[3] More help on text-files can be found under http://support.smqueue.com/support/solutions/articles/31751-how-to-create-a-plain-text-file-on-a-mac-computer-for-bulk-uploads.

`[ _mydir_ ]/.ipython/profile_default/startup`. This will open a *Finder* window with a header named "startup". On the left of this text there should be a blue folder icon. Drag and drop the folder into the *Save as...* window open in the editor. IPython has a *README* file explaining the naming conventions. In our case the file must begin with `00-`, so we could name it `00-[ _myname_ ].`

- Open your `.bash_profile` (which contains the startup commands for your shellscripts), and enter the line
  `alias ipy='jupyter qtconsole'`
- To see all *Jupyter* Notebooks, do the following:
  - Go to `[_mydir_]`
  - Create the file ipynb.sh, containing the lines

```
#!/bin/bash
cd [wherever_you_have_the_ipynb_files]
jupyter notebook
```

  - Make the file executable, with `chmod 755 ipynb.sh`

2.1.6 Python Resources

If you have some programming experience, this book may be all you need to get the statistical analysis of your data going. But if required, very good additional information can be found on the web, where tutorials as well as good free books are available online. The following links are all recommendable sources of information if you are starting with *Python*:

- *Python Scientific Lecture Notes* If you don't read anything else, read this! (http://scipy-lectures.github.com)
- *NumPy for Matlab Users* Start here if you have *Matlab* experience. (https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html; also check http://mathesaurus.sourceforge.net/matlab-numpy.html)
- *Lectures on scientific computing with Python* Great *Jupyter Notebooks*, from JR Johansson! (https://github.com/jrjohansson/scientific-python-lectures)
- *The Python tutorial* The official introduction. (http://docs.python.org/3/tutorial)

In addition free *Python* books are available, for different levels of programming skills:

- *A Byte of Python* A very good book, at the introductory level. (http://swaroopch.com/notes/python)
- *Learn Python the Hard Way* (3rd Ed) A popular book that you can work through. (http://learnpythonthehardway.org/book/)

- *Think Python* For advanced programmers. (http://www.greenteapress.com/thinkpython)
- *Introduction to Python for Econometrics, Statistics and Data Analysis* Introduces *Python* with a focus on statistics (Sheppard 2015).
- *Probabilistic Programming and Bayesian Methods for Hackers* An excellent introduction into Bayesian thinking. The section on Bayesian statistics in this book is also based on that book (Pilon 2015).

I have not seen many textbooks on *Python* that I have really liked. My favorite introductory books are Harms and McDonald (2010), and the more recent Scopatz and Huff (2015).

When I run into a problem while developing a new piece of code, most of the time I just google; thereby I stick primarily (a) to the official *Python* documentation pages, and (b) to http://stackoverflow.com/. Also, I have found user groups surprisingly active and helpful!

2.1.7 First Python Programs

a) Hello World

Python Shell

Python is an interpreted language. The simplest way to start *Python* is to type `python` on the command line. (When I say *command line* I refer in *Windows* to the command shell started with `cmd`, and in *Linux* or *Mac OS X* to the `terminal`.) Then you can already start to execute *Python* commands, e.g., the command to print "Hello World" to the screen: `print('Hello World')`. On my Windows computer, this results in

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25) [
    MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
    information.
>>> print('Hello World')
Hello World
>>>
```

However, I never use the basic *Python* shell any more, but always start out with the *IPython/Jupyter qtconsole* described in more detail in Sect. 2.3. The *Qt console* is an interactive programming environment which offers a number of advantages. For example, when you type `print(` in the *Qt console*, you immediately see information about the possible input arguments for the command `print`.

Python Modules

Often we want to store our commands in a file for later reuse. *Python* files have the extension `.py`, and are referred to as *Python modules*. Let us create a new file with the name `helloWorld.py`, containing the line

```
print('Hello World')
```

This file can now be executed by typing `python helloWorld.py` on the command line.

In *Windows* you can actually run the file by double-clicking it, or by simply typing `helloWorld.py` if the extension `.py` is associated with the *Python* program installed on your computer. In *Linux* and *Mac OS X* the procedure is slightly more involved. There, the file needs to contain an additional first line specifying the path to the *Python* installation.

```
#! \usr\bin\python
print('Hello World')
```

On these two systems, you also have to make the file executable, by typing `chmod +x helloWorld.py`, before you can run it with `helloWorld.py`.

b) SquareMe

To increase the level of complexity, let us write a *Python* module which prints out the square of the numbers from zero to five. We call the file `squareMe.py`, and it contains the following lines

Listing 2.1 squareMe.py

```
1 # This file shows the square of the numbers from 0 to 5.
2
3 def squared(x):
4     return x**2
5
6 for ii in range(6):
7     print(ii, squared(ii))
8
9 print('Done')
```

Let me explain what happens in this file, line-by-line:

1 The first line starts with "#", indicating a comment-line.

3–4 These two lines define the function *squared*, which takes the variable *x* as input, and returns the square (`x**2`) of this variable.
Note: The range of the function is defined by the indentation! This is a feature loved by many *Python* programmers, but often found confusing by newcomers. Here the last indented line is *line 4*, which ends the function definition.

6–7 Here the program loops over the first 6 numbers. Also the range of the `for`-loop is defined by the indentation of the code.
In *line 7*, each number and its corresponding square are printed to the output.

9 This command is not indented, and therefore is executed after the `for`-loop has ended.

Notes

- Since *Python* starts at 0, the loop in *line 6* includes the numbers from 0 to 5.
- In contrast to some other languages *Python* distinguishes the syntax for function calls from the syntax for addressing elements of an array etc: function calls, as in *line 7*, are indicated with round brackets `( ... )`; and individual elements of arrays or vectors are addressed by square brackets `[ ... ]`.

2.2 Python Data Structures

2.2.1 Python Datatypes

Python offers a number of powerful data structures, and it pays off to make yourself familiar with them. One can use

- *Tuples* to group objects of different types.
- *Lists* to group objects of the same types.
- *Arrays* to work with numerical data. (*Python* also offers the data type *matrix*. However, it is recommended to use *arrays*, since many numerical and scientific functions will not accept input data in *matrix* format.)
- *Dictionaries* for named, structured data sets.
- *DataFrames* for statistical data analysis.

Tuple ( ) A collection of different things. Tuples are "immutable", i.e., they cannot be modified after creation.

```
In [1]: import numpy as np

In [2]: myTuple = ('abc', np.arange(0,3,0.2), 2.5)

In [3]: myTuple[2]
Out[3]: 2.5
```

List [] Lists are "mutable", i.e., their elements can be modified. Therefore lists are typically used to collect items of the same type (numbers, strings, ...). Note that "+" concatenates lists.

```
In [4]: myList = ['abc', 'def', 'ghij']

In [5]: myList.append('klm')

In [6]: myList
Out[6]: ['abc', 'def', 'ghij', 'klm']

In [7]: myList2 = [1,2,3]

In [8]: myList3 = [4,5,6]

In [9]: myList2 + myList3
Out[9]: [1, 2, 3, 4, 5, 6]
```

Array [] *vectors* and *matrices*, for numerical data manipulation. Defined in *numpy*. Note that vectors and 1-d arrays are different: vectors CANNOT be transposed! With arrays, "+" adds the corresponding elements; and the array-method *.dot* performs a scalar multiplication of two arrays. (From Python 3.5 onward, this can also be achieved with the "@" operator.).

```
In [10]: myArray2 = np.array(myList2)

In [11]: myArray3 = np.array(myList3)

In [12]: myArray2 + myArray3
Out[12]: array([5, 7, 9])

In [13]: myArray2.dot(myArray3)
Out[13]: 32
```

Dictionary { } Dictionaries are unordered *(key/value)* collections of content, where the content is addressed as `dict['key']`. Dictionaries can be created with the command `dict`, or by using curly brackets {...}:

```
In [14]: myDict = dict(one=1, two=2, info='some information')

In [15]: myDict2 = {'ten':1, 'twenty':20,
                    'info':'more information'}

In [16]: myDict['info']
Out[16]: 'some information'

In [17]: myDict.keys()
Out[17]: dict_keys(['one', 'info', 'two'])
```

DataFrame Data structure optimized for working with named, statistical data. Defined in *pandas*. (See Sect. 2.5.)

2.2.2 Indexing and Slicing

The rules for addressing individual elements in *Python* lists or tuples or in *numpy* arrays are pretty simple really, and have been nicely summarized by Greg Hewgill on *stackoverflow*[4]:

```
a[start:end] # items start through end-1
a[start:]    # items start through the rest of the array
a[:end]      # items from the beginning through end-1
a[:]         # a copy of the whole array
```

There is also the `step` value, which can be used with any of the above:

```
a[start:end:step] # start through not past end, by step
```

The key points to remember are that indexing starts at 0, *not* at 1; and that the `:end` value represents the first value that is *not* in the selected slice. So, the difference between `end` and `start` is the number of elements selected (if `step` is 1, the default).

The other feature is that `start` or `end` may be a negative number, which means it counts from the end of the array instead of the beginning. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

As a result, `a[:5]` gives you the first five elements (*Hello* in Fig. 2.2), and `a[-5:]` the last five elements (*World*).

2.2.3 Vectors and Arrays

numpy is the *Python* module that makes working with numbers efficient. It is commonly imported with

```
import numpy as np
```

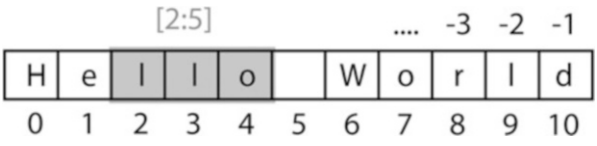


Fig. 2.2 Indexing starts at 0, and slicing does *not* include the last value

[4] http://stackoverflow.com/questions/509211/explain-pythons-slice-notation.

By default, it produces vectors. The commands most frequently used to generate numbers are:

np.zeros generates zeros. Note that it takes only one(!) input. If you want to generate a matrix of zeroes, this input has to be a tuple, containing the number of rows/columns!

```
In [1]: import numpy as np

In [2]: np.zeros(3)
Out[2]: array([ 0.,  0.,  0.])

In [3]: np.zeros( (2,3) )
Out[3]: array([[ 0.,  0.,  0.],
               [ 0.,  0.,  0.]])
```

np.ones generates ones.

np.random.randn generates normally distributed numbers, with a mean of 0 and a standard deviation of 1.

np.arange generates a range of numbers. Parameters can be `start, end, steppingInterval`. Note that the end-value is excluded! While this can sometimes be a bit awkward, it has the advantage that consecutive sequences can be easily generated, without any overlap, and without missing any data points:

```
In [4]: np.arange(3)
Out[4]: array([0, 1, 2])

In [5]: np.arange(1,3,0.5)
Out[5]: array([ 1. ,  1.5,  2. ,  2.5])

In [6]: xLow = np.arange(0,3,0.5)
In [7]: xHigh = np.arange(3,5,0.5)

In [8]: xLow
Out[8]: array([ 0., 0.5, 1., 1.5, 2., 2.5])

In [9]: xHigh
Out[9]: array([ 3., 3.5, 4., 4.5])
```

np.linspace generates linearly spaced numbers.

```
In [10]: np.linspace(0,10,6)
Out[10]: array([ 0., 2., 4., 6., 8.,  10.])
```

np.array generates a numpy array from given numerical data.

```
In [11]: np.array([[1,2], [3,4]])
Out[11]: array([ [1, 2],
                 [3, 4] ])
```

There are a few points that are peculiar to *Python*, and that are worth noting:

- Matrices are simply "lists of lists". Therefore the first element of a matrix gives you the first row:

```
In [12]: Amat = np.array([ [1, 2],
                           [3, 4] ])

In [13]: Amat[0]
Out[13]: array([1, 2])
```

- A vector is not the same as a one-dimensional matrix! This is one of the few really un-intuitive features of *Python*, and can lead to mistakes that are hard to find. For example, vectors cannot be transposed, but matrices can.

```
In [14]: x = np.arange(3)

In [15]: Amat = np.array([ [1,2], [3,4] ])

In [16]: x.T == x
Out[16]: array([ True,  True,  True], dtype=bool)

In [17]: Amat.T == Amat
Out[17]: array([[ True, False],
                [False,  True]], dtype=bool)
```

2.3 IPython/*Jupyter*: An Interactive Programming Environment

A good workflow for source code development can make a very big difference for coding efficiency. For me, the most efficient way to write new code is as follows: I first get the individual steps worked out interactively in *IPython* (http://ipython.org/). *IPython* provides a programming environment that is optimized for interactive computing with *Python*, similar to the command-line in *Matlab*. It comes with a command history, interactive data visualization, command completion, and lots of features that make it quick and easy to try out code. When the *pylab* mode is activated with `%pylab inline`, *IPython* automatically loads `numpy` and `matplotlib.pyplot` (which is the package used for generating plots) into the active workspace, and provides a very convenient, *Matlab*-like programming environment. The optional argument `inline` directs plots into the current *qtconsole/notebook*.

IPython uses *Jupyter* to provide different interface options, my favorite being the *qtconsole*:

```
jupyter qtconsole
```

A very helpful addition is the browser-based *notebook*, with support for code, text, mathematical expressions, inline plots and other rich media.

```
jupyter notebook
```

Note that many of the examples that come with this book are also available as *Jupyter Notebooks*, which are available at *github*: https://github.com/thomas-haslwanter/statsintro_python.git.

2.3.1 First Session with the Qt Console

An important aspect of statistical data analysis is the interactive, visual inspection of the data. Therefore I strongly recommend to start the data analysis in the *ipython qtonsole*.

For maximum flexibility, I start my *IPython* sessions from the command-line, with the command `jupyter qtconsole`. (Under *WinPython*: if you have problems starting *IPython* from the `cmd` console, use the *WinPython Command Prompt* instead—it is nothing else but a command terminal with the environment variables set such that *Python* is readily found.)

To get started with *Python* and *IPython*, let me go step-by-step through the *IPython* session in Fig. 2.3:

- *IPython* starts out listing the version of *IPython* and *Python* that are used, and showing the most important help calls.
- **In [1]:** The first command `%pylab inline` loads *numpy* and *matplotlib* into the current workspace, and directs *matplotlib* to show plots "inline".

  To understand what is happening here requires a short detour into the structure of scientific *Python*.

  Figure 2.1 shows the connection of the most important *Python* packages that are used in this book. *Python* itself is an interpretative programming language, with no optimization for working with vectors or matrices, or for producing plots. *Packages* which extend the abilities of *Python* must be loaded explicitly. The most important package for scientific applications is *numpy* , which makes working with vectors and matrices fast and efficient, and *matplotlib*, which is the most common package used for producing graphical output. *scipy* contains important scientific algorithms. For the statistical data analysis, `scipy.stats` contains the majority of the algorithms that will be used in this book. *pandas* is a more recent addition, which has become widely adopted for statistical data analysis. It provides *DataFrames*, which are labeled, two-dimensional data structures, making work with data more intuitive. *seaborn* extends the plotting

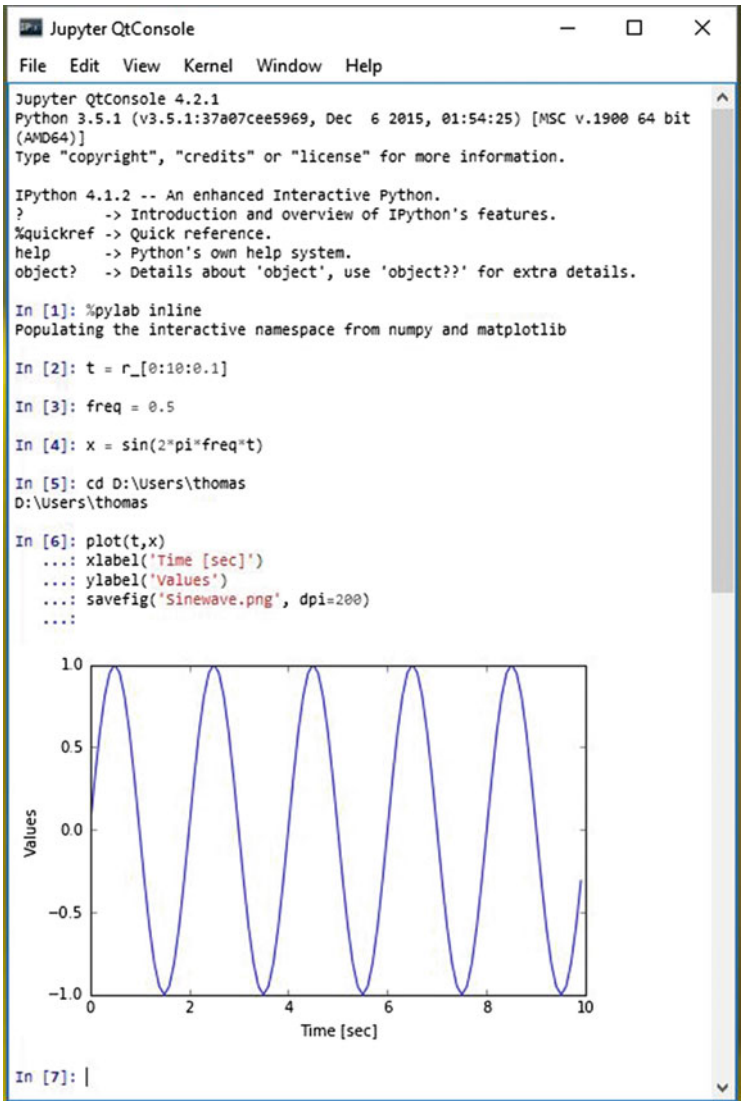


Fig. 2.3 Sample session in the *Jupyter QtConsole*

abilities of *matplotlib*, with a focus on statistical graphs. And *statsmodels* contains many modules for statistical modeling, and for advanced statistical analysis. Both *seaborn* and *statsmodels* make use of *pandas* DataFrames.

IPython provides the tools for interactive data analysis. It lets you quickly display graphs and change directories, explore the workspace, provides a command history etc. The ideas and base structure of *IPython* have been so successful that

the front end has been turned into a project of its own, *Jupyter*, which is now also used by other languages like *Julia*, *R*, and *Ruby*.

- **In [2]:** The command `t = r_[0:10:0.1]` is a shorthand version for `t = arange(0, 10, 0.1)`, and generates a vector from 0 to 10, with a step size of 0.1. `r_` (and `arange`) are commands in the *numpy* package. (`r_` generates *row* vectors, and `c_` is the corresponding *numpy* command to generate *column* vectors.) However, since `numpy` has already been imported into the current workspace by `%pylab inline`, we can use these commands right away.
- **In [4]:** Since *t* is a vector, and *sin* is a function from *numpy*, the sine-value is calculated automatically for each value of *t*.
- **In [5]:** In *Python* scripts, changes of the current folder have to be performed with `os.chdir()`. However, tasks common with interactive computing, such as directory changes (`%cd`), bookmarks for directories (`%bookmark`), inspection of the workspace (`%who` and `%whos`), etc., are implemented as "IPython magic functions". If no *Python* variable with the same name exists, the "%" sign can be left away, as here.
- **In [6]:** Since we have started out with the command `%pylab inline`, *IPython* generates plots in the *Jupyter QtConsole*, as shown in Fig. 2.3. To enter multi-line commands in *IPython*, one can use `CTRL+Enter` for additional command lines, indicated in the terminal by .... (The command sequence gets executed after the next empty line.)
  Note that also generating graphics files is very simple: here I generate the PNG-file "Sinewave.png", with a resolution of 200 dots-per-inch.

I have mentioned above that *matplotlib* handles the graphics output. In the *Jupyter QtConsole*, you can switch between inline graphs and output into an external graphics-window with `%matplotlib inline` and `%matplotlib qt4` (see Fig. 2.4). (Depending on your version of *Python*, you may have to replace `%matplotlib qt4` with `%matplotlib tk`.) An external graphics window allows to zoom and pan in the figure, get the cursor position (which can help to find outliers), and get interactive input with the command `ginput`. *matplotlib*'s plotting commands closely follow the *Matlab* conventions.

2.3.2 Notebook and rpy2

Many of the code samples accompanying this book are also available as *Jupyter Notebooks*, and can be downloaded from https://github.com/thomas-haslwanter/statsintro_python.git. Therefore the concept of *Notebooks* and their integration with the *R*-language are briefly presented here.

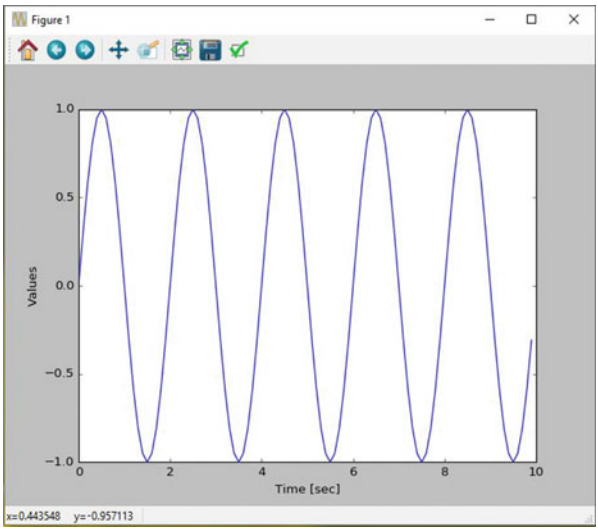


Fig. 2.4 Graphical output window, using the Qt-framework. This allows you to pan, zoom, and get interactive input

a) The Notebook

Since approximately 2013 the *IPython Notebook* has become a very popular way to share research and results in the *Python* community. In 2015 the development of the interface has become its own project, called *Jupyter*, since the *notebook* can be used not only with *Python* language, but also with *Julia*, *R*, and 40 other programming languages. The *notebook* is a browser based interface, which is especially well suited for teaching and for documentation. It allows to combine a structured layout, equations in the popular LaTeX format, and images, and can include resulting graphs and videos, as well as the output from *Python* commands (see Fig. 2.5).

b) rpy2

While *Python* is my preferred programming language, the world of advanced statistics is clearly dominated by *R*. Like *Python*, *R* is completely free and has a very active user community. While *Python* is a general programming language, *R* is optimized for the interactive work with statistical data. Many users swear that *ggplot* provides the best-looking graphs for statistical data.

To combine the best of both worlds, the package *rpy2* provides a way to transfer data from *Python* into *R*, execute commands in *R*, and transfer the results back into *Python*. In the *Jupyter Notebook*, with *rpy2* even *R* graphics can be fully utilized (see Fig. 2.6)!

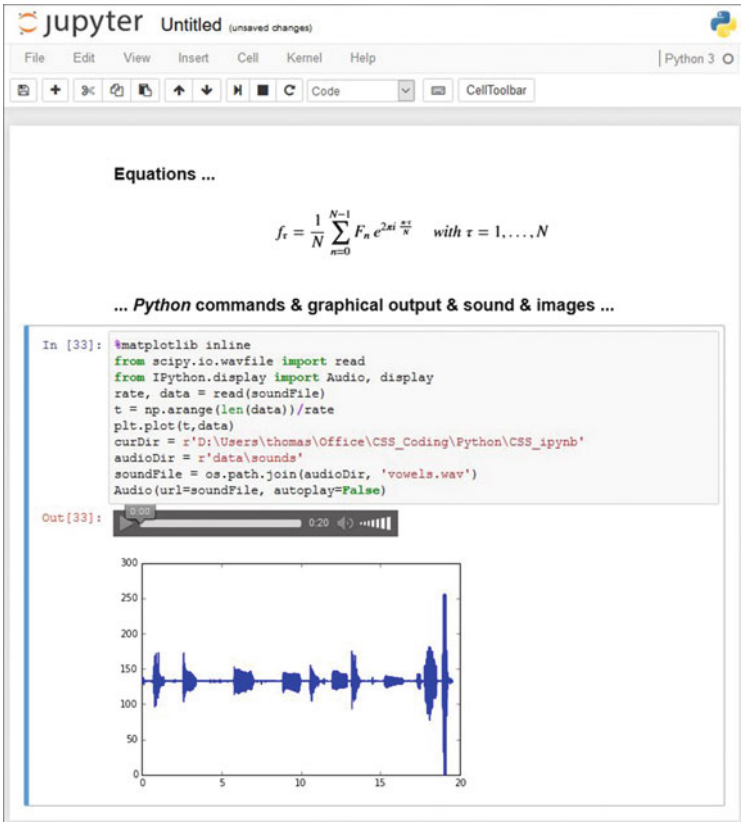


Fig. 2.5 The *Jupyter Notebook* makes it easy to share research, formulas, and results

2.3.3 IPython Tips

1. Use *IPython* in the *Jupyter QtConsole*, and customize your startup as described in Sect. 2.1.5: it will save you time in the long run!
2. For help on e.g., `plot`, use `help(plot)` or `plot?`. With one question mark the help gets displayed, with two question marks (e.g., `plot??`) also the source code is shown.
3. Check out the help tips displayed at the start of *IPython*.
4. Use TAB-completion, for file- and directory names, variable names, AND for commands.
5. To switch between inline and external graphs, use `%matplotlib inline` and `%matplotlib qt4`.
6. By default, *IPython* displays data with a very high precision. For a more concise display, use `%precision 3`.
7. You can use `edit [_fileName_]` to edit files in the local directory, and `%run [_fileName_]` to execute *Python* scripts in your current workspace.

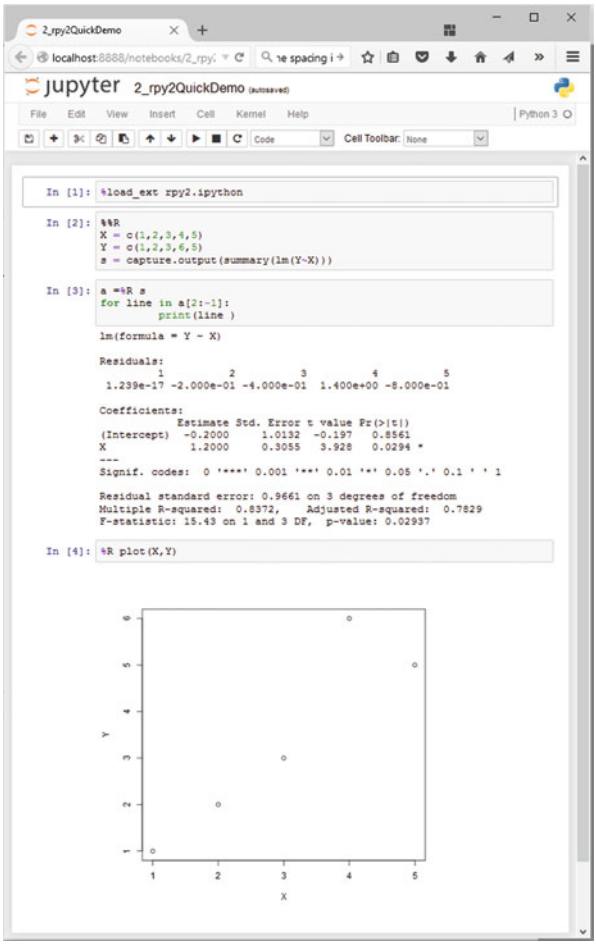


Fig. 2.6 The output from *R*-commands is not working properly yet, and has been "hacked" here. However, this issue should be resolved soon

2.4 Developing Python Programs

2.4.1 Converting Interactive Commands into a Python Program

IPython is very helpful in working out the command syntax and sequence. The next step is to turn these commands into a *Python* program with comments, that can be run from the command-line. This section introduces a fairly large number of *Python* conventions and syntax.

An efficient way to turn *IPython* commands into a function is to

- first obtain the command history with the command `%hist` or `%history`.

- copy the history into a good IDE (integrated development environment): I either use *Wing* (my clear favorite *Python* IDE, although it is commercial; see Fig. 2.7) or *Spyder* (which is good and free; see Fig. 2.8). *PyCharm* is another IDE with a good debugger, and with very good *vim*-emulation.
- turn it into a working *Python* program by adding the relevant package information, etc.

Converting the commands from the interactive session in Fig. 2.3 into a program, we get

Listing 2.2 L2_4_pythonScript.py

```
'''
Short demonstration of a Python script.

author: Thomas Haslwanter
date:   May-2015
ver:    1.0
'''

# Import standard packages
import numpy as np
import matplotlib.pyplot as plt

# Generate the time-values
t = np.r_[0:10:0.1]

# Set the frequency, and calculate the sine-value
freq = 0.5
x = np.sin(2*np.pi*freq*t)

# Plot the data
plt.plot(t,x)

# Format the plot
plt.xlabel('Time[sec]')
plt.ylabel('Values')

# Generate a figure, one directory up
plt.savefig(r'..\Sinewave.png', dpi=200)

# Put it on the screen
plt.show()
```

The following modifications were made from the *IPython* history:

- The commands were put into a files with the extension ".py", a so-called *Python module*.
- **1–7:** It is common style to precede a *Python* module with a header block. Multiline comments are given between triple inverted commas `''' [_ xxx _] '''`. The first comment block describing the module should also contain information about author, date, and version number.

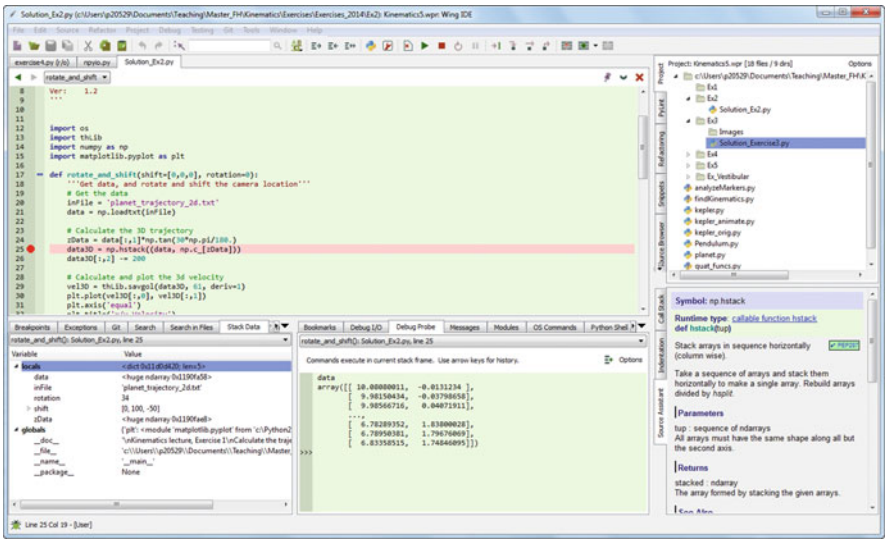


Fig. 2.7 *Wing* is my favorite development environment, with probably the best existing debugger for *Python*. *Tip*: If *Python* does not run right away, you may have to go to `Project -> Project Properties` and set the *Custom* `Python Executable` and/or `Python Path`

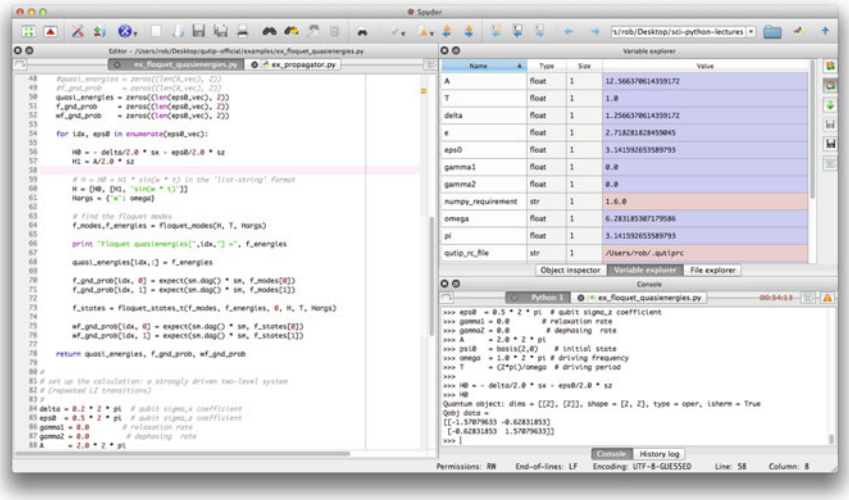


Fig. 2.8 *Spyder* is a very good, free IDE

- **9:** Single-line comments use "#".
- **10–11:** The required *Python* packages have to be imported explicitly. (In IPython, this is done for *numpy* and *matplotlib.pyplot* by the command `%pylab`.) It is customary to import *numpy as np*, and *matplotlib.pyplot*, the matplotlib module containing all the plotting commands, as *plt*.
- **14 etc:** The numpy command `r_` has to be addressed through the corresponding package name, i.e., `np.r_`. (In *IPython*, `%pylab` took care of that.)
- **18:** Note that also "pi" is in *numpy*, so `np.pi` is needed!
- **21 etc:** All the plotting commands are in the package `plt`.
- **28:** Care has to be taken with backslashes in pathnames: in *Windows*, directories in path-names are separated by `"\"`, which is also used as the escape-character in strings. To take `"\"` literally, a string has to be preceded by "r" (for "r"aw string), e.g., `r'C:\Users\Peter'` instead of `'C:\\Users\\Peter'`.
- **34:** While *IPython* automatically shows graphical output, *Python* programs don't show the output until this is explicitly requested by `plt.show()`. The idea behind this is to optimize the program speed, only showing the graphical output when required. The output looks the same as in Fig. 2.4.

2.4.2 Functions, Modules, and Packages

Python has three different levels of modularization:

Function A *function* is defined by the keyword `def`, and can be defined anywhere in *Python*. It returns the object in the `return` statement, typically at the end of the function.

Modules A *module* is a file with the extension ".py". Modules can contain function and variable definitions, as well as valid *Python* statements.

Packages A *package* is a folder containing multiple *Python* modules, and must have a file named `__init__.py`. For example, *numpy* is a *Python* package. Since *packages* are mainly important for grouping a larger number of modules, they won't be discussed in this book.

a) Functions

The following example shows how functions can be defined and used.

Listing 2.3 L2_4_pythonFunction.py

```
1 '''Demonstration of a Python Function
2
3 author: thomas haslwanter, date: May-2015
4 '''
5
6 # Import standard packages
7 import numpy as np
8
```

```
9  def incomeAndExpenses(data):
10     '''Find the sum of the positive numbers, and the sum of
           the negative ones.'''
11     income = np.sum(data[data>0])
12     expenses = np.sum(data[data<0])
13
14     return (income, expenses)
15
16 if __name__=='__main__':
17     testData = np.array([-5, 12, 3, -6, -4, 8])
18
19     # If only real banks would be so nice ;)
20     if testData[0] < 0:
21         print('Your first transaction was a loss, and will be
               dropped.')
22         testData = np.delete(testData, 0)
23     else:
24         print('Congratulations: Your first transaction was a
               gain!')
25
26     (myIncome, myExpenses) = incomeAndExpenses(testData)
27     print('You have earned {0:5.2f} EUR, and spent {1:5.2f}
           EUR.'.format(myIncome, -myExpenses))
```

- **1–4:** Comment header.
- **6:** Since *numpy* will be required in that module, it has to be imported. To reduce the writing to a minimum, it is conventionally called *np*.
- **9/10:** Function definition, and a comment describing the function. Note that in *Python* the function block is defined by the indentation, not by any brackets or *end* statements! This is a feature that irritates many *Python* novices, but really helps to keep code clear and nicely formatted. Important: *Python* makes a difference between a tab and the equivalent amount of spaces. This can lead to errors which are really hard to detect, so use a good IDE that automatically converts tabs to spaces!
- **11:**
  - The `sum` command is taken from *numpy*, so it has to be preceded by `.np`.
  - In *Python*, function arguments are indicated by round brackets `(...)`, whereas elements of lists, tuples, vectors, and arrays are indicated by square brackets `[...]`.
  - In *numpy* you can select elements of an array either with an index (see line **20**), or with a boolean array (line **11**).
- **14:** *Python* also uses round brackets to form groups of elements, the so-called *tuples*. And the `return` statement does the obvious things: it returns elements from a function.
- **16:** Here quite a few new aspects of *Python* come together:
  - Just like function definitions, `if`-loops or `for`-loops use indentation to define their context.

  - *Python* conventionally uses underscores (_) to indicate private variables, which are not used for typical programming tasks.
  - Here we check the variable with the name `__name__`, which is denoting the context of a module evaluation. If the module is run as a *Python* script, `__name__` is set to `__main__`. But if a module is imported, it is set to the name of the importing module. This way it is possible to add code to a function that is only used when the module is executed, but not when the functions in this module are `import`ed by other modules (see below).

- **17:** Definition of a *numpy* array.
- **26:** The two elements returned as a tuple from the function `incomeAndExpenses` can be immediately assigned to two different *Python* objects (`myIncome, myExpenses`).
- **27:** While there are different ways to produce formatted strings, this is probably the most elegant one: curly brackets `{ ... }` indicate values that will be inserted, and can also contain formatting statements. The corresponding values are then passed into the string by the method `format`, e.g., `print('The value of pi is {0}'.format(np.py))`.

b) Modules

To execute the module `pythonFunction.py` from the command-line, type `python pythonFunction.py`. In Windows, if the extension ".py" is associated with the *Python* program, it suffices to double-click the module, or to type `pythonFunction.py` on the command-line. In WinPython the association of the extension ".py" with the *Python* function is set by the *WinPython Control Panel.exe*, by the command *Register Distribution ...* in the menu *Advanced*.

To run a module in *IPython*, use the magic function `%run`:

```
In [56]: %run pythonFunction
Your first transaction was a loss, and will be dropped.
You have earned 23.00 EUR, and spent 10.00 EUR.
```

Note that you either have to be in the directory where the function is defined, or you have to give the full pathname.

If you want to use a function or variable that is defined in a different module, you have to import that module. This can be done in three different ways. For the following example, assume that the other module is called `newModule.py`, and the function that we want from there `newFunction`.

- `import newModule`: The function can then be accessed with `newModule.newFunction()`.
- `from newModule import newFunction`: In this case, the function can be called directly `newFunction()`.
- `from newModule import *`: This imports all variables and functions from `newModule` into the current workspace; again, the function can be called directly

with `newFunction()`. However, use of this syntax is discouraged as it clutters up the current workspace.

If you import a module multiple times, *Python* recognizes that the module is already known, and skips later imports. If you want to override this, and explicitly want to re-import a module that has changed, you have to use the command `reload` from the package `importlib`:

```
from importlib import reload
reload(pythonFunction)
```

Python 2.x: `reload` does NOT need to be imported from `importlib`, but is available as a core module.

The next example shows you how to import functions from one module into another module:

Listing 2.4 L2_4_pythonImport.py

```
1 '''Demonstration of importing a Python module
2
3 author: ThH, date: May-2015'''
4
5 # Import standard packages
6 import numpy as np
7
8 # additional packages: this imports the function defined
      above
9 import L2_4_pythonFunction
10
11 # Generate test-data
12 testData = np.arange(-5, 10)
13
14 # Use a function from the imported module
15 out = L2_4_pythonFunction.incomeAndExpenses(testData)
16
17 # Show some results
18 print('You have earned {0:5.2f} EUR, and spent {1:5.2f} EUR.'
      .format(out[0], -out[1]))
```

- **9:** Here the module `pythonFunction` (that we have just discussed above) is imported. Note that the code in the section `if __name__ == '__main__'` in `pythonFunction.py` is NOT executed when the module is imported!
- **15:** To access the function `incomeAndExpenses` from the module `pythonFunction`, module- and function-name have to be given: `incomeAndExpenses.pythonFunction(...)`

2.4.3 Python Tips

1. Stick to the standard conventions.
   - Every function should have a documentation string on the line below the function definition.
   - Packages should be imported with their commonly used names:
     ```
     import numpy as np
     import matplotlib.pyplot as plt
     import scipy as sp
     import pandas as pd
     import seaborn as sns
     ```
2. To get the current directory, use `os.path.abspath(os.curdir)`. And in Python modules a change of directories can NOT be executed with `cd` (as in *IPython*), but instead requires the command `os.chdir(...)`.
3. Everything in *Python* is an object: to find out about "obj", use `type(obj)` and `dir(obj)`.
4. Learn to use the debugger. Personally, I always use the debugger from the IDE, and rarely resort to the built-in debugger *pdb*.
5. Know lists, tuples, and dictionaries; also, know about *numpy* arrays and *pandas* DataFrames.
6. Use functions a lot, and understand the `if __name__=='__main__':` construct.
7. If you have all your personal functions in the directory `mydir`, you can add this directory to your `PYTHONPATH` with the command
   ```
   import sys
   sys.path.append('mydir')
   ```
8. If you are using non-ASCII characters, such as the German `\"{o}\"{a}\"{u}{\ss}` or the French `\`{e}\'{e}`, you have to let *Python* know, by adding
   ```
   # -*- coding: utf-8 -*-
   ```
   in the first or second line of your *Python* module. This has to be done, even if the non-ASCII characters only appear in the comments! This requirement arises from the fact that *Python* will default to ASCII as standard encoding if no other encoding hints are given.

2.4.4 Code Versioning

Computer programs rarely come out perfect at the first try. Typically they are developed iteratively, by successively eliminating the known errors. *Version control* programs, also known as *revision control* programs, allow tracking only the modifications, and storing previous versions of the program under development. If the latest changes cause a new problem, it is then easy to compare them to earlier versions, and to restore the program to a previous state.

I have been working with a number of version control programs, and *git* is the first one I am really happy with. *git* is a version control program, and *github* is a central source code repository. If you are developing computer software, I strongly recommend the use of *git*. It can be used locally, with very little overhead. And it can also be used to maintain and manage a remote backup copy of the programs. While the real power of *git* lies in its features for collaboration, I have been very happy with it for my own data and software. An introduction to *git* goes beyond the scope of this book, but a very good instruction is available under https://git-scm.com/. Good, short, and simple starting instructions—in many languages—can be found at http://rogerdudler.github.io/git-guide/.

I am mostly working under Windows, and *tortoisegit* (https://tortoisegit.org/) provides a very useful Windows shell interface for *git*. For example, in order to clone a repository from *github* to a computer where *tortoisegit* is installed, simply right-click in the folder on your computer where you want the repository to be installed, select `Git Clone ...`, and enter the repository name—and the whole repository will be cloned there. Done!

github (https://github.com/) is an online project using *git*, and the place where the source code for the majority of *Python* packages is hosted.

2.5 Pandas: Data Structures for Statistics

pandas is a widely used *Python* package which has been contributed by Wes McKinney. It provides data structures suitable for statistical analysis, and adds functions that facilitate data input, data organization, and data manipulation. It is common to `import pandas as pd`, which reduces the typing a bit (http://pandas.pydata.org/).

A good introduction to pandas has been written by Olson (2012).

2.5.1 Data Handling

a) Common Procedures

In statistical data analysis, labeled data structures have turned out to be immensely useful. To handle labeled data in *Python*, *pandas* introduces the so-called *DataFrame* objects. A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table. DataFrames are the most commonly used pandas objects.

Let me start with a specific example, by creating a DataFrame with three columns, called "Time," "x," and "y":

```
import numpy as np
import pandas as pd

t = np.arange(0,10,0.1)
x = np.sin(t)
y = np.cos(t)

df = pd.DataFrame({'Time':t, 'x':x, 'y':y})
```

In *pandas*, rows are addressed through indices and columns through their name. To address the first column only, you have two options:

```
df.Time
df['Time']
```

If you want to extract two columns at the same time, ask for several variables in a list:

```
data = df[['Time', 'y']]
```

To display the first or last rows, use

```
data.head()
data.tail()
```

To extract the six rows from 5 to 10, use

```
data[4:10]
```

as $10 - 4 = 6$. (I know, the array indexing takes some time to get used to. Just keep in mind that *Python* addresses the *locations between* entries, not the entries, and that it starts at 0!)

The handling of DataFrames is somewhat different from the handling of *numpy* arrays. For example, (numbered) rows and (labeled) columns can be addressed simultaneously as follows:

```
df[['Time', 'y']][4:10]
```

You can also apply the standard row/column notation, by using the method `iloc`:

```
df.iloc[4:10, [0,2]]
```

Finally, sometimes you want to have direct access to the data, not to the DataFrame. You can do this with

```
data.values
```

which returns a *numpy* array.

b) Notes on Data Selection

While *pandas*' DataFrames are similar to numpy arrays, their philosophy is different, and I have wasted a lot of nerves addressing data correctly. Therefore I want to explicitly point out the differences here:

numpy handles "rows" first. E.g., `data[0]` is the first row of an array

pandas starts with the columns. E.g., `df['values'][0]` is the first element of the column `'values'`.

If a DataFrame has labeled rows, you can extract for example the row "rowlabel" with `df.loc['rowlabel']`. If you want to address a row by its number, e.g., row number "15," use `df.iloc[15]`. You can also use `iloc` to address "rows/columns," e.g., `df.iloc[2:4,3]`.

Slicing of rows also works, e.g., `df[0:5]` for the first 5 (!) rows. A sometimes confusing convention is that if you want to slice out a single row, e.g., row "5," you have to use `df[5:6]`. If you use `df[5]` alone, you get an error!

2.5.2 Grouping

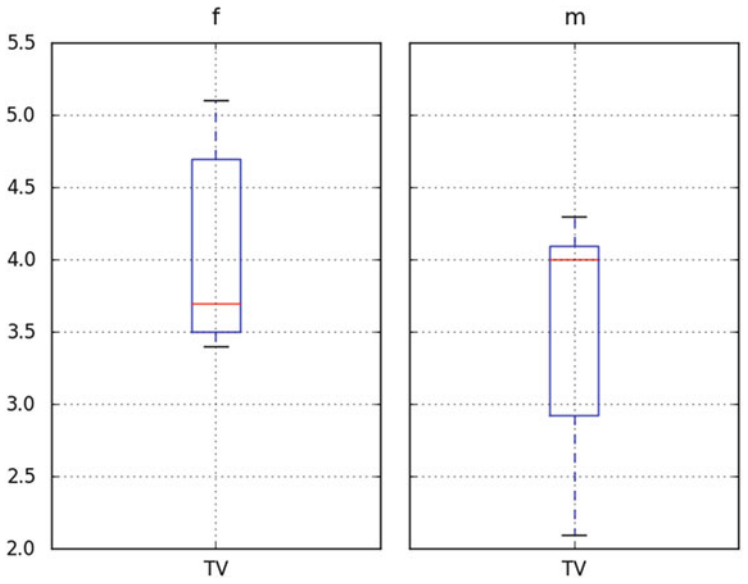
pandas offers powerful functions to handle missing data which are often replaced by *nan's ("Not-A-Number")*. It also allows more complex types of data manipulation like pivoting. For example, you can use data-frames to efficiently group objects, and do a statistical evaluation of each group. The following data are simulated (but realistic) data of a survey on how many hours a day people watch the TV, grouped into "m"ale and "f"emale responses:

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

data = pd.DataFrame({
    'Gender': ['f', 'f', 'm', 'f', 'm',
               'm', 'f', 'm', 'f', 'm', 'm'],
    'TV': [3.4, 3.5, 2.6, 4.7, 4.1, 4.1,
            5.1, 3.9, 3.7, 2.1, 4.3]
    })

#----------------------------------------
```

```
# Group the data
grouped = data.groupby('Gender')

# Do some overview statistics
print(grouped.describe())

# Plot the data:
grouped.boxplot()
plt.show()

#-----------------------------------------
# Get the groups as DataFrames
df_female = grouped.get_group('f')

# Get the corresponding numpy-array
values_female = grouped.get_group('f').values
```

produces

```
                    TV
Gender
f      count  5.000000
       mean   4.080000
       std    0.769415
       min    3.400000
       25%    3.500000
       50%    3.700000
       75%    4.700000
       max    5.100000
m      count  5.000000
       mean   3.360000
       std    0.939681
       min    2.100000
       25%    2.600000
       50%    4.000000
       75%    4.000000
       max    4.100000
```



For statistical analysis, *pandas* becomes really powerful if you combine it with *statsmodels* (see below).

2.6 Statsmodels: Tools for Statistical Modeling

statsmodels is a *Python* package contributed to the community by the *statsmodels* development team (http://www.statsmodels.org/). It has a very active user community, and has in the last five years massively increased the functionality of *Python* for statistical data analysis. *statsmodels* provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration. An extensive list of result statistics are available for each estimator.

statsmodels also allows the formulation of models with the popular formula language based on the notation introduced by Wilkinson and Rogers (1973), and also used by *S* and *R*. For example, the following example would fit a model that assumes a linear relationship between x and y to a given dataset:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm

# Generate a noisy line, and save the data in a pandas-DataFrame
x = np.arange(100)
y = 0.5*x - 20 + np.random.randn(len(x))
df = pd.DataFrame({'x':x, 'y':y})
```

```
# Fit a linear model, using the "formula" language
# added by the package "patsy"
model = sm.ols('y~x', data=df).fit()
print( model.summary() )
```

Another example would be a model that assumes that "success" is determined by intelligence" and "diligence," as well as the interaction of the two. Such a model could be described by

$$success \sim intelligence * diligence$$

More information on that topic is presented in Chap. 11 ("Statistical Models").

An extensive list of result statistics are available for each estimator. The results of all *statsmodels* commands have been tested against existing statistical packages to ensure that they are correct. Features include:

- Linear Regression
- Generalized Linear Models
- Generalized Estimating Equations
- Robust Linear Models
- Linear Mixed Effects Models
- Regression with Discrete Dependent Variables
- ANOVA
- Time Series analysis
- Models for Survival and Duration Analysis
- Statistics (e.g., Multiple Tests, Sample Size Calculations, etc.)
- Nonparametric Methods
- Generalized Method of Moments
- Empirical Likelihood
- Graphics functions
- A Datasets Package

2.7 Seaborn: Data Visualization

seaborn is a *Python* visualization library based on *matplotlib*. Its primary goal is to provide a concise, high-level interface for drawing statistical graphics that are both informative and attractive http://stanford.edu/~mwaskom/software/seaborn/ (Fig. 2.9).

For example, the following code already produces a nice regression plot (Fig. 2.9), with line-fit and confidence intervals:

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

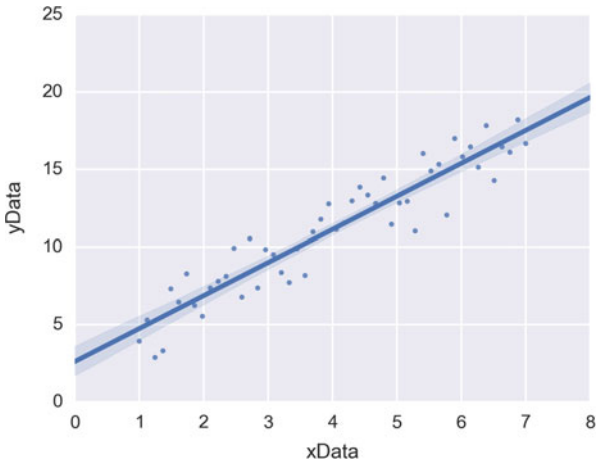


Fig. 2.9 Regression plot, from *seaborn*. The plot shows the data, the best-fit line, and the confidence intervals for the fit

```
import pandas as pd
import seaborn as sns

x = np.linspace(1, 7, 50)
y = 3 + 2*x + 1.5*np.random.randn(len(x))
df = pd.DataFrame({'xData':x, 'yData':y})
sns.regplot('xData', 'yData', data=df)
plt.show()
```

2.8 General Routines

In the examples used later in this book, a few tasks come up repeatedly: reading in data, setting the desired font size and formatting parameters, and generating graphical output files. The two following modules handle those tasks. If you are interested you can check them out; but their understanding is not really required:

Code: "ISP_mystyle.py"[5]: sets commonly used formatting options, and provides functions for standardized graphics-output into files.

[5] https://github.com/thomas-haslwanter/statsintro_python/blob/master/ISP/Code_Quantlets/Utilities.

2.9 Exercises

2.1 Data Input

Read in data from different sources:

- A CVS-file with a header ('.\Data\data_kaplan\swim100m.csv'). Also show the first 5 data points.
- An MS-Excel file ('.\Data\data_others\Table 2.8 Waist loss.xls'). Show the last five data points.
- Read in the same file, but this time from the zipped archive http://cdn.crcpress.com/downloads/C9500/GLM_data.zip.

2.2 First Steps with *Pandas*

- Generate a *pandas* DataFrame, with the x-column time stamps from 0 to 10 s, at a rate of 10 Hz, the y-column data values with a sine with 1.5 Hz, and the z-column the corresponding cosine values. Label the x-column "Time", and the y-column "YVals", and the z-column "ZVals".
- Show the head of this DataFrame.
- Extract the data in lines 10–15 from "Yvals" and "ZVals", and write them to the file "out.txt".
- Let the user know where the data have been written to.

Chapter 3
Data Input

This chapter shows how to read data into *Python*. Thus it forms the link between the chapter on *Python*, and the first chapter on statistical data analysis. It may be surprising, but reading data into the system in the correct format and checking for erroneous or missing entries is often one of the most time consuming parts of the data analysis.

Data input can be complicated by a number of problems, like different separators between data entries (such as spaces and/or tabulators), or empty lines at the end of the file. In addition, data may have been saved in different formats, such as MS Excel, *Matlab*, HDF5 (which also includes the *Matlab*-format), or in databases. Understandably, we cannot cover all possible input options. But I will try to give an overview of where and how to start with data input.

3.1 Input from Text Files

3.1.1 Visual Inspection

When the data are available as ASCII-files, you should always start out with a visual inspection of the data! In particular, you should check

- Do the data have a header and/or a footer?
- Are there empty lines at the end of the file?
- Are there white-spaces before the first number, or at the end of each line? (The latter is a lot harder to see.)
- Are the data separated by tabulators, and/or by spaces? (Tip: you should use a text-editor which can visualize tabs, spaces, and end-of-line (EOL) characters.)

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_3

3.1.2 Reading ASCII-Data into Python

In *Python*, I strongly suggest that you start out reading in and inspecting your data in the *Jupyter QtConsole* or in an *Jupyter Notebook*. It allows you to move around much more easily, try things out, and quickly get feedback on how successful your commands have been. When you have your command syntax worked out, you can obtain the command history with `%history`, copy it into your favorite IDE, and turn it into a program.

While the a numpy command `np.loadtxt` allows to read in simply formatted text data, most of the time I go straight to *pandas*, as it provides significantly more powerful tools for data-entry. A typical workflow can contain the following steps:

- Changing to the folder where the data are stored.
- Listing the items in that folder.
- Selecting one of these items, and reading in the corresponding data.
- Checking if the data have been read in completely, and in the correct format.

These steps can be implemented in *IPython* with the following commands:

```
In [1]: import pandas as pd
In [2]: cd 'C:\Data\storage'
In [3]: pwd     # Check if you were successful
In [4]: ls      # List the files in that directory
In [5]: inFile = 'data.txt'
In [6]: df = pd.read_csv(inFile)
In [7]: df.head()   # Check if first line is ok
In [8]: df.tail()   # Check the last line
```

After "In [7]" I often have to adjust the options of `pd.read_csv`, to read in all the data correctly. Make sure you check that the number of column headers is equal to the number of columns that you expect. It can happen that everything gets read in—but into one large single column!

a) Simple Text-Files

For example, a file `data.txt` with the following content

```
1, 1.3, 0.6
2, 2.1, 0.7
3, 4.8, 0.8
4, 3.3, 0.9
```

can be read in and displayed with

```
In [9]: data = np.loadtxt('data.txt', delimiter=',')

In [10]: data
Out[10]:
    array([[ 1. ,  1.3,  0.6],
           [ 2. ,  2.1,  0.7],
           [ 3. ,  4.8,  0.8],
           [ 4. ,  3.3,  0.9]])
```

where `data` is a *numpy array*. Without the flag `delimiter=','`, the function *np.loadtxt* crashes. An alternative way to read in these data is with

```
In [11]: df = pd.read_csv('data.txt', header=None)

In [12]: df
Out[12]:
   0    1    2
0  1  1.3  0.6
1  2  2.1  0.7
2  3  4.8  0.8
3  4  3.3  0.9
```

where `df` is a *pandas DataFrame*. Without the flag `header=None`, the entries of the first row are falsely interpreted as the column labels!

```
In [13]: df = pd.read_csv('data.txt')

In [14]: df
Out[14]:
     1   1.3   0.6
  0  2   2.1   0.7
  1  3   4.8   0.8
  2  4   3.3   0.9
```

The *pandas* routine has the advantage that the first column is recognized as integer, whereas the second and third columns are float.

b) More Complex Text-Files

The advantage of using *pandas* for data input becomes clear with more complex files. Take for example the input file "data2.txt," containing the following lines:

```
ID, Weight, Value
1, 1.3, 0.6
2, 2.1, 0.7
3, 4.8, 0.8
4, 3.3, 0.9

Those are dummy values, created by ThH.
June, 2015
```

One of the input flags of `pd.read_csv` is `skipfooter`, so we can read in the data easily with

```
In [15]: df2 = pd.read_csv('data.txt',
           skipfooter=3, delimiter='[ ,]*')
```

The last option, `delimiter='[ ,]*'`, is a *regular expression* (see below) specifying that "one or more spaces and/or commas may be used to separate entry values." Also, when the input file includes a header row with the column names, the data can be accessed immediately with their corresponding column name, e.g.:

```
In [16]: df2
Out[16]:
   ID   Weight   Value
0   1      1.3     0.6
1   2      2.1     0.7
2   3      4.8     0.8
3   4      3.3     0.9

In [17]: df2.Value
Out[17]:
0    0.6
1    0.7
2    0.8
3    0.9
Name: Value, dtype: float64
```

c) Regular Expressions

Working with text data often requires the use of simple *regular expressions*. Regular expressions are a very powerful way of finding and/or manipulating text strings. Many books have been written about them, and good, concise information on regular expressions can be found on the web, for example at:

- https://www.debuggex.com/cheatsheet/regex/python provides a convenient cheat sheet for regular expressions in *Python*.
- http://www.regular-expressions.info gives a comprehensive description of regular expressions.

Let me give two examples how *pandas* can make use of regular expressions:

1. Reading in data from a file, separated by a combination of commas, semicolons, or white-spaces:
   `df = pd.read_csv(inFile, sep='[ ;,]+')`
   The square brackets indicate a *combination* ("[. . .]") of . . .
   The plus indicates *one or more* ("+")

2. Extracting columns with certain name-patterns from a *pandas* DataFrame. In the following example, I will extract all the columns starting with `Vel`:

```
In [18]: data = np.round(np.random.randn(100,7), 2)

In [19]: df = pd.DataFrame(data, columns=['Time',
         'PosX', 'PosY', 'PosZ', 'VelX', 'VelY', 'VelZ'])

In [20]: df.head()
Out[20]:
   Time  PosX  PosY  PosZ  VelX  VelY  VelZ
0  0.30 -0.13  1.42  0.45  0.42 -0.64 -0.86
1  0.17  1.36 -0.92 -1.81 -0.45 -1.00 -0.19
2 -3.03 -0.55  1.82  0.28  0.29  0.44  1.89
3 -1.06 -0.94 -0.95  0.77 -0.10 -1.58  1.50
4  0.74 -1.81  1.23  1.82  0.45 -0.16  0.12

In [21]: vel = df.filter(regex='Vel*')

In [22]: vel.head()
Out[22]:
   VelX  VelY  VelZ
0  0.42 -0.64 -0.86
1 -0.45 -1.00 -0.19
2  0.29  0.44  1.89
3 -0.10 -1.58  1.50
4  0.45 -0.16  0.12
```

3.2 Input from MS Excel

There are two approaches to reading a *Microsoft Excel* file in *pandas*: the function `read_excel`, and the class `ExcelFile`.[1]

- `read_excel` is for reading one file with file-specific arguments (i.e., identical data formats across sheets).
- `ExcelFile` is for reading one file with sheet-specific arguments (i.e., different data formats across sheets).

Choosing the approach is largely a question of code readability and execution speed.

[1]The following section has been taken from the *pandas* documentation.

The following commands show equivalent class and function approaches to read a single sheet:

```
# using the ExcelFile class
xls = pd.ExcelFile('path_to_file.xls')
data = xls.parse('Sheet1', index_col=None,
                    na_values=['NA'])

# using the read_excel function
data = pd.read_excel('path_to_file.xls', 'Sheet1',
        index_col=None, na_values=['NA'])
```

If this fails, give it a try with the *Python* package *xlrd*.

The following advanced script shows how to directly import data from an Excel file which is stored in a zipped archive on the web:

Listing 3.1 L3_2_readZip.py

```
'''Get data from MS-Excel files, which are stored zipped on
    the WWW. '''

# author: Thomas Haslwanter, date: Nov-2015

# Import standard packages
import pandas as pd

# additional packages
import io
import zipfile

# Python 2/3 use different packages for "urlopen"
import sys
if sys.version_info[0] == 3:
    from urllib.request import urlopen
else:
    from urllib import urlopen

def getDataDobson(url, inFile):
    '''Extract data from a zipped-archive on the web'''

    # get the zip-archive
    GLM_archive = urlopen(url).read()

    # make the archive available as a byte-stream
    zipdata = io.BytesIO()
    zipdata.write(GLM_archive)

    # extract the requested file from the archive, as a
        pandas XLS-file
    myzipfile = zipfile.ZipFile(zipdata)
    xlsfile = myzipfile.open(inFile)

    # read the xls-file into Python, using Pandas, and return
         the extracted data
```

```
34      xls = pd.ExcelFile(xlsfile)
35      df  = xls.parse('Sheet1', skiprows=2)
36
37      return df
38
39  if __name__ == '__main__':
40      # Select archive (on the web) and the file in the archive
41      url = 'http://cdn.crcpress.com/downloads/C9500/GLM_data.
            zip'
42      inFile = r'GLM_data/Table 2.8 Waist loss.xls'
43
44      df = getDataDobson(url, inFile)
45      print(df)
46
47      input('All done!')
```

3.3 Input from Other Formats

Matlab Support for data input from *Matlab* files is built into *scipy*, with the command `scipy.io.loadmat`.

Clipboard If you have data in your clipboard, you can import them directly with `pd.read_clipboard()`

Other file formats Also SQL databases and a number of additional formats are supported by *pandas*. The simplest way to access them is typing `pd.read_ + TAB`, which shows all currently available options for reading data into *pandas* DataFrames.

3.3.1 Matlab

The following commands return string, number, vector, and matrix variables from a *Matlab* file "data.mat", as well as the content of a structure with two entries (a vector and a string). The *Matlab* variables containing the scalar, string, vector, matrix, and structure are called *number, text, vector, matrix*, and *structure*, respectively.

```
from scipy.io import loadmat
data = loadmat('data.mat')

number = data['number'][0,0]
text   = data['text'][0]
vector = data['vector'][0]
matrix = data['matrix']
struct_values = data['structure'][0,0][0][0]
strunct_string = data['structure'][0,0][1][0]
```

Chapter 4
Display of Statistical Data

The dominant task of the human cortex is to extract visual information from the activity patters on the retina. Our visual system is therefore exceedingly good at detecting patterns in visualized data sets. As a result, one can almost always *see* what is happening before it can be demonstrated through a quantitative analysis of the data. Visual data displays are also helpful at finding extreme data values, which are often caused by mistakes in the execution of the paradigm or mistakes in the data acquisition.

This chapter shows a number of different ways to visualize statistical data sets.

4.1 Datatypes

The choice of appropriate statistical procedure depends on the data type. Data can be *categorical* or *numerical*. If the variables are numerical, we are led to a certain statistical strategy. In contrast, if the variables represent qualitative categorizations, then we follow a different path.

In addition, we distinguish between *univariate*, *bivariate*, and *multivariate* data. *Univariate data* are data of only one variable, e.g., the size of a person. *Bivariate data* have two parameters, for example, the x/y position in a plane, or the income as a function of age. *Multivariate data* have three or more variables, e.g., the position of a particle in space, etc.

4.1.1 Categorical

a) Boolean

Boolean data are data which can only have two possible values. For example,

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_4

- female/male
- smoker/nonsmoker
- True/False

b) Nominal

Many classifications require more than two categories. Such data are called *nominal data*. An example is *married/single/divorced*.

c) Ordinal

In contrast to nominal data, *ordinal data* are ordered and have a logical sequence, e.g., *very few/few/some/many/very many*.

4.1.2 Numerical

a) Numerical Continuous

Whenever possible, it is best to record the data in their original continuous format, and only with a meaningful number of decimal places. For example, it does not make sense to record the body size with more than 1 mm accuracy, as there are larger changes in body height between the size in the morning and the size in the evening, due to compression of the intervertebral disks.

b) Numerical Discrete

Some numerical data can only take on integer values. These data are called *numerical discrete*. For example *Number of children: 0 1 2 3 4 5 . . .*

4.2 Plotting in Python

Visualization is most important for numerical data, so I will focus on the following on this data type.

In practice the display of data can be a bit tricky, as there are so many options: graphical output can be displayed as a picture in an HTML-page, or in an interactive

graphics window; plots can force the attention of the user, or can automatically close after a few seconds, etc. This section will therefore focus on general aspects of plotting data; the next section will then present different types of plots, e.g., histograms, errorbars, 3D-plots, etc.

The *Python* core does not include any tools to generate plots. This functionality is added by other packages. By far the most common package for plotting is *matplotlib*. If you have installed *Python* with a scientific distribution like *WinPython* or *Anaconda*, it will already be included. *matplotlib* is intended to mimic the style of *Matlab*. As such, users can either generate plots in the *Matlab* style, or in the traditional *Python* style (see below).

matplotlib contains different modules and features:

matplotlib.pyplot This is the module that is commonly used to generate plots. It provides the interface to the plotting library in *matplotlib*, and is by convention imported in *Python* functions and modules with
`import matplotlib.pyplot as plt`.
pyplot handles lots of little details, such as creating figures and axes for the plot, so that the user can concentrate on the data analysis.

matplotlib.mlab Contains a number of functions that are commonly used in *Matlab*, such as `find`, `griddata`, etc.

"backends" *matplotlib* can produce output in many different formats, which are referred to as *backends*:

- In a *Jupyter Notebook*, or in a *Jupyter QtConsole*, the command `%matplotlib inline` directs output into the current browser window. (`%pylab inline` is a combination of loading pylab, and directing plot-output inline.)
- In the same environment, `%matplotlib qt4`[1] directs the output into a separate graphics window (Fig. 2.4). This allows panning and zooming the plot, and interactive selection of points on the plot by the user, with the command `plt.ginput`.
- With `plt.savefig` output can be directed to external files, e.g., in PDF, PNG, or JPG format.

pylab is a convenience module that bulk imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space. Although many examples use *pylab*, it is no longer recommended, and should only be used in *IPython*, to facilitate interactive development of code.

[1] Depending on your build of Python, this command may also be `%matplotlib tk`.

4.2.1 Functional and Object-Oriented Approaches to Plotting

Python plots can be generated in a *Matlab*-like style, or in an object oriented, more pythonic way. These styles are all perfectly valid, and each have their pros and cons. The only caveat is to avoid mixing the coding styles in your own code.

First, consider the frequently used *pyplot* style:

```
# Import the required packages,
# with their conventional names
import matplotlib.pyplot as plt
import numpy as np

# Generate the data
x = np.arange(0, 10, 0.2)
y = np.sin(x)

# Generate the plot
plt.plot(x, y)

# Display it on the screen
plt.show()
```

Note that the creation of the required figure and an axes is done automatically by *pyplot*.

Second, a more pythonic, object oriented style, which may be clearer when working with multiple figures and axes. Compared to the example above, only the section entitled *"# Generate the plot"* changes:

```
# Generate the plot
fig = plt.figure()         # Generate the figure
ax = fig.add_subplot(111) # Add an axis to that figure
ax.plot(x,y)               # Add a plot to that axis
```

For interactive data analysis, it is convenient to load the most common commands from *numpy* and *matplotlib.pyplot* into the current workspace. This is achieved with `pylab`, and leads to a *Matlab*-like coding style:

```
from pylab import *
x = arange(0, 10, 0.2)
y = sin(x)
plot(x, y)
show()
```

So, why all the extra typing as one moves away from the pure *Matlab*-style? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and to maintain. For example, the following lines of code produce a figure with two plots above each other, and clearly indicate which plot goes into which axis:

```
# Import the required packages
import matplotlib.pyplot as plt
import numpy as np

# Generate the data
x = np.arange(0, 10, 0.2)
y = np.sin(x)
z = np.cos(x)

# Generate the figure and the axes
fig, axs = plt.subplots(nrows=2, ncols=1)

# On the first axis, plot the sine and label the ordinate
axs[0].plot(x,y)
axs[0].set_ylabel('Sine')

# On the second axis, plot the cosine
axs[1].plot(x,z)
axs[1].set_ylabel('Cosine')

# Display the resulting plot
plt.show()
```

Code: "ISP_gettingStarted.py"[2] gives a short demonstration of *Python* for scientific data analysis.

4.2.2 Interactive Plots

matplotlib provides different ways to interact with the user. Unfortunately, this interaction is less intuitive than in *Matlab*. The examples below may help to bypass most of these problems. They show how to

- Exactly position figures on the screen.
- Pause between two plots, and proceed automatically after a few seconds.
- Proceed on a click or keyboard hit.
- Evaluate keyboard entries.

Listing 4.1 L4_1_interactivePlots.py

```
# Source: http://scipy-central.org/item/84/1/simple-
    interactive-matplotlib-plots
```

[2] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/04_DataDisplay/gettingStarted.

```
'''Interactive graphs with Matplotlib have haunted me. So
    here I have collected a number of
tricks that should make interactive use of plots simpler. The
     functions below show how to

- Position figures on the screen (e.g. top left half of
    display)
- Pause to display the plot, and proceed automatically after
    a few sec
- Proceed on a click, or a keyboard hit
- Evaluate keyboard inputs

author: Thomas Haslwanter
date:   Nov-2015
ver:    1.1
license: CC BY-SA 4.0

'''
# Import standard packages
import numpy as np
import matplotlib.pyplot as plt

# additional packages
try:
    import tkinter as tk
except ImportError:     #different capitalization in Python
    2.x
    import Tkinter as tk

t = np.arange(0,10,0.1)
c = np.cos(t)
s = np.sin(t)

def normalPlot():
    '''Just show a plot. The progam stops, and only continues
         when the plot is closed,
    either by hitting the "Window Close" button, or by typing
         "ALT+F4". '''

    plt.plot(t,s)
    plt.title('Normal plot: you have to close it to continue\
        nby clicking the "Window Close" button, or by hitting
        "ALT+F4"')
    plt.show()

def positionOnScreen():
    '''Position two plots on your screen. This uses the
        Tickle-backend, which I think is the default on all
        platforms.'''

    # Get the screen size
    root = tk.Tk()
    (screen_w, screen_h) = (root.winfo_screenwidth(), root.
        winfo_screenheight())
```

```
    root.destroy()

    def positionFigure(figure, geometry):
        '''Position one figure on a given location on the
            screen.
        This works for Tk and for Qt5 backends, but may fail
            on others.'''

        mgr = figure.canvas.manager
        (pos_x, pos_y, width, height) = geometry
        try:
            # positioning commands for Tk
            position = '{0}x{1}+{2}+{3}'.format(width, height
                , pos_x, pos_y)
            mgr.window.geometry(position)
        except TypeError:
            # positioning commands for Qt5
            mgr.window.setGeometry(pos_x, pos_y, width,
                height)

    # The program continues after the first plot
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(t,c)
    ax.set_title('Top Left: Close this one last')

    # Position the first graph in the top-left half of the
        screen
    topLeft = (0, 0, screen_w//2, screen_h//2)
    positionFigure(fig, topLeft)

    # Put another graph in the top right half
    fig2 = plt.figure()
    ax2 = fig2.add_subplot(111)
    ax2.plot(t,s)
    # I don't completely understand why this one has to be
        closed first. But otherwise the program gets unstable.
    ax2.set_title('Top Right: Close this one first (e.g. with
         ALT+F4)')

    topRight = (screen_w//2, 0, screen_w//2, screen_h//2)
    positionFigure(fig2, topRight)

    plt.show()

def showAndPause():
    '''Show a plot only for 2 seconds, and then proceed
        automatically'''
    plt.plot(t,s)
    plt.title('Don''t touch! I will proceed automatically.')

    plt.show(block=False)
    duration = 2    # [sec]
    plt.pause(duration)
```

```
        plt.close()

    def waitForInput():
        ''' This time, proceed with a click or by hitting any key
            '''
        plt.plot(t,c)
        plt.title('Click in that window, or hit any key to
            continue')

        plt.waitforbuttonpress()
        plt.close()

    def keySelection():
        '''Wait for user intput, and proceed depending on the key
            entered.
        This is a bit complex. But None of the versions I tried
            without
        key binding were completely stable.'''

        fig, ax = plt.subplots()
        fig.canvas.mpl_connect('key_press_event', on_key_event)

        # Disable default Matplotlib shortcut keys:
        keymaps = [param for param in plt.rcParams if param.find(
            'keymap') >= 0]
        for key in keymaps:
            plt.rcParams[key] = ''

        ax.plot(t,c)
        ax.set_title('First, enter a vowel:')
        plt.show()

    def on_key_event(event):
        '''Keyboard interaction'''

        #print('you pressed %s'%event.key)
        key = event.key

        # In Python 2.x, the key gets indicated as "alt+[key]"
        # Bypass this bug:
        if key.find('alt') == 0:
            key = key.split('+')[1]

        curAxis = plt.gca()
        if key in 'aeiou':
            curAxis.set_title('Well done!')
            plt.pause(1)
            plt.close()
        else:
            curAxis.set_title(key + ' is not a vowel: try again
                to find a vowel ....')
            plt.draw()

    if __name__ == '__main__':
```

```
normalPlot()
positionOnScreen()
showAndPause()
waitForInput()
keySelection()
```

4.3 Displaying Statistical Datasets

The first step in the data analysis should always be a visual inspection of the raw-data. Between 30 and 50 % of our cortex are involved in the processing of visual information, and as a result our brain is very good at recognizing patterns in visually represented data. The trick is choosing the most informative type of display for your data.

The easiest way to find and implement one of the many image types that *matplotlib* offers is to browse their gallery (http://matplotlib.org/gallery.html), and copy the corresponding *Python* code into your program.

For statistical data analysis, the *Python* package *seaborn* (http://www.stanford.edu/~mwaskom/software/seaborn/) builds on *matplotlib*, and aims to provide a concise, high-level interface for drawing statistical graphics that are both informative and attractive. Also *pandas* builds on *matplotlib* and offers many convenient ways to visualize DataFrames.

Other interesting plotting packages are:

- *plot.ly* is a package that is available for *Python*, *Matlab*, and *R*, and makes beautiful graphs (https://plot.ly).
- *bokeh* is a *Python* interactive visualization library that targets modern web browsers for presentation. *bokeh* can help anyone who would like to quickly and easily create interactive plots, dashboards, and data applications (http://bokeh.pydata.org/).
- *ggplot* for *Python*. It emulates the *R*-package *ggplot*, which is loved by many *R*-users (http://ggplot.yhathq.com/).

4.3.1 Univariate Data

The following examples all have the same format. Only the “Plot-command” line changes.

```
# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as stats
import seaborn as sns
```

```
# Generate the data
x = np.random.randn(500)

# Plot-command start ---------------------
plt.plot(x, '.')
# Plot-command end -----------------------

# Show plot
plt.show()
```

a) Scatter Plots

This is the simplest way to represent univariate data: just plot each individual data point. The corresponding plot-command is either

```
plt.plot(x, '.')
```

or, equivalently,

```
plt.scatter(np.arange(len(x), x))
```

Note: In cases where we only have few discrete values on the *x*-axis (e.g., *Group1, Group2, Group3*), it may be helpful to spread overlapping data points slightly (also referred to as *"adding jitter")* to show each data point. An example can be found at http://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.stripplot.html)

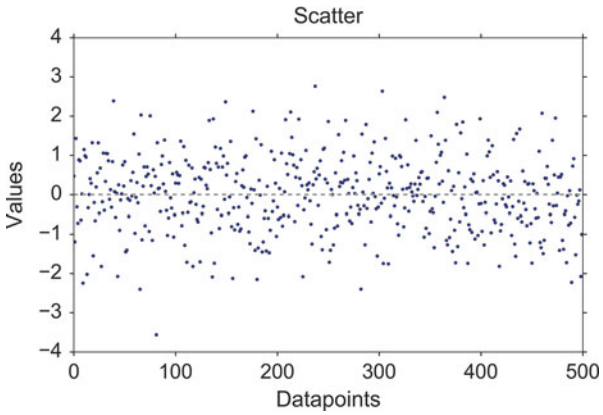


Fig. 4.1 Scatter plot

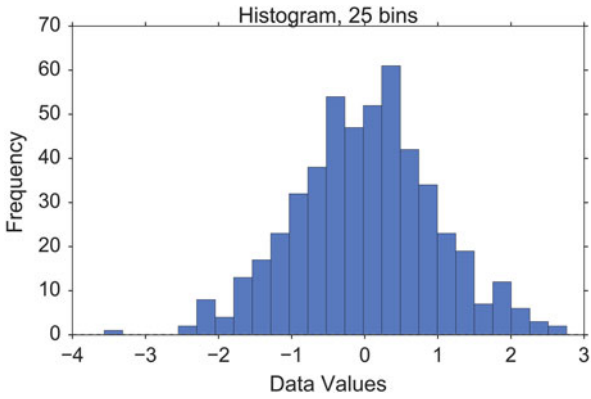


Fig. 4.2 Histogram

b) Histograms

Histograms provide a first good overview of the distribution of your data. If you divide by the overall number of data points, you get a *relative frequency histogram*; and if you just connect the top center points of each bin, you obtain a *relative frequency polygon*.

```
plt.hist(x, bins=25)
```

c) Kernel-Density-Estimation (KDE) Plots

Histograms have the disadvantage that they are discontinuous, and that their shape critically depends on the chosen bin-width. In order to obtain smooth *probability densities*, i.e., curves describing the likelihood of finding an event in any given interval, the technique of *Kernel Density Estimation* (KDE) can be used. Thereby a normal distribution is typically used for the kernel. The width of this kernel function determines the amount of smoothing. To see how this works, we compare the construction of histogram and kernel density estimators, using the following six data points:

```
x = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2].
```

For the histogram, first the horizontal axis is divided into subintervals or bins which cover the range of the data. In Fig. 4.3, left, we have six bins each of width 2. Whenever a data point falls inside this interval, we place a box of height 1/12. If more than one data point falls inside the same bin, we stack the boxes on top of each other.

For the kernel density estimate, we place a normal kernel with variance 2.25 (indicated by the red dashed lines in Fig. 4.3, right) on each of the data points x_i. The kernels are summed to make the kernel density estimate (solid blue curve). The smoothness of the kernel density estimate is evident. Compared to the discreteness

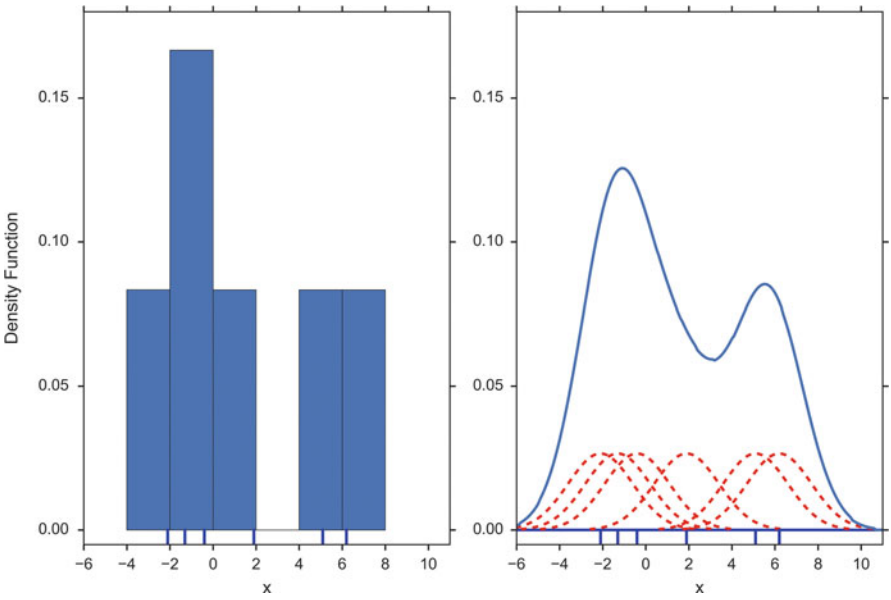


Fig. 4.3 Comparison of the histogram (*left*) and kernel density estimate (*right*) constructed using the same data. The six individual kernels are the *red dashed curves*, the kernel density estimate the *solid blue curve*. The data points are the rug plot on the horizontal axis

of the histogram, the kernel density estimates converge faster to the true underlying density for continuous random variables.

```
sns.kdeplot(x)
```

The bandwidth of the kernel is the parameter which determines how much we smooth out the contribution from each event. To illustrate its effect, we take a simulated random sample from the standard normal distribution, plotted as the blue spikes in the *rug plot* on the horizontal axis in Fig. 4.4, left. (A *rug plot* is a plot where every data entry is visualized by a small vertical tick.) The right plot shows the true density in blue. (A normal density with mean 0 and variance 1.) In comparison, the gray curve is undersmoothed since it contains too many spurious data artifacts arising from using a bandwidth $h = 0.1$ which is too small. The green dashed curve is oversmoothed since using the bandwidth $h = 1$ obscures much of the underlying structure. The red curve with a bandwidth of $h = 0.42$ is considered to be optimally smoothed since its density estimate is close to the true density.

It can be shown that under certain conditions the optimal choice for h is

$$h = \left(\frac{4\hat{\sigma}^5}{3n}\right)^{\frac{1}{5}} \approx 1.06\hat{\sigma}n^{-1/5}, \tag{4.1}$$

where $\hat{\sigma}$ is the standard deviation of the samples ("Silverman's rule of thumb").

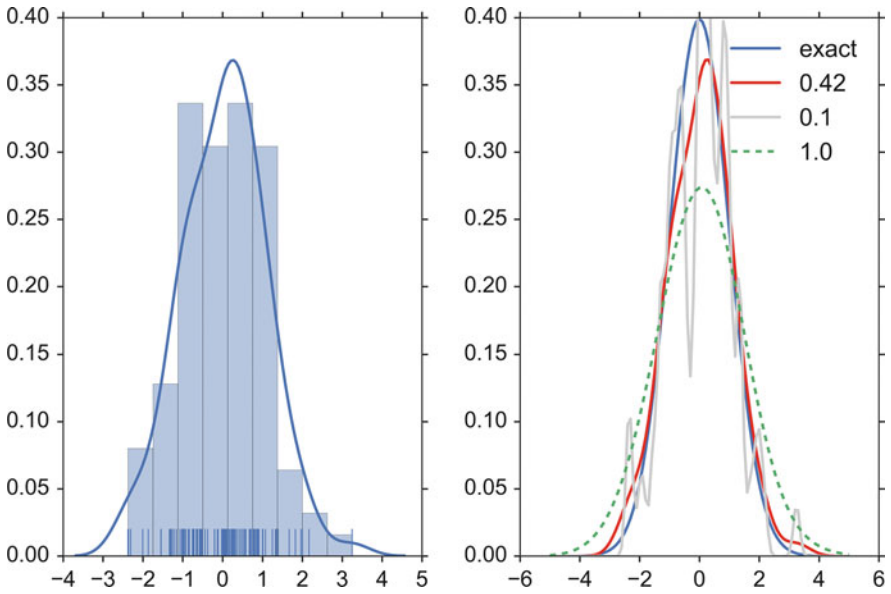


Fig. 4.4 *Left*: Rug plot, histogram, and Kernel density estimate (KDE) of a random sample of 100 points from a standard normal distribution. *Right*: True density distribution (*blue*), and KDE with different bandwidths. *Gray dashed*: KDE with $h = 0.1$; *red*: KDE with $h = 0.42$; *green dashed*: KDE with $h = 1.0$

d) Cumulative Frequencies

A *cumulative frequency* curve indicates the number (or percent) of data with less than a given value. This curve is very useful for statistical analysis, for example when we want to know the data range containing 95 % of all the values. Cumulative frequencies are also useful for comparing the distribution of values in two or more different groups of individuals.

When you use percentage points, the cumulative frequency presentation has the additional advantage that it is bounded: $0 \leq cumfreq(x) \leq 1$

```
plt.plot(stats.cumfreq(x,numbins)[0])
```

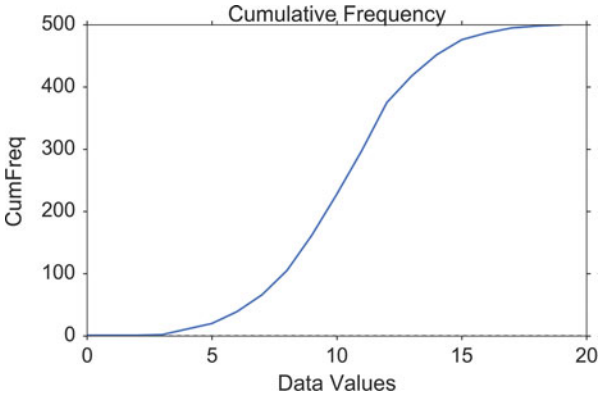


Fig. 4.5 Cumulative frequency function for a normal distribution

e) Error-Bars

Error-bars are a common way to show mean value and variability when comparing measurement values. Note that it always has to be stated explicitly if the error-bars correspond to the *standard deviation* or to the *standard error* of the data. Using *standard errors* has a nice feature: When error bars for the standard errors for two groups overlap, one can be sure the difference between the two means is not statistically significant ($p > 0.05$). However, the opposite is not always true!

```
index = np.arange(5)
y = index**2
errorBar = index/2  # just for demonstration
plt.errorbar(index,y, yerr=errorBar, fmt='o',
             capsize=5, capthick=3)
```

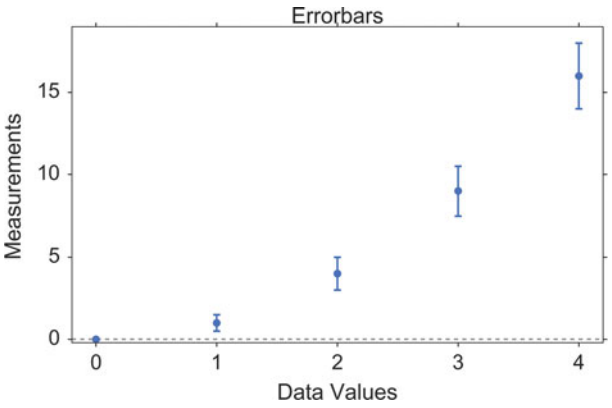


Fig. 4.6 Errorbars

f) Box Plots

Boxplots are frequently used in scientific publications to indicate values in two or more groups. The bottom and top of the box indicate the first quartile and third quartile, respectively, and the line inside the box shows the median. Care has to be taken with the whiskers, as different conventions exist for them. The most common form is that the lower whisker indicates the lowest value still within 1.5 * *inter-quartile-range* (IQR) of the lower quartile, and the upper whisker the highest value still within 1.5 * IQR of the upper quartile. Outliers (outside the whiskers) are plotted separately. Another convention is to have the whiskers indicate the full data range.

There are a number of tests to check for outliers. The method suggested by Tukey, for example, is to check for data which lie more than 1.5 * IQR above or below the first/third quartile (see Sect. 6.1.2).

```
plt.boxplot(x, sym='*')
```

Boxplots can be combined with KDE-plots to produce the so-called *violin plots*, where the vertical axis is the same as for the box-plot, but in addition a KDE-plot is shown symmetrically along the horizontal direction (Fig. 4.8).

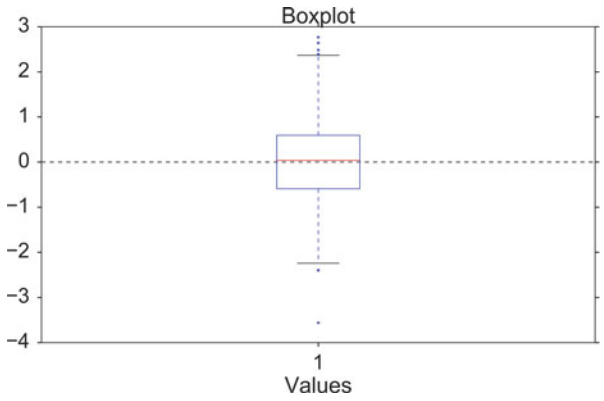


Fig. 4.7 Box plot

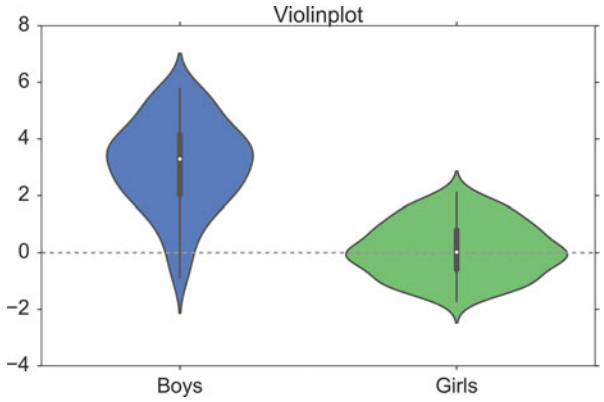


Fig. 4.8 Violinplot, produced with *seaborn*

```
# Generate the data
nd = stats.norm
data = nd.rvs(size=(100))

nd2 = stats.norm(loc = 3, scale = 1.5)
data2 = nd2.rvs(size=(100))

# Use pandas and the seaborn package
# for the violin plot
df = pd.DataFrame({'Girls':data, 'Boys':data2})
sns.violinplot(df)
```

g) Grouped Bar Charts

For some applications the plotting abilities of *pandas* can facilitate the generation of useful graphs, e.g., for grouped barplots (Figs. 4.9, 4.10, 4.11, and 4.12):

```
df = pd.DataFrame(np.random.rand(10, 4),
        columns=['a', 'b', 'c', 'd'])
df.plot(kind='bar', grid=False)
```

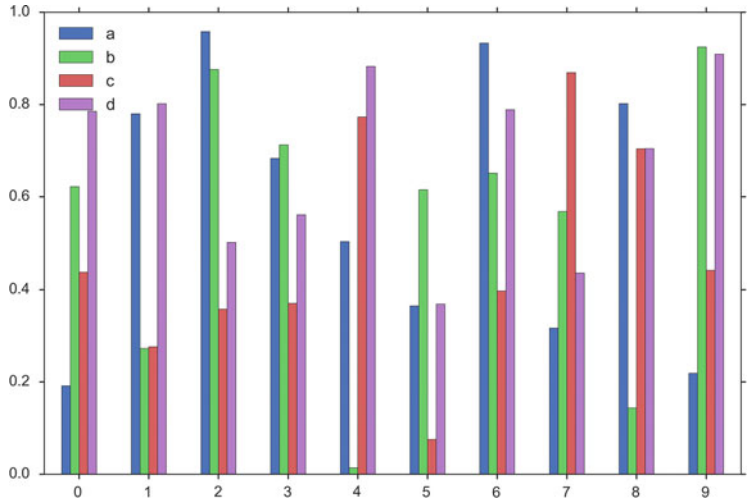


Fig. 4.9 Grouped barplot, produced with *pandas*

h) Pie Charts

Pie charts can be generated with a number of different options, e.g.

```
import seaborn as sns
import matplotlib.pyplot as plt

txtLabels = 'Cats', 'Dogs', 'Frogs', 'Others'
fractions = [45, 30, 15, 10]
offsets =(0, 0.05, 0, 0)

plt.pie(fractions, explode=offsets, labels=txtLabels,
         autopct='%1.1f%%', shadow=True, startangle=90,
         colors=sns.color_palette('muted') )
plt.axis('equal')
```

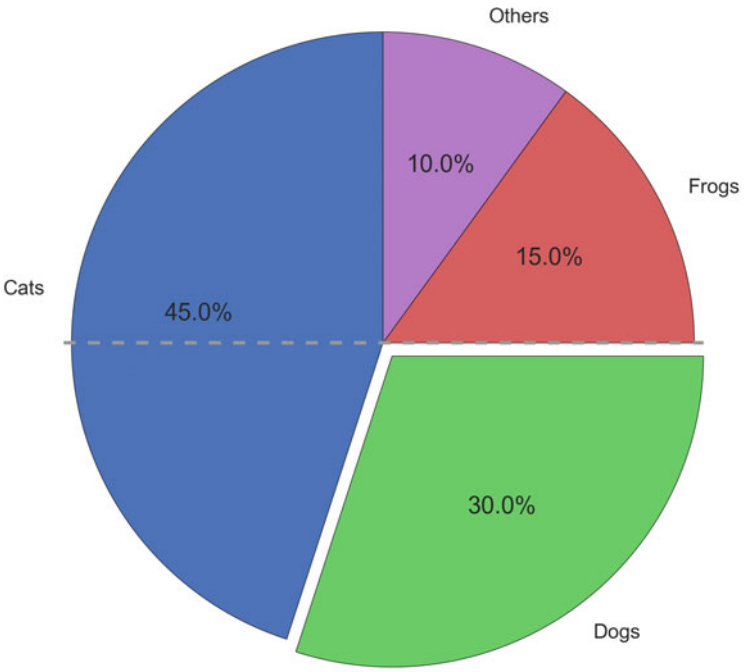


Fig. 4.10 "Sometimes it is raining cats and dogs"

i) Programs: Data Display

Code: "ISP_showPlots.py"[3] shows how the plots in this section have been generated.

4.3.2 Bivariate and Multivariate Plots

a) Bivariate Scatter Plots

Simple scatter plots are trivial. But *pandas* also makes fancy scatter plots easy:

```
df2 = pd.DataFrame(np.random.rand(50, 4),
                   columns=['a', 'b', 'c', 'd'])
df2.plot(kind='scatter', x='a', y='b', s=df['c']*300);
```

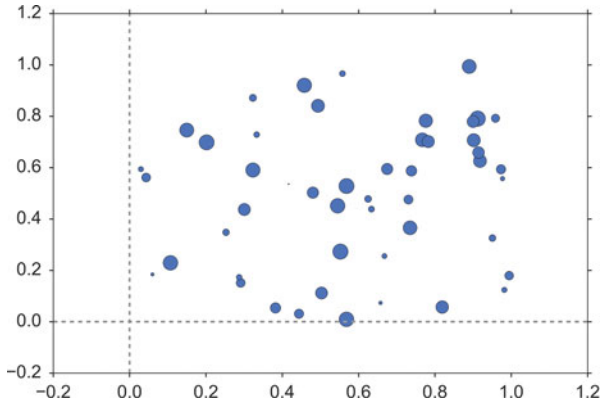


Fig. 4.11 Scatterplot, with scaled datapoints

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/04_DataDisplay/showPlots.

b) 3D Plots

3D plots in *matplotlib* are a bit awkward, because separate modules have to be imported, and axes for 3D plots have to be explicitly declared. However, once the axis is correctly defined, the rest is straightforward. Here are two examples:

```
# imports specific to the plots in this example
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# Twice as wide as it is tall.
fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot
# Note that the declaration "projection='3d'"
# is required for 3d plots!
ax = fig.add_subplot(1, 2, 1, projection='3d')

# Generate the grid
X = np.arange(-5, 5, 0.1)
Y = np.arange(-5, 5, 0.1)
X, Y = np.meshgrid(X, Y)

# Generate the surface data
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
        cmap=cm.GnBu, linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

outfile = '3dGraph.png'
plt.savefig(outfile, dpi=200)
print('Image saved to {0}'.format(outfile))
plt.show()
```

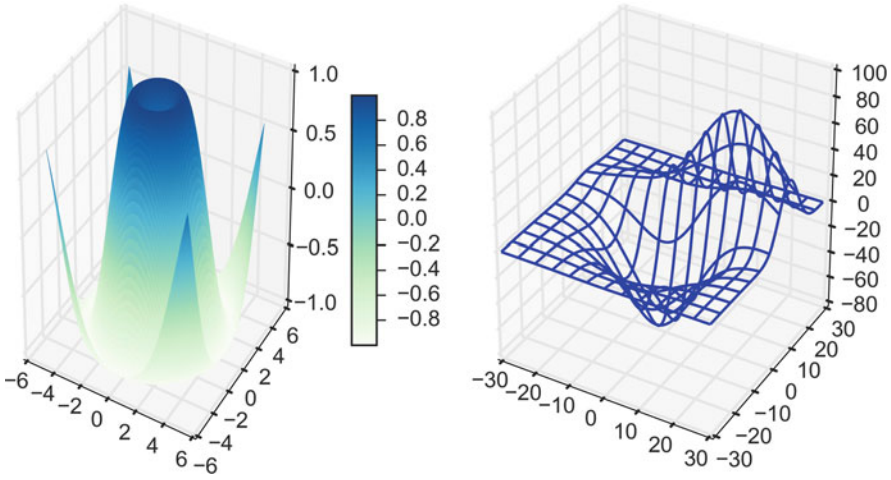


Fig. 4.12 Two types of 3D graphs. *(Left)* surface plot. *(Right)* wireframe plot

4.4 Exercises

4.1 Data Display

1. Read in the data from 'Data\amstat\babyboom.dat.txt'.
2. Inspect them visually, and give a numerical description of the data.
3. Are the data normally distributed?

Part II
Distributions and Hypothesis Tests

This part of the book moves the focus from *Python* to statistics.

The first chapter serves to define the statistical basics, like the concepts of *populations* and *samples*, and of *probability distributions*. It also includes a short overview of *study design*. The design of statistical studies is seriously underestimated by most beginning researchers: faulty study design produces garbage data, and the best analysis cannot remedy those problems ("Garbage in–garbage out"). However, if the study design is good, but the analysis faulty, the situation can be fixed with a new analysis, which typically takes much less time than an entirely new study.

The next chapter shows how to characterize the position and the variability of a distribution, and then uses the normal distribution to describe the most important *Python* methods common to all distribution functions. After that, the most important discrete and continuous distributions are presented.

The third chapter in this part first describes a typical workflow in the analysis of statistical data. Then the concept of *hypothesis tests* is explained, as well as the different types of errors, and common concepts like *sensitivity* and *specificity.*

The remaining chapters explain the most important hypothesis tests, for continuous variables and for categorical variables. A separate chapter is dedicated to survival analysis (which also encompasses the statistical characterization of material failures and machine breakdowns), as this question requires a somewhat different approach than the other hypothesis tests presented here. Each of these chapters also includes working *Python* sample code (including the required data) for each of the tests presented. This should make it easy to implement the tests for different data sets.

Chapter 5
Background

This chapter briefly introduces the main concepts underlying the statistical analysis of data. It defines discrete and continuous probability distributions, and then gives an overview of various types of study designs.

5.1 Populations and Samples

In the statistical analysis of data, we typically use data from a few selected *samples* to draw conclusions about the *population* from which these samples were taken. Correct *study design* should ensure that the sample data are representative of the population from which the samples were taken.

The main difference between a *population* and a *sample* has to do with how observations are assigned to the data set (see Fig. 5.1).

Population Includes all of the elements from a set of data.
Sample Consists of one or more observations from the population.

More than one sample can be derived from the same population.

When estimating a *parameter* of a *population*, e.g., the weight of male Europeans, we typically cannot measure all subjects. We have to limit ourselves to investigating a (hopefully representative) random *sample* taken from this group. Based on the *sample statistic*, i.e., the corresponding value calculated from the sample data, we use *statistical inference* to find out what we know about the corresponding parameter in the population.

Parameter Characteristic of a population, such as a mean or standard deviation. Often notated using Greek letters.
Statistic A measurable characteristic of a sample. Examples of statistics are:

- the mean value of the sample data

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_5

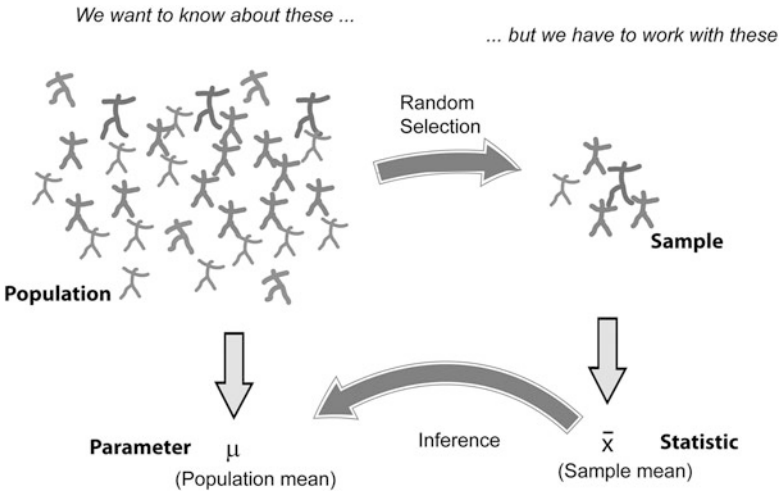


Fig. 5.1 With statistical inference, information from samples is used to estimate parameters from populations

Table 5.1 Comparison of sample statistics and population parameters

	Population parameter	Sample statistic
Mean	μ	$\bar{x}$
Standard deviation	σ	s

- the range of the sample data
- deviation of the data from the sample mean

Sampling distribution The probability distribution of a given statistic based on a random sample.

Statistical inference Enables you to make an educated guess about a population parameter based on a statistic computed from a sample randomly drawn from that population.

Examples of parameters and statistics are given in Table 5.1. Population parameters are often indicated using Greek letters, while sample statistics typically use standard letters.

5.2 Probability Distributions

The mathematical tools to describe the distribution of numerical data in populations and samples are *probability distributions*.

5.2.1 Discrete Distributions

A simple example of a discrete probability distribution is the game of throwing dice: for each of the numbers $i = 1, \ldots, 6$, the probability that at the throw of a die the side showing the number i faces upward, P_i, is

$$P_i = \frac{1}{6},\ i = 1 \ldots 6 \tag{5.1}$$

The set of all these probabilities $\{P_i\}$ makes up the *probability distribution* for rolling dice.

Note that the smallest possible value for P_i is 0. And since one of the faces of the die has to turn up at every throw of the dice, we have

$$\sum_{i=1}^{6} P_i = 1 \tag{5.2}$$

Generalizing this, we can say that a discrete probability distribution has the following properties

- $0 \leq P_i \leq 1\ \forall\, i \in \mathbf{N}$
- $\sum_{i=1}^{n} P_i = 1$

For a given discrete distribution, the P_i are called the *probability mass function* (*PMF*) of that distribution.

5.2.2 Continuous Distributions

Many measurements have an outcome that is not restricted to discrete integer values. For example, the weight of a person can be any positive number. In this case, the curve describing the probability for each value, i.e., the *probability distribution*, is a continuous function, the *probability density function* (*PDF*).

The PDF, or *density* of a continuous random variable, is a function that describes the relative likelihood of a random variable X to take on a given value x. In the mathematical fields of probability and statistics, a *random variate* x is a particular outcome of a *random variable* X: the random variates which are other outcomes of the same random variable might have different values.

Since the likelihood of finding any given value cannot be less than zero, and since the variable has to have some value, the PDF $p(x)$ has the following properties (Fig. 5.2):

- $p(x) \geq 0\ \forall\, x \in \mathbf{R}$
- $\int_{-\infty}^{\infty} p(x)\mathrm{d}x = 1$

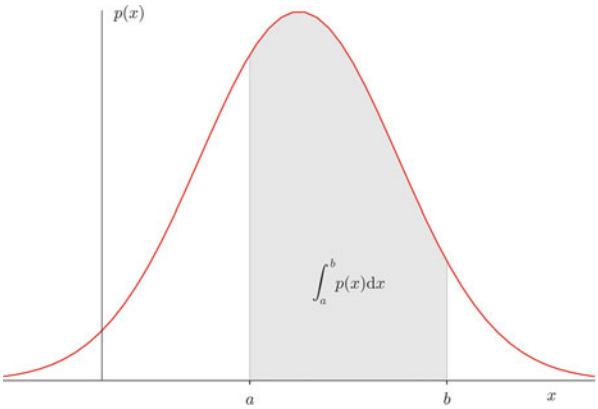


Fig. 5.2 $p(x)$ is the Probability Density Function of a value x. The integral over $p(x)$ between a and b represents the likelihood of finding the value of x in that range

5.2.3 Expected Value and Variance

a) Expected Value

The PDF also defines the *expected value* E[X] of a continuous distribution of X:

$$E[X] = \int_{-\infty}^{\infty} xf(x)\,\mathrm{d}x. \tag{5.3}$$

For discrete distributions, the integral over x is replaced by the sum over all possible values:

$$E[X] = \sum_i x_i P_i = 1. \tag{5.4}$$

where x_i represents all possible values that the measured variable can have.

The *expected value* is a function of the probability distribution of the observed value in our *population*. The *sample mean* of our *sample* is the observed mean value of our data. If the experiment has been designed correctly, the sample mean should converge to the expected value as more and more samples are included in the analysis.

b) Variance

The *variability* of the data is characterized by the *variance* of the data:

$$\begin{aligned} Var(X) &= E[(X - E[X])^2] \\ &= E[X^2] - (E[X])^2 \end{aligned} \tag{5.5}$$

5.3 Degrees of Freedom

The concept of degrees of freedom (DOF), which in mechanics appears to be crystal clear, is harder to grasp for statistical applications.

In mechanics, a particle which moves in a plane has "2 DOF": at each point in time, two parameters (the x/y-coordinates) define the location of the particle. If the particle moves about in space, it has "3 DOF": the $x/y/z$-coordinates.

In statistics, a group of n values has n DOF. If we look only at the shape of the distribution of the values, we can subtract from each value the sample mean. Then, the remaining data only have $n-1$ DOF. (This is clearest for $n=2$: if we know the mean value and the value of $sample_1$, then we can calculate the value of $sample_2$ by $val_2 = 2 * mean - val_1$.)

The case becomes more complex when we have many groups. For example, in Sect. 8.3.1, there is an example with 22 patients, divided into 3 groups. In the *analysis of variance* (*ANOVA*), the DOFs in this example are divided as follows:

- 1 DOF for the total mean value.
- 2 DOF for the mean value of each of the three groups (remember, if we know the mean values of two groups *and* the total mean, we can calculate the mean value of the third group).
- 19 DOF ($= 22-1-2$) are left for the residual deviations from the group means.

5.4 Study Design

The importance of a good study design has been demonstrated recently by an investigation showing the effect of the introduction of the *clinicaltrials.gov* registry (Kaplan and Irvin 2015): A 1997 US law mandated the creation of the registry, requiring researchers from 2000 onwards to record their trial methods and outcome measures *before* collecting data. Kaplan et al looked at studies evaluating drugs or dietary supplements for the treatment or prevention of cardiovascular disease. They found that before the introduction of *clinicaltrials.gov*, 57 % of the studies showed a positive outcome, while after the introduction, this number was reduced dramatically to only 8 %. In other words, without rigorous study design, there is a significant bias towards getting the result that you hope for.

5.4.1 Terminology

In the context of study design, various terminology can be found (Fig. 5.3):

- The controlled inputs are often called *factors* or *treatments*.
- The uncontrolled inputs are called *cofactors*, *nuisance factors*, or *confoundings*.

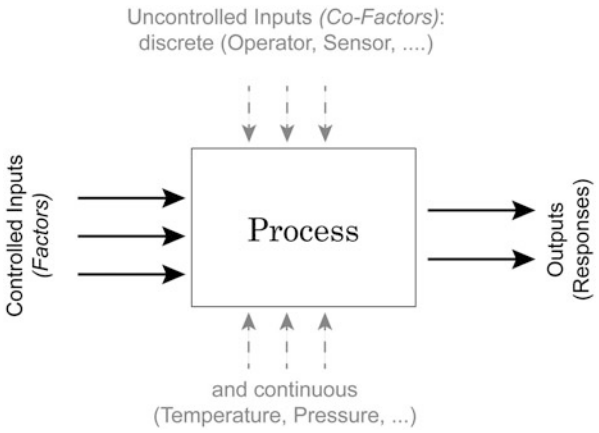


Fig. 5.3 Process schematic

The term *covariate* refers to a variable that is possibly predictive of the outcome being studied, and can be a factor or a cofactor.

When we try to model a process with two inputs and one output, we can formulate a mathematical model for example as

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_{12} X_1 X_2 + \epsilon \tag{5.6}$$

The terms with the single X (β_1, β_2) are called *main effects*, and the terms with multiple X (β_{12}) *interaction terms*. And since the β parameters enter the equation only linearly, this is referred to as a *general linear model*. The ϵ are called *residuals*, and are expected to be distributed approximately normally around zero if the model describes the data correctly.

5.4.2 Overview

The first step in the design of a study is the explicit clarification of the goal of the study. Do we want to

1. Compare two or more groups, or one group to a fixed value?
2. Screen the observed responses to identify factors/effects that are important?
3. Maximize or minimize a response (variability, distance to target, robustness)?
4. Develop a regression model to quantify the dependence of a response variable on the process input?

The first question leads to a *hypothesis test*. The second one is a *screening investigation*, where one must be watchful of artifacts if the factors in the model

are not completely independent. The third task is an *optimization problem*. And the last one brings us into the realm of *statistical modeling*.

Once we have determined *what* we want to do, we have to decide *how* we want to do this. Either *controlled experiments* or *observations* can be used to obtain the necessary data. In a controlled experiment we typically try to vary only a single parameter, and investigate the effects of that parameter on the output.

5.4.3 Types of Studies

a) Observational or Experimental

In an *observational study*, the researcher only collects information, but does not interact with the study population. In contrast, in an *experimental study* the researcher deliberately influences events (e.g., treats the patient with a new type of medication) and investigates the effects of these interventions.

b) Prospective or Retrospective

In a *prospective study*, the data are collected from the beginning of the study. In contrast, a *retrospective study* takes data acquired from previous events, e.g., routine tests done at a hospital.

c) Longitudinal or Cross-Sectional

In *longitudinal* investigations, the researcher collects information over a period of time, maybe multiple times from each patient. In contrast, in *cross-sectional* studies individuals are observed only once. For example, most surveys are cross-sectional, but experiments are usually longitudinal.

d) Case–Control and Cohort studies

In a *case–control study*, first the patients are treated, and then they are selected for inclusion in the study, based on certain criteria (e.g., whether they responded to a certain medication). In contrast, in a *cohort study*, subjects of interest are selected first, and then these subjects are studied over time, e.g., for their response to a treatment.

e) Randomized Controlled Trial

The gold standard for experimental scientific clinical trials, and the basis for the approval of new medications, is the *randomized controlled trial*. Here bias is avoided

by splitting the subjects to be tested into an *intervention group* and a *control group*. Group allocation is *random*.

In a designed experiment, there may be several conditions, called *factors*, that are controlled by the experimenter. By having the groups differ in only one aspect, the factor *treatment*, one should be able to detect the effect of the treatment on the patients.

Through randomization, confoundings should be balanced across the groups.

f) Crossover Studies

An alternative to randomization is the *crossover* design of studies. A crossover study is a longitudinal study in which subjects receive a sequence of different treatments. Every subject receives every treatment. (The subject "crosses over" from one treatment to the next.) To avoid causal effects, the sequence of the treatment allocation should be randomized.

For example, in an investigation that tests the effect of standing and sitting on the concentration of subjects, each subject performs both the execution of tasks while standing and the execution of tasks while sitting. The sequence of standing/sitting is randomized, to cancel out any sequence effects.

5.4.4 Design of Experiments

Block whatever you can; and randomize the rest!

I have mentioned above that we have *factors* (which we can control) and *nuisance factors*, which influence the results, but which we cannot control and/or manipulate. Assume, for example, that we have an experiment where the results depend on the person who performs the experiment (e.g., the nurse who tests the subject), and on the time of the day. In that case we can block the factor *nurse*, by having all tests performed by the same nurse. But it won't be possible to test all subjects at the same time. So we try to average out time effects, by *randomly* mixing the timing of the subjects. If, in contrast, we measure our patients in the morning and our healthy subjects in the afternoon, we will invariably bring some *bias* into our data.

a) Sample Selection

When selecting the subjects, one should pay attention to the following:

1. The samples should be representative of the group to be studied.
2. In comparative studies, groups must be similar with respect to known sources of variation (e.g., age, . . .).

3. **Important:** Make sure that your selection of samples (or subjects) sufficiently covers all of the parameters that you need! For example, if age is a nuisance factor, make sure you have enough young, middle aged, and elderly subjects.

Ad (1) For example, randomly selected subjects from patients at a hospital automatically bias the sample towards subjects with health problems.

Ad (3) For example, tests of the efficacy of a new rehabilitation therapy for stroke patients should *not* only include patients who have had a stroke: make sure that there are equal numbers of patients with mild, medium, and severe symptoms. Otherwise, one may end up with data which primarily include patients with few or no after-effects of the stroke. (This is one of the easiest mistakes to make, and cost me many months of work!)

Many surveys and studies fall short on these criteria (see section on *Bias* above). The field of "matching by propensity scores" (Rosenbaum and Rubin 1983) attempts to correct these problems.

b) Sample Size

Many studies also fail because the sample size is too small to observe an effect of the desired magnitude. In determining the sample size, one has to know

- What is the variance of the parameter under investigation?
- What is the magnitude of the expected effect, relative to the standard deviation of the parameter?

This is known as *power analysis*. It is especially important in behavioral research, where research plans are not approved without careful sample size calculations. (This topic will be discussed in more detail in Sect. 7.2.5.)

c) Bias

To explain the effects of selection bias on a statistical analysis, consider the 1936 presidential elections in the USA. The Republican Landon challenged the incumbent president, F.D. Roosevelt. *Literary Digest*, at the time one of the most respected magazines, asked ten million Americans who they would vote for. 2.4 million responded, and Literary Digest predicted Landon would win 57 % of the vote compared with 41 % for Roosevelt. However, the actual election results were 62 % for Roosevelt and 38 % for Landon. In other words, despite the huge sample size, the predictions were a whopping 19 % off!

What went wrong?

First, the sample was poorly chosen, and not representative of the American voter: the mailing lists for the survey were taken from telephone directories, club membership lists, and lists of magazine subscribers. Thus, they were strongly biased towards the American middle- and upper-class. And second, only about one-fourth of the people asked responded. And people who respond to surveys are different

from people who don't, the so-called *non-response bias*. This example shows that a large sample size alone does not guarantee a representative response. One has to watch out for selection bias and non-response bias.

In general, when selecting the subjects one tries to make them representative of the group to be studied; and one tries to conduct the experiments in a way representative of investigations by other researchers. However, it is very easy to get biased data.

Bias can have a number of sources:

- The selection of subjects.
- The structure of the experiment.
- The measurement device.
- The analysis of the data.

Care should be taken to avoid bias in the data as much as possible.

d) Randomization

This may be one of the most important aspects of experimental planning. Randomization is used to avoid bias as much as possible, and there are different ways to randomize an experiment. For randomization, *random number generators*, which are available with most computer languages, can be used. To minimize the chance of bias, the randomly allocated numbers should be presented to the experimenter as late as possible.

Depending on the experiment, there are various ways to randomize group assignment:

Simple Randomization

This procedure is robust against selection and accidental bias. The disadvantage is that the resulting group size can differ significantly.

For many types of data analysis it is important to have the same sample number in each group. To achieve this, other options are possible:

Block Randomization

This is used to keep the number of subjects in the different groups closely balanced at all times. For example, with two types of treatment, A and B, and a block-size of four, one can allocate the two treatments to the blocks of four subjects in the following sequences:

1. AABB
2. ABAB

3. ABBA
4. BBAA
5. BABA
6. BAAB

Based on this, one can use a random number generator to generate random integers between 1 and 6, and use the corresponding blocks to allocate the respective treatments. This will keep the number of subjects in each group always almost equal.

Minimization

A closely related, but not completely random way to allocate a treatment is *minimization*. Here one takes whichever treatment has the smallest number of subjects, and allocates this treatment with a probability greater than 0.5 to the next patient.

Assume, for example, that you are conducting a randomized controlled trial of a new medication, with a "placebo-group" and a "real medication group." Halfway through the trials you realize that your placebo-group already contains 60 subjects, while your medication-group only has 40. You can now solve this imbalance, by giving each remaining subject with 60 % probability (instead of the previously used 50 %) the medication instead of the placebo.

Stratified Randomization

Sometimes one may want to include a wider variety of subjects, with different characteristics. For example, one may choose to have younger as well as older subjects. In this case, one should try to keep the number of subjects within each *stratum* balanced. In order to do this, separate lists of random numbers should be kept for each group of subjects.

e) Blinding

Consciously or not, the experimenter can significantly influence the outcome of an experiment. For example, a young researcher with a new "brilliant" idea for a new treatment will be biased in the execution of the experiment, as well in the analysis of the data, to see the hypothesis confirmed. To avoid such subjective influence, ideally the experimenter as well as the subject should be blinded to the therapy. This is referred to as *double blinding*. When also the person who does the analysis does not know which group the subject has been allocated to, we speak about *triple blinding*.

f) Factorial Design

When each combination of factors is tested, we speak of *full factorial design* of the experiment.

In planning the analysis, one must distinguish between *within subject* comparisons, and *between subjects* comparisons. The former, *within subject comparisons*, allows to detect smaller differences with the same number of subjects than *between subject comparisons*.

5.4.5 Personal Advice

1. Be realistic about your task.
2. Plan in sufficient control/calibration experiments.
3. Take notes.
4. Store your data in a well-structured way.

1) Preliminary Investigations and Murphy's Law

Most investigations require more than one round of experiments and analyses. Theoretically, you state your hypothesis first, then do the experiments, and finally accept or reject the hypothesis. Done.

Most of my real investigations have been less straightforward, and often took two rounds of experiments. Typically, I start out with an idea. After making sure that nobody else has found the solution yet, I sit down, do the first rounds of measurements, and write the analysis programs required to analyze the data. Through this I find most of the things that can go wrong (they typically do, as stated by *Murphy's Law*: "Anything that can go wrong will go wrong."), and what I should have done differently in the first place. If the experiments are successful, that first round of investigation provides me with a "proof of principle" that my question is tractable; in addition, I also obtain data on the variability of typical responses. This allows me to obtain a reasonable estimate of the number of subjects/samples needed in order to accept or reject my hypothesis. By this time I also know whether my experimental setup is sufficient or whether a different or better setup is required. The second round of investigations is in most cases the real thing, and (if I am lucky) provides me with enough data to publish my findings.

2) Calibration Runs

Measurements of data can be influenced by numerous artifacts. To control these artifacts as much as possible, one should always start and end experimental recordings with something known. For example, during movement recordings, I try

to start out by recording a stationary point, and then move it 10 cm forward, left, and up. Having a recording with exact knowledge of what is happening not only helps to detect drift in the sensors and problems in the experimental setup. These recordings also help to verify the accuracy of the analysis programs.

3) Documentation

Make sure that you document all the factors that may influence your results, and everything that happens during the experiment:

- The date and time of the experiment.
- Information about the experimenters and the subjects.
- The exact paradigm that you have decided on.
- Anything noteworthy that happens during the experiment.

Be as brief as possible, but take down everything noteworthy that happens during the experiment. Be especially clear about the names of the recorded data-files, as they will be the first thing you need when you analyze the data later. Often you won't need all the details from your notes. But when you have outliers or unusual data points, these notes can be invaluable for the data analysis.

4) Data Storage

Try to have clear, intuitive, and practical naming conventions. For example, when you perform experiments with patients and with normals on different days, you could name these recordings "[p/n][yyyy/mm/dd]_[x].dat," e.g., *n20150329_a*. With this convention you have a natural grouping of your data, and the data are automatically sorted by their date.

Always store the raw data immediately, preferably in a separate directory. I prefer to make this directory read-only, so that I don't inadvertently delete valuable raw-data. You can in most cases easily redo an analysis run. But often, you will not be able to repeat an experiment.

5.4.6 Clinical Investigation Plan

To design a medical study properly, a clinical investigation plan is not only advisable, it is even required by ISO 14155-1:2003, for *Clinical investigations of medical devices for human subjects*. This norm specifies many aspects of clinical studies. It enforces the preparation of a *clinical investigation plan* (*CIP*), specifying

1. Type of study (e.g., double-blind, with or without control group, etc.).
2. Discussion of the control group and the allocation procedure.

3. Description of the paradigm.
4. Description and justification of primary endpoint of study.
5. Description and justification of chosen measurement variable.
6. Measurement devices and their calibration.
7. Inclusion criteria for subjects.
8. Exclusion criteria for subjects.
9. Point of inclusion ("When is a subject part of the study?")
10. Description of the measurement procedure.
11. Criteria and procedures for subjects who drop out.
12. Chosen sample number and level of significance, and their justification.
13. Procedure for documentation of negative effects or side-effects.
14. List of factors that can influence the measurement results or their interpretation.
15. Procedure for documentation, also for missing data.
16. Statistical analysis procedure.
17. The designation of a *monitor* for the investigation.
18. The designation of a *clinical investigator*.
19. Specifications for data handling.

Chapter 6
Distributions of One Variable

Distributions of one variable are called *univariate distributions*. They can be divided into *discrete distributions*, where the observations can only take on integer values (e.g., the number of children); and *continuous distributions*, where the observation variables are float values (e.g., the weight of a person).

The beginning of this chapter shows how to describe and work with statistical distributions. Then the most important discrete and continuous distributions are presented.

6.1 Characterizing a Distribution

6.1.1 Distribution Center

When we have a data sample from a distribution, we can characterize the center of the distribution with different parameters. Thereby the data can be evaluated in two ways:

1. By their value.
2. By their rank (i.e., their list-number when they are ordered according to magnitude).

a) Mean

By default, when we talk about the mean value we refer to the arithmetic mean $\bar{x}$:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{6.1}$$

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_6

Not surprisingly, the mean of an array x can be found with the command `np.mean`.

Real life data often include missing values, which are in many cases replaced by *nan*'s (*nan* stands for "Not-A-Number"). For statistics of arrays which include nan's, *numpy* has a number of functions starting with *nan*...

```
In [1]: import numpy as np

In [2]: x = np.arange(10)

In [3]: np.mean(x)
Out[3]: 4.5

In [4]: xWithNan = np.hstack( (x, np.nan) )     # append nan

In [5]: np.mean(xWithNan)
Out[5]: nan

In [6]: np.nanmean(xWithNaN)
Out[6]: 4.5
```

b) Median

The *median* is the value that comes half-way when the data are ranked in order. In contrast to the mean, the median is not affected by outlying data points. The median can be found with

```
In [7]: np.median(x)
Out[7]: 4.5
```

Note that when a distribution is symmetrical, as is the case here, the mean and the median value coincide.

c) Mode

The *mode* is the most frequently occurring value in a distribution.

The easiest way to find the mode value is the corresponding function in *scipy.stats*, which provides value and frequency of the mode value.

```
In [8]: from scipy import stats

In [9]: data = [1, 3, 4, 4, 7]

In [10]: stats.mode(data)
Out[10]: (array([4]), array([ 2.]))
```

d) Geometric Mean

In some situations the *geometric mean* can be useful to describe the location of a distribution. It can be calculated via the arithmetic mean of the log of the values.

$$mean_{geometric} = \left(\prod_{i=1}^{N} x_i\right)^{1/N} = exp\left(\frac{\sum_i ln(x_i)}{n}\right) \tag{6.2}$$

Again, the corresponding function is located in *scipy.stats*:

```
In [11]: x = np.arange(1,101)

In [12]: stats.gmean(x)
Out[12]: 37.992689344834304
```

Note that the input numbers for the geometric mean have to be positive.

6.1.2 Quantifying Variability

a) Range

The *range* is simply the difference between the highest and the lowest data value, and can be found with

```
range = np.ptp(x)
```

ptp stands for "peak-to-peak." The only thing that should be watched is outliers, i.e., data points with a value much higher or lower than the rest of the data. Often, such points are caused by errors in the selection of the sample or in the measurement procedure.

There are a number of tests to check for outliers. One of them is to check for data which lie more than 1.5*inter-quartile-range (IQR) above or below the first/third quartile ("quartiles" are defined in the next section).

b) Percentiles

The simplest way to understand *centiles*, also called *percentiles*, is to first define the *Cumulative Distribution Function* (*CDF*):

$$CDF(x) = \int_{-\infty}^{x} PDF(x)dx \tag{6.3}$$

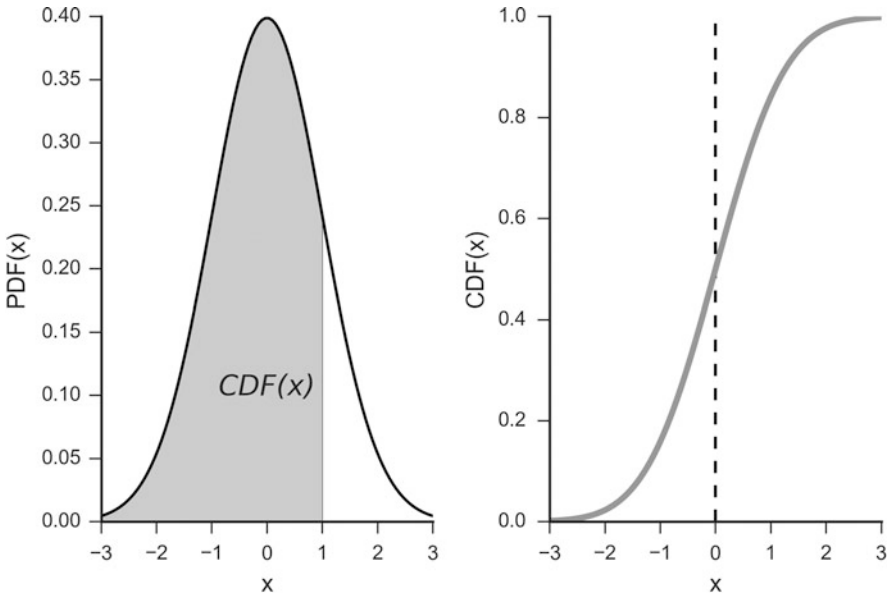


Fig. 6.1 Probability density function (*left*) and cumulative distribution function (*right*) of a normal distribution

The CDF is the integral of the PDF from minus infinity up to the given value (see Fig. 6.1), and thus specifies the percentage of the data that lie below this value. Knowing the CDF simplifies the calculation of how likely it is to find a value x within the range a to b (Fig. 5.2): The probability to find a value between a and b is given by the integral over the PDF in that range (see Fig. 5.2), and can be found by the difference of the corresponding CDF-values:

$$\mathbf{P}(a \leq X \leq b) = \int_a^b PDF(x)dx = CDF(b) - CDF(a) \tag{6.4}$$

For discrete distributions, the integral has to be replaced by a sum.

Coming back to *percentiles*: those are just the inverse of the CDF, and give the value below which a given percentage of the data values occur (see Fig. 6.5, lower left plot). While the expression "percentiles" does not come up very often, one will frequently encounter specific centiles:

- To find the range which includes 95 % of the data, one has to find the 2.5th and the 97.5th percentile of the sample distribution.
- The 50th percentile is the *median*.
- Also important are the *quartiles*, i.e., the 25th and the 75th percentile. The difference between them is called the *inter-quartile range* (*IQR*).

Median, upper, and lower quartile are used for the data display in box plots (Fig. 4.7).

c) Standard Deviation and Variance

Figure 5.1 shows a sketch of how a sample statistic relates to the corresponding population parameter. Applying this concept to the *variance* of a data set, we distinguish between the *sample variance*, i.e., the variance in the data sampled, and the *population variance*, i.e., the variance of the full population. The maximum likelihood estimator of the sample variance is given by

$$var = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n} \tag{6.5}$$

However, Eq. 6.5 systematically underestimates the population variance, and is therefore referred to as a "biased estimator" of the population variance. In other words, if you take a population with a given *population standard deviation*, and one thousand times selects n random samples from that population and calculate the standard deviation for each of these samples, then the mean of those one thousand *sample standard deviations* will be below the *population standard deviation*.

Figure 6.2 tries to motivate why the sample variance systematically underestimates the population variance: the sample mean is always chosen such that the variance of the given sample data is minimized, and thereby underestimates the variance of the population.

It can be shown that the best unbiased estimator for the population variance is given by

$$var = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} \tag{6.6}$$

Equation 6.6 is referred to as *sample variance*.

The *standard deviation* is the square root of the variance, and the *sample standard deviation* the square root of the sample variance:

$$s = \sqrt{var} \tag{6.7}$$

As indicated in Table 5.1, in statistics it is common to denote the population standard deviation with σ, and the sample standard deviation with s.

Watch out: in contrast to other languages like *Matlab* or *R*, *numpy* by default calculates the variance for "n." To obtain the sample variance one has to set

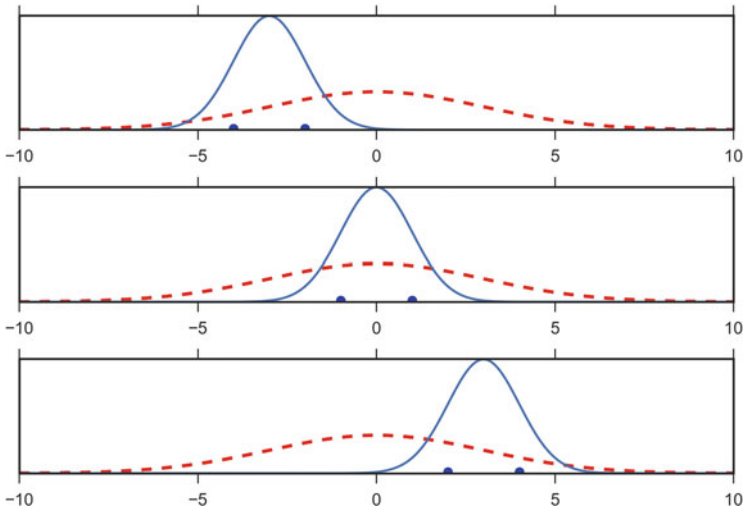


Fig. 6.2 Gaussian distributions fitted to selections of data from the underlying distribution: While the average mean of a number of samples converges to the real mean, the sample standard deviation underestimates the standard deviation from the distribution

"ddof=1":

```
In [1]: data = np.arange(7,14)

In [2]: np.std(data, ddof=0)
Out[2]: 2.0

In [3]: np.std(data, ddof=1)
Out[3]: 2.16025
```

Note: In *pandas*, the default for the calculation of the standard deviation is set to *ddof=1*.

d) Standard Error

The *standard error* is the estimate of the standard deviation of a coefficient. For example, in Fig. 6.3, we have 100 data points from a normal distribution about 5. The more data points we have to estimate the mean value, the better our estimate of the mean becomes.

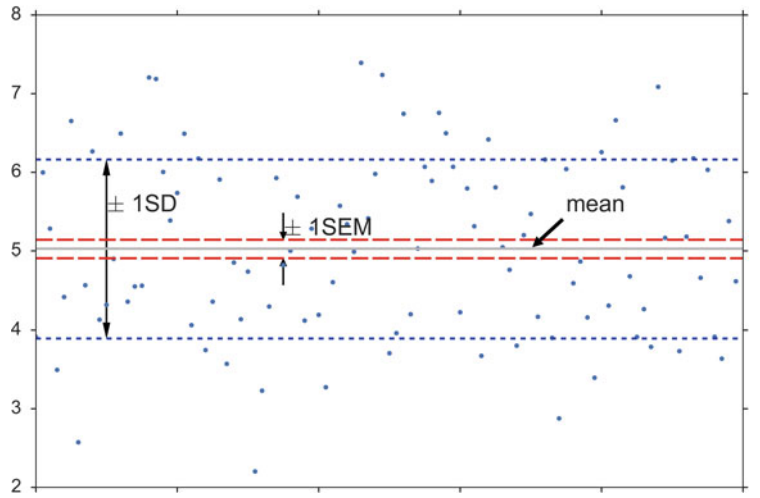


Fig. 6.3 One hundred random data points, from a normal distribution about 5. The sample mean (*solid line*) is very close to the real mean. The standard deviation of the mean (*long dashed line*), or standard error of the mean (SEM), is ten times smaller than the standard deviation of the samples (*short dashed line*)

For normally distributed data, the *sample standard error of the mean* (SE or SEM) is

$$SEM = \frac{s}{\sqrt{n}} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \cdot \frac{1}{\sqrt{n}} \tag{6.8}$$

So with 100 data points the standard deviation of our estimate, i.e., the standard error of the mean, is ten times smaller than the sample standard deviation.

e) Confidence Intervals

In the statistical analysis of data it is common to state the confidence interval of an estimated parameter. The $\alpha\%$ *confidence interval* (*CI*) reports the range that contains the true value for the parameter with a likelihood of $\alpha\%$.

If the sampling distribution is symmetrical and unimodal (i.e., decaying smoothly on both sides of the maximum), it will often be possible to approximate the confidence interval by

$$ci = mean \pm std * N_{PPF}\left(\frac{1-\alpha}{2}\right) \tag{6.9}$$

where *std* is the standard deviation, and N_{PPF} the *percentile point function* (*PPF*) for the standard normal distribution (see Fig.6.5). For the 95 % two-sided confidence intervals, for example, you have to calculate the *PPF(0.025)* of the standard normal distribution to get the lower and upper limit of the confidence interval. For a *Python* implementation for a normal distribution, see for example the code-sample on p. 106.

Notes

- To calculate the confidence interval for the mean value, the standard deviation has to be replaced by the standard error.
- If the distribution is skewed, Eq. 6.9 is *not* appropriate and does not provide the correct confidence intervals!

6.1.3 Parameters Describing the Form of a Distribution

In `scipy.stats`, continuous distribution functions are characterized by their *location* and their *scale*. To give two examples: for the normal distribution, (*location*/*shape*) are given by (*mean*/*standard deviation*) of the distribution; and for the uniform distribution, they are given by the (*start*/*end-start*) of the range where the distribution is different from zero.

a) Location

A *location parameter* x_0 determines the location or shift of a distribution:

$$f_{x0}(x) = f(x - x_0)$$

Examples of location parameters include the mean, the median, and the mode.

b) Scale

The *scale parameter* describes the width of a probability distribution. If the scale parameter s is large, then the distribution will be more spread out; if s is small then it will be more concentrated. If the probability density exists for all values of s, then the density (as a function of the scale parameter only) satisfies

$$f_s(x) = f(x/s)/s$$

where f is the density of a standardized version of the density.

c) Shape Parameters

It is customary to refer to all of the parameters beyond location and scale as *shape parameters*. Thankfully, almost all of the distributions that we use in statistics have only one or two parameters. It follows that the *skewness* and *kurtosis* of these distribution are constants.

Skewness

Distributions are *skewed* if they depart from symmetry (Fig. 6.4, left). For example, for measurements that cannot be negative, which is usually the case, we can infer that the data have a skewed distribution if the standard deviation is more than half the mean. Such an asymmetry is referred to as *positive skewness*. The opposite, negative skewness, is rare.

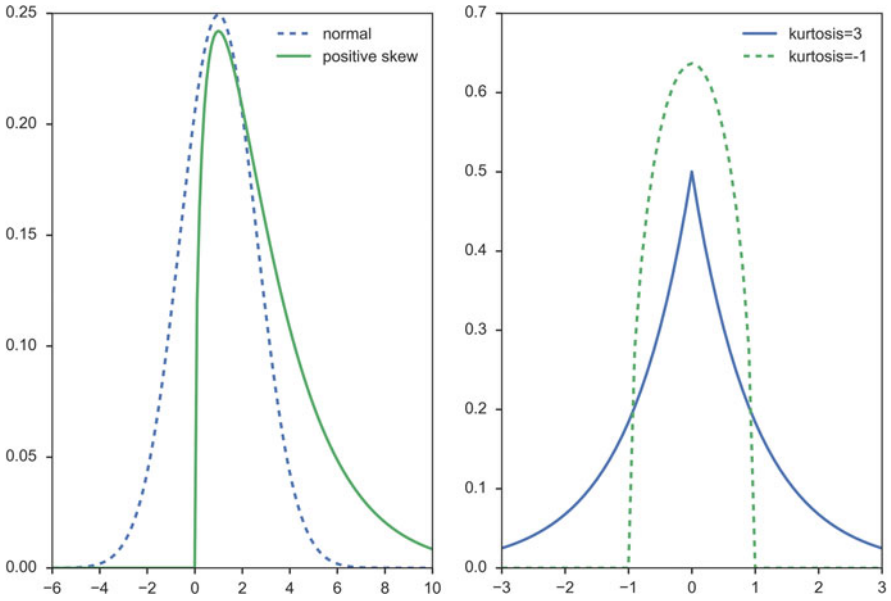


Fig. 6.4 (*Left*) Normal distribution, and distribution with positive skewness. (*Right*) The (leptokurtic) Laplace distribution has an excess kurtosis of 3, and the (platykurtic) Wigner semicircle distribution an excess kurtosis of −1

Kurtosis

Kurtosis is a measure of the "peakedness" of the probability distribution (Fig. 6.4, right). Since the normal distribution has a kurtosis of 3, the *excess kurtosis* = *kurtosis-3* is *0* for the normal distribution. Distributions with negative or positive excess kurtosis are called *platykurtic distributions* or *leptokurtic distributions*, respectively.

6.1.4 Important Presentations of Probability Densities

Figure 6.5 shows a number of functions that are equivalent to the PDF, but each represents a different aspect of the probability distribution. I will give examples which demonstrate each aspect for a normal distribution describing the size of male subjects.

- *Probability density function* (*PDF*): Note that to obtain the probability for the variable appearing in a certain interval, you have to integrate the PDF over that range.
  Example: What is the chance that a man is between 160 and 165 cm tall?

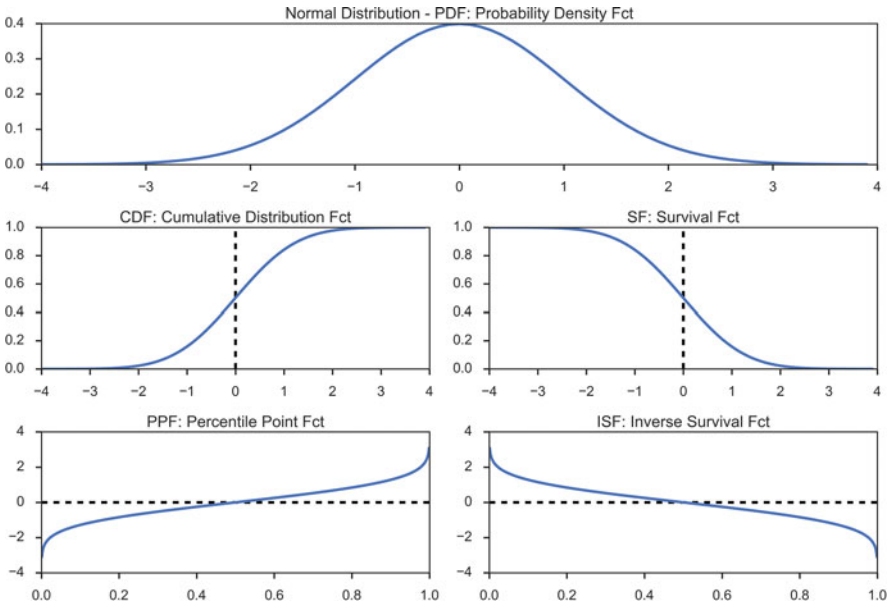


Fig. 6.5 Utility functions for continuous distributions, here for the standard normal distribution

- *Cumulative distribution function* (*CDF*): gives the probability of obtaining a value smaller than the given value.
  Example: What is the chance that a man is less than 165 cm tall?
- *Survival Function* (*SF*) $= 1 -$ CDF: gives the probability of obtaining a value larger than the given value. It can also be interpreted as the proportion of data "surviving" above a certain value.
  Example: What is the chance that a man is larger than 165 cm?
- *Percentile Point Function* (*PPF*): the inverse of the CDF. The PPF answers the question "Given a certain probability, what is the corresponding input value for the CDF?"
  Example: Given that I am looking for a man who is smaller than 95 % of all other men, what size does the subject have to be?
- *Inverse Survival Function* (*ISF*): the name says it all.
  Example: Given that I am looking for a man who is larger than 95 % of all other men, what size does the subject have to be?
- Random Variate Sample (*RVS*): random variates from a given distribution. (A *variable* is the general type, a *variate* is a specific number.)

Note: In *Python*, the most elegant way of working with distribution functions is a two-step procedure:

- In the first step, you create the distribution (e.g., `nd = stats.norm()`). Note that this is a distribution (in *Python* parlance a "frozen distribution"), not a function yet!
- In the second step, you decide which function you want to use from this distribution, and calculate the function value for the desired x-input (e.g., `y = nd.cdf(x)`).

```
import numpy as np
from scipy import stats

myDF = stats.norm(5,3)     # Create the frozen distribution

x = np.linspace(-5, 15, 101)
y = myDF.cdf(x)        # Calculate the corresponding CDF
```

6.2 Discrete Distributions

Two discrete distributions are frequently encountered: the *binomial distribution* and the *Poisson distribution*.

The big difference between those two functions: applications of the binomial distribution have an inherent upper limit (e.g., when you throw dice five times, each side can come up a maximum of five times); in contrast, the Poisson distribution does not have an inherent upper limit (e.g., how many people you know).

6.2.1 Bernoulli Distribution

The simplest case of a univariate distribution, and also the basis of the binomial distribution, is the *Bernoulli distribution* which has only two states, e.g., the simple coin flipping test. If we flip a coin (and the coin is not rigged), the chance that "heads" comes up is $p_{heads} = 0.5$. And since it has to be *heads* or *tails*, we must have

$$p_{heads} + p_{tails} = 1 \tag{6.10}$$

so the chance for "tails" is $p_{tails} = 1 - p_{heads}$.

We see that one parameter, $p = p_{heads}$, completely determines everything, and we can fix the distribution with the commands

```
In [1]: from scipy import stats
In [2]: p = 0.5
In [3]: bernoulliDist = stats.bernoulli(p)
```

In *Python* this is called a "frozen distribution function", and it allows us to calculate everything we want for this distribution. For example, the probability if head comes up zero or one times is given by the *probability mass function* (*PMF*)

```
In [4]: p_tails = bernoulliDist.pmf(0)
In [5]: p_heads = bernoulliDist.pmf(1)
```

And we can simulate 10 Bernoulli trials with

```
In [6]: trials = bernoulliDist.rvs(10)

In [7]: trials
Out[7]: array([0, 0, 0, 1, 0, 0, 0, 1, 1, 0])
```

In `In[6]`, *rvs* stands for *random variates*.

6.2.2 Binomial Distribution

If we flip a coin multiple times, and ask "How often did heads come up?" we have the *binomial distribution* (Fig. 6.6). In general, the binomial distribution is associated with the question "Out of a given (fixed) number of trials, how many will succeed?" Some example questions that are modeled with a binomial distribution are:

- Out of ten tosses, how many times will this coin land heads?
- From the children born in a given hospital on a given day, how many of them will be girls?
- How many students in a given classroom will have green eyes?
- How many mosquitoes, out of a swarm, will die when sprayed with insecticide?

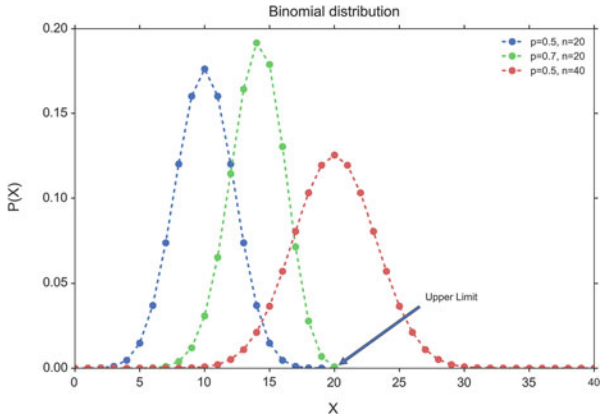


Fig. 6.6 Binomial distribution. Note that legal values exist only for integer x. The *dotted lines* in between only facilitate the grouping of the values to individual distribution parameters

We conduct n repeated experiments where the probability of success is given by the parameter p and add up the number of successes. This number of successes is represented by the random variable X. The value of X is then between 0 and n.

When a random variable X has a binomial distribution with parameters p and n we write it as $X \in B(n, p)$ and the probability mass function at $X = k$ is given by the equation:

$$P[X = k] = \begin{cases} \binom{n}{k} p^k (1-p)^{n-k} & 0 \le k \le n \\ 0 & \text{otherwise} \end{cases} \quad 0 \le p \le 1, \quad n \in \mathbf{N} \tag{6.11}$$

$$P[X = k] = p^k (1-p)^{n-k} \quad 0 \le p \le 1, \quad n \in \mathbf{N} \tag{6.12}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

In *Python*, the procedure is the same as above for the Bernoulli distribution, with one additional parameter, the number of coin tosses. First we generate the frozen distribution function, for example for four coin tosses:

```
In [1]: from scipy import stats
In [2]: import numpy as np

In [3]: (p, num) = (0.5, 4)
In [4]: binomDist = stats.binom(num, p)
```

and then we can calculate, e.g., the probabilities how often heads come up during those four tosses, given by the PMF for the values zero to four:

```
In [5]: binomDist.pmf(np.arange(5))
Out[5]: array([ 0.0625,  0.25  ,  0.375 ,  0.25  ,  0.0625])
```

For example, the chance that heads never comes up is about 6 %, the chance that it comes up exactly once is 25 %, etc.

Also note that the sum of all probabilities has to add up exactly to one:

$$p_0 + p_1 + \ldots + p_{n-1} = \sum_{i=0}^{n-1} p_i = 1 \tag{6.13}$$

b) Example: Binomial Test

Suppose we have a board game that depends on the roll of a die and attaches special importance to rolling a 6. In a particular game, the die is rolled 235 times, and 6 comes up 51 times. If the die is fair, we would expect 6 to come up 235/6 = 39.17 times. Is the proportion of 6's significantly higher than would be expected by chance, on the null hypothesis of a fair die?

To find an answer to this question using the *Binomial Test*, we consult the binomial distribution with $n = 235$ and $p = 1/6$, to determine the probability of finding exactly 51 sixes in a sample of 235 if the true probability of rolling a 6 on each trial is 1/6. We then find the probability of finding exactly 52, exactly 53, and so on up to 235, and add all these probabilities together. In this way, we calculate the probability of obtaining the observed result (51 sixes) or a more extreme result (> 51 sixes) assuming that the die is fair. In this example, the result is 0.0265, which indicates that observing 51 sixes is unlikely (not significant at the 5 % level) to come from a die that is not loaded to give many sixes (one-tailed test).

Clearly a die could roll too few sixes as easily as too many and we would be just as suspicious, so we should use the two-tailed test which (for example) splits the 5 % probability across the two tails. Note that to do this we cannot simply double the one-tailed p-value unless the probability of the event is 1/2. This is because the binomial distribution becomes asymmetric as that probability deviates from 1/2. "scipy.stats" therefore provides for the two-sided test the function "binom_test". (See also the explanation of one- and two-tailed *t*-tests, p. 141.)

Code: "ISP_binomial.py"[1]: Example of a one- and two-sided binomial test.

Table 6.1 Properties of discrete distributions

	Mean	Variance
Binomial	$n \cdot p$	$np(1-p)$
Poisson	λ	λ

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/06_Distributions/binomialTest.

6.2.3 *Poisson Distribution*

Any French speaker will notice that "Poisson" means "fish," but really there's nothing fishy about this distribution. It's actually pretty straightforward. The name comes from the mathematician Siméon-Denis Poisson (1781–1840).

The Poisson distribution is very similar to the binomial distribution. We are examining the number of times an event happens. The difference is subtle. Whereas the binomial distribution looks at how many times we register a success over a fixed total number of trials, the Poisson distribution measures how many times a discrete event occurs, over a period of continuous space or time. There is no "total" value n, and the Poisson distribution is defined by a single parameter.

The following questions can be answered with the Poisson distribution:

- How many pennies will I encounter on my walk home?
- How many children will be delivered at the hospital today?
- How many products will I sell after airing a new television commercial?
- How many mosquito bites did you get today after having sprayed with insecticide?
- How many defects will there be per 100 m of rope sold?

What's a little different about this distribution is that the random variable X which counts the number of events can take on any nonnegative integer value. In other words, I could walk home and find no pennies on the street. I could also find one penny. It's also possible (although unlikely, short of an armored-car exploding nearby) that I would find 10 or 100 or 10,000 pennies.

Instead of having a parameter p that represents a component probability as in the binomial distribution, this time we have the parameter "lambda" or λ which represents the "average or expected" number of events to happen within our experiment (Fig. 6.7). The probability mass function of the Poisson distribution is

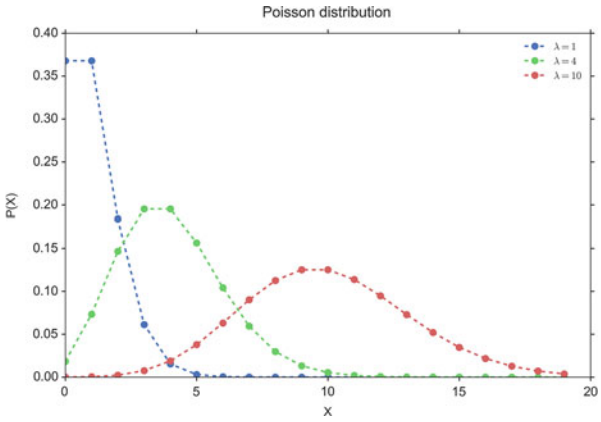


Fig. 6.7 Poisson distribution. Again note that legal values exist only for integer x. The *dotted lines* in between only facilitate the grouping of the values to individual distribution parameters

given by

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!} \tag{6.14}$$

.

Code: "ISP_distDiscrete.py"[2] shows different discrete distribution functions.

6.3 Normal Distribution

The Normal distribution or Gaussian distribution is by far the most important of all the distribution functions (Fig. 6.8). This is due to the fact that the mean values of *all* distribution functions approximate a normal distribution for large enough sample numbers (see Sect. 6.3.2). Mathematically, the normal distribution is characterized by a mean value μ, and a standard deviation σ:

$$f_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{6.15}$$

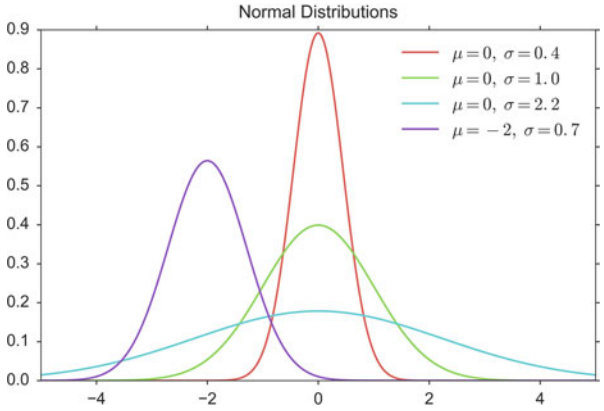


Fig. 6.8 Normal distributions, with different parameters for μ and σ

[2] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/06_Distributions/distDiscrete.

where $-\infty < x < \infty$, and $f_{\mu,\sigma}$ is the *Probability Density Function* (*PDF*) of the normal distribution. In contrast to the PMF (probability mass function) of discrete distributions, which is defined only for discrete integers, the PDF is defined for continuous values. The *standard normal distribution* is a normal distribution with a mean of zero and a standard deviation of one, and is sometimes referred to as *z-distribution*.

For smaller sample numbers, the sample distribution can show quite a bit of variability. For example, look at 25 distributions generated by sampling 100 numbers from a normal distribution (Fig. 6.9)

The normal distribution with parameters μ and σ is denoted as $N(\mu, \sigma)$ (Table 6.2). If the random variates (rvs) of X are normally distributed with expectation μ and standard deviation σ, one writes: $X \in N(\mu, \sigma)$ (Fig. 6.10).

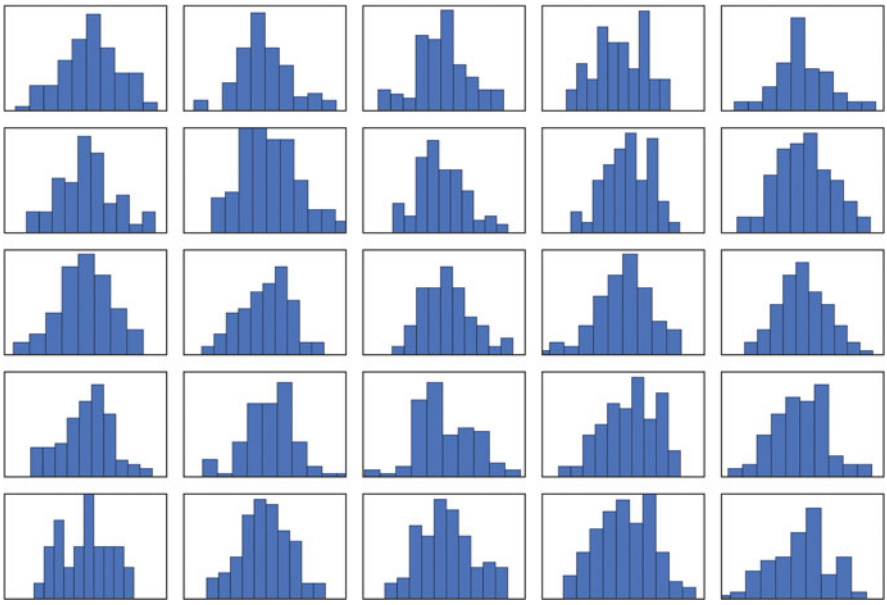


Fig. 6.9 Twenty-five randomly generated samples of 100 points from a standard normal distribution

Table 6.2 Tails of a normal distribution, with the distance from the mean expressed in standard deviations (SDs)

	Probability of being	
Range	Within range	Outside range
Mean ± 1SD	68.3 %	31.7 %
Mean ± 2SD	95.4 %	4.6 %
Mean ± 3SD	99.7 %	0.27 %

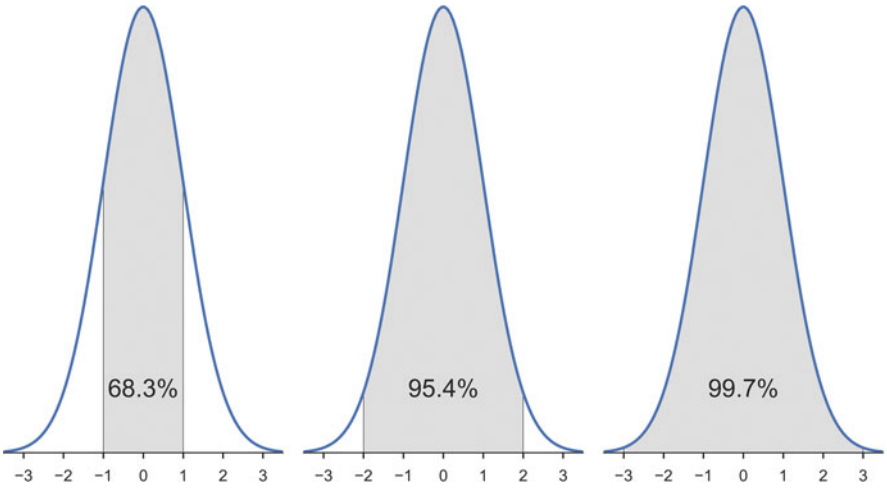


Fig. 6.10 Area under ± 1, 2, and 3 standard deviations of a normal distribution

Code: "ISP_distNormal.py"[3] shows simple manipulations of normal distribution functions.

```
In [1]:  import numpy as np
In [2]:  from scipy import stats

In [3]:  mu = -2
In [4]:  sigma = 0.7
In [5]:  myDistribution = stats.norm(mu, sigma)
In [6]:  significanceLevel = 0.05

In [7]:  myDistribution.ppf(
             [significanceLevel/2, 1-significanceLevel/2] )
Out[8]:  array([-3.38590382, -0.61409618]
```

Example of how to calculate the interval of the PDF containing 95 % of the data, for the blue curve in Fig. 6.8

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/06_Distributions/distNormal.

Sum of Normal Distributions
An important property of normal distributions is that the sum (or difference) of two normal distributions is also normally distributed. i.e., if

$$X \in N(\mu_X, \sigma_X^2)$$
$$Y \in N(\mu_Y, \sigma_Y^2)$$
$$Z = X \pm Y,$$

then

$$Z \in N(\mu_X \pm \mu_Y, \sigma_X^2 + \sigma_Y^2). \tag{6.16}$$

In words, the variance of the sum is the sum of the variances.

6.3.1 Examples of Normal Distributions

- If the average man is 175 cm tall with a standard deviation of 6 cm, what is the probability that a man selected at random will be 183 cm tall?
- If cans are assumed to have a standard deviation of 4 g, what does the average weight need to be in order to ensure that 99 % of all cans have a weight of at least 250 g?
- If the average man is 175 cm tall with a standard deviation of 6 cm, and the average woman is 168 cm tall with a standard deviation of 3 cm, what is the probability that a randomly selected man will be shorter than a randomly selected woman?

6.3.2 Central Limit Theorem

The central limit theorem states that the mean of a sufficiently large number of identically distributed random variates will be approximately normally distributed. Or in other words, the sampling distribution of the mean tends toward normality, regardless of the distribution. Figure 6.11 shows that averaging over ten uniformly distributed data already produces a smooth, almost Gaussian distribution.

Code: "ISP_centralLimitTheorem.py"[4] demonstrates that already averaging over ten uniformly distributed data points produces an almost Gaussian distribution.

[4] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/06_Distributions/centralLimitTheorem.

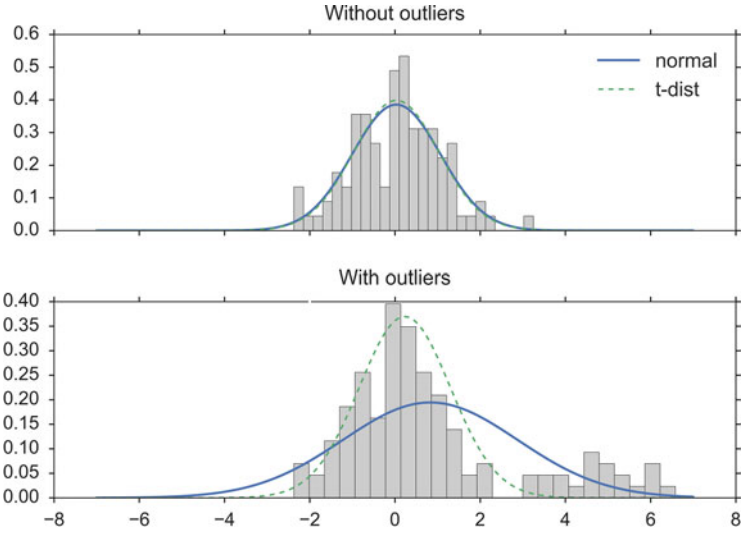


Fig. 6.11 Demonstration of the *Central Limit Theorem* for a uniform distribution: (*Left*) Histogram of uniformly distributed random data between 0 and 1. (*Center*) Histogram of average over two data points. (*Right*) Histogram of average over ten data points

6.3.3 Distributions and Hypothesis Tests

To illustrate the connection between distribution functions and hypothesis tests, let me go step-by-step through the analysis of the following problem:

The average weight of a newborn child in the USA is 3.5 kg, with a standard deviation of 0.76 kg. If we want to check all children that are significantly different from the typical baby, what should we do with a child that is born with a weight of 2.6 kg?

We can rephrase that problem in the form of a *hypothesis test*: our hypothesis is that *the baby comes from the population of healthy babies*. Can we keep the hypothesis, or does the weight of the baby suggest that we should reject that hypothesis?

To answer that question, we can proceed as follows:

- Find the distribution that characterizes healthy babies $\rightarrow \mu = 3.5$, $\sigma = 0.76$.
- Calculate the CDF at the value of interest $\rightarrow CDF(2.6\,kg) = 0.118$. In other words, the probability that a healthy baby is at least 0.9 kg lighter than the average baby is 11.8 %.
- Since we have a normal distribution, the probability that a healthy baby is at least 0.9 kg heavier than the average baby is also 11.8 %.

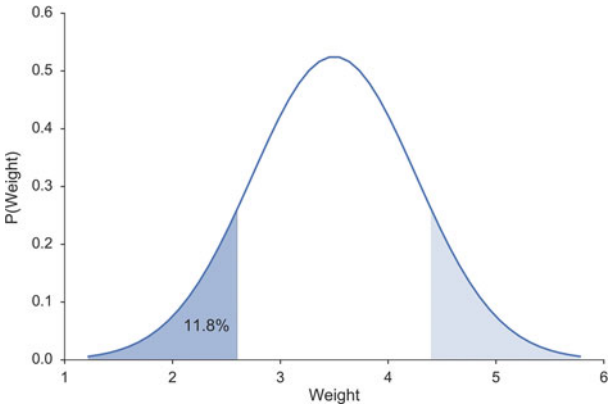


Fig. 6.12 The chance that a healthy baby weighs 2.6 kg or less is 11.8 % (*darker area*). The chance that the difference from the mean is as extreme or more extreme than for 2.6 kg is twice that much, as the lighter area must also be considered

- Interpret the result → *If the baby is healthy, the chance that its weight deviates by at least 0.9 kg from the mean is 2*11.8 % = 23.6 %. This is not significant, so we do not have sufficient evidence to reject our hypothesis, and our baby is regarded as healthy* (see Fig. 6.12).

```
In [1]: from scipy import stats

In [2]: nd = stats.norm(3.5, 0.76)

In [3]: nd.cdf(2.6)
Out[3]: 0.11816
```

Note: The starting hypothesis is often referred to as *null hypothesis*. In our example it would mean that we assume that there is *null* difference between the distribution the baby comes from and the population of healthy babies.

6.4 Continuous Distributions Derived from the Normal Distribution

Some frequently encountered continuous distributions are closely related to the normal distribution:

- ***t*-Distribution**: The sample distribution of mean values for samples from a normally distributed population. Typically used for small sample numbers, when the true mean/SD are not known.
- **χ-Square distribution**: For describing variability of normally distributed data.

- ***F*-distribution**: For comparing variabilities of two sets of normally distributed data.

  In the following we will discuss these continuous distribution functions.

Code: "ISP_distContinuous.py"[5] shows different continuous distribution functions.

6.4.1 t-Distribution

In 1908 W.S. Gosset, who worked for the Guinness brewery in Dublin, was interested in the problems of small samples, for example the chemical properties of barley where sample sizes might be as low as 3. Since in these measurements the true variance of the mean was unknown, it must be approximated by the sample standard error of the mean. And the ratio between the sample mean and the standard error had a distribution that was unknown till Gosset, under the pseudonym "Student," solved that problem. The corresponding distribution is the *t-distribution*, and converges for larger values towards the normal distribution (Fig. 6.13). Due to Gosset's pseudonym, "Student," it is now also known as *Student's distribution*.

Since in most cases the population mean and its variance are unknown, one typically works with the *t*-distribution when analyzing sample data.

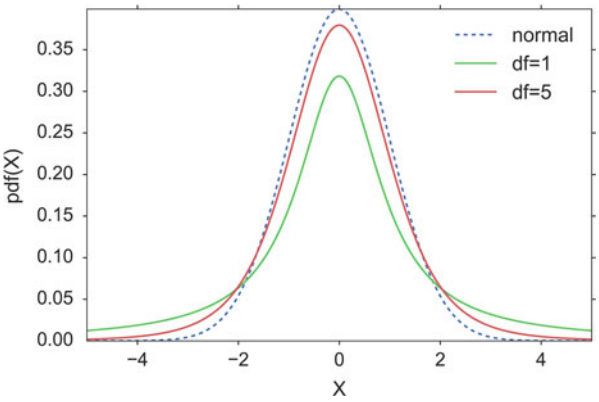


Fig. 6.13 *t*-Distribution

[5] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/06_Distributions/distContinuous.

If $\bar{x}$ is the sample mean, and s the sample standard deviation, the resulting statistic is

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}} = \frac{\bar{x} - \mu}{SE} \tag{6.17}$$

A very frequent application of the t-distribution is in the calculation of confidence intervals for the mean. The width of the 95 %-confidence interval (CI), i.e., the interval that contains the true mean with a chance of 95 %, is the same width about the population mean that contains 95 % of the sample means (Eq. 6.17):

$$ci = mean \pm se * t_{df,\alpha} \tag{6.18}$$

The following example shows how to calculate the t-values for the 95 %-CI, for n = 20. The lower end of the 95 % CI is the value that is larger than 2.5 % of the distribution; and the upper end of the 95 %-CI is the value that is larger than 97.5 % of the distribution. These values can be obtained either with the *percentile point function* (*PPF*), or with the *inverse survival function* (*ISF*). For comparison, I also calculate the corresponding value from the normal distribution:

```
In [1]: import numpy as np
In [2]: from scipy import stats
In [3]: n = 20
In [4]: df = n-1
In [5]: alpha = 0.05

In [6]: stats.t(df).isf(alpha/2)
Out[6]: 2.093

In [7]: stats.norm.isf(alpha/2)
Out[7]: 1.960
```

In *Python*, the 95 %-CI for the mean can be obtained with a one-liner:

```
In [8]: alpha = 0.95
In [9]: df = len(data)-1
In [10]: ci = stats.t.interval(alpha, df,
                 loc=np.mean(data), scale=stats.sem(data))
```

Since the t-distribution has longer tails than the normal distribution, it is much less affected by extreme cases (see Fig. 6.14).

6.4.2 Chi-Square Distribution

a) Definition

The chi-square distribution is related to the normal distribution in a simple way: if a random variable X has a normal distribution ($X \in N(0, 1)$), then X^2 has a chi-square distribution, with one degree of freedom ($X^2 \in \chi_1^2$). The sum squares of n

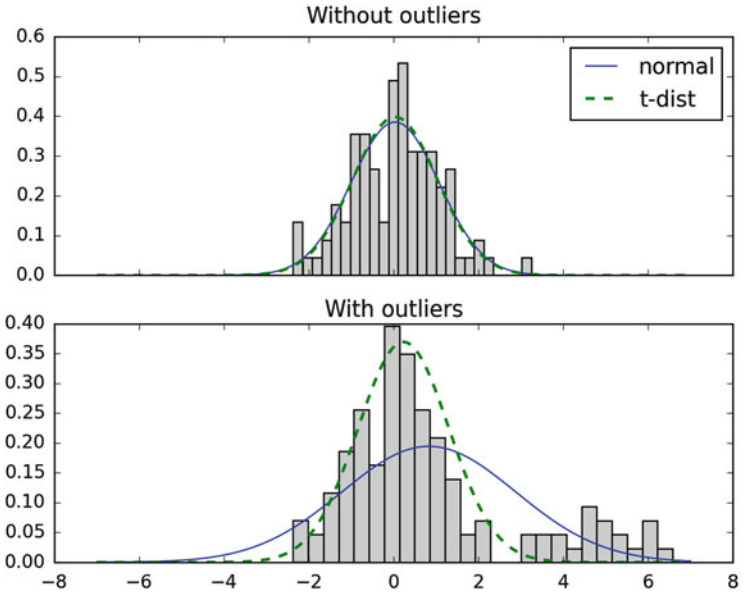


Fig. 6.14 The *t*-distribution is much more robust against outliers than the normal distribution. (*Top*) Best-fit normal and *t*-distribution, for a sample from a normal population. (*Bottom*) Same fits, with 20 "outliers," normally distributed data about 5, added

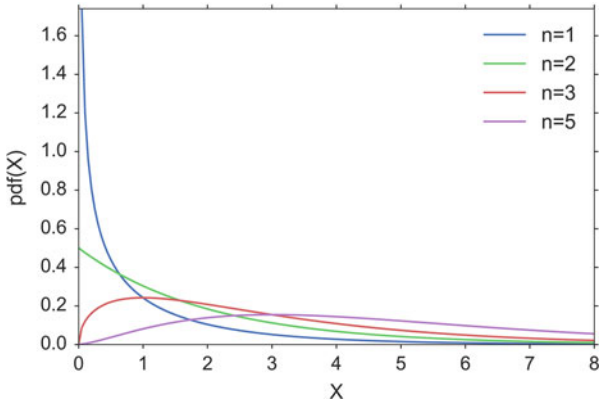


Fig. 6.15 Chi-square distribution

independent and standard normal random variables have a chi-square distribution with n degrees of freedom (Fig. 6.15):

$$\sum_{i=1}^{n} X_i^2 \in \chi_n^2 \tag{6.19}$$

b) Application Example

A pill producer is ordered to deliver pills with a standard deviation of $\sigma = 0.05$. From the next batch of pills $n = 13$ random samples have a weight of 3.04, 2.94, 3.01, 3.00, 2.94, 2.91, 3.02, 3.04, 3.09, 2.95, 2.99, 3.10, 3.02 g.

Question Is the standard deviation larger than allowed?

Answer Since the chi-square distribution describes the distribution of the summed squares of random variates from a *standard normal distribution*, we have to normalize our data before we calculate the corresponding CDF-value:

$$SF_{\chi^2_{(n-1)}} = 1 - CDF_{\chi^2_{(n-1)}} \left(\sum (\frac{x - \bar{x}}{\sigma})^2 \right) = 0.1929 \tag{6.20}$$

Interpretation If the batch of pills is from a distribution with a standard deviation of $\sigma = 0.05$, the likelihood of obtaining a chi-square value as large or larger than the one observed is about 19 %, so it is not atypical. In other words, the batch matches the expected standard deviation.

Note The number of the DOF is $n - 1$, because we are only interested in the shape of the distribution, and the mean value of the n data is subtracted from all data points.

```
In [1]: import numpy as np
In [2]: from scipy import stats
In [3]: data = np.r_[3.04, 2.94, 3.01, 3.00, 2.94, 2.91, 3.02,
                     3.04, 3.09, 2.95, 2.99, 3.10, 3.02]
In [4]: sigma = 0.05
In [5]: chi2Dist = stats.chi2(len(data)-1)
In [6]: statistic = sum( ((data-np.mean(data))/sigma)**2 )

In [7]: chi2Dist.sf(statistic)
Out[7]: 0.19293
```

6.4.3 F-Distribution

a) Definition

This distribution is named after Sir Ronald Fisher, who developed the *F* distribution for use in determining critical values in ANOVAs ("ANalysis Of VAriance," see Sect. 8.3.1).

If we want to investigate whether two groups have the same variance, we have to calculate the ratio of the sample standard deviations squared:

$$F = \frac{S_x^2}{S_y^2} \tag{6.21}$$

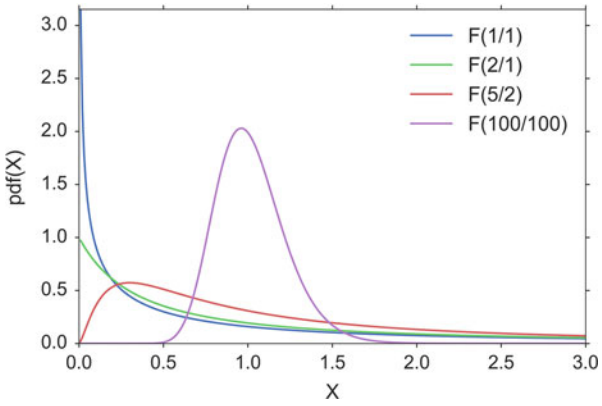


Fig. 6.16 *F*-distribution

where S_x is the sample standard deviation of the first sample, and S_y the sample standard deviation of the second sample.

The distribution of this statistic is the *F distribution*. For applications in ANOVAs (see Sect. 8.3.1), the cutoff values for an F distribution are generally found using three variables:

- ANOVA numerator degrees of freedom
- ANOVA denominator degrees of freedom
- significance level

An ANOVA compares the size of the variance between two different samples. This is done by dividing the larger variance by the smaller variance. The formula for the resulting F statistic is (Fig. 6.16):

$$F(r_1, r_2) = \frac{\chi^2_{r1}/r_1}{\chi^2_{r2}/r_2} \tag{6.22}$$

where χ^2_{r1} and χ^2_{r2} are the chi-square statistics of sample one and two respectively, and r_1 and r_2 are their degrees of freedom.

b) Application Example

Take for example the case where we want to compare the precision of two methods to measure eye movements. The two methods can have different accuracy and different precision. As shown in Fig. 6.17, the *accuracy* gives the deviation between the real and the measured value, while the *precision* is determined by the variance of the measurements. With the test we want to determine if the precision of the two methods is equivalent, or if one method is more precise than the other.

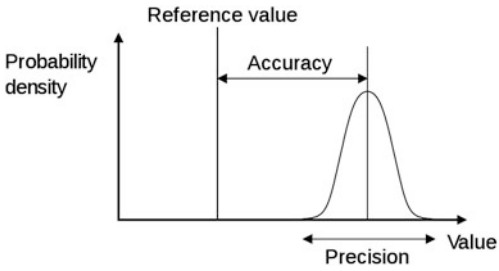


Fig. 6.17 Accuracy and precision of a measurement are two different characteristics

When you look 20° to the right, you get the following results:

Method 1: [20.7, 20.3, 20.3, 20.3, 20.7, 19.9, 19.9, 19.9, 20.3, 20.3, 19.7, 20.3]
Method 2: [19.7, 19.4, 20.1, 18.6, 18.8, 20.2, 18.7, 19.]

The F statistic is $F = 0.244$, and has $n-1$ and $m-1$ degrees of freedom, where n and m are the number of recordings with each method. The code sample below shows that the F statistic is in the tail of the distribution (p_oneTail=0.019), so we reject the hypothesis that the two methods have the same precision.

```
import numpy as np
from scipy import stats

method1 = np.array([20.7,  20.3,  20.3,  20.3,  20.7,  19.9,
                19.9, 19.9,  20.3, 20.3,  19.7,  20.3])
method2 = np.array([ 19.7,  19.4,  20.1,  18.6,  18.8,  20.2,
                    18.7,  19. ])

fval = np.var(method1, ddof=1)/np.var(method2, ddof=1)
fd = stats.f(len(method1)-1,len(method2)-1)
p_oneTail = fd.cdf(fval)    # -> 0.019

if (p_oneTail<0.025) or (p_oneTail>0.975):
    print('There is a significant difference'
          ' between the two distributions.')
else:
    print('No significant difference.')
```

6.5 Other Continuous Distributions

Some common distributions which are not directly related to the normal distribution are described briefly in the following:

- **Lognormal distribution**: A normal distribution, plotted on an exponential scale. A logarithmic transformation of the data is often used to convert a strongly skewed distribution into a normal one.
- **Weibull distribution**: Mainly used for reliability or survival data.

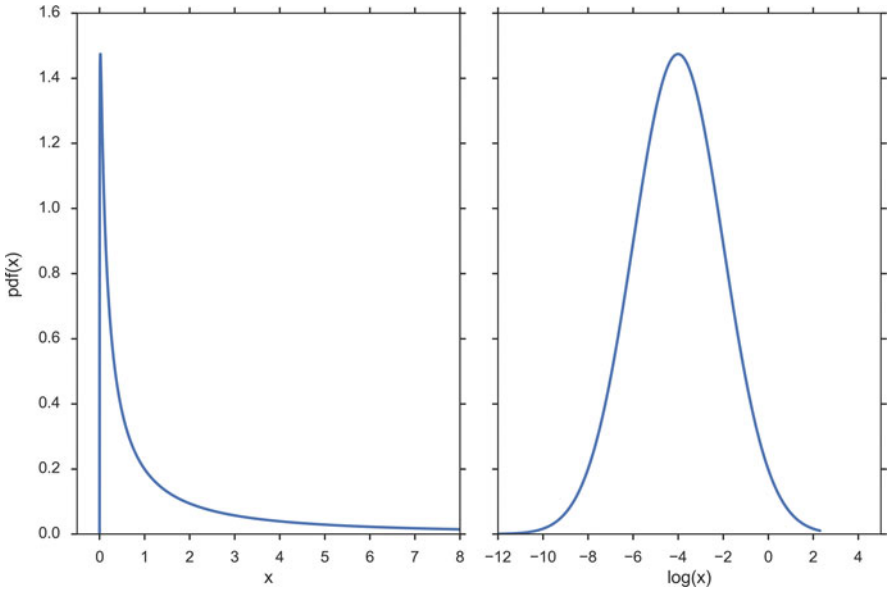


Fig. 6.18 Lognormal distribution, plotted against a linear abscissa (*left*) and against a logarithmic abscissa (*right*)

- **Exponential distribution**: Exponential curves.
- **Uniform distribution**: When everything is equally likely.

6.5.1 Lognormal Distribution

Normal distributions are the easiest ones to work with. In some circumstances a set of data with a positively skewed distribution can be transformed into a symmetric, normal distribution by taking logarithms. Taking logs of data with a skewed distribution will often give a distribution that is near to normal (see Fig. 6.18).

6.5.2 Weibull Distribution

The Weibull distribution is the most commonly used distribution for modeling reliability data or "survival" data. It has two parameters, which allow it to handle increasing, decreasing, or constant failure-rates (see Fig. 6.19). It is defined as

$$f_x(x) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0, \\ 0 & x < 0, \end{cases} \tag{6.23}$$

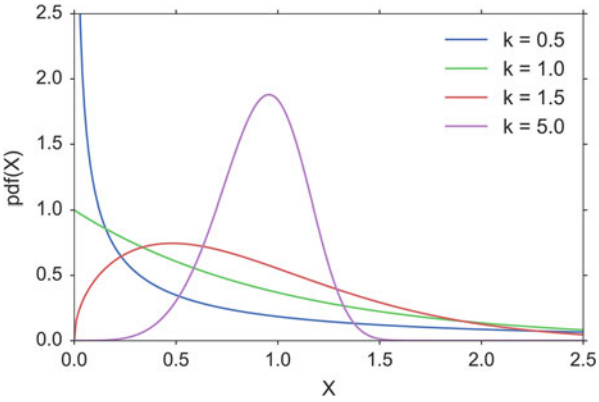


Fig. 6.19 Weibull distribution. The scale parameters for all curves is $\lambda = 1$

where $k > 0$ is the *shape parameter* and $\lambda > 0$ is the scale parameter of the distribution. (It is one of the rare cases where we use a shape parameter different from skewness and kurtosis.) Its complementary cumulative distribution function is a stretched exponential function.

If the quantity x is a "time-to-failure," the Weibull distribution gives a distribution for which the failure rate is proportional to a power of time. The shape parameter, k, is that power plus one, and so this parameter can be interpreted directly as follows:

- A value of $k < 1$ indicates that the failure rate decreases over time. This happens if there is significant "infant mortality," or defective items failing early and the failure rate decreasing over time as the defective items are weeded out of the population.
- A value of $k = 1$ indicates that the failure rate is constant over time. This might suggest random external events are causing mortality, or failure.
- A value of $k > 1$ indicates that the failure rate increases with time. This happens if there is an "aging" process, or parts that are more likely to fail as time goes on. An example would be products with a built-in weakness that fail soon after the warranty expires.

In the field of materials science, the shape parameter k of a distribution of strengths is known as the *Weibull modulus*.

6.5.3 Exponential Distribution

For a stochastic variable X with an exponential distribution, the probability distribution function is:

$$f_x(x) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{6.24}$$

The exponential PDF is shown in Fig. 6.20.

6.5.4 Uniform Distribution

This is a simple one: an even probability for all data values (Fig. 6.21). Not very common for real data.

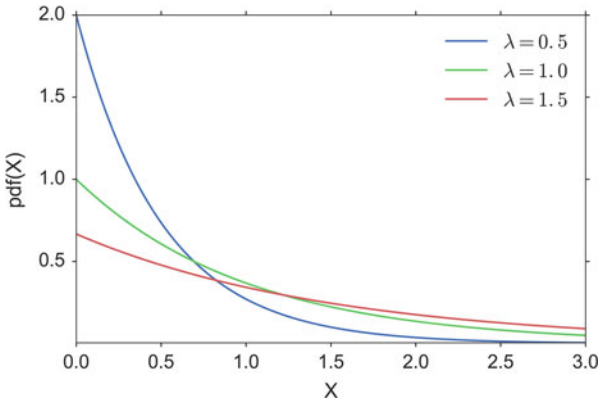


Fig. 6.20 Exponential distribution

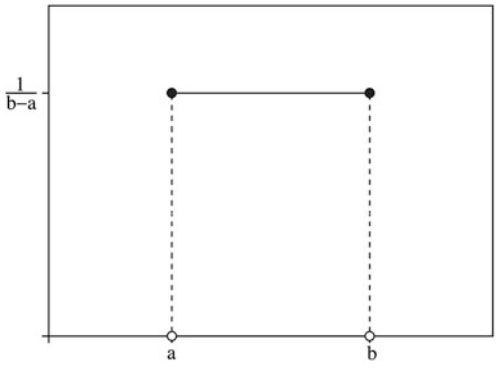


Fig. 6.21 Uniform distribution

6.6 Exercises

6.1 Sample Standard Deviation

Create an numpy-array, containing the data $1, 2, 3, \ldots, 10$. Calculate mean and sample(!)-standard deviation. (Correct answer for the SD: 3.03.)

6.2 Normal Distribution

- Generate and plot the Probability Density Function (PDF) of a normal distribution, with a mean of 5 and a standard deviation of 3.
- Generate 1000 random data from this distribution.
- Calculate the standard error of the mean of these data. (Correct answer: ca. 0.096.)
- Plot the histogram of these data.
- From the PDF, calculate the interval containing 95 % of these data. (Correct answer: $[-0.88, 10.88]$.)
- Your doctor tells you that he can use hip implants for surgery even if they are 1 mm bigger or smaller than the specified size. And your financial officer tells you that you can discard 1 out of 1000 hip implants, and still make a profit.

  What is the required standard deviation for the producer of the hip implants, to simultaneously satisfy both requirements? (Correct answer: $\sigma = 0.304\,\text{mm}$.)

6.3 Continuous Distributions

- ***t*-Distribution:** Measuring the weight of your colleagues, you have obtained the following weights: 52, 70, 65, 85, 62, 83, 59 kg. Calculate the corresponding mean, and the 99 % confidence interval for the mean. Note: with n values you have $n - 1$ DOF for the t-distribution. (Correct answer: $68.0 + / - 17.2\,\text{kg}$.)
- **Chi-square Distribution:** Create three normally distributed data sets (mean $=$ 0, SD $= 1$), with 1000 samples each. Then square them, sum them (so that you have 1000 data-points), and create a histogram with 100 bins. This should be similar to the curve for the chi-square distribution, with 3 DOF (i.e., it should come down at the left, see Fig. 6.22).
- ***F*-Distribution:** You have two apple trees. There are three apples from the first tree that weigh 110, 121, and 143 g, respectively, and four from the other which weigh 88, 93, 105, and 124 g, respectively. Are the variances from the two trees different?

  Note: calculate the corresponding F-value, and check if the CDF for the corresponding F-distribution is < 0.025. (Correct answer: no.)

6.4 Discrete Distributions

- **Binomial Distribution**: "According to research, pure blue eyes in Europe approach greatest frequency in Finland, Sweden, and Norway (at 72 %), followed by Estonia, Denmark (69 %); Latvia, Ireland (66 %); Scotland (63 %); Lithuania (61 %); The Netherlands (58 %); Belarus, England (55 %); Germany (53 %); Poland, Wales (50 %); Russia, The Czech Republic (48 %); Slovakia (46 %);

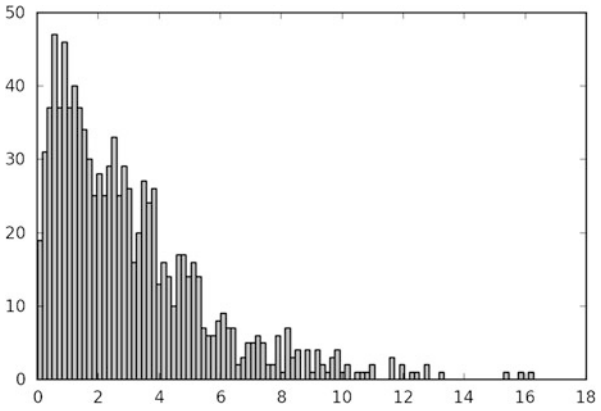


Fig. 6.22 Chi2 distribution with three degrees of freedom

Belgium (43 %); Austria, Switzerland, Ukraine (37 %); France, Slovenia (34 %); Hungary (28 %); Croatia (26 %); Bosnia and Herzegovina (24 %); Romania (20 %); Italy (18 %); Serbia, Bulgaria (17 %); Spain (15 %); Georgia, Portugal (13 %); Albania (11 %); Turkey and Greece (10 %). Further analysis shows that the average occurrence of blue eyes in Europe is 34 %, with 50 % in Northern Europe and 18 % in Southern Europe."

If we have 15 Austrian students in the classroom, what is the chance of finding three, six, or ten students with blue eyes? (Correct answer: 9, 20.1, and 1.4 %.)

- **Poisson Distribution**: In 2012 there were 62 fatal accidents on streets in Austria. Assuming that those are evenly distributed, we have on average $62/(365/7) = 1.19$ fatal accidents per week. How big is the chance that in a given week there are no, two, or five accidents? (Correct answer: 30.5, 21.5, 0.6 %.)

Chapter 7
Hypothesis Tests

This chapter describes a typical workflow in the analysis of statistical data. Special attention is paid to visual and quantitative tests of normality for the data. Then the concept of a *hypothesis tests* is explained, as well as the different types of errors, and the interpretation of *p-values* is discussed. Finally, the common test concepts of *sensitivity* and *specificity* are introduced and explained.

7.1 Typical Analysis Procedure

In "the old days" (before computers with almost unlimited computational power were available), the statistical analysis of data was typically restricted to hypothesis tests: you formulate a hypothesis, collect your data, and then accept or reject the hypothesis. The resulting hypothesis tests form the basic framework for by far most analyses in medicine and life sciences, and the most important hypotheses tests will be described in the following chapters.

The advent of powerful computers has changed the game. Nowadays, the analysis of statistical data is (or at least should be) a highly interactive process: you look at the data, and generate models which may explain your data. Then you determine the best fit parameters for these models, and check these models, typically by looking at the residuals. If you are not happy with the results, you modify the model to improve the correspondence between models and data; when you are happy, you calculate the confidence intervals for your model parameters, and form your interpretation based on these values. An introduction into this type of statistical analysis is provided in Chap. 11.

In either case, one should start off with the following steps:

- Visually inspect the data.
- Find extreme samples, and check them carefully.
- Determine the data-type of the values.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_7

- If the data are continuous, check whether or not they are normally distributed.
- Select and apply the appropriate test, or start with the model-based analysis of the data.

7.1.1 Data Screening and Outliers

The first step in data analysis is the visual inspection of the data. Our visual system is enormously powerful, and if the data are properly displayed, trends that characterize the data can be clearly visible. In addition to checking if the first and the last data values have been read in correctly, it is recommendable to check for missing data and outliers.

There is no unique definition for outliers. However, for normally distributed samples they are often defined as data that lie either more than 1.5*IQR (interquartile range), or more than two standard deviations, from the sample mean. Outliers often fall in one of two groups: they are either caused by mistakes in the recording, in which case they should be excluded; or they constitute very important and valuable data points, in which case they have to be included in the data analysis. To decide which of the two is the case, you have to check the underlying raw data (for saturation or invalid data values), and the protocols from your experiments (for mistakes that may have occurred during the recording). If an underlying problem is detected, then—and only then—one may eliminate the outliers from the analysis. In every other case, the data have to be kept!

7.1.2 Normality Check

Statistical hypothesis tests can be grouped into *parametric tests* and *nonparametric tests*. Parametric tests assume that the data can be well described by a distribution that is defined by one or more parameters, in most cases by a normal distribution. For the given data set, the best-fit parameters for this distribution are then determined, together with their confidence intervals, and interpreted.

However, this approach only works if the given data set is in fact well approximated by the chosen distribution. If not, the results of the parametric test can be completely wrong. In that case nonparametric tests have to be used which are less sensitive, but therefore do not depend on the data following a specific distribution.

a) Probability-Plots

In statistics different tools are available for the visual assessments of distributions. A number of graphical methods exist for comparing two probability distributions by

plotting their quantiles, or closely related parameters, against each other:

QQ-Plots The "Q" in QQ-plot stands for *quantile*. The quantiles of a given data set are plotted against the quantiles of a reference distribution, typically the standard normal distribution.

PP-Plots Plot the CDF (cumulative-distribution-function) of a given data set against the CDF of a reference distribution.

Probability Plots Plot the ordered values of a given data set against the quantiles of a reference distribution.

In all three cases the results are similar: if the two distributions being compared are similar, the points will approximately lie on the line $y = x$. If the distributions are linearly related, the points will approximately lie on a line, but not necessarily on the line $y = x$ (Fig. 7.1).

In *Python*, a probability plot can be generated with the command

```
stats.probplot(data, plot=plt)
```

To understand the principle behind those plots, look at the right plot in Fig. 7.2. Here we have *100* random data points from a chi2-distribution, which is clearly asymmetrical (Fig. 7.2, left). The x-value of the first data point is (approximately) the $1/100$-quantile of a standard normal distribution (`stats.norm().ppf(0.01)`), which corresponds to -2.33. (The exact value is slightly shifted, because of a small correction, called "Filliben's estimate".) The y-value is the smallest value of our data-set. Similarly, the second x-value corresponds approximately to `stats.norm().ppf(0.02)`, and the second y-value is the second-lowest value of the data set, etc.

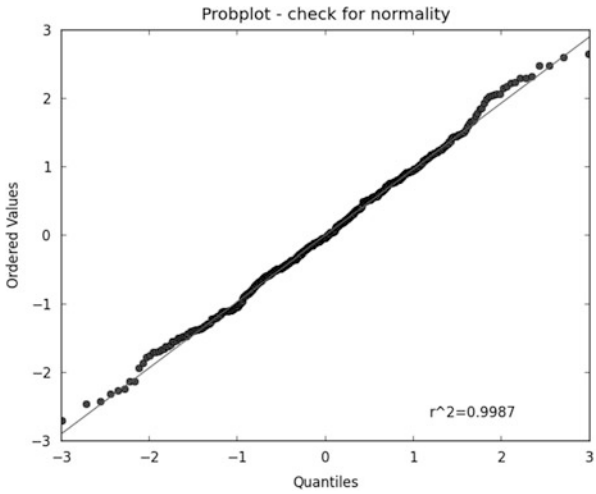


Fig. 7.1 Probability-plot, to check for normality of a sample distribution

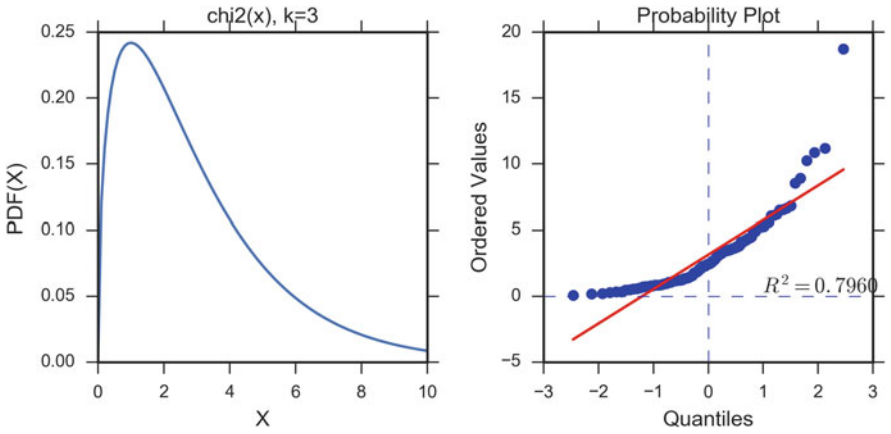


Fig. 7.2 (*Left*) Probability-density-function for a Chi2-distribution ($k = 3$), which is clearly non-normal. (*Right*) Corresponding probability-plot

b) Tests for Normality

In tests for normality, different challenges can arise: sometimes only few samples may be available, while other times one may have many data, but some extremely outlying values. To cope with the different situations different tests for normality have been developed. These tests to evaluate normality (or similarity to some specific distribution) can be broadly divided into two categories:

1. Tests based on comparison ("best fit") with a given distribution, often specified in terms of its CDF. Examples are the Kolmogorov–Smirnov test, the Lilliefors test, the Anderson–Darling test, the Cramer–von Mises criterion, as well as the Shapiro–Wilk and Shapiro–Francia tests.
2. Tests based on descriptive statistics of the sample. Examples are the skewness test, the kurtosis test, the D'Agostino–Pearson omnibus test, or the Jarque–Bera test.

For example, the Lilliefors test, which is based on the Kolmogorov–Smirnov test, quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution (Fig. 7.3), or between the empirical distribution functions of two samples. (The original Kolmogorov–Smirnov test should not be used if the number of samples is ca. $\leq$300.)

The Shapiro–Wilk W test, which depends on the covariance matrix between the order statistics of the observations, can also be used with $\leq$50 samples, and has been recommended by Altman (1999) and by Ghasemi and Zahediasl (2012).

The *Python* command `stats.normaltest(x)` uses the D'Agostino–Pearson *omnibus test*. This test combines a skewness and kurtosis test to produce a single, global "omnibus" statistic.

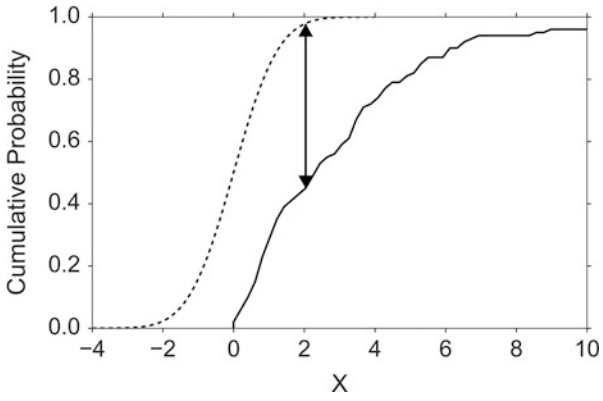


Fig. 7.3 Illustration of the Kolmogorov–Smirnov statistic. The *dashed line* is the CDF for the normal distribution, the *solid line* is the empirical CDF for a chi2-distribution (Fig.7.2), and the black arrow is the K–S statistic which is integrated

Below the output of the *Python*-module `C7_1_checkNormality.py` is shown, which checks 1000 random variates from a normal distribution for normality. Note that while for the full data set all tests correctly indicate that the underlying distribution is normal, the effects of extreme values strongly depend on the type of test if only the first 100 random variates are included.

```
p-values for all 1000 data points: ----------------
Omnibus                0.913684
Shapiro-Wilk           0.558346
Lilliefors             0.569781
Kolmogorov-Smirnov     0.898967

p-values for the first 100 data points: ----------------
Omnibus                0.004530
Shapiro-Wilk           0.047102
Lilliefors             0.183717
Kolmogorov-Smirnov     0.640677
```

Code: "ISP_checkNormality.py"[1] shows how to check graphically, as well as with different quantitative tests, if a given distribution is normal.

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/07_CheckNormality_CalcSamplesize/checkNormality.

7.1.3 Transformation

If the data deviate significantly from a normal distribution, it is sometimes possible to make the distribution approximately normal by transforming the data. For example, data often have values that can only be positive (e.g., the size of persons), and that have long positive tail: such data can often be made normal by applying a log transform. This is demonstrated in Fig. 6.18.

7.2 Hypothesis Concept, Errors, *p*-Value, and Sample Size

7.2.1 An Example

Assume that you are running a private educational institution. Your contract says that if your students score 110 in the final exam, where the national average is 100, you get a bonus. When the results are significantly lower, you loose your bonus (because the students are not good enough), and you have to hire more teachers; and when the results are significantly higher, you also loose your bonus (because you have spent too much money on teachers), and you have to cut back on the number of teachers.

The final exam of your ten students produce the following scores (Fig. 7.4):

```
In [1]: import numpy as np
In [2]: scores = np.array([ 109.4, 76.2, 128.7, 93.7, 85.6,
                    117.7, 117.2, 87.3, 100.3, 55.1])
```

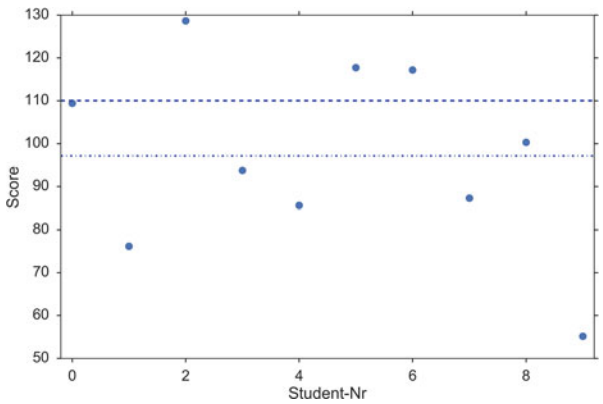


Fig. 7.4 The question we ask: based on our sample mean (*dashed-dotted line*) and the observed variance of the data (the sample variance), do we believe that the population mean is different from 110 (*dashed line*)? Or in other words: our null hypothesis is that the difference between the population mean and 110 is zero. Can we keep our null hypothesis, or do we have to reject it based on the data?

The question we want to answer: Is the mean value of the scores (*97.1*) significantly different from 110?

A *normality test* (`stats.normaltest(scores)`) indicates that the data are probably taken from a normal distribution. Since we don't know the population variance of the results of students tested, we have to take our best guess, the sample variance (see also Fig. 5.1). And we know that the normalized difference between sample and the population mean, the *t*-statistic, follows the *t*-distribution (Eq. 6.17).

The difference between our sample mean and the value we want to compare it to (`np.mean(scores) - 110`) is *–12.9*. Normalized by the sample standard error (Eq. 6.17), this gives a value of $t = -1.84$. Since the *t*-distribution is a known curve that depends only on the number of samples, and we can calculate the likelihood that we obtain a *t*-statistic of $|t| > 1.84$:

```
In [3]: tval = (110-np.mean(scores))/stats.sem(scores)    # 1.84
In [4]: td = stats.t(len(scores)-1)    # "frozen" t-distribution
In [5]: p = 2*td.sf(tval)                                  # 0.0995
```

(The factor *2* in the last line of the code is required, since we have to combine the probability of $t < -1.84$ and $t > 1.84$.) Expressed in words, given our sample data, we can state that the likelihood that the population mean is *110* is 9.95 %. But since a *statistical difference* is only given by convention if the likelihood is less than 5 %, we conclude that the observed value of 97.1 is not significantly different from 110, and the bonus has to be paid out.

7.2.2 Generalization and Applications

a) Generalization

Based on the previous example, the general procedure for hypothesis tests can be described as follows (the sketch in Fig. 5.1 indicates the meaning of many upcoming terms):

- A random sample is drawn from a population. (In our example, the random sample is our *scores*).
- A null hypothesis is formulated. ("There is null difference between the population mean and the value of 110.")
- A test-statistic is calculated, of which we know the probability distribution. (Here the sample mean, since we know that the mean value of samples from a normal distribution follows the *t*-distribution.)
- Comparing the observed value of the statistic (here the obtained *t*-value) with the corresponding distribution (the *t*-distribution), we can find the likelihood that a value as extreme as or more extreme than the observed one is found by chance. This is the so-called *p-value*.
- If the *p*-value is $p < 0.05$, we reject the null hypothesis, and speak of a *statistically significant difference*. If a value of $p < 0.001$ is obtained, the result

is typically called *highly significant*. The critical region of a hypothesis test is the set of all outcomes which cause the null hypothesis to be rejected.

In other words, the p-value states how likely it is to obtain a value as extreme or more extreme by chance alone, *if the null hypothesis is true*.

The value against which the p-value is compared is the *significance level*, and is often indicated with the letter α. The significance level is a user choice, and typically set to 0.05.

This way of proceeding to test a hypothesis is called *statistical inference*.

Remember, p only indicates the likelihood of obtaining a certain value for the test statistic if the null hypothesis is true—nothing else!

And keep in mind that improbable events do happen, even if not very frequently. For example, back in 1980 a woman named Maureen Wilcox bought tickets for both the Rhode Island lottery and the Massachusetts lottery. And she got the correct numbers for both lotteries. Unfortunately for her, she picked all the correct numbers for Massachusetts on her Rhode Island ticket, and all the right numbers for Rhode island on her Massachusetts ticket. Seen statistically, the p-value for such an event would be extremely small—but it did happen anyway.

b) Additional Examples

Example 1 Let us compare the weight of two groups of subjects. The null hypothesis is that there is null difference in the weight between the two groups. If a statistical comparison of the weight produces a p-value of 0.03, this means that the probability that the null hypothesis is correct is 0.03, or 3 %. Since this probability is less than 0.05, we say that "there is a significant difference between the weight of the two groups."

Example 2 If we want to check the assumption that the mean value of a group is 7, then the corresponding null hypothesis would be: "We assume that there is null difference between the mean value in our population and the value 7."

Example 3 (Test for Normality) If we check if a data sample is normally distributed, the null hypothesis is that "there is no difference between my data and normally distributed data": here a large p-value indicates that the data are in fact normally distributed!

7.2.3 The Interpretation of the p-Value

A value of $p < 0.05$ for the null hypothesis has to be interpreted as follows: *If the null hypothesis is true, the chance to find a test statistic as extreme as or more extreme than the one observed is less than 5 %.* This is *not* the same as saying that the null hypothesis is false, and even less so, that an alternative hypothesis is true!

Stating a *p*-value alone is no longer state of the art for the statistical analysis of data. In addition also the confidence intervals for the parameters under investigation should be given.

To reduce errors in the data interpretation research is sometimes divided into *exploratory research* and *confirmatory research*. Take for example the case of Matt Motyl, a Psychology PhD student at the University of Virginia. In 2010, data from his study of nearly 2000 people indicated that political moderates saw shades of grey more accurately than people with more extreme political opinions, with a *p*-value of 0.01. However, when he tried to reproduce the data, the *p*-value dropped down to 0.59. So while the *exploratory research* showed that a certain hypothesis may be likely, the *confirmatory research* showed that the hypothesis did not hold (Nuzzo 2014).

An impressive demonstration of the difference between exploratory and confirmatory research is a collaborative science study, where 270 researchers set down and tried to replicate the findings of 100 experimental and correlational studies published in three leading psychology journals in 2008. While 97 % of the studies had statistically significant results, only 36 % of replications were statistically significant (OSC 2015)!

Sellke (2001) has investigated this question in detail, and recommends to use a "calibrated *p*-value" to estimate the probability of making a mistake when rejecting the null hypothesis, when the data produce a *p*-value *p*:

$$\alpha(p) = \frac{1}{1 + \frac{1}{-e\, p\, log(p)}} \tag{7.1}$$

with $e = exp(1)$, and *log* the natural logarithm. For example, $p = 0.05$ leads to $\alpha = 0.29$, and $p = 0.01$ to $\alpha = 0.11$. However, I have to admit that I have not seen that idea applied in practical research.

7.2.4 Types of Error

In hypothesis testing, two types of errors can occur.

a) Type I Errors

Type I errors are errors where the result is significant despite the fact that the null hypothesis is true. The likelihood of a Type I error is commonly indicated with α, and is set before the start of the data analysis. In quality control, a Type I error is called *producer risk*, because you reject a produced item despite the fact that it meets the regulatory requirements.

In Fig. 7.7, a Type I error would be a diagnosis of cancer ("positive" test result), even though the subject is healthy.

Example

For example, assume that the population of young Austrian adults has a mean IQ of 105 (i.e., if Austrian males were smarter than the rest) and a standard deviation of 15. We now want to check if the average FH student in Linz has the same IQ as the average Austrian, and we select 20 students. We set $\alpha = 0.05$, i.e., we set our significance level to 95 %. Let us now assume that the average student has in fact the same IQ as the average Austrian. If we repeat our study 20 times, we will find one of those 20 times that our sample mean is significantly different from the Austrian average IQ. Such a finding would be a false result, despite the fact that our assumption is correct, and would constitute a Type I error.

b) Type II Errors and Test Power

If we want to answer the question "How much chance do we have to reject the null hypothesis when the alternative is in fact true?," or in other words, "What's the probability of detecting a real effect?," we are faced with a different problem. To answer these questions, we need an *alternative hypothesis*.

Type II errors are errors where the result is *not* significant, despite the fact that the null hypothesis is false. In quality control, a Type II error is called a *consumer risk*, because the consumer obtains an item that does not meet the regulatory requirements.

In Fig. 7.7, a Type II error would be a "healthy" diagnosis ("negative" test result), even though the subject has cancer.

The probability for this type of error is commonly indicated with β. The "power" of a statistical test is defined as $(1-\beta)*100$, and is the chance of correctly accepting the alternative hypothesis. Figure 7.5 shows the meaning of the power of a statistical test. Note that for finding the power of a test, you need an alternative hypothesis.

c) Pitfalls in the Interpretation of *p*-Values

In other words, *p*-values measure evidence for a hypothesis. Unfortunately, they are often incorrectly viewed as an error probability for rejection of the hypothesis, or, even worse, as the posterior probability (i.e., after the data have been collected) that the hypothesis is true. As an example, take the case where the alternative hypothesis is that the mean is just a fraction of one standard deviation larger than the mean under the null hypothesis: in that case, a sample that produces a *p*-value of 0.05 may just as likely be produced if the alternative hypothesis is true as if the null hypothesis is true!

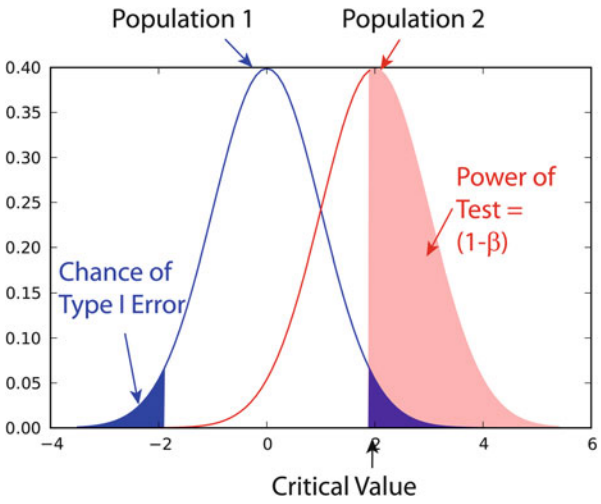


Fig. 7.5 *Power* of a statistical test, for comparing the mean value of two sampling distributions

7.2.5 Sample Size

The *power* or *sensitivity* of a binary hypothesis test is the probability that the test correctly rejects the null hypothesis when the alternative hypothesis is true.

The determination of the power of a statistical test, and the calculation of the minimum sample size required to reveal an effect of a given magnitude, is called *power analysis*. It involves four factors:

1. α, the probability for Type I errors
2. β, the probability for Type II errors ($\Rightarrow$ power of the test)
3. d, the effect size, i.e., the magnitude of the investigated effect relative to σ, the standard deviation of the sample
4. n, the sample size

Only three of these four parameters can be chosen, the 4th is then automatically fixed. Figure 7.6 shows for example how an increase in the number of subjects increases the power of the test.

The absolute size of the difference $D(= d*\sigma)$ between mean treatment outcomes that will answer the clinical question being posed is often called *clinical significance* or *clinical relevance*.

For example, the motor function of arms and hands after a stroke is commonly graded with the *Fugl-Meyer Assessment of the Upper Extremities*, which has a maximum score of 66. In 2014, a multi-center study of task-specific robot therapy of the arm after stroke showed that this therapy leads to a significant improvement of 0.78 points (Klamroth-Marganska 2014). But while this improvement is significant, it is so small that it does not make any difference in the treatment of the patient, and therefore is of no clinical relevance.

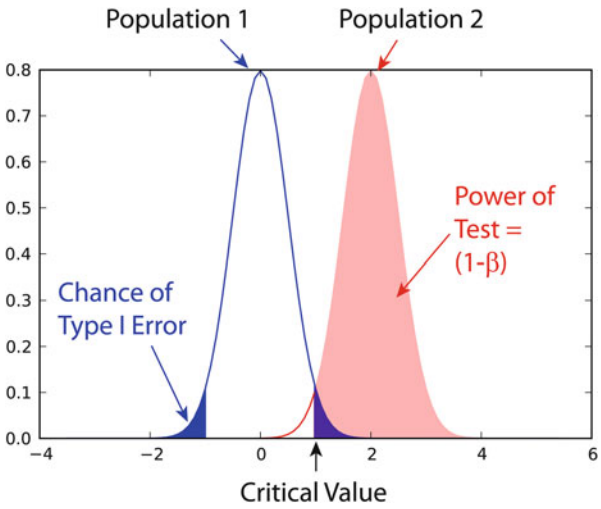


Fig. 7.6 Effect of an increase in sample size on the power of a test

a) Examples

Test on One Mean

If we have the hypothesis that the population we draw our samples from has a mean value of x_1 and a standard deviation of σ, and the actual population has a mean value of $x_1 + D$ and the same standard deviation, we can find such a difference with a minimum sample number of

$$n = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2}{d^2} \tag{7.2}$$

Here z is the standardized normal variable (see also Sect. 6.3)

$$z = \frac{x - \mu}{\sigma}. \tag{7.3}$$

and $d = \frac{D}{\sigma}$ the effect size.

In words, if the real mean has a value of x_1, we want to detect this correctly in at least $1 - \alpha\%$ of all tests; and if the real mean is shifted by D or more, we want to detect this with a likelihood of at least $1 - \beta\%$.

Test Between Two Different Populations

For finding a difference between two normally distributed means, with standard deviations of σ_1 and σ_2, the minimum number of samples we need in each group to

detect an absolute difference D is

$$n_1 = n_2 = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2(\sigma_1^2 + \sigma_2^2)}{D^2}. \tag{7.4}$$

b) Python Solution

statsmodels makes clever use of the fact that three of the four factors mentioned above are independent, and combines it with the *Python* feature of "named parameters" to provide a program that takes 3 of those parameters as input, and calculates the remaining 4th parameter. For example,

```
In [1]: from statsmodels.stats import power

In [2]: nobs = power.tt_ind_solve_power(
                 effect_size = 0.5, alpha =0.05, power=0.8 )

In [3]: print(nobs)
Out[3]: 63.76561177540974
```

tells us that if we compare two groups with the same number of subjects and the same standard deviation, require an $\alpha = 0.05$ and a test power of 80 %, and we want to detect a difference between the groups that is half the standard deviation, we need to test 64 subjects in each group.

Similarly,

```
In [4]: effect_size = power.tt_ind_solve_power(
                      alpha =0.05, power=0.8, nobs1=25 )

In [5]: print(effect_size)
Out[5]: 0.8087077886680407
```

tells us that if we have an $\alpha = 0.05$, a test power of 80 %, and 25 subjects in each group, then the smallest difference between the groups is 81 % of the sample standard deviation.

The corresponding command for one sample t-tests is `tt_solve_power`.

c) Programs: Sample Size

Code: "ISP_sampleSize.py"[2]: direct sample size calculation for normally distributed data with arbitrary standard deviations,

[2] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/07_CheckNormality_CalcSamplesize/sampleSize.

for detecting changes within a group, and for comparison of two independent groups with different variances; more flexible than the *statsmodel* function `power.tt_in_solve_power`.

7.3 Sensitivity and Specificity

Some of the more confusing terms in statistical analysis are *sensitivity* and *specificity*. Related topics are the *positive predictive value* (*PPV*) and the *negative predictive value* (*NPV*) of statistical tests. The diagram in Fig. 7.7 shows how the four are related:

Sensitivity Also called *power*. Proportion of positives that are correctly identified by a test (= probability of a positive test, given the patient is ill).

Specificity Proportion of negatives that are correctly identified by a test (= probability of a negative test, given that patient is well).

Positive Predictive Value (PPV) Proportion of patients with positive test results who are correctly diagnosed.

Negative Predictive Value (NPV) Proportion of patients with negative test results who are correctly diagnosed.

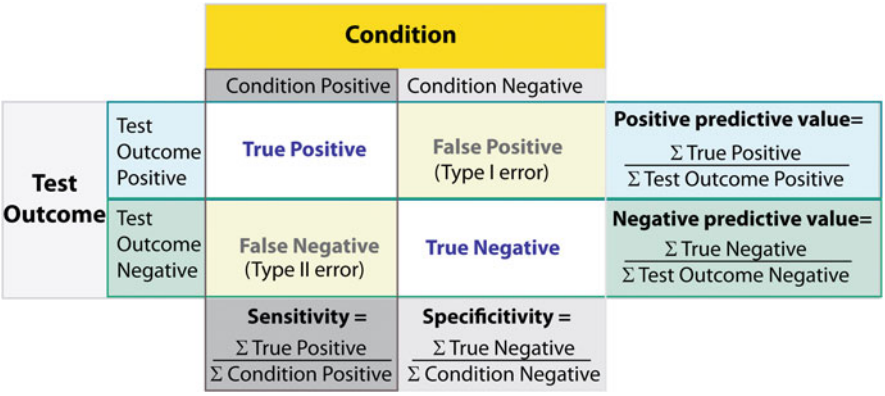


Fig. 7.7 Relationship between sensitivity, specificity, positive predictive value, and negative predictive value

For example, pregnancy tests have a high sensitivity: when a woman is pregnant, the probability that the test is positive is very high.

In contrast, an indicator for an attack with atomic weapons on the White House should have a very high specificity: if there is no attack, the probability that the indicator is positive should be very, very small.

While sensitivity and specificity characterize a test and are independent of prevalence, they do not indicate what portion of patients with abnormal test results are truly abnormal. This information is provided by the positive/negative predictive value (PPV/NPV). These are the values relevant for a doctor diagnosing a patient: when a patient has a positive test result, how likely is it that the patient is in fact sick? Unfortunately, as Fig. 7.8 indicates, these values are affected by the prevalence of the disease. The *prevalence* of a disease indicates how many out of 100,000 people are affected by it; in contrast, the *incidence* gives the number of newly diagnosed cases per 100,000 people. In summary, we need to know the prevalence of the disease as well as the PPV/NPV of a test to provide a sensible interpretation of medical test results.

Take for example a test where a positive test results implies a 50 % chance of having a certain medical condition. If half the population has this condition, a positive test result tells the doctor nothing. But if the condition is very rare, a positive test result indicates that the patient has a fifty–fifty chance of having this rare condition—a piece of information that is very valuable.

Figure 7.8 shows how the prevalence of a disease affects the interpretation of diagnostic results, with a test with a given specificity and sensitivity: a high prevalence of the disease increases the PPV of the test, but decreases the NPV; and a low prevalence does exactly the opposite. Figure 7.9 gives a worked example.

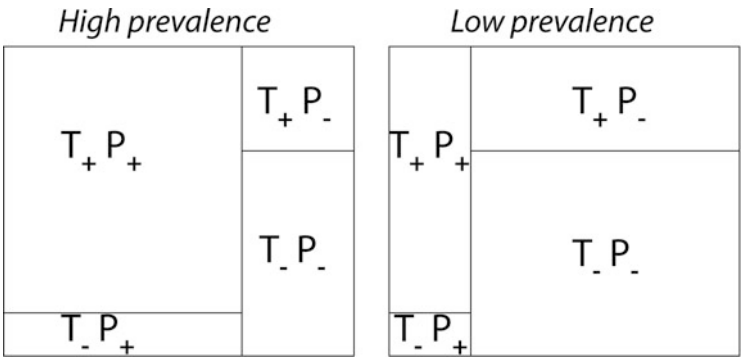


Fig. 7.8 Effect of prevalence on PPV and NPV. "T" stands for "test," and "P" for "patient." (For comparison with below: T+P+ = TP, T−P− = TN, T+P− = FP, and T−P+ = FN.)

		Condition		
		Condition Positive	Condition Negative	
Test Outcome	Test Outcome Positive	True Positive (TP) = 25	False Positive (FP) = 175	Positive predictive value= = TP / (TP+FP) = 25 / (25+175) = 12.5%
	Test Outcome Negative	False Negative (FN) = 10	True Negative (TN) = 2000	Negative predictive value= = TN / (FN+TN) = 2000 / (10+2000) = 99.5%
		Sensitivity = = TP / (TP+FN) = 25 / (25+10) = 71%	Specificitivity = = TN / (FP+TN) = 2000 / (175+2000) = 92%	

Fig. 7.9 Worked example

7.3.1 Related Calculations

In evidence-based medicine, *likelihood ratios* are used for assessing the value of performing a diagnostic test. They use the sensitivity and specificity of the test to determine whether a test result changes the probability that a condition (such as a disease state) exists. Two versions of the likelihood ratio exist, one for positive and one for negative test results. They are known as the *positive likelihood ratio* (*LR*+) and *negative likelihood ratio* (*LR*−), respectively.

- False positive rate (α) = type I error = $1 - specificity = \frac{FP}{FP+TN} = \frac{175}{175+2000} = 8\,\%$
- False negative rate (β) = type II error = $1 - sensitivity = \frac{FN}{TP+FN} = \frac{10}{25+10} = 29\,\%$
- Power = sensitivity = $1 - \beta$
- *positive likelihood ratio* = $\frac{sensitivity}{1-specificity} = \frac{71\,\%}{1-92\,\%} = 8.9$
- *negative likelihood ratio* = $\frac{1-sensitivity}{specificity} = \frac{1-71\,\%}{92\,\%} = 0.32$

Hence with large numbers of false positives and few false negatives, a positive test outcome in this example is in itself poor at confirming cancer (PPV = 12.5 %) and further investigations must be undertaken; it does, however, correctly identify 71 % of all cancers (the sensitivity). However as a screening test, a negative result is very good at reassuring that a patient does not have cancer (NPV = 99.5 %), and this initial screen correctly identifies 92 % of those who do not have cancer (the specificity).

7.4 Receiver-Operating-Characteristic (ROC) Curve

Closely related to sensitivity and specificity is the *Receiver-Operating-Characteristic* (*ROC*) curve. This is a graph displaying the relationship between the true positive rate (on the vertical axis) and the false positive rate (on the horizontal

axis). The technique comes from the field of engineering, where it was developed to find the predictor which best discriminates between two given distributions: ROC curves were first used during WWII to analyze radar effectiveness. In the early days of radar, it was sometimes hard to tell a bird from a plane. The British pioneered using ROC curves to optimize the way that they relied on radar for discriminating between incoming German planes and birds.

Take the case that we have two different distributions, for example one from the radar signal of birds and one from the radar signal of German planes, and we have to determine a cut-off value for an indicator in order to assign a test result to distribution one ("bird") or to distribution two ("German plane"). The only parameter that we can change is the cut-off value, and the question arises: is there an optimal choice for this cut-off value?

The answer is *yes*: it is the point on the ROC-curve with the largest distance to the diagonal (arrow in Fig.7.10).[3]

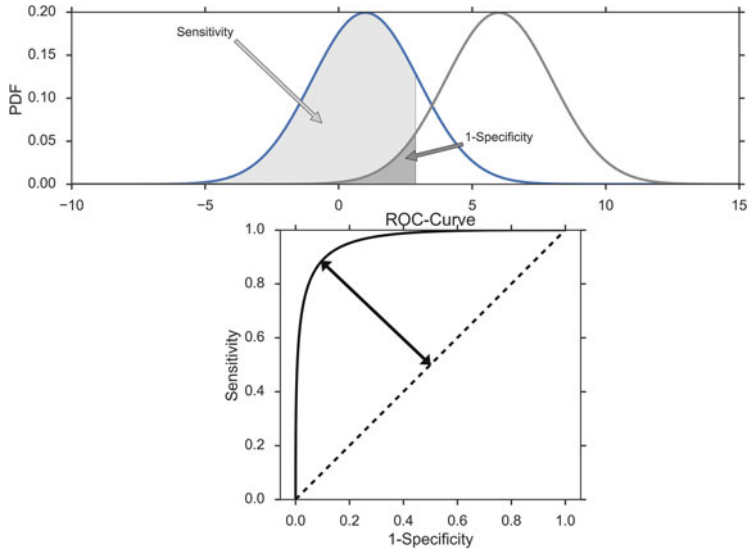


Fig. 7.10 *Top*: Probability density functions for two distributions. *Bottom*: Corresponding *ROC-curve*, with the largest distance between the diagonal and the curve indicated by the *arrow*

[3]Strictly speaking this only holds if Type I errors and Type II errors are equally important. Otherwise, the weight of each type of error also has to be considered.

Chapter 8
Tests of Means of Numerical Data

This chapter covers hypothesis tests for the mean values of groups, and shows how to implement each of these tests in *Python*:

- Comparison of one group with a fixed value.
- Comparison of two groups with respect to each other.
- Comparison of three or more groups with each other.

In each case we distinguish between two cases. If the data are approximately normally distributed, the so-called *parametric tests* can be used. These tests are more sensitive than *nonparametric tests*, but require that certain assumptions are fulfilled. If the data are not normally distributed, or if they are only available in ranked form, the corresponding nonparametric tests should be used.

8.1 Distribution of a Sample Mean

8.1.1 One Sample t-Test for a Mean Value

To check the mean value of normally distributed data against a reference value, we typically use the *one sample t-test*, which is based on the *t*-distribution.

If we knew the mean and the standard deviation of a normally distributed population, we could calculate the corresponding standard error, and use values from the normal distribution to determine how likely it is to find a certain value. However, in practice we have to estimate the mean and standard deviation from the sample; and the *t*-distribution, which characterizes the distribution of sample means for normally distributed data, deviates slightly from the normal distribution.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_8

a) Example

Since it is very important to understand the basic principles of how to calculate the *t*-statistic and the corresponding *p*-value for this test, let me illustrate the underlying statistics by going through the analysis of a specific example, step-by-step. As an example we take 100 normally distributed data, with a mean of 7 and with a standard deviation of 3. What is the chance of finding a mean value at a distance of 0.5 or more from the mean? *Answer: The probability from the t-test in the example is 0.057, and from the normal distribution 0.054.*

- We have a population, with a mean value of 7 and a standard deviation of 3.
- From that population an observer takes 100 random samples. The sample mean of the example shown in Fig. 8.1 is 7.10, close to but different from the real mean. The sample standard deviation is 3.12, and the standard error of the mean 0.312. This gives the observer an idea about the variability of the population.
- The observer knows that the distribution of the sample mean follows a *t*-distribution, and that the standard error of the mean (SEM) characterizes the width of that distribution.

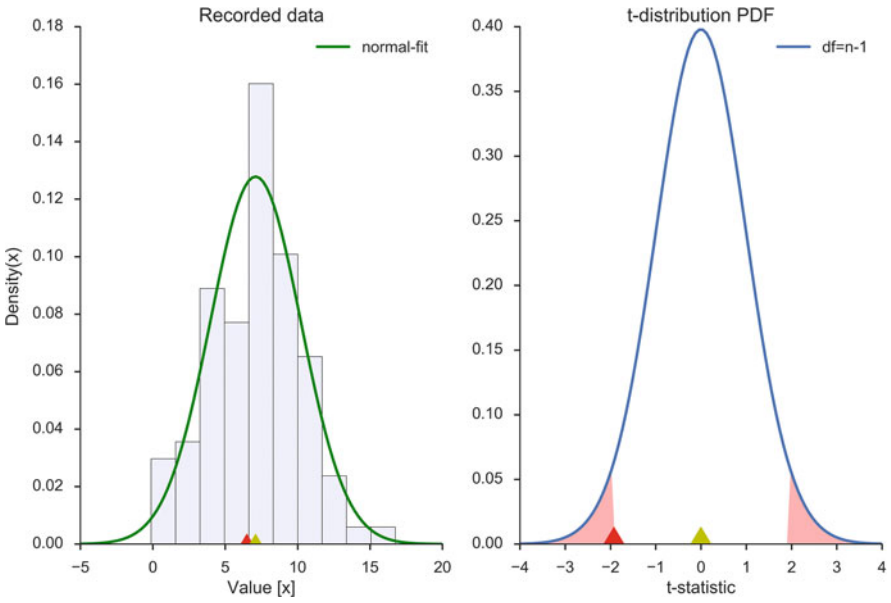


Fig. 8.1 *Left*: Frequency histogram of the sample data, together with a normal fit (*green line*). The sample mean, which is very close to the population mean, is indicated with a *yellow triangle*; the value to be checked is indicated with a *red triangle*. *Right*: sampling distribution of the mean value (*t*-distribution, for $n - 1$ degrees of freedom). At the bottom the normalized value of the sample mean (*yellow triangle*) and the normalized value to be checked (*red triangle*). The sum of the red shaded areas, indicating values as extreme or more extreme than the red arrow, corresponds to the *p*-value

- How likely it is that the real mean has a value of x_0 (e.g., 6.5, indicated by the red triangle in Fig. 8.1, left)? To find this out, the value has to be transformed by subtracting the sample mean and dividing by the standard error (Fig. 8.1, right). This provides the *t-statistic* for this test (−1.93).
- The corresponding *p*-value, which tells us how likely it is that the real mean has a value of 6.5 or more extreme relative to the sample mean, is given by the red shaded area under the curve-wings: $2 * CDF(t\text{-}statistic) = 0.057$, which means that the difference to 6.5 is just not significant. The factor "2" comes from the fact that we have to check in both tails, and this test is therefore referred to as a *two-tailed t-test*.
- The probability to find a value of 6.5 or less is half as much (*p=0.0285*. Since in this case we only look in one tail of the distribution, this is called a *one-tailed t-test*.

In *Python*, test statistic and *p*-value for the one sample *t*-test can be calculated with

```
t, pVal = stats.ttest_1samp(data, checkValue)
```

Code: "ISP_oneGroup.py"[1]: Sample analysis for one group of continuous data.

8.1.2 Wilcoxon Signed Rank Sum Test

If the data are not normally distributed, the one-sample *t*-test should not be used (although this test is fairly robust against deviations from normality, see Fig. 6.14). Instead, we must use a nonparametric test on the mean value. We can do this by performing a *Wilcoxon signed rank sum test*. Note that in contrast to the one-sample *t*-test, this test checks for a difference from null:

```
(rank, pVal) = stats.wilcoxon(data-checkValue)
```

This method has three steps[2]:

1. Calculate the difference between each observation and the value of interest.
2. Ignoring the signs of the differences, rank them in order of magnitude.
3. Calculate the sum of the ranks of all the negative (or positive) ranks, corresponding to the observations below (or above) the chosen hypothetical value.

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/oneGroup.

[2] The following description and example have been taken from (Altman 1999, Table 9.2).

Table 8.1 Daily energy intake of 11 healthy women with rank order of differences (ignoring their signs) from the recommended intake of 7725 kJ

Subject	Daily energy intake (kJ)	Difference from 7725 kJ	Ranks of differences
1	5260	2465	11
2	5470	2255	10
3	5640	2085	9
4	6180	1545	8
5	6390	1335	7
6	6515	1210	6
7	6805	920	4
8	7515	210	1.5
9	7515	210	1.5
10	8230	−505	3
11	8770	−1045	5

In Table 8.1 you see an example, where the significance to a deviation from the value of 7725 is tested. The rank sum of the negative values gives $3+5=8$, and can be looked up in the corresponding tables to be significant. In practice, your computer program will nowadays do this for you. This example also shows another feature of rank evaluations: tied values (here 7515) get accorded their mean rank (here 1.5).

8.2 Comparison of Two Groups

8.2.1 Paired t-Test

In the comparison of two groups with each other, two cases have to be distinguished. In the first case, two values recorded from the same subject at different times are compared to each other. For example, the size of students when they enter primary school and after their first year, to check if they have grown. Since we are only interested in the difference in each subject between the first and the second measurement, this test is called *paired t-test*, and is essentially equivalent to a one-sample *t*-test for the mean difference (Fig. 8.2).

Therefore the two tests `stats.ttest_1samp` and `stats.ttest_rel` provide the same result (apart from minute numerical differences):

```
In [1]: import numpy as np
In [2]: from scipy import stats

In [3]: np.random.seed(1234)
In [4]: data = np.random.randn(10)+0.1
In [5]: data1 = np.random.randn(10)*5  # dummy data
In [6]: data2 = data1 + data    # same group-difference as "data"
```

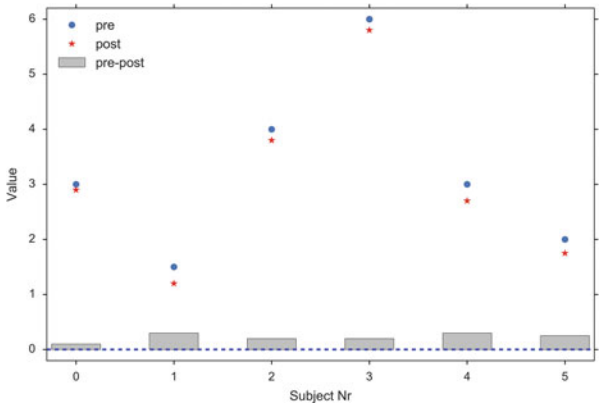


Fig. 8.2 Paired *t*-tests can detect differences that would be insignificant otherwise. For this example all the inner-subject differences are positive, and the paired *t*-test yields a *p*-value of $p < 0.001$, while the unpaired *t*-test leads to $p = 0.81$

```
In [7]: stats.ttest_1samp(data, 0)
Out[7]: (-0.12458492298731401, 0.90359045085470857)

In [8]: stats.ttest_rel(data2, data1)
Out[8]: (-0.1245849229873135, 0.9035904508547089)
```

8.2.2 t-Test between Independent Groups

An *unpaired t-test*, or *t-test for two independent groups*, compares two groups. An example would be the comparison of the effect of two medications given to two different groups of patients.

The basic idea is the same as for the one-sample *t*-test. But instead of the variance of the mean, we now need the variance of the difference between the means of the two groups. Since *the variance of a sum (or difference) of independent random variables equals the sum of the variances*, we have:

$$\begin{aligned} se(\bar{x}_1 \pm \bar{x}_2) &= \sqrt{\text{var}(\bar{x}_1) + \text{var}(\bar{x}_2)} \\ &= \sqrt{\{se(\bar{x}_1)\}^2 + \{se(\bar{x}_2)\}^2} \\ &= \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \end{aligned} \tag{8.1}$$

where $\bar{x}_i$ is the mean of the ith sample, and *se* indicates the *standard error*.

```
t_statistic, pVal = stats.ttest_ind(group1, group2)
```

8.2.3 *Nonparametric Comparison of Two Groups: Mann–Whitney Test*

If the measurement values from two groups are not normally distributed we have to resort to a nonparametric test. The most common nonparametric test for the comparison of two independent groups is the *Mann–Whitney(–Wilcoxon) test*. Watch out, because this test is sometimes also referred to as *Wilcoxon rank-sum test*. This is different from the *Wilcoxon signed rank sum test*! The test-statistic for this test is commonly indicated with *u*:

```
u_statistic, pVal = stats.mannwhitneyu(group1, group2)
```

Code: "ISP_twoGroups.py"[3]: Comparison of two groups, paired and unpaired.

8.2.4 *Statistical Hypothesis Tests vs Statistical Modeling*

With the advent of cheap computing power, statistical modeling has been a booming field. This has also affected classical statistical analysis, as most problems can be viewed from two perspectives: one can either make a statistical hypothesis, and verify or falsify that hypothesis; or one can make a statistical model, and analyze the significance of the model parameters.

Let me use a classical *t*-test as an example.

a) Classical *t*-Test

Let us take performance measurements from a racing team, on two different occasions. During Race1, the members of the team achieve a score of [79, 100, 93, 75, 84, 107, 66, 86, 103, 81, 83, 89, 105, 84, 86, 86, 112, 112, 100, 94], and during Race2 a score of [92, 100, 76, 97, 72, 79, 94, 71, 84, 76, 82, 57, 67, 78, 94, 83, 85, 92, 76, 88].

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/twoGroups.

To generate these numbers and to compare the two groups with a *t*-test, the following *Python* commands can be used:

```
import numpy as np
from scipy import stats

# Generate the data
np.random.seed(123)
race_1 = np.round(np.random.randn(20)*10+90)
race_2 = np.round(np.random.randn(20)*10+85)

# t-test
(t, pVal) = stats.ttest_rel (race_1, race_2)

# Show the result
print('The probability that the two distributions '
       'are equal is {0:5.3f} .'.format(pVal))
```

which produce

```
The probability that the two distributions are equal is 0.033 .
```

The command `random.seed(123)` initializes the random number generator with the number 123, which ensures that two consecutive runs of this code produce the same result, corresponding to the numbers given above.

b) Statistical Modeling

Expressed as a statistical model, we assume that the difference between the first and the second race is simply a constant value. (The null hypothesis would be that this value is equal to zero.) This model has one parameter: the constant value. We can find this parameter, as well as its confidence interval and a lot of additional information, with the following *Python* code:

```
import pandas as pd
import statsmodels.formula.api as sm

np.random.seed(123)
df = pd.DataFrame({'Race1': race_1, 'Race2':race_2})

result = sm.ols(formula='I(Race2-Race1) ~ 1', data=df).fit()

print(result.summary())
```

The important line is the last but one, which produces the *result*. Thereby the function `sm.ols` ("ols" for "ordinary least square") function from *statsmodels* tests the model which describes the difference between the results of *Race1* and those of *Race2* with only an offset, also called *Intercept* in the language of modeling. The results below show that the probability that this intercept is zero is only 0.033: the difference from zero is significant.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:       I(Race2 - Race1)   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                  0.000
Method:                 Least Squares   F-statistic:                       nan
Date:                Sun, 08 Feb 2015   Prob (F-statistic):                nan
Time:                        18:48:06   Log-Likelihood:                -85.296
No. Observations:                  20   AIC:                             172.6
Df Residuals:                      19   BIC:                             173.6
Df Model:                           0
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept     -9.1000      3.950     -2.304      0.033       -17.367    -0.833
==============================================================================
Omnibus:                        0.894   Durbin-Watson:                   2.009
Prob(Omnibus):                  0.639   Jarque-Bera (JB):                0.793
Skew:                           0.428   Prob(JB):                        0.673
Kurtosis:                       2.532   Cond. No.                         1.00
==============================================================================
```

The output from OLS-models is explained in more detail in Chap. 11. The important point here is that the *t*- and *p*-value for the intercept obtained with the statistical model are the same as with the classical *t*-test above.

8.3 Comparison of Multiple Groups

8.3.1 Analysis of Variance (ANOVA)

a) Principle

The idea behind the Analysis of Variance (ANOVA) is to divide the variance into the variance *between* groups, and that *within* groups, and see if those distributions match the null hypothesis that all groups come from the same distribution (Fig. 8.3). The variables that distinguish the different groups are often called *factors* or *treatments*.

(By comparison, *t*-tests look at the mean values of two groups, and check if those are consistent with the assumption that the two groups come from the same distribution.)

For example, if we compare a group with *No treatment*, another with *treatment A*, and a third with *treatment B*, then we perform a *one factor ANOVA*, sometimes also called *one-way ANOVA*, with *treatment* the one analysis factor. If we do the same test with men and with women, then we have a *two-factor* or *two-way ANOVA*, with *gender* and *treatment* as the two treatment factors. Note that with ANOVAs, it is quite important to have exactly the same number of samples in each analysis group! (This is called a *balanced ANOVA*: a balanced design has an equal number of observations for all possible combinations of factor levels.)

Because the null hypothesis is that there is no difference between the groups, the test is based on a comparison of the observed variation between the groups (i.e., between their means) with that expected from the observed variability within the

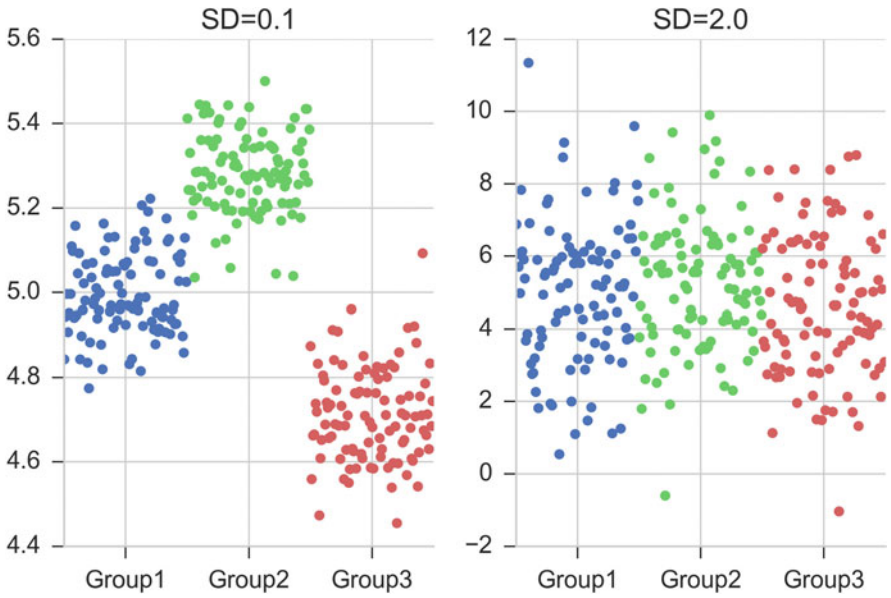


Fig. 8.3 In both cases, the difference between the two groups is the same. But left, the difference within the groups is smaller than the differences between the groups, whereas right, the difference within the groups is larger than the difference between

groups (i.e. between subjects). The comparison takes the general form of an F-test to compare variances, but for two groups the t-test leads to exactly the same result, as demonstrated in the code-sample `C8_3_anovaOneway.py`.

The one-way ANOVA assumes all the samples are drawn from normally distributed populations with equal variance. The assumption of equal variance can be checked with the Levene test.

ANOVA uses traditional terminology. DF indicates the degrees of freedom (DF) (see also Sect. 5.3), the summation is called the Sum-of-Squares (SS), the ratio between the two is called the Mean Square (MS), and the squared terms are deviations from the sample mean. In general, the sample variance is defined by

$$s^2 = \frac{1}{DF}\sum(y_i - \bar{y})^2 = \frac{SS}{DF} \tag{8.2}$$

The fundamental technique is a partitioning of the total sum of squares SS into components related to the effects used in the model (Fig. 8.4). Thereby ANOVA estimates three sample variances: a *total variance* based on all the observation deviations from the grand mean (calculated from SS_{Total}), a *treatment variance* (from $SS_{Treatments}$), and an *error variance* based on all the observation deviations from their appropriate treatment means (from SS_{Error}). The treatment variance is based on the deviations of treatment means from the grand mean, the result being multiplied by the number of observations in each treatment to account for the difference between

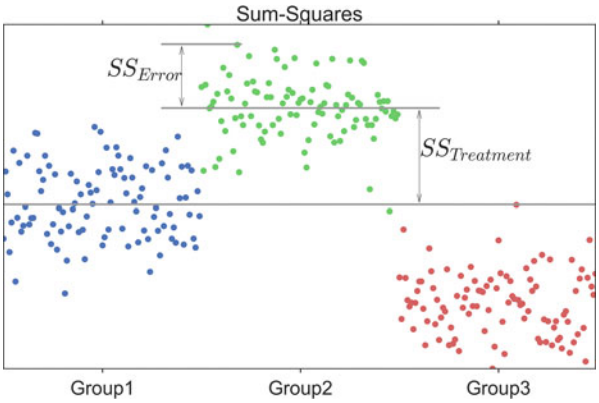


Fig. 8.4 The *long blue line* indicates the grand mean over all data. The SS_{Error} describes the variability *within* the groups, and the $SS_{Treamtent}$ (summed over all respective points!) the variability *between* groups

the variance of observations and the variance of means. The three sums-of-squares are related by:

$$SS_{\text{Total}} = SS_{\text{Error}} + SS_{\text{Treatment}} \tag{8.3}$$

where SS_{Total} is sum-squared deviation from the overall mean, the SS_{Error} the sum-squared deviation from the mean within a group, and the $SS_{Treatment}$ the sum-squared deviation between each group and the overall mean (Fig. 8.4). If the null hypothesis is true, all three variance estimates (Eq. 8.2) are equal (within sampling error).

The number of degrees of freedom DF can be partitioned in a similar way: one of these components (that for error) specifies a chi-squared distribution which describes the associated sum of squares, while the same is true for *treatments* if there is no treatment effect.

$$DF_{\text{Total}} = DF_{\text{Error}} + DF_{\text{Treatments}} \tag{8.4}$$

b) Example: One-Way ANOVA

As an example, take the red cell folate levels ($\mu g/l$) in three groups of cardiac bypass patients given different levels of nitrous oxide ventilation (Amess et al. 1978), described in the *Python* code example below. In total 22 patients were included in the analysis.

The null hypothesis of ANOVAs is that all groups come from the same population. A test whether to keep or reject this null hypothesis can be done with

```
from scipy import stats
F_statistic, pVal = stats.f_oneway(group_1, group_2, group_3)
```

where the data of group *i* are in a vector `group_i`. (The whole program can be found in `ISP_anovaOneway.py`.)

A more detailed output of the ANOVA is provided by the implementation in *statsmodels*:

```
import pandas as pd
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm

df = pd.DataFrame(data, columns=['value', 'treatment'])
model = ols('value ~ C(treatment)', df).fit()
anovaResults = anova_lm(model)
print(anovaResults)
```

where the numerical values are in the first column of the array `data`, and the (categorical) group variable in the second column. This produces the following output:

```
                DF     SS           MS      F     p(>F)
C(treatment)     2  15515.76  7757.88  3.71   0.043
Residual        19  39716.09  2090.32   NaN    NaN
```

- First the "Sums of squares (SS)" are calculated. Here the SS between treatments is 15,515.76, and the SS of the residuals is 39,716.09 . The total SS is the sum of these two values.
- The mean squares (MS) is the SS divided by the corresponding degrees of freedom (DF).
- The *F-test* or *variance ratio test* is used for comparing the factors of the total deviation. The *F*-value is the larger mean squares value divided by the smaller value. (If we only have two groups, the *F*-value is the square of the corresponding *t*-value. See `ISP_anovaOneway.py`).

$$F = \frac{variance\_between\_treatments}{variance\_within\_treatments}$$

$$F = \frac{MS_{\text{Treatments}}}{MS_{\text{Error}}} = \frac{SS_{\text{Treatments}}/(n_{groups} - 1)}{SS_{\text{Error}}/(n_{total} - n_{groups})} \tag{8.5}$$

- Under the null hypothesis that two normally distributed populations have equal variances we expect the ratio of the two sample variances to have an *F Distribution* (see Sect. 6.5). From the *F*-value, we can look up the corresponding *p*-value.

Code: "ISP_anovaOneway.py"[4]: Different aspects of one-way ANOVAs: how to check the assumptions (with the Levene

[4] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/anovaOneway.

test), different ways to calculate a one-way ANOVA, and a demonstration that for the comparison between two groups, a one-way ANOVA is equivalent to a *t*-test.

8.3.2 Multiple Comparisons

The null hypothesis in a one-way ANOVA is that the means of all the samples are the same. So if a one-way ANOVA yields a significant result, we only know that they are *not* the same.

However, often we are not just interested in the joint hypothesis if all samples are the same, but we would also like to know for which pairs of samples the hypothesis of equal values is rejected. In this case we conduct several tests at the same time, one test for each pair of samples. (Typically, this is done with *t*-tests.)

These tests are sometimes referred to as *post-hoc analysis*. In the design and analysis of experiments, a post hoc analysis (from Latin post hoc, "after this") consists of looking at the data—after the experiment has concluded—for patterns that were not specified beforehand. Here this is the case, because the null hypothesis of the ANOVA is that there is no difference between the groups.

This results, as a consequence, in a *multiple testing problem*: since we perform multiple comparison tests, we should compensate for the risk of getting a significant result, even if our null hypothesis is true. This can be done by correcting the *p*-values to account for this. We have a number of options to do so:

- Tukey HSD
- Bonferroni correction
- Holms correction
- ... and others ...

a) Tukey's Test

Tukey's test, sometimes also referred to as the Tukey *Honest Significant Difference test* (*HSD*) method, controls for the Type I error rate across multiple comparisons and is generally considered an acceptable technique. It is based on a statistic that we have not come across yet, the *studentized range*, which is commonly represented by the variable q. The *studentized range* computed from a list of numbers $x_1, \ldots, x_n$ is given by

$$q_n = \frac{\max\{x_1, \ \ldots \ x_n\} - \min\{x_1, \ \ldots \ x_n\}}{s} \tag{8.6}$$

where s is the sample standard deviation. In the Tukey HSD method the sample $x_1, \ldots, x_n$ is a sample of means and q is the basic test-statistic. It can be used as post-hoc analysis to test between which two group-means there is a significant difference

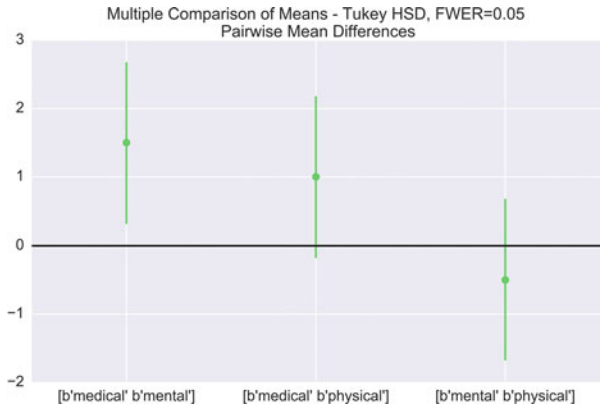


Fig. 8.5 Comparing the means of multiple groups—here three different treatment options

(pairwise comparisons) after rejecting the null hypothesis that all groups are from the same population, i.e. all means are equal (Fig. 8.5).

Code: "ISP_multipleTesting.py"[5]: This script provides an example where three treatments are compared.

b) Bonferroni Correction

Tukey's studentized range test (HSD) is a test specific to the comparison of all pairs of k independent samples. Instead we can run t-tests on all pairs, calculate the p-values and apply one of the p-value corrections for multiple testing problems. The simplest—and at the same time quite conservative—approach is to divide the required p-value by the number of tests that we do (*Bonferroni correction*). For example, if you perform four comparisons, you check for significance not at $p = 0.05$, but at $p = 0.05/4 = 0.0125$.

While multiple testing is not yet included in *Python* standardly, you can get a number of multiple-testing corrections done with the *statsmodels* package:

```
In [7]: from statsmodels.sandbox.stats.multicomp \
  ... :     import multipletests

In [8]: multipletests([.05, 0.3, 0.01], method='bonferroni')
Out[8]:
```

[5] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/multipleTesting.

```
(array([False, False,  True], dtype=bool),
array([ 0.15,  0.9 ,  0.03]),
0.016952427508441503,
0.016666666666666666)
```

c) Holm Correction

The Holm adjustment, sometimes also referred to as *Holm–Bonferroni method*, sequentially compares the lowest *p*-value with a Type I error rate that is reduced for each consecutive test. For example, if you have three groups (and thus three comparisons), this means that the first *p*-value is tested at the 0.05/3 level (0.017), the second at the 0.05/2 level (0.025), and third at the 0.05/1 level (0.05). As stated by Holm (1979) "Except in trivial non-interesting cases the sequentially rejective Bonferroni test [i.e., the Holm–Bonferroni method] has strictly larger probability of rejecting false hypotheses and thus it ought to replace the classical Bonferroni test at all instants where the latter usually is applied."

8.3.3 Kruskal–Wallis Test

When we compare *two* groups to each other, we use the *t-test* when the data are normally distributed, and the nonparametric *Mann–Whitney test* otherwise. For *three or more* groups, the test for normally distributed data is the *ANOVA-test*; for not-normally distributed data, the corresponding test is the *Kruskal–Wallis test*. When the null hypothesis is true the test statistic for the Kruskal–Wallis test follows the chi-square distribution.

Code: "ISP_kruskalWallis.py"[6]: Example of a Kruskal–Wallis test (for not normally distributed data).

8.3.4 Two-Way ANOVA

Compared to one-way ANOVAs, the analysis with two-way ANOVAs has a new element. We can look not only if each of the factors is significant; we can also check if the *interaction* of the factors has a significant influence on the distribution

[6] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/kruskalWallis.

of the data. Let us take for example measurements of fetal head circumference, by four observers in three fetuses, from a study investigating the reproducibility of ultrasonic fetal head circumference data.

The most elegant way of implementing a two-way ANOVAs for these data is with *statsmodels*:

```
import pandas as pd
from C2_8_getdata import getData
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm

# Get the data
data = getData('altman_12_6.txt', subDir='..\Data\data_altman')

# Bring them in DataFrame-format
df = pd.DataFrame(data, columns=['hs', 'fetus', 'observer'])

# Determine the ANOVA with interaction
formula = 'hs ~ C(fetus) + C(observer) + C(fetus):C(observer)'
lm = ols(formula, df).fit()
anovaResults = anova_lm(lm)

print(anovaResults)
```

This leads to the following result:

```
                      df  sum_sq mean_sq        F    PR(>F)
C(fetus)               2  324.00  162.00  2113.10  1.05e-27
C(observer)            3    1.19    0.39     5.21  6.497-03
C(fetus):C(observer)   6    0.56    0.09     1.22  3.29e-01
Residual              24    1.84    0.07      NaN       NaN
```

In words: While—as expected—different fetuses show highly significant differences in their head size ($p < 0.001$), also the choice of the observer has a significant effect ($p < 0.05$). However, no individual observer was significantly off with any individual fetus ($p > 0.05$).

Code: "ISP_anovaTwoway.py"[7]: Two-way Analysis of Variance (ANOVA).

[7] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/08_TestsMeanValues/anovaTwoway.

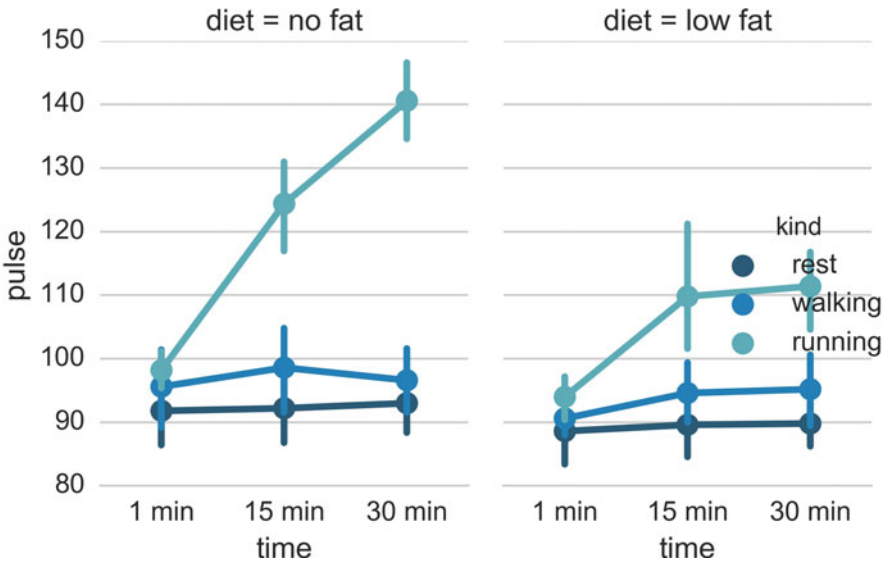


Fig. 8.6 Three-way ANOVA

8.3.5 Three-Way ANOVA

With more than two factors, it is recommendable to use *statistical modeling* for the data analysis (see Chap. 11). However, as always with the analysis of statistical data, one should first inspect the data visually. *seaborn* makes this quite simple. Figure 8.6 shows for example the pulse-rate after *(1/15/30)* minutes of *(resting/walking/running)*, in two groups who are eating different diets.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")

df = sns.load_dataset("exercise")

sns.factorplot("time", "pulse", hue="kind", col="diet", data=df,
               hue_order=["rest", "walking", "running"],
               palette="YlGnBu_d", aspect=.75).despine(left=True)
plt.show()
```

8.4 Summary: Selecting the Right Test for Comparing Groups

8.4.1 Typical Tests

Table 8.2 shows typical tests for statistical problems for nominal and ordinal data. When we have univariate data and two groups, we can ask the question: "Are they different?" The answer is provided by hypothesis tests: by a *t*-test if the data are normally distributed, or by a Mann–Whitney test otherwise.

So what happens when we have more than two groups?

To answer the question "Are they different?" for more than two groups, we have to use the Analysis of Variance (ANOVA)-test for data where the residuals are normally distributed. If this condition is not fulfilled, the Kruskal–Wallis Test has to be used.

What should we do if we have paired data?

If we have matched pairs for two groups, and the differences are not normally distributed, we can use the Wilcoxon-signed-rank-sum test. The rank test for more than two groups of matched data is the *Friedman test*.

An example for the application of the Friedman test: Ten professional piano players are blindfolded, and are asked to judge the quality of three different pianos. Each player rates each piano on a scale of 1–10 (1 being the lowest possible grade, and 10 the highest possible grade). The null hypothesis is that all three pianos rate equally. To test the null hypothesis, the Friedman test is used on the ratings of the ten piano players.

Table 8.2 Typical tests for statistical problems, for nominal and ordinal data

No. of groups compared	Independent samples	Paired samples
Groups of nominal data		
1	One-sample *t*-test or Wilcoxon signed rank sum test	–
2 or more	Fisher's exact test or chi-square test	McNemar's test
Groups of ordinal data		
2	Mann–Whitney U test	Wilcoxon signed rank test
3 or more	Kruskal–Wallis test	Friedman test
Groups of continuous data		
2	Student's *t*-test or Mann–Whitney test	Paired *t*-test or Wilcoxon signed-rank sum test
3 or more	ANOVA or Kruskal–Wallis test	Repeated measures ANOVA or Friedman test

Note that the tests for comparing one group to a fixed value are the same as comparing two groups with paired samples

It may be worth mentioning that in a blog Thom Baguley suggested that in cases where one-way repeated measures ANOVA is not appropriate, rank transformation followed by ANOVA will provide a more robust test with greater statistical power than the Friedman test.
(http://www.r-bloggers.com/beware-the-friedman-test/)

8.4.2 Hypothetical Examples

1 group, nominal	Average calory intake. E.g. "Do our children eat more than they should?"
1 group, ordinal	Sequence of giant-planets. E.g. "In our solar system, are giant planets further out than average in the sequence of planets?"
2 groups, nominal	male/female, blond-hair/black-hair. E.g. "Are females more blond than males?"
2 groups, nominal, paired	2 labs, analysis of blood samples. E.g. "Does the blood analysis from Lab1 indicate more infections than the analysis from Lab2?"
2 groups, ordinal	Jamaican/American, ranking 100 m sprint. E.g. "Are Jamaican sprinters more successful than American sprinters?"
2 groups, ordinal, paired	sprinters, before/after diet. E.g. "Does a chocolate diet make sprinters more successful?"
3 groups, ordinal	single/married/divorces, ranking 100 m sprint. E.g. "Does the marital status have an effect on the success of sprinters?"
3 groups, ordinal, paired	sprinters, before/after diet. E.g. "Does a rice diet make Chinese sprinters more successful?"
2 groups, continuous	male/female, IQ. E.g. "Are women more intelligent than men?"
2 groups, continuous, paired	male/female, looking at diamonds. E.g. "Does looking at sports cars raise the male heart-beat more than the female?
3 groups, continuous	Tyrolians, Viennese, Styrians; IQ. E.g. "Are Tyrolians smarter than people from other Austrian federal states?"
3 groups, continuous, paired	Tyrolians, Viennese, Styrians; looking at mountains. E.g. "Does looking at mountains raise the heart-beat of Tyrolians more than those of other people?"

8.5 Exercises

8.1 One or Two Groups

- **Paired *t*-Test** and **Wilcoxon signed rank sum test**
  The daily energy intake from 11 healthy women is [5260., 5470., 5640., 6180., 6390., 6515., 6805., 7515., 7515., 8230., 8770.] kJ.
  Is this value significantly different from the recommended value of 7725?
  (Correct answer: yes, $p_{ttest} = 0.018$, and $p_{Wilcoxon} = 0.026$.)
- ***t*-Test of independent samples**
  In a clinic, 15 lazy patients weigh [76, 101, 66, 72, 88, 82, 79, 73, 76, 85, 75, 64, 76, 81, 86.] kg, and 15 sporty patients weigh [64, 65, 56, 62, 59, 76, 66, 82, 91, 57, 92, 80, 82, 67, 54] kg.
  Are the lazy patients significantly heavier? (Correct answer: yes, $p = 0.045$.)
- **Normality test**
  Are the two data sets normally distributed? (Correct answer: yes, they are.)
- **Mann–Whitney test**
  Are the lazy patients still heavier, if you check with the Mann–Whitney test? (Correct answer: no, $p = 0.077$. Note, however, that the answer would be "yes" for a one-sided test!)

8.2 Multiple Groups

The following example is taken from the really good, but somewhat advanced book by A.J. Dobson: “An Introduction to Generalized Linear Models”:

- **Get the data**
  The file *Data/data_others/Table 6.6 Plant experiment.xls*, which can also be found on https://github.com/thomas-haslwanter/statsintro/tree/master/Data/data_others, contains data from an experiment with plants in three different growing conditions. Read the data into *Python*. Hint: use the module *xlrd*.
- **Perform an ANOVA**
  Are the three groups different? (Correct answer: yes, they are.)
- **Multiple Comparisons**
  Using the Tukey test, which of the pairs are different? (Correct answer: only TreamtmentA and TreatmentB differ.)
- **Kruskal–Wallis**
  Would a nonparametric comparison lead to a different result? (Correct answer: no.)

Chapter 9
Tests on Categorical Data

In a data sample the number of data falling into a particular group is called the *frequency*, so the analysis of categorical data is the *analysis of frequencies*. When two or more groups are compared the data are often shown in the form of a *frequency table*, sometimes also called *contingency table*. For example, Table 9.1 gives the number of right/left-handed subjects, *contingent* on the subject being male or female.

If we have only one *factor* (i.e., a table with only one row), the analysis options are somewhat limited (Sect. 9.2.1). In contrast, a number of statistical tests exist for the analysis of frequency tables:

Chi-square test This is the most common type. It is a hypothesis test, which checks if the entries in the individual cells in a frequency table (e.g., Table 9.1) all come from the same distribution. In other words, it checks the null hypothesis H_0 that the results are independent of the row or column in which they appear. The alternative hypothesis H_a does not specify the type of association, so close attention to the data is required to interpret the information provided by the test.

Fisher's Exact Test While the chi-square test is approximate, the *Fisher's Exact Test* is an exact test. It is computationally more expensive and intricate than the chi-square test, and was originally used only for small sample numbers. However, in general it is now the more advisable test to use.

McNemar's Test This is a *matched pair test* for 2×2 tables. For example, if you want to see if two doctors obtain comparable results when checking (the same) patients, you would use this test.

Cochran's Q Test Cochran's Q test is an extension to the McNemar's test for related samples that provides a method for testing for differences between three or more matched/paired sets of frequencies or proportions. For example, if you have exactly

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_9

Table 9.1 Example of a frequency table. (The row and column totals are always written in italics.)

	Right handed	Left handed	*Total*
Males	43	9	*52*
Females	44	4	*48*
Total	*87*	*13*	*100*

the same samples analyzed by 3 different laboratories, and you want to check if the results are statistically equivalent, you would use this test.

9.1 One Proportion

9.1.1 Confidence Intervals

If we have one sample group of data, we can check if the sample is representative of the standard population. To do so, we have to know the proportion p of the characteristic in the standard population. The occurrence of a characteristic in a group of n people is described by the binomial distribution, with $mean = p * n$. The standard error of samples with this characteristic is given by

$$se(p) = \sqrt{p(1-p)/n} \tag{9.1}$$

and the corresponding 95 % confidence interval is

$$ci = mean \pm se * t_{n,0.025}$$

Thereby $t_{n,0.025}$ can be calculated as the inverse survival function (ISF) of the t-distribution, at the value 0.025. If the data lie outside this confidence interval, they are *not* representative of the population.

9.1.2 Explanation

The innocent looking equation 9.1 is more involved than it seems at first:

If we have n independent samples from a binomial distribution $B(k, p)$, the variance of their sample mean is

$$var\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\sum_{i=1}^{n} var(X_i) = \frac{n\, var(X_i)}{n^2} = \frac{var(X_i)}{n} = \frac{kpq}{n}$$

where $q = 1 - p$. This follows since

1. $var(cX) = c^2 var(X)$, for any random variable, X, and any constant c.
2. the variance of a sum of independent random variables equals the sum of the variances.

The standard error of the sample mean $\bar{X}$ is the square root of the variance: $\sqrt{\frac{kpq}{n}}$. Therefore,

- When $k = n$, we get $se = \sqrt{pq}$.
- When $k = 1$, and the Binomial variables are just Bernoulli trials, the standard error is given by $se = \sqrt{\frac{pq}{n}}$.

9.1.3 Example

For example, let us look at incidence and mortality for breast cancer, and try to answer the following two questions: among the students at the Upper Austria University of Applied Sciences, how many occurrences of breast cancer should we expect per year? And how many of the female FH-students will probably die from breast cancer at the end of their life?

We know that:

- the Upper Austrian University of Applied Sciences has about 5000 students, about half of which are female.
- breast cancer hits predominantly women.
- the *incidence* of breast cancer in the age group 20–30 is about 10, where *incidence* is typically defined as the new occurrences of a disease per year per 100,000 people.
- 3.8 % of all women die of cancer.

From these pieces of information, we can obtain the following parameters for our calculations:

- $n = 2500$
- $p_{incidence} = 10/100{,}000$
- $p_{mortality} = 3.8/100$.

The 95 % confidence interval for the incidence of breast cancer is −0.7–1.2, and for the number of deaths 76–114. So we expect that every year most likely none or one of the FH-students will be diagnosed with breast cancer; but between 76 and 114 of the current female students will eventually die from this disease.

9.2 Frequency Tables

If the data can be organized in a set of categories, and they are given as *frequencies*, i.e., the total number of samples in each category (not as percentages), the tests described in this section are appropriate for the data analysis.

Many of these tests analyze the *deviation from an expected value*. Since the chi-square distribution characterizes the variability of data (in other words, their deviation from a mean value), many of these tests refer to this distribution, and are accordingly termed *chi-square tests*.

When n is the total number of observations included in the table, the expected value for each cell in a two-way table is

$$expectedFrequency = \frac{RowTotal * ColumnTotal}{n} \tag{9.2}$$

Assume that we have observed absolute frequencies o_i and expected absolute frequencies e_i. Under the null hypothesis all the data come from the same population, and the test statistic

$$V = \sum_i \frac{(o_i - e_i)^2}{e_i} \approx \chi_f^2 \tag{9.3}$$

follows a chi square distribution with f degrees of freedom. i might denote a simple index running from $1, \ldots, f$ or a multi-index $(i_1, \ldots, i_n)$ running from $(1, \ldots, 1)$ to $(f_1, \ldots, f_n)$, and $f = \sum_{i=1}^{n} f_i$.

9.2.1 One-Way Chi-Square Test

For example, assume that you go hiking with your friends. Every evening, you draw lots who has to do the washing up. But at the end of the trip, you seem to have done most of the work:

You	Peter	Hans	Paul	Mary	Joe
10	6	5	4	5	3

You expect that there has been some foul play, and calculate how likely it is that this distribution came up by chance. The

$$expectedFrequency = \frac{n_{total}}{n_{people}} \tag{9.4}$$

is 5.5. The likelihood that this distribution came up by chance is

```
V, p = stats.chisquare(data)
print(p)
>>> 0.373130385949
```

In other words, you doing a lot of the washing up really could have been by chance!

9.2.2 Chi-Square Contingency Test

When data can be arranged in rows and columns, we can check if the numbers in the individual columns are contingent on the row value. For this reason, this test is sometimes called *contingency test*. Using the example in Table 9.1, if females were more left-handed than males, the ratio $\frac{left\text{-}handed}{right\text{-}handed}$ would be contingent on the row and larger for females than for males.

The chi-square contingency test is based on a test statistic that measures the divergence of the observed data from the values that would be expected under the null hypothesis of no association (e.g., Table 9.2).

a) Assumptions

The test statistic V is approximately χ^2 distributed, if

- for all absolute expected frequencies e_i holds: $e_i \geq 1$, and
- for at least 80 % of the absolute expected frequencies e_i holds: $e_i \geq 5$.

For small sample numbers, corrections should be made for some bias that is caused by the use of the continuous chi-squared distribution, while the frequencies are by definition integers. This correction is referred to as *Yates correction*.

b) Degrees of Freedom

The degrees of freedom (DOF) can be computed by the numbers of absolute observed frequencies which can be chosen freely. For example, only one cell of a 2×2 table with the sums at the side and bottom needs to be filled, and the others

Table 9.2 Corresponding *expected values* for Table 9.1

	Right handed	Left handed	*Total*
Males	45.2	6.8	*52*
Females	41.8	6.2	*48*
Total	*87*	*13*	*100*

Table 9.3 General structure of 2 × 2 frequency tables

		B		
		0	1	*Total*
A	0	a	b	$a+b$
	1	c	d	$c+d$
Total		$a+c$	$b+d$	$N=a+b+c+d$

can be found by subtraction. In general, an $r \times c$ table with r rows and c columns has

$$df = (r-1) \times (c-1) \tag{9.5}$$

degrees of freedom. We know that the sum of absolute expected frequencies is

$$\sum_i o_i = n \tag{9.6}$$

We might have to subtract from the number of degrees of freedom the number of parameters we need to estimate from the sample, since this implies further relationships between the observed frequencies.

c) Example 1

The *Python* command *stats.chi2_contingency* returns the following list:
(χ^2-value, p*-value, degrees-of-freedom, expected values).*

```
data = np.array([[43,9],
                 [44,4]])
V, p, dof, expected = stats.chi2_contingency(data)
print(p)
>>> 0.300384770391
```

For the example data in Table 9.1, the results are ($\chi^2 = 1.1, p = 0.3, df = 1$). In other words, there is no indication that there is a difference in left-handed people vs right-handed people between males and females.

Note: These values assume the default setting, which uses the *Yates correction*. Without this correction, i.e., using Eq. 9.3, the results are $\chi^2 = 1.8, p = 0.18$.

d) Example 2

The chi-square test can be used to generate a "quick and dirty" test of normality, e.g.

H_0 : The random variable X is symmetrically distributed versus
H_1 : the random variable X is not symmetrically distributed.

We know that in case of a symmetrical distribution the arithmetic mean $\bar{x}$ and median should be nearly the same. So a simple way to test this hypothesis would be to count how many observations are less than the mean (n_-) and how many observations are larger than the arithmetic mean (n_+). If mean and median are the same, then 50 % of the observation should be smaller than the mean and 50 % should be larger than the mean. It holds

$$V = \frac{(n_- - n/2)^2}{n/2} + \frac{(n_+ - n/2)^2}{n/2} \approx \chi^2_1 \tag{9.7}$$

e) Comments

The chi-square test is a pure hypothesis test. It tells you if the observed frequency can be due to a random sample selection from a single population. A number of different expressions have been used for chi-square tests, which are due to the original derivation of the formulas (from the time before computers were pervasive). Expression such as 2×2 *tables*, *r–c tables*, or *chi-square test of contingency* all refer to frequency tables and are typically analyzed with chi-square tests.

9.2.3 Fisher's Exact Test

If the requirement that 80 % of cells should have expected values of at least 5 is not fulfilled, *Fisher's exact test* should be used. This test is based on the observed row and column totals. The method consists of evaluating the probability associated with all possible 2×2 tables which have the same row and column totals as the observed data, making the assumption that the null hypothesis (i.e., that the row and column variables are unrelated) is true. In most cases, Fisher's exact test is preferable to the chi-square test. But until the advent of powerful computers, it was not practical. You should use it up to approximately 10–15 cells in the frequency tables. It is called "exact" because the significance of the deviation from a null hypothesis can be calculated exactly, rather than relying on an approximation that becomes exact in the limit as the sample size grows to infinity, as with many statistical tests.

In using the test, you have to decide if you want to use a one-tailed test or a two-tailed test. The former one looks for the probability to find a distribution as extreme as or more extreme than the observed one. The latter one (which is the default in *Python*) also considers tables as extreme in the opposite direction.

Note: The python command `stats.fisher_exact` returns by default the *p*-value for *finding a value as extreme or more extreme than the observed one*. According to Altman (1999), this is a reasonable approach, although not all statisticians agree on that point.

Fig. 9.1 First milk, then tea (*left*)—or first tea, then milk (*right*): Could you taste the difference? (Published with the kind permission of © Thomas Haslwanter 2015. All rights reserved.)

a) Example: "A Lady Tasting Tea"

R.A. Fisher[1] was one of the founding fathers of modern statistics. One of his early experiments, and perhaps the most famous, was to test an English lady's claim that she could tell whether milk was poured before tea or not (Fig. 9.1). Here is an account of the seemingly trivial event that had the most profound impact on the history of modern statistics, and hence, arguably, modern quantitative science (Box 1978).

> Already, quite soon after he had come to Rothamstead, his presence had transformed one commonplace tea time to an historic event. It happened one afternoon when he drew a cup of tea from the urn and offered it to the lady beside him, Dr. B. Muriel Bristol, an algologist. She declined it, stating that she preferred a cup into which the milk had been poured first. "Nonsense," returned Fisher, smiling, "Surely it makes no difference." But she maintained, with emphasis, that of course it did. From just behind, a voice suggested, "Let's test her." It was William Roach who was not long afterward to marry Miss Bristol. Immediately, they embarked on the preliminaries of the experiment, Roach assisting with the cups and exulting that Miss Bristol divined correctly more than enough of those cups into which tea had been poured first to prove her case.
>
> Miss Bristol's personal triumph was never recorded, and perhaps Fisher was not satisfied at that moment with the extempore experimental procedure. One can be sure, however, that even as he conceived and carried out the experiment beside the trestle table, and the onlookers, no doubt, took sides as to its outcome, he was thinking through the questions it raised.

The real scientific significance of this experiment is in these questions. These are, allowing incidental particulars, the questions one has to consider before designing an experiment. We will look at these questions as pertaining to the "lady tasting tea," but you can imagine how these questions should be adapted to different situations.

- *What should be done about chance variations in the temperature, sweetness, and so on?* Ideally, one would like to make all cups of tea identical except for the order of pouring milk first or tea first. But it is never possible to control all of the ways in which the cups of tea can differ from each other. If we cannot

[1] Adapted from Stat Labs: Mathematical statistics through applications by D. Nolan and T. Speed, Springer-Verlag, New York, 2000.

control these variations, then the best we can do—we do mean the "best"—is by randomization.

- *How many cups should be used in the test? Should they be paired? In what order should the cups be presented?* The key idea here is that the number and ordering of the cups should allow a subject ample opportunity to prove his or her abilities and keep a fraud from easily succeeding at correctly discriminating the order of pouring in all the cups of tea served.
- *What conclusion could be drawn from a perfect score or from one with one or more errors?* If the lady is unable to discriminate between the different orders of pouring, then by guessing alone, it should be highly unlikely for that person to determine correctly which cups are which for all of the cups tested. Similarly, if she indeed possesses some skill at differentiating between the orders of pouring, then it may be unreasonable to require her to make no mistakes so as to distinguish her ability from a pure guesser.

An actual scenario described by Fisher and told by many others as the "lady tasting tea" experiment is as follows.

- For each cup, we record the order of actual pouring and what the lady says the order is. We can summarize the result by a table like this:

		Order of actual pouring		
		Tea first	Milk first	*Total*
Lady says	Tea first	a	b	$a+b$
	Milk first	c	d	$c+d$
	Total	$a+c$	$b+d$	n

Here n is the total number of cups of tea made. The number of cups where tea is poured first is $a+c$ and the lady classifies $a+b$ of them as tea first. Ideally, if she can taste the difference, the counts b and c should be small. On the other hand, if she cannot really tell, we would expect a and c to be about the same.

- Suppose now that to test the lady's abilities, eight cups of tea are prepared, four tea first, four milk first, and she is informed of the design (that there are four cups milk first and four cups tea first). Suppose also that the cups are presented to her in random order. Her task then is to identify the four cups milk first and four cups tea first.

  This design fixes the row and column totals in the table above to be 4 each. That is,

$$a+b=a+c=c+d=b+d=4.$$

With these constraints, when any one of a, b, c, d is specified, the remaining three are uniquely determined:

$$b = 4 - a,\ c = 4 - a,\ \text{and}\ d = a$$

In general, for this design, no matter how many cups (n) are served, the row total $a + b$ will equal $a + c$ because the subject knows how many of the cups are "tea first" (or one kind as supposed to the other). So once a is given, the other three counts are specified.

- We can test the discriminating skill of the lady, if any, by randomizing the order of the cups served. If we take the position that she has no discriminating skill, then the randomization of the order makes the four cups chosen by her as tea first equally likely to be any four of the eight cups served. There are $\binom{8}{4} = 70$ (in *Python*, choose `scipy.misc.comb(8,4,exact=True)`) possible ways to classify four of the eight cups as "tea first." If the subject has no ability to discriminate between two preparations, then by the randomization, each of these 70 ways is equally likely. Only one of 70 ways leads to a completely correct classification. So someone with no discriminating skill has 1/70 chance of making no errors.
- It turns out that, if we assume that she has no discriminating skill, the number of correct classifications of tea first ("a" in the table) has a "hypergeometric" probability distribution (`hd=stats.hypergeom(8,4,4)` in *Python*). There are five possibilities: 0, 1, 2, 3, 4 for a and the corresponding probabilities (and *Python* commands for computing the probabilities) are tabulated below.

Number of correct calls	*Python* command	Probability
0	hd.pmf(0)	1/70
1	hd.pmf(1)	16/70
2	hd.pmf(2)	36/70
3	hd.pmf(3)	16/70
4	hd.pmf(4)	1/70

- With these probabilities, we can compute the p-value for the test of the hypothesis that the lady cannot tell between the two preparations. Recall that the p-value is the probability of observing a result as extreme or more extreme than the observed result assuming the null hypothesis. If she makes all correct calls, the p-value is 1/70 and if she makes one error (three correct calls) then the p-value is $1/70 + 16/70 \sim 0.24$.

The test described above is known as "Fisher's exact test," and the implementation is quite trivial:

```
oddsratio, p = stats.fisher_exact(obs, alternative='greater')
```

where *obs* is the matrix containing the observations.

9.2.4 McNemar's Test

Although the McNemar test bears a superficial resemblance to a test of categorical association, as might be performed by a 2×2 chi-square test or a 2×2 Fisher exact probability test, it is doing something quite different. The test of association examines the relationship that exists among the cells of the table. The McNemar test examines the difference between the proportions that derive from the marginal sums of the table (see Table 9.3): $p_A = (a + b)/N$ and $p_B = (a + c)/N$. The question in the McNemar test is: do these two proportions, p_A and p_B, significantly differ? And the answer it receives must take into account the fact that the two proportions are not independent. The correlation of p_A and p_B is occasioned by the fact that both include the quantity a in the upper left cell of the table.

McNemar's test can be used for example in studies in which patients serve as their own control, or in studies with "before and after" design.

a) Example

In the following example, a researcher attempts to determine if a drug has an effect on a particular disease. Counts of individuals are given in the table, with the diagnosis (disease: present or absent) before treatment given in the rows, and the diagnosis after treatment in the columns (Table 9.4). The test requires the same subjects to be included in the before-and-after measurements (matched pairs).

In this example, the null hypothesis of "marginal homogeneity" would mean there was no effect of the treatment. From the above data, the McNemar test statistic with Yates's continuity correction is

$$\chi^2 = \frac{(|b - c| - correctionFactor)^2}{b + c}. \tag{9.8}$$

where χ^2 has a chi-squared distribution with one degree of freedom. For small sample numbers the *correctionFactor* should be 0.5 (*Yates's correction*) or 1.0 (*Edward's correction*). (For $b + c < 25$, the binomial calculation should be performed, and indeed, most software packages simply perform the binomial calculation in all cases, since the result then is an exact test in all cases.) Using Yates's correction, we get

$$\chi^2 = \frac{(|121 - 59| - 0.5)^2}{121 + 59} \tag{9.9}$$

Table 9.4 McNemar's test: example

	After: present	After: absent	*Total*
Before: present	101	121	*222*
Before: absent	59	33	*92*
Total	*160*	*154*	*314*

The resulting value is 21.01, which is extremely unlikely from the distribution implied by the null hypothesis ($p_b = p_c$). Thus the test provides strong evidence to reject the null hypothesis of no treatment effect.

To implement the McNemar's test in *Python*, use

```
from statsmodels.sandbox.stats.runs import mcnemar

obs = [[a,b], [c, d]]
chi2, p = mcnemar(obs)
```

with *obs* again representing the observation matrix.

9.2.5 *Cochran's* Q *Test*

Cochran's Q test is a hypothesis test where the response variable can take only two possible outcomes (coded as 0 and 1). It is a nonparametric statistical test to verify if k treatments have identical effects. Cochran's Q test should not be confused with *Cochran's C test*, which is a variance outlier test.

a) Example

Twelve subjects are asked to perform three tasks. The outcome of each task is *success* or *failure*. The results are coded 0 for *failure* and 1 for *success*. In the example, subject 1 was successful in task 2, but failed tasks 1 and 3 (see Table 9.5).

Table 9.5 Cochran's Q test: success or failure for 12 subjects on 3 tasks

Subject	Task 1	Task 2	Task 3
1	0	1	0
2	1	1	0
3	1	1	1
4	0	0	0
5	1	0	0
6	0	1	1
7	0	0	0
8	1	1	0
9	0	1	0
9	0	1	0
10	0	1	0
11	0	1	0
12	0	1	0

The null hypothesis for the Cochran's Q test is that there are no differences between the variables. If the calculated probability p is below the selected significance level, the null hypothesis is rejected, and it can be concluded that the proportions in at least 2 of the variables are significantly different from each other. For our example (Table 9.5), the analysis of the data provides *Cochran's Q = 8.6667* and a significance of $p = 0.013$. In other words, at least one of the three tasks is easier or harder than the others.

To implement the Cochran's Q test in *Python*, use

```
from statsmodels.sandbox.stats.runs import cochrans_q

obs = [[a,b], [c, d]]
q_stat, p = cochrans_q(obs)
```

Code: "ISP_compGroups.py"[2]: Analysis of categorical data: once the correct test is selected, the computational steps are trivial.

9.3 Exercises

9.1 Fisher's Exact Test: The Tea Experiment

At a party, a lady claimed to be able to tell whether the tea or the milk was added first to a cup. Fisher proposed to give her eight cups, four of each variety, in random order. One could then ask what the probability was for her getting the number she got correct, but just by chance.

The experiment provided the Lady with eight randomly ordered cups of tea—four prepared by first adding milk, four prepared by first adding the tea. She was to select the four cups prepared by one method. (This offered the Lady the advantage of judging cups by comparison.)

The null hypothesis was that the Lady had no such ability. (In the real, historical experiment, the lady got all eight cups correct.)

- Calculate if the claim of the Lady is supported if she gets three out of the four pairs correct.
  (Correct answer: No. If she gets three correct, that chance that a selection of "three or greater" was random is 0.243. She needs to get all four correct, if we set the rejection threshold at 0.05.)

[2] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/09_TestsCategoricalData/compGroups .

9.2 Chi2 Contingency Test (1 DOF)

A test of the effect of a new drug on the heart rate has yielded the following results:

	Heart rate		
	Increased	NOT-increased	*Total*
Treated	36	14	*50*
Not treated	30	25	*55*
Total	*66*	*39*	*105*

- Does the drug affect the heart rate? (Correct answer: no.)
- What would be the result if the response in one of the not-treated persons would have been different? Perform this test with and without the Yates-correction. (Correct answer:
  without Yates correction: yes, $p = 0.042$
  with Yates correction: no, $p = 0.067$.)

	Heart rate		
	Increased	NOT-increased	Total
Treated	36	14	*50*
Not treated	29	26	*55*
Total	*65*	*40*	*105*

9.3 One Way Chi2-Test (>1 DOF)

The city of Linz wants to know if people want to build a long beach along the Danube. They interview local people, and decide to collect 20 responses from each of the five age groups: (<15, 15–30, 30–45, 45–60, >60)

The questionnaire states: *"A beach-side development will benefit Linz."* and the possible answers are

1	2	3	4
Strongly agree	Agree	Disagree	Strongly disagree

The city council wants to find out if the age of people influenced feelings about the development, particularly of those who felt negatively (i.e., "disagreed" or "strongly disagreed") about the planned development.

Age group (type)	Frequency of negative responses (observed values)
<15	4
15–30	6
30–45	14
45–60	10
>60	16

The categories seem to show large differences of opinion between the groups.

- Are these differences significant? (Correct answer: yes, $p = 0.034$.)
- How many degrees of freedom does the resulting analysis have? (Correct answer: 4.)

9.4 McNemar's Test

In a lawsuit regarding a murder the defense uses a questionnaire to show that the defendant is insane. As a result of the questionnaire, the accused claims "not guilty by reason of insanity."

In reply, the state attorney wants to show that the questionnaire does not work. He hires an experienced neurologist, and presents him with 40 patients, 20 of whom have completed the questionnaire with an "insane" result, and 20 with a "sane" result. When examined by the neurologist, the result is mixed: 19 of the "sane" people are found sane, but 6 of the 20 "insane" people are labeled as sane by the expert.

	Sane by expert	Insane by expert	Total
Sane	19	1	*20*
Insane	6	14	*20*
Total	*22*	*18*	*40*

- Is this result significantly different from the questionnaire? (Correct answer: no.)
- Would the result be significantly different, if the expert had diagnosed all "sane" people correctly? (Correct answer: yes.)

Chapter 10
Analysis of Survival Times

When analyzing survival times, different problems come up than the ones discussed so far. One question is how to deal with subjects dropping out of a study. For example, assume that we test a new cancer drug. While some subjects die, others may believe that the new drug is not effective, and decide to drop out of the study before the study is finished.

The term used for this type of study is *survival analysis*, although the same methods are also used to analyze similar problems in other areas. For example, these techniques can be used to investigate how long a machine lasts before it breaks down, or how long people subscribe to mailing lists (where the "death" corresponds to unsubscribing from a mailing list).

10.1 Survival Distributions

The Weibull distribution is often used for modeling reliability data or survival data. Since it was first identified by Fréchet (in 1927), but described in detail by Weibull (in 1951), it is sometimes also found under the name Fréchet distribution.

In `scipy.stats`, the Weibull distribution is available under the name `weibull_min`, or equivalently `frechet_r` for *Fréchet right*. (The complementary `weibull_max`, also called `frechet_l` for *Fréchet left*, is simply mirrored about the origin.)

The Weibull distribution is characterized by a shape parameter, the *Weibull Modulus k* (see also Sect. 6.5.2). All *Python*-distributions offer a convenient method

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_10

fit, which allows quick fitting of the distribution parameters:

Listing 10.1 L10_1_WeibullDemo.py

```
''' Example of fitting the Weibull modulus. '''

# author: Thomas Haslwanter, date: Jun-2015

# Import standard packages
import matplotlib.pyplot as plt
import scipy as sp
from scipy import stats

# Generate some sample data, with a Weibull modulus of 1.5
WeibullDist = stats.weibull_min(1.5)
data = WeibullDist.rvs(500)

# Now fit the parameter
fitPars = stats.weibull_min.fit(data)

# Note: fitPars contains (WeibullModulus, Location, Scale)
print('The fitted Weibull modulus is {0:5.2f}, compared to
    the exact value of 1.5 .'.format(fitPars[0]))
```

10.2 Survival Probabilities

For the statistical analysis of survival data, Cam Davidson-Pilon has developed the *Python* package *lifelines*. It can be installed with

```
pip install lifelines
```

A very extensive documentation, which also includes an introduction to survival analysis and survival regression modeling, is available under http://lifelines.readthedocs.org/.

10.2.1 Censorship

The difficulty of using data for survival analysis is that at the end of a study, many individuals may be still "alive." In statistics, the expression for measurement values that are only partially known is *censorship* or *censoring*. As an example let us consider a mailing list, whose subscribers fall into two subgroups. Group One quickly gets tired of the emails, and unsubscribes after three months. Group Two

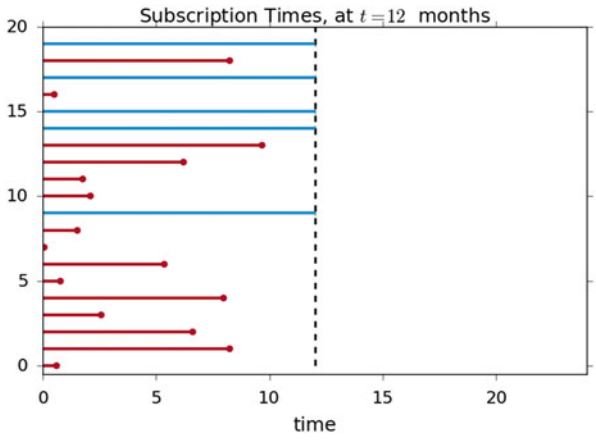


Fig. 10.1 Dummy results from a study on the subscription behavior of a mailing list

enjoys it, and typically subscribes for one and a half years (Fig. 10.1). We perform a study which lasts one year, and want to investigate the average subscription duration:

Code: "ISP_lifelinesDemo.py"[1]: Graphical representation of lifelines.

The red lines denote the subscription time of individuals where the dropout event has been observed, and the blue lines denote the subscription time of the right-censored individuals (dropouts have not been observed). If we are asked to estimate the average subscription time of our population, and we naively decided not to include the right-censored individuals, it is clear that we would be severely underestimating the true average subscription time.

A similar, further problem occurs if some subjects increase their privacy-settings in the middle of the study, i.e., they forbid us to monitor them before the study is over. Also these data are right-censored data.

10.2.2 Kaplan–Meier Survival Curve

A clever way to deal with these problems is the description of such data with the Kaplan–Meier curve, described in detail in Altman (1999). First, the time is subdivided into small periods. Then the likelihood is calculated that a subject

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/10_SurvivalAnalysis/lifelinesDemo.

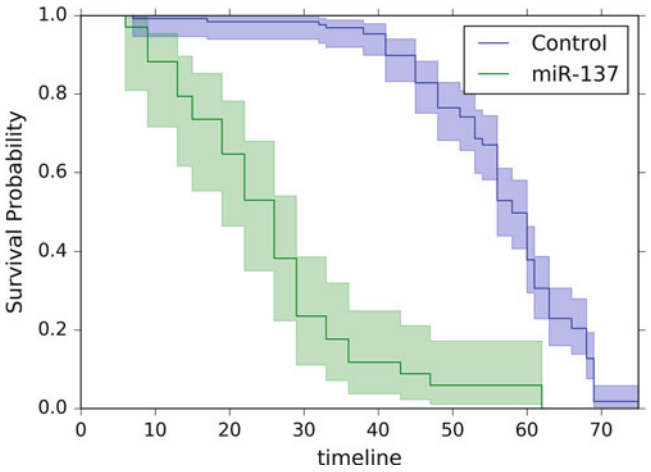


Fig. 10.2 Survival probability of two groups of Drosophila flies. The *shaded areas* indicate the 95 % confidence intervals

survives a given period. The survival probability is given by

$$p_k = p_{k-1} * \frac{r_k - f_k}{r_k} \tag{10.1}$$

where p_k is the probability to survive period k; r_k is the number of subjects still at risk (i.e., still being followed up) immediately before the kth day, and f_k is the number of observed failures on the day k. The curve describing the resulting survival probability is called *life table*, *survival curve*, or *Kaplan–Meier curve* (see Fig. 10.2).

The following data show results from a study with fruitflies of the genus Drosophila. The numbers give the genotypes of the flies and the number of days survived. Since we work with flies, we don't need to worry about left-censoring: we know the birth date of all flies. We do have issues with accidentally killing some or if some escape. These would be right-censored as we do not actually observe their death due to "natural" causes.

Listing 10.2 L10_2_lifelinesSurvival.py

```
''' Graphical representation of survival curves, and
    comparison of two
curves with logrank test.
"miR-137" is a short non-coding RNA molecule that functions
    to regulate
the expression levels of other genes.
'''
# author: Thomas Haslwanter, date: Jun-2015

# Import standard packages
```

```
import matplotlib.pyplot as plt

# additional packages
import sys
sys.path.append(r'..\Quantlets\Utilities')
import ISP_mystyle

from lifelines.datasets import load_waltons
from lifelines import KaplanMeierFitter
from lifelines.statistics import logrank_test

# Set my favorite font
ISP_mystyle.setFonts(18)

# Load and show the data
df = load_waltons() # returns a Pandas DataFrame

print(df.head())
'''
    T  E    group
0   6  1  miR-137
1  13  1  miR-137
2  13  1  miR-137
3  13  1  miR-137
4  19  1  miR-137
'''

T = df['T']
E = df['E']

groups = df['group']
ix = (groups == 'miR-137')

kmf = KaplanMeierFitter()

kmf.fit(T[~ix], E[~ix], label='control')
ax = kmf.plot()

kmf.fit(T[ix], E[ix], label='miR-137')
kmf.plot(ax=ax)

plt.ylabel('Survival Probability')
outFile = 'lifelines_survival.png'
ISP_mystyle.showData(outFile)

# Compare the two curves
results = logrank_test(T[ix], T[~ix], event_observed_A=E[ix],
     event_observed_B=E[~ix])
results.print_summary()
```

This code produces the following output:

```
Results
   t 0: -1
   alpha: 0.95
   df: 1
   test: logrank
   null distribution: chi squared

p-value _|_ test statistic _|_ test result _|_ is significant
 0.00000 |          122.249 |  Reject Null  |       True
```

Note that the survival curve changes only when a "failure" occurs, i.e., when a subject dies. *Censored* entries, describing either when a subject drops out of the study or when the study finishes, are taken into consideration at the "failure" times, but otherwise do not affect the survival curve.

10.3 Comparing Survival Curves in Two Groups

The most common test for comparing independent groups of survival times is the *logrank test*. This test is a nonparametric hypothesis test, testing the probability that both groups come from the same underlying population. To explore the effect of different variables on survival, more advanced methods are required. For example, the *Cox Regression model*, also called *Cox Proportional Hazards model* introduced by Cox in 1972 is used widely when it is desired to investigate several variables at the same time.

These tests, as well as other models for the analysis of survival data, are available in the *lifelines* package, and are easy to apply once you know how to use *Python*.

Part III
Statistical Modeling

While hypothesis tests can decide if two or more sets of data samples come from the same population or from different ones, they cannot quantify the strength of a relationship between two or more variables. This question, which also includes the quantitative prediction of variables, is addressed in the third part of this book. The basic algebraic tools that come with *Python* may suffice for simple problems like line-fits or the determination of correlation coefficients. But a number of packages significantly extend the power of *Python* for statistical data analysis and modeling. This part will show applications of the following packages:

- statsmodels
- PyMC
- scikit-learn
- scikits.bootstrap

In addition, a very short introduction to generalized linear models is included. The section on logistic regression has also been placed in this part, as logistic regression is a generalized linear model. An introduction to Bayesian statistics, including a practical example of a running Markov-chain–Monte-Carlo simulation, rounds off the chapter.

Chapter 11
Linear Regression Models

There is a substantial difference in approach between hypothesis tests and statistical modeling. With hypothesis tests, one typically starts out with a null hypothesis. Based on the question and the data, one then selects the appropriate statistical test as well as the desired significance level, and either accepts or rejects the null hypothesis.

In contrast, statistical modeling typically involves a more interactive analysis of the data. One starts out with a visual inspection of the data, looking for correlations and/or relationships. Based on this first inspection, a statistical model is selected that may describe the data. In simple cases, the relationship in the data can be described with a linear model

$$y = k * x + d.$$

Next

- the model parameters (e.g., k and d) are determined,
- the quality of the model is assessed,
- and the residuals (i.e., the remaining errors) are inspected, to check if the proposed model has missed essential features in the data.

If the residuals are too large, or if the visual inspection of the residuals shows outliers or suggests another model, the model is modified. This procedure is repeated until the results are satisfactory.

This chapter describes how to implement and solve linear regression models in *Python*. The resulting model parameters are discussed, as well as the assumptions of the models and interpretations of the model results. Since *bootstrapping* can be helpful in the evaluation of some models, the final section in this chapter shows a *Python* implementation of a bootstrapping example.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_11

11.1 Linear Correlation

For two related variables, the *correlation* measures the association between the two variables. In contrast, a *linear regression* is used for the prediction of the value of one variable from another.

11.1.1 Correlation Coefficient

The *correlation coefficient* between two variables answers the question: "Are the two variables related? That is, if one variable changes, does the other also change?" If the two variables are normally distributed, the standard measure of determining the correlation coefficient, often ascribed to Pearson, is

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{11.1}$$

With the sample covariance s_{xy} defined as

$$s_{xy} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{n-1} \tag{11.2}$$

and s_x, s_y the sample standard deviations of the x and y values, respectively, Eq. 11.1 can also be written as

$$r = \frac{s_{xy}}{s_x \cdot s_y}. \tag{11.3}$$

Pearson's correlation coefficient, sometimes also referred to as *population correlation coefficient* or *sample correlation*, can take any value from -1 to $+1$. Examples are given in Fig. 11.1. Note that the formula for the correlation coefficient is symmetrical between x and y—which is not the case for linear regression!

11.1.2 Rank Correlation

If the data distribution is not normal, a different approach is necessary. In that case one can rank the set of data for each variable and compare the orderings. There are two commonly used methods of calculating the rank correlation.

Spearman's ρ is exactly the same as the Pearson correlation coefficient r, but calculated on the ranks of the observations and not on the original numbers.

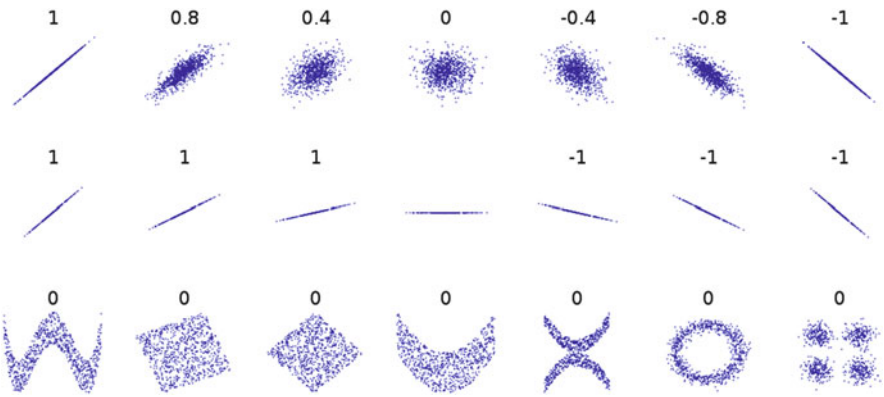


Fig. 11.1 Several sets of (x, y) points, with the correlation coefficient of x and y for each set. Note that the correlation reflects the noisiness and direction of a linear relationship (*top row*), but not the slope of that relationship (*middle*), nor many aspects of nonlinear relationships (*bottom*). N.B.: the figure in the center has a slope of 0 but in that case the correlation coefficient is undefined because the variance of Y is zero. (In Wikipedia. Retrieved May 27, 2015, from http://en.wikipedia.org/wiki/Correlation_and_dependence.)

Kendall's τ is also a rank correlation coefficient, measuring the association between two measured quantities. It is harder to calculate than Spearman's ρ, but it has been argued that confidence intervals for Spearman's ρ are less reliable and less interpretable than confidence intervals for Kendall's τ-parameters.

Code: "ISP_bivariate.py"[1]: Analysis of multivariate data (regression, correlation).

11.2 General Linear Regression Model

We can use the method of *Linear Regression* when we want to predict the value of one variable from the value(s) of one or more other variables (Fig. 11.2).

For example, when we search for the best-fit line to a given data set (x_i, y_i), we are looking for the parameters (k, d) which minimize the sum of the squared residuals ϵ_i in

$$y_i = k * x_i + d + \epsilon_i \tag{11.4}$$

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/bivariate.

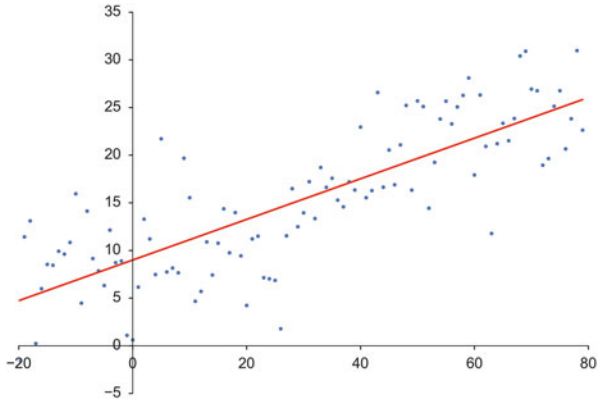


Fig. 11.2 Best-fit linear regression line to a given set of data

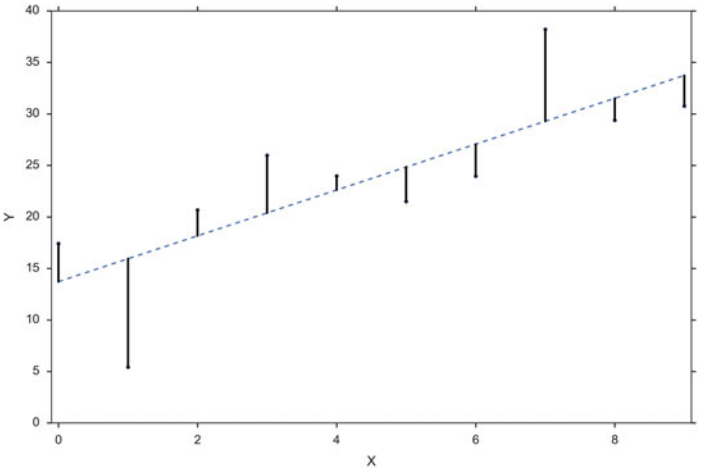


Fig. 11.3 Best-fit linear regression line (*dashed line*) and residuals (*solid lines*)

where k is the *slope* or *inclination* of the line, and d the *intercept*. The *residuals* are the differences between observed values and predicted values (see Fig.11.3).

Since the linear regression equation is solved to minimize the square sum of the residuals, linear regression is sometimes also called *Ordinary Least-Squares* (*OLS*) *Regression*.

This is in fact just the one-dimensional example of a more general technique, which is described in the next section.

Note that in contrast to the correlation, this relationship between x and y is not symmetrical any more: it is assumed that the x-values are known exactly, and that all the variability lies in the residuals.

11.2.1 Example 1: Simple Linear Regression

Suppose there are seven data points $\{y_i, x_i\}$, where $i = 1, 2, \ldots, 7$. The simple linear regression model is

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i, \tag{11.5}$$

where β_0 is the y-intercept and β_1 is the slope of the regression line. This model can be represented in matrix form as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \\ 1 & x_6 \\ 1 & x_7 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \tag{11.6}$$

where the first column of ones in the matrix on the right-hand side represents the y-intercept term, while the second column is the x-values associated with the y-values. (Section 11.4 shows how to solve these equations for β_i with *Python*.)

11.2.2 Example 2: Quadratic Fit

The equation for a quadratic fit to the given data is

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i, \tag{11.7}$$

This can be rewritten in matrix form:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \\ 1 & x_6 & x_6^2 \\ 1 & x_7 & x_7^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \tag{11.8}$$

Note that unknown parameters β_i enter only linearly, and the quadratic components are restricted to the (known) data matrix.

11.2.3 Coefficient of Determination

A data set has values y_i, each of which has an associated modeled value f_i (sometimes also referred to as $\hat{y}_i$). Here, the values y_i are called the *observed values*, and the modeled values f_i are sometimes called the *predicted values*.

In the following $\bar{y}$ is the mean of the observed data:

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{11.9}$$

where n is the number of observations.

The "variability" of the data set is measured through different sums of squares:

- $SS_{\text{mod}} = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$ is the *Model Sum of Squares*, or the sum of squares for the regression. This value is sometimes also called the *Explained Sum of Squares*.
- $SS_{\text{res}} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$ is the *Residuals Sum of Squares*, or the sum of squares for the errors.
- $SS_{\text{tot}} = \sum_{i=1}^{n} (y_i - \bar{y})^2$ is the *Total Sum of Squares*, and is equivalent to the sample variance multiplied by $n - 1$.

For linear regression models (Fig. 11.4),

$$SS_{\text{mod}} + SS_{\text{res}} = SS_{\text{tot}} \tag{11.10}$$

The notations SS_R and SS_E should be avoided, since in some texts their meaning is reversed to "Regression sum of squares" and "Explained sum of squares," respectively.

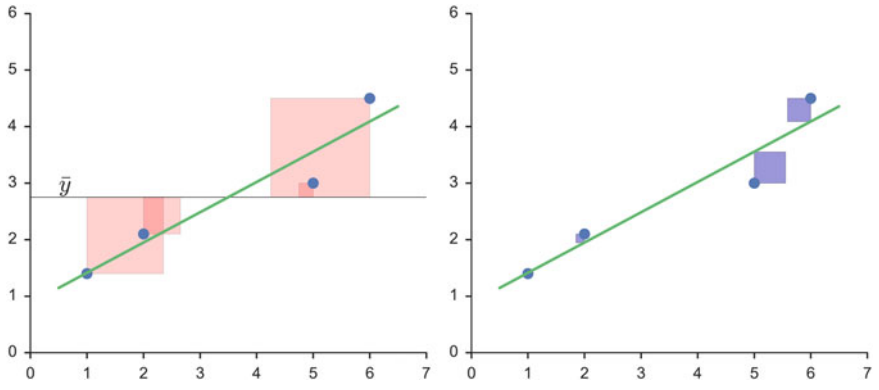


Fig. 11.4 The better the linear regression (on the *right*) fits the data in comparison to the simple average (on the *left graph*), the closer the value of R^2 is to one. The areas of the *blue squares* represent the squared residuals with respect to the linear regression. The areas of the *red squares* represent the squared residuals with respect to the average value

With these expressions, the most general definition of the *coefficient of determination*, R^2, is

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}. \tag{11.11}$$

Since

$$SS_{\text{tot}} = SS_{\text{mod}} + SS_{\text{res}} \tag{11.12}$$

Equation 11.11 is equivalent to

$$R^2 = \frac{SS_{\text{mod}}}{SS_{\text{tot}}} \tag{11.13}$$

In words: The coefficient of determination is the ratio between the sum-of-squares explained by the model, and the total sum-of-squares.

For simple linear regression (i.e., line-fits), the coefficient of determination or R^2 is the square of the correlation coefficient r. It is easier to interpret than the correlation coefficient r: values of R^2 close to 1 correspond to a close correlation, values close to 0 to a poor one. Note that for general models it is common to write R^2, whereas for simple linear regression r^2 is used.

a) Relation to Unexplained Variance

In a general form, R^2 can be seen to be related to the unexplained variance, since the second term in Eq. 11.11 compares the unexplained variance (variance of the model's errors) with the total variance (of the data).

b) "Good" Fits

How large R^2 values must be to be considered as "good" depends on the discipline. They are usually expected to be larger in the physical sciences than in biology or the social sciences. In finance or marketing, it also depends on what is being modeled.

Caution: the sample correlation and R^2 are misleading if there is a nonlinear relationship between the independent and dependent variables (see Fig. 11.1)!

11.3 Patsy: The Formula Language

The mini-language commonly used now in statistics to describe formulas was first used in the languages *R* and *S*, but is now also available in *Python* through the *Python* package *patsy*.

For instance, if we have a variable y, and we want to regress it against another variable x, we can simply write

$$y \sim x \tag{11.14}$$

A more complex situation where y depends on the variables x, a, b, and the interaction of a and b, can be expressed by

$$y \sim x + a + b + a : b \tag{11.15}$$

This formula language is based on the notation introduced by Wilkinson and Rogers (1973). The symbols in Table 11.1 are used on the right-hand side to denote different interactions. A complete set of the description can be found under http://patsy.readthedocs.org.

11.3.1 Design Matrix

a) Definition

A very general definition of a regression model is the following:

$$y = f(x, \epsilon) \tag{11.16}$$

In the case of a linear regression model, the model can be rewritten as:

$$y = \mathbf{X}\beta + \epsilon, \tag{11.17}$$

Table 11.1 Most important elements of the formula syntax

Operator	Meaning
$\sim$	Separates the left-hand side from the right-hand side. If omitted, a formula is assumed right-hand side only
+	Combines terms on either side (set union)
−	Removes terms on the right from set of terms on the left (set difference)
*	$a * b$ is shorthand for the expansion $a + b + a : b$
/	a/b is shorthand for the expansion $a + a : b$. It is used when b is nested within a (e.g., states and counties)
:	Computes the interaction between terms on the left and right
**	Takes a set of terms on the left and an integer n on the right and computes the * of that set of terms with itself n times

The matrix $\mathbf{X}$ is sometimes called the *design matrix* for the model. For a simple linear regression and for multilinear regression, the corresponding design matrices are given in 11.6 and 12.2, respectively.

Given a data set $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^n$ of n statistical units,[2] a linear regression model assumes that the relationship between the dependent variable y_i and the p-vector of regressors x_i is linear. This relationship is modeled through a disturbance term or error variable ϵ_i, an unobserved random variable that adds noise to the linear relationship between the dependent variable and regressors. Thus the model takes the form

$$y_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^{\mathrm{T}} \beta + \varepsilon_i, \qquad i = 1, \ldots, n, \tag{11.18}$$

where ^{T} denotes the transpose, so that $x_i^T \beta$ is the inner product between the vectors $\mathbf{x}_i$ and β.

Often these n equations are stacked together and written in vector form as

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon, \tag{11.19}$$

where

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} \mathbf{x}_1^{\mathrm{T}} \\ \mathbf{x}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{x}_n^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1p} \\ x_{21} & \cdots & x_{2p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{np} \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \quad \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}. \tag{11.20}$$

Some remarks on terminology and general use:

- y_i is called the *regressand*, *endogenous variable*, *response variable*, *measured variable*, or *dependent variable*. The decision as to which variable in a data set is modeled as the dependent variable and which are modeled as the independent variables may be based on a presumption that the value of one of the variables is caused by, or directly influenced by the other variables. Alternatively, there may be an operational reason to model one of the variables in terms of the others, in which case there need be no presumption of causality.
- $\mathbf{x}_i$ are called *regressors*, *exogenous variables*, *explanatory variables*, *covariates*, *input variables*, *predictor variables*, or *independent variables*. (The expression *independent variables* is meant in contrast to *dependent variables*, but not to be confused with *independent random variables*, where "independent" indicates that those variables don't depend on anything else).

[2]This section has been taken from Wikipedia https://en.wikipedia.org/wiki/Linear_regression, last accessed 21 Oct 2015.

  - Usually a constant is included as one of the regressors. For example we can take $x_{i1} = 1$ for $i = 1, \ldots, n$. The corresponding element of β is called the *intercept*. Many statistical inference procedures for linear models require an intercept to be present, so it is often included even if theoretical considerations suggest that its value should be zero.
  - Sometimes one of the regressors can be a nonlinear function of another regressor or of the data, as in polynomial regression and segmented regression. The model remains *linear* as long as it is linear in the parameter vector β (see Eq. 11.8, where a linear regression is used to fit a quadratic curve to the data.).

- β is a p-dimensional parameter vector. Its elements are also called *effects*, or *regression coefficients*. Statistical estimation and inference in linear regression focuses on β.
- ε_i is called the *residuals*, *error term*, *disturbance term*, or *noise*. This variable captures all other factors which influence the dependent variable y_i other than the regressors x_i. The relationship between the error term and the regressors, for example whether they are correlated, is a crucial step in formulating a linear regression model, as it will determine the method to use for estimation.
- If $i = 1$ and $p = 1$ in Eq.11.18, we have a *simple linear regression*, corresponding to Eq.11.4. If $i > 1$ we talk about *multilinear regression* or *multiple linear regression* (see Eq. 12.2).

b) Examples

One-Way ANOVA (Cell Means Model)

This example demonstrates a one-way analysis of variance (ANOVA) with three groups and seven observations. The given data set has the first three observations belonging to the first group, the following two observations belong to the second group, and the final two observations are from the third group. If the model to be fit is just the mean of each group, then the model is

$$y_{ij} = \mu_i + \epsilon_{ij} \ , \ i = 1, 2, 3 \tag{11.21}$$

which can be written as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \tag{11.22}$$

It should be emphasized that in this model μ_i represents the mean of the ith group.

One-Way ANOVA (Offset from Reference Group)

The ANOVA model could be equivalently written as each group parameter τ_i being an offset from some overall reference. Typically this reference point is taken to be one of the groups under consideration. This makes sense in the context of comparing multiple treatment groups to a control group and the control group is considered the "reference." In this example, group 1 was chosen to be the reference group. As such the model to be fit is:

$$y_{ij} = \mu + \tau_i + \epsilon_{ij} \ , \ i = 1, 2, 3 \tag{11.23}$$

with the constraint that τ_1 is zero.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mu \\ \tau_2 \\ \tau_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \tag{11.24}$$

In this model μ is the mean of the reference group and τ_i is the difference from group i to the reference group. τ_1 is not included in the matrix because its difference from the reference group (itself) is necessarily zero.

11.4 Linear Regression Analysis with Python

11.4.1 Example 1: Line Fit with Confidence Intervals

For univariate distributions, the confidence intervals based on the standard deviation indicate the interval that we expect to contain 95 % of the data, and the confidence intervals based on the standard error of the mean indicate the interval that contains the true mean with 95 % probability. We also have these two types of confidence intervals (one for the data, and one for the fitted parameters) for line fits, and they are shown in Fig. 11.5.

The corresponding equation is Eq.11.4, and the formula syntax is given by Eq.11.14.

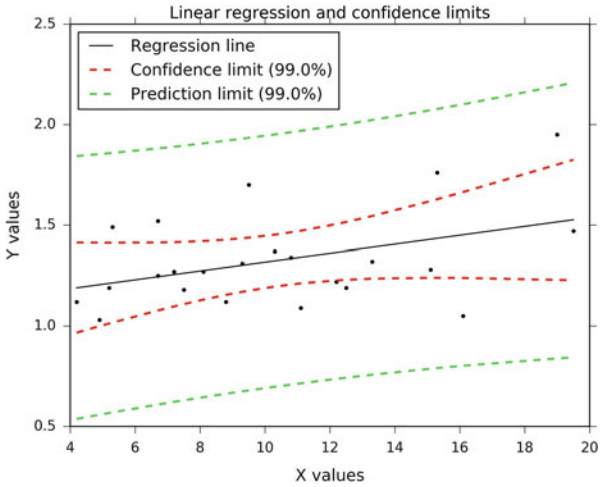


Fig. 11.5 Regression, with confidence intervals for the mean, as well as for the predicted data. The *red dotted line* shows the confidence interval for the mean; and the *green dotted line* the confidence interval for predicted data. The corresponding code can be found in *ISP_fitLine.py*

Code: "ISP_fitLine.py"[3]: Linear regression fit, with the output shown in Fig. 11.5.

11.4.2 Example 2: Noisy Quadratic Polynomial

To see how different models can be used to evaluate a given set of data, let us look at a simple example: fitting a noisy, slightly quadratic curve. Let us start with the algorithms implemented in *numpy*, and fit a linear, quadratic, and a cubic curve to the data.

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt

In [2]: ''' Generate a noisy, slightly quadratic dataset '''
   ...: x = np.arange(100)
   ...: y = 150 + 3*x + 0.03*x**2 + 5*np.random.randn(len(x))
   ...:

In [3]: # Create the Design Matrices for the linear, quadratic,
   ...  # and cubic fit
```

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/fitLine.

```
   ...: M1 = np.vstack( (np.ones_like(x), x) ).T
   ...: M2 = np.vstack( (np.ones_like(x), x, x**2) ).T
   ...: M3 = np.vstack( (np.ones_like(x), x, x**2, x**3) ).T
   ...:
   ...: # an equivalent alternative solution with statsmodels would be
   ...: # M1 = sm.add_constant(x)
   ...:
In [4]: # Solve the equations
   ...: p1 = np.linalg.lstsq(M1, y)
   ...: p2 = np.linalg.lstsq(M2, y)
   ...: p3 = np.linalg.lstsq(M3, y)
   ...:

In [5]: np.set_printoptions(precision=3)

In [6]: print('The coefficients from the linear fit: {0}'
   ...:       .format(p1[0]))
The coefficients from the linear fit:
[ 100.42    5.98]

In [7]: print('The coefficients from the quadratic fit: {0}'
   ...:       .format(p2[0]))
The coefficients from the quadratic fit:
[  1.48e+02   3.10e+00   2.91e-02]

In [8]: print('The coefficients from the cubic fit: {0}'
   ...:       .format(p3[0]))
The coefficients from the cubic fit:
[  1.47e+02   3.12e+00   2.84e-02   4.81e-06]
```

With this simple analytical solution we can obtain the fitted coefficients (β_i in Eq.11.18) for a linear, a quadratic, and a cubic model. And as we see in Fig.11.6, the quadratic and the cubic fit are both very good and essentially undistinguishable.

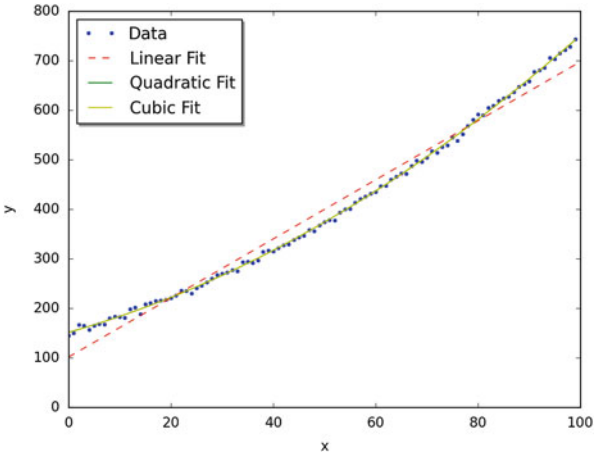


Fig. 11.6 A noisy, slightly quadratic data set, with a linear, a quadratic, and a cubic fit superposed. The quadratic and the cubic lines are almost exactly the same

If we want to find out which is the "better" fit, we can use the tools provided by *statsmodels* to again fit the model. Using *statsmodels* we obtain not only the best-fit parameters, but a wealth of additional information about the model:

```
In [9]: '''Solution with the tools from statsmodels'''
  ...: import statsmodels.api as sm
  ...:
  ...: Res1 = sm.OLS(y, M1).fit()
  ...: Res2 = sm.OLS(y, M2).fit()
  ...: Res3 = sm.OLS(y, M3).fit()

 In [10]: print(Res1.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.983
Dependent Variable: y                AIC:                909.6344
Date:               2015-06-27 13:50 BIC:                914.8447
No. Observations:   100              Log-Likelihood:     -452.82
Df Model:           1                F-statistic:        5818.
Df Residuals:       98               Prob (F-statistic): 4.46e-89
R-squared:          0.983            Scale:              512.18
-----------------------------------------------------------------
         Coef.     Std.Err.      t      P>|t|     [0.025    0.975]
-----------------------------------------------------------------
const   100.4163     4.4925   22.3519   0.0000   91.5010   109.3316
x1        5.9802     0.0784   76.2769   0.0000    5.8246     6.1358
-----------------------------------------------------------------
Omnibus:               10.925       Durbin-Watson:          0.131
Prob(Omnibus):         0.004        Jarque-Bera (JB):       6.718
Skew:                  0.476        Prob(JB):               0.035
Kurtosis:              2.160        Condition No.:          114
=================================================================
```

```
In [11]: print('The AIC-value is {0:4.1f} for the linear fit,\n
    ...: {1:4.1f} for the quadratic fit, and \n
    ...: {2:4.1f} for the cubic fit'.format(Res1.aic, Res2.aic,
    ...:                                    Res3.aic))
The AIC-value is 909.8 for the linear fit,
    578.7 for the quadratic fit, and
    580.2 for the cubic fit
```

In the next section we will explain the meaning of all these parameters in detail. Here I just want to point out the *AIC*-value, the *Akaike Information Criterion*, which can be used to assess the quality of the model: the lower the AIC value, the better the model. We see that the quadratic model has the lowest AIC value and therefore is the best model: it provides the same quality of fit as the cubic model, but uses fewer parameters to achieve that quality.

Before we move to the next example, let me show how the formula language can be used to perform the same fits, but without having to manually generate the design matrices, and how to extract, e.g., the model parameters, standard errors, and confidence intervals. Note that the use of *pandas* DataFrames allows *Python* to add information about the individual parameters.

```
In [14]: '''Formula-based modeling '''
    ...: import pandas as pd
    ...: import statsmodels.formula.api as smf
    ...:
    ...: # Turn the data into a pandas DataFrame, so that we
    ...: # can address them in the formulas with their name
    ...: df = pd.DataFrame({'x':x, 'y':y})
    ...:
    ...: # Fit the models, and show the results
    ...: Res1F = smf.ols('y~x', df).fit()
    ...: Res2F = smf.ols('y ~ x+I(x**2)', df).fit()
    ...: Res3F = smf.ols('y ~ x+I(x**2)+I(x**3)', df).fit()

In [15]: #As example, display parameters for the quadratic fit
    ...: Res2F.params
Out[15]:
Intercept    148.022539
x              3.043490
I(x ** 2)      0.029454
dtype: float64

In [16]: Res2F.bse
Out[16]:
Intercept    1.473074
x            0.068770
I(x ** 2)    0.000672
dtype: float64

In [17]: Res2F.conf_int()
Out[17]:
                     0           1
Intercept  145.098896  150.946182
x            2.907001    3.179978
I(x ** 2)    0.028119    0.030788
```

The confidence intervals (Out[17]) are of particular interest, as parameters whose confidence intervals overlap zero are not significant.

 **Code:** "ISP_modelImplementations.py"[4]: Three ways how to solve a linear regression model in *Python*.

11.5 Model Results of Linear Regression Models

The output of linear regression models, such as the one on page 196, can at first be daunting. Since understanding this type of output is a worthwhile step towards more complex models, I will in the following present a simple example, and explain the output step-by-step. We will use *Python* to explore measures of fits for linear regression: the coefficient of determination (R^2), hypothesis tests (F, T, Omnibus), and other measures.[5]

11.5.1 Example: Tobacco and Alcohol in the UK

First we will look at a small data set from the DASL library (http://lib.stat.cmu.edu/DASL/Stories/AlcoholandTobacco.html), regarding the correlation between tobacco and alcohol purchases in different regions of the United Kingdom. The interesting feature of this data set is that Northern Ireland is reported as an outlier (Fig. 11.7). Notwithstanding, we will use this data set to describe two tools for calculating a linear regression. We will alternatively use the *statsmodels* and *sklearn* modules for calculating the linear regression, while using *pandas* for data management, and *matplotlib* for plotting. To begin, we will import the modules, get the data into *Python*, and have a look at them:

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: import matplotlib as mpl
   ...: import matplotlib.pyplot as plt
   ...: import statsmodels.formula.api as sm
   ...: from sklearn.linear_model import LinearRegression
   ...: from scipy import stats
   ...:
```

[4] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/modelImplementations.

[5] The following is based on the blog of Connor Johnson (http://connor-johnson.com/2014/02/18/linear-regression-with-python/), with permission from the author.

```
In [2]: data_str = '''Region Alcohol Tobacco
   ...: North 6.47 4.03
   ...: Yorkshire 6.13 3.76
   ...: Northeast 6.19 3.77
   ...: East_Midlands 4.89 3.34
   ...: West_Midlands 5.63 3.47
   ...: East_Anglia 4.52 2.92
   ...: Southeast 5.89 3.20
   ...: Southwest 4.79 2.71
   ...: Wales 5.27 3.53
   ...: Scotland 6.08 4.51
   ...: Northern_Ireland 4.02 4.56'''
   ...:
   ...: # Read in the data. Note that for Python 2.x,
   ...: # you have to change the "import" statement
   ...: from io import StringIO
   ...: df = pd.read_csv(StringIO(data_str), sep=r'\s+')
   ...:

In [3]: # Plot the data
   ...: df.plot('Tobacco', 'Alcohol', style='o')
   ...: plt.ylabel('Alcohol')
   ...: plt.title('Sales in Several UK Regions')
   ...: plt.show()
   ...:
```

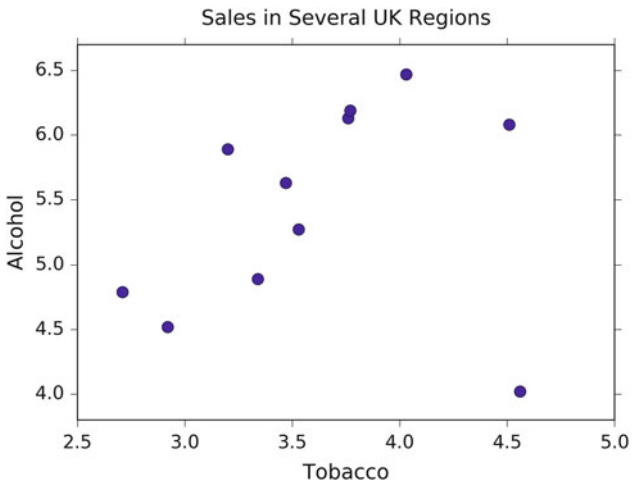


Fig. 11.7 Sales of alcohol vs tobacco in the UK. We notice that there seems to be a linear trend, and one outlier, which corresponds to Northern Ireland

Fitting the model, leaving the outlier (which is the last data point) for the moment away is then very easy:

```
In [4]: result = sm.ols('Alcohol ~ Tobacco', df[:-1]).fit()
   ...: print(result.summary())
   ...:
```

Note that using the `formula.api` module from statsmodels, an intercept is automatically added. This gives us

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Alcohol   R-squared:                       0.615
Model:                            OLS   Adj. R-squared:                  0.567
Method:                 Least Squares   F-statistic:                     12.78
Date:                Sun, 27 Apr 2014   Prob (F-statistic):            0.00723
Time:                        13:19:51   Log-Likelihood:                -4.9998
No. Observations:                  10   AIC:                             14.00
Df Residuals:                       8   BIC:                             14.60
Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      2.0412      1.001      2.038      0.076        -0.268     4.350
Tobacco        1.0059      0.281      3.576      0.007         0.357     1.655
==============================================================================
Omnibus:                        2.542   Durbin-Watson:                   1.975
Prob(Omnibus):                  0.281   Jarque-Bera (JB):                0.904
Skew:                          -0.014   Prob(JB):                        0.636
Kurtosis:                       1.527   Cond. No.                         27.2
==============================================================================
```

And now we have a very nice table of numbers—which at first looks fairly daunting. To explain what the individual numbers mean, I will go through and explain each one. The left column of the first table is mostly self explanatory. The degrees of freedom (*Df*) of the model are the number of predictors, or explanatory, variables. The *Df* of the residuals is the number of observations minus the degrees of freedom of the model, minus one (for the offset).

Most of the values listed in the summary are available via the `result` object. For instance, the R^2 value can be obtained by `result.rsquared`. If you are using *IPython*, you may type `result.` and hit the TAB key, to see a list of all possible attributes for the `result`-object.

11.5.2 Definitions for Regression with Intercept

The Sum-of Squares variables SS_{xx} have already been defined above, in Sect. 11.2.3. n is the number of observations, and k is the number of regression parameters. For example, if you fit a straight line, $k = 2$. And as above (Sect. 11.2.3) $\hat{y}_i$ will indicate the fitted model values, and $\bar{y}$ the mean. In addition to these, the following variables

will be used:

- $DF_{\text{mod}} = k - 1$ is the *(Corrected) Model Degrees of Freedom*. (The "−1" comes from the fact that we are only interested in the correlation, not in the absolute offset of the data.)
- $DF_{\text{res}} = n - k$ is the *Residuals Degrees of Freedom*
- $DF_{\text{tot}} = n - 1$ is the *(Corrected) Total Degrees of Freedom*. The Horizontal line regression is the null hypothesis model.

  For multiple regression models with intercept, $DF_{\text{mod}} + DF_{\text{res}} = DF_{\text{tot}}$.

- $MS_{\text{mod}} = SS_{\text{mod}}/DF_{\text{mod}}$: *Model Mean of Squares*
- $MS_{\text{res}} = SS_{\text{res}}/DF_{\text{res}}$: *Residuals Mean of Squares*. MS_{res} is an unbiased estimate for σ^2 for multiple regression models.
- $MS_{\text{tot}} = SS_{\text{tot}}/DF_{\text{tot}}$: *Total Mean of Squares*, which is the sample variance of the y-variable.

11.5.3 The R^2 Value

As we have already seen in Sect. 11.2.3, the R^2 value indicates the proportion of variation in the y-variable that is due to variation in the x-variables. For simple linear regression, the R^2 value is the square of the sample correlation r_{xy}. For multiple linear regression with intercept (which includes simple linear regression), the R^2 value is defined as

$$R^2 = \frac{SS_{\text{mod}}}{SS_{\text{tot}}} \tag{11.25}$$

11.5.4 $\bar{R}^2$: The Adjusted R^2 Value

For assessing the quality of models, many researchers prefer the *adjusted* R^2 *value*, commonly indicated with the bar above the $\bar{R}$, which is penalized for having a large number of parameters in the model.

Here is the logic behind the definition of $\bar{R}^2$: R^2 is defined as $R^2 = 1 - SS_{\text{res}}/SS_{\text{tot}}$ or $1 - R^2 = SS_{\text{res}}/SS_{\text{tot}}$. To take into account the number of regression parameters p, define the *adjusted R-squared* value as

$$1 - \bar{R}^2 = \frac{ResidualVariance}{TotalVariance} \tag{11.26}$$

where *(Sample) Residual Variance* is estimated by $SS_{\text{res}}/DF_{\text{res}} = SS_{\text{res}}/(n-k)$, and *(Sample) Total Variance* is estimated by $SS_{\text{tot}}/DF_{\text{tot}} = SS_{\text{tot}}/(n-1)$. Thus,

$$\begin{aligned} 1 - \bar{R}^2 &= \frac{SS_{\text{res}}/(n-k)}{SS_{\text{tot}}/(n-1)} \\ &= \frac{SS_{\text{res}}}{SS_{\text{tot}}} \frac{n-1}{n-k} \end{aligned} \tag{11.27}$$

so

$$\begin{aligned} \bar{R}^2 &= 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \frac{n-1}{n-k} \\ &= 1 - (1-R^2)\frac{n-1}{n-k} \end{aligned} \tag{11.28}$$

a) The *F*-Test

For a multiple regression model with intercept,

$$\begin{aligned} Y_j &= \alpha + \beta_1 X_{1j} + \ldots + \beta_n X_{nj} + \epsilon_i \\ &= \alpha + \sum_{i=1}^{n} \beta_i X_{ij} + \epsilon_j \\ &= E(Y_j|X) + \epsilon_j \end{aligned} \tag{11.29}$$

In the last line $E(Y_j|X)$ indicates the "expected value for Y, given X."

We want to test the following null hypothesis and alternative hypothesis:

H_0: $\beta_1 = \beta_2 = \ldots = \beta_n = 0$
H_1: $\beta_j \neq 0$, for at least one value of j

This test is known as the *overall F-test for regression*.

Remember, if $t_1, t_2, \ldots, t_m$ are independent, $N(0, \sigma^2)$ random variables, then $\sum_{i=1}^{m} \frac{t_i^2}{\sigma^2}$ is a χ^2 (chi-squared) random variable with m degrees of freedom. It can be shown that if H_0 is true and the residuals are unbiased, homoscedastic (i.e., all function values have the same variance), independent, and normal (see Sect. 11.6), then:

1. SS_{res}/σ^2 has a χ^2 distribution with DF_{res} degrees of freedom.
2. SS_{mod}/σ^2 has a χ^2 distribution with DF_{mod} degrees of freedom.
3. SS_{res} and SS_{mod} are independent random variables.

If u is a χ^2 random variable with n degrees of freedom, v is a χ^2 random variable with m degrees of freedom, and u and v are independent, then $F = \frac{u/n}{v/m}$ has an F distribution with (n, m) degrees of freedom.

If H_0 is true,

$$\begin{aligned} F &= \frac{(SS_{\text{mod}}/\sigma^2)/DF_{\text{mod}}}{(SS_{\text{res}}/\sigma^2)/DF_{\text{res}}} \\ &= \frac{SS_{\text{mod}}/DF_{\text{mod}}}{SS_{\text{res}}/DF_{\text{res}}} \\ &= \frac{MS_{\text{mod}}}{MS_{\text{res}}}, \end{aligned} \tag{11.30}$$

has an F distribution with $(DF_{\text{mod}}, DF_{\text{res}})$ degrees of freedom, and is independent of σ.

We can test this directly in *Python* with

```
In [5]: N = result.nobs
   ...: k = result.df_model+1
   ...: dfm, dfe = k-1, N - k
   ...: F = result.mse_model / result.mse_resid
   ...: p = 1.0 - stats.f.cdf(F,dfm,dfe)
   ...: print('F-statistic: {:.3f}, p-value: {:.5f}'
   ...:          .format( F, p ))
   ...:
   F-statistic: 12.785, p-value: 0.00723
```

which corresponds to the values in the model summary above.

Here, `stats.f.cdf(F, m, n)` returns the cumulative sum of the F-distribution with shape parameters `m = k-1 = 1`, and `n = N - k = 8`, up to the F-statistic F. Subtracting this quantity from one, we obtain the probability in the tail, which represents the probability of observing F-statistics more extreme than the one observed.

b) Log-Likelihood Function

A very common approach in statistics is the idea of *maximum likelihood estimation*. The basic idea is quite different from the *OLS (least square)* approach: in the least square approach the model is constant, and the errors of the response are variable; in contrast, in the maximum likelihood approach, the data response values are regarded as constant, and the likelihood of the model is maximized. (The concept of maximum likelihood estimation is very well explained in Duda (2004).)

For the classical linear regression model (with normal errors) we have

$$\epsilon = y_i - \sum_{k=1}^{n} \beta_k x_{ik} = y_i - \hat{y}_i \ \in \ N(0, \sigma) \tag{11.31}$$

so the probability density is given by

$$p(\epsilon_i) = \Phi(\frac{y_i - \hat{y}_i}{\sigma}) \tag{11.32}$$

where $\Phi(z)$ is the standard normal probability distribution function. The probability of independent samples is the product of the individual probabilities

$$\Pi_{total} = \prod_{i=1}^{n} p(\epsilon_i) \tag{11.33}$$

The *Log Likelihood function* is defined as

$$\begin{aligned} ln(\mathscr{L}) &= ln(\Pi_{total}) \\ &= ln\left[\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2}\right)\right] \\ &= \sum_{i=1}^{n}\left[ln\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \left(\frac{(y_i - \hat{y}_i)^2}{2\sigma^2}\right)\right] \end{aligned}$$

It can be shown that the maximum likelihood estimator of σ^2 is

$$E(\sigma^2) = \frac{SS_{\text{res}}}{n} \tag{11.34}$$

We can calculate this in *Python* as follows:

```
In [6]: N = result.nobs
   ...: SSR = result.ssr
   ...: s2 = SSR / N
   ...: L = (1.0/np.sqrt(2*np.pi*s2)) ** N*np.exp(-SSR/(s2*2.0))
   ...: print('ln(L) =', np.log( L ))
   ...:
ln(L) = -4.99975869739
```

which again matches the model summary.

c) Information Content of Statistical Models: AIC and BIC

To judge the quality of a model, one should first visually inspect the residuals. In addition, one can also use a number of numerical criteria to assess the quality of a statistical model. These criteria represent various approaches for balancing model accuracy with parsimony.

We have already encountered the *adjusted* R^2 value (Sect. 11.5.4), which—in contrast to the R^2 value—decreases if there are too many regressors in the model.

Other commonly encountered criteria are the *Akaike Information Criterion* (*AIC*) and the *Schwartz* or *Bayesian Information Criterion* (*BIC*), which are based on the log-likelihood described in the previous section. Both measures introduce a penalty for model complexity, but the AIC penalizes complexity less severely than the BIC. The *Akaike Information Criterion* (*AIC*) is given by

$$AIC = 2 * k - 2 * ln(\mathscr{L}) \tag{11.35}$$

and the Schwartz or *Bayesian Information Criterion* (*BIC*) by

$$BIC = k * ln(N) - 2 * ln(\mathscr{L}). \tag{11.36}$$

Here, N is the number of observations, k is the number of parameters, and $\mathscr{L}$ is the likelihood. We have two parameters in this example, the slope and intercept. The AIC is a relative estimate of information loss between different models. The BIC was initially proposed using a Bayesian argument and does not relate to ideas of information. Both measures are only used when trying to decide between different models. So, if we have one regression for alcohol sales based on cigarette sales, and another model for alcohol consumption that incorporated cigarette sales and lighter sales, then we should choose the model with the lower AIC or BIC value.

11.5.5 Model Coefficients and Their Interpretation

The second table in the model summary on page 200 contains the model coefficients and their interpretation.

a) Coefficients

The *coefficients* or weights of the linear regression are contained in `result.params`, and returned as a *pandas* Series object, since we used a *pandas* DataFrame as input. This is nice, because the coefficients are named for convenience.

```
In [7]: result.params
Out[7]:
Intercept    2.041223
Tobacco      1.005896
dtype: float64
```

We can obtain this directly by computing

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \cdot \mathbf{y}. \tag{11.37}$$

Here, $\mathbf{X}$ is the design matrix, i.e., the matrix of predictor variables as columns, with an extra column of ones for the constant term; $\mathbf{y}$ is the column vector of the

response variable, and $\boldsymbol{\beta}$ is the column vector of coefficients corresponding to the columns of $\mathbf{X}$. In *Python*:

```
In [8]: df['Ones'] = np.ones( len(df) )
   ...: Y = df.Alcohol[:-1]
   ...: X = df[['Tobacco','Ones']][:-1]
   ...:
```

Note: the "−1" in the indices excludes the last data point, i.e., the outlier Northern Ireland.

b) Standard Error

To obtain the *standard errors of the coefficients* we will calculate the *covariance-variance matrix*, also called the *covariance matrix*, for the estimated coefficients $\boldsymbol{\beta}$ of the predictor variables using

$$C = cov(\beta) = \sigma^2(\mathbf{X}\mathbf{X}^T)^{-1}. \tag{11.38}$$

Here, σ^2 is the variance, or the mean-squared-error of the residuals. The standard errors are the square roots of the elements on the main diagonal of this covariance matrix. We can perform the operation above and calculate the element-wise square root using the following *Python* code,

```
In [9]: X = df.Tobacco[:-1]
   ...:
   ...: # add a column of ones for the constant intercept term
   ...: X = np.vstack(( np.ones(X.size), X ))
   ...:
   ...: # convert the numpy array to a matrix
   ...: X = np.matrix( X )
   ...:
   ...: # perform the matrix multiplication,
   ...: # and then take the inverse
   ...: C = np.linalg.inv( X * X.T )
   ...:
   ...: # multiply by the mean squared error of the residual
   ...: C *= result.mse_resid
   ...:
   ...: # take the square root
   ...: SE = np.sqrt(C)
   ...:
   ...: print(SE)
   ...:
[[ 1.00136021         nan]
 [        nan  0.28132158]]
```

c) *t*-Statistic

We use the *t*-test to test the null hypothesis that the coefficient of a given predictor variable is zero, implying that a given predictor has no appreciable effect on the response variable. The alternative hypothesis is that the predictor does contribute to the response. In testing we set some threshold, $\alpha = 0.05$ or 0.001, and if $\Pr(T \geq |t|) < \alpha$, then we reject the null hypothesis at our threshold α, otherwise we fail to reject the null hypothesis. The *t*-test generally allows us to evaluate the importance of different predictors, *assuming that the residuals of the model are normally distributed about zero*. If the residuals do not behave in this manner, then that suggests that there is some nonlinearity between the variables, and that their *t*-tests should not be used to assess the importance of individual predictors. Furthermore, it might be best to try to modify the model so that the residuals do tend the cluster normally about zero.

The *t* statistic is given by the ratio of the coefficient (or factor) of the predictor variable of interest and its corresponding standard error. If $\boldsymbol{\beta}$ is the vector of coefficients or factors of our predictor variables, and SE is our standard error, then the *t*-statistic is given by,

$$t_i = \beta_i / SE_{i,i} \tag{11.39}$$

So, for the first factor, corresponding to the slope in our example, we have the following code

```
In [10]: i = 1
    ...: beta = result.params[i]
    ...: se = SE[i,i]
    ...: t = beta / se
    ...: print('t =', t)
    ...:
t = 3.57560845424
```

Once we have a *t*-statistic, we can calculate the probability of observing a statistic at least as extreme as what we have already observed, given our assumptions about the normality of our errors by using the code

```
In [11]: N = result.nobs
    ...: k = result.df_model + 1
    ...: dof = N - k
    ...: p_onesided = 1.0 - stats.t( dof ).cdf( t )
    ...: p = p_onesided * 2.0
    ...: print('p = {0:.3f}'.format(p))
    ...:
p = 0.007
```

Here, `dof` are the degrees of freedom, which should be eight: the number of observations, *N*, minus the number of parameters, which is two. The CDF is the cumulative sum of the PDF. We are interested in the area under the right-hand tail, beyond our *t*-statistic, *t*, so we subtract the cumulative sum up to that statistic from

one in order to obtain the tail probability on the other side. Then we multiply this tail probability by two to obtain a two-tailed probability.

d) Confidence Interval

The confidence interval is built using the standard error, the p-value from our t-test, and a critical value from a t-test having $N - k$ degrees of freedom, where k is the number of observations and P is the number of model parameters, i.e., the number of predictor variables. The confidence interval is the range of values in which we would expect to find the parameter of interest, based on what we have observed. You will note that we have a confidence interval for the predictor variable coefficient, and for the constant term. A smaller confidence interval suggests that we are confident about the value of the estimated coefficient, or constant term. A larger confidence interval suggests that there is more uncertainty or variance in the estimated term. Again, let me reiterate that hypothesis testing is only one perspective. Furthermore, it is a perspective that was developed in the late nineteenth and early twentieth centuries when data sets were generally smaller and more expensive to gather, and data scientists were using books of logarithm tables for arithmetic.

The confidence interval is given by,

$$CI = \beta_i \pm z \cdot SE_{i,i} \tag{11.40}$$

Here, β_i is one of the estimated coefficients, z is a *critical-value*, which is the t-statistic required to obtain a probability less than the alpha significance level, and $SE_{i,i}$ is the standard error. The critical value is calculated using the inverse of the cumulative distribution function. In code, the confidence interval using a t-distribution looks like

```
In [12]: i = 0
    ...:
    ...: # the estimated coefficient, and its variance
    ...: beta, c = result.params[i], SE[i,i]
    ...:
    ...: # critical value of the t-statistic
    ...: N = result.nobs
    ...: P = result.df_model
    ...: dof = N - P - 1
    ...: z = stats.t( dof ).ppf(0.975)
    ...:
    ...: # the confidence interval
    ...: print(beta - z * c, beta + z * c)
    ...:
-0.267917709371 4.35036388305
```

11.5.6 Analysis of Residuals

The third table in the model summary on page 200 contains the parameters that characterize the residuals. If those clearly deviate from a normal distribution, then the model most likely has missed an essential element of the data.

The `OLS` command from `statsmodels.formula.api` provides some additional information about the residuals of the model: Omnibus, Skewness, Kurtosis, Durbin–Watson, Jarque–Bera, and the Condition number. In the following we will briefly describe these parameters.

a) Skewness and Kurtosis

Skew and kurtosis refer to the shape of a distribution. *Skewness* is a measure of the asymmetry of a distribution, and *kurtosis* is a measure of its curvature, specifically how pointed the curve is. (For normally distributed data it is approximately 3.) These values are defined by

$$S = \frac{\hat{\mu}_3}{\hat{\sigma}^3} = \frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^3}{\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2\right)^{3/2}}, \tag{11.41a}$$

$$K = \frac{\hat{\mu}_4}{\hat{\sigma}^4} = \frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^4}{\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2\right)^{2}} \tag{11.41b}$$

As you see, the $\hat{\mu}_3$ and $\hat{\mu}_4$ are the third and fourth central moments of a distribution.

The *excess kurtosis* is defined as $K - 3$, to ensure that its value for a normal distribution is equal to zero.

One possible *Python* implementation would be

```
In [13]: d = Y - result.fittedvalues
    ...:
    ...: S = np.mean( d**3.0 ) / np.mean( d**2.0 )**(3.0/2.0)
    ...: # equivalent to:
    ...: # S = stats.skew(result.resid, bias=True)
    ...:
    ...: K = np.mean( d**4.0 ) / np.mean( d**2.0 )**(4.0/2.0)
    ...: # equivalent to:
    ...: # K = stats.kurtosis(result.resid, fisher=False,
    ...: #                   bias=True)
```

```
    ...: print('Skewness: {:.3f},  Kurtosis: {:.3f}'.format(S,K))
    ...:
Skewness: -0.014,  Kurtosis: 1.527
```

b) Omnibus Test

The *Omnibus test* uses skewness and kurtosis to test the null hypothesis that a distribution is normal. In this case, we are looking at the distribution of the residuals. If we obtain a very small value for *P(Omnibus)*, then the residuals are not normally distributed about zero, and we should maybe look at our model more closely. The *statsmodels* OLS function uses the `stats.normaltest()` function:

```
In [14]: (K2, p) = stats.normaltest(result.resid)
    ...: print('Omnibus: {0:.3f}, p = {1:.3f}'.format(K2, p))
    ...:
Omnibus: 2.542, p = 0.281
```

Thus, if either the skewness or kurtosis suggests non-normality, this test should pick it up.

c) Durbin–Watson

The *Durbin–Watson test* is used to detect the presence of autocorrelation (a relationship between values separated from each other by a given time lag) in the residuals. Here the lag is one:

$$DW = \frac{\sum_{i=2}^{N}((y_i - \hat{y}_i) - (y_{i-1} - \hat{y}_{i-1}))^2}{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{11.42}$$

```
In [15]: DW = np.sum( np.diff( result.resid.values )**2.0 ) \
    ...:       / result.ssr
    ...: print('Durbin-Watson: {:.5f}'.format( DW ))
    ...:
Durbin-Watson: 1.97535
```

d) Jarque–Bera Test

The *Jarque–Bera test* is another test that considers skewness (S), and kurtosis (K). The null hypothesis is that the distribution is normal, that both the skewness and excess kurtosis equal zero, or alternatively, that the skewness is zero and the regular run-of-the-mill kurtosis is three. Unfortunately, with small samples the Jarque–Bera

test is prone to rejecting the null hypothesis—that the distribution is normal—when it is in fact true.

$$JB = \frac{N}{6}\left(S^2 + \frac{1}{4}(K-3)^2\right) \tag{11.43}$$

Calculating the Jarque–Bera statistic using the χ^2 distribution with two degrees of freedom we have

```
In [16]: JB = (N/6.0) * ( S**2.0 + (1.0/4.0)*( K - 3.0 )**2.0 )
    ...: p = 1.0 - stats.chi2(2).cdf(JB)
    ...: print('JB-statistic: {:.5f},  p-value: {:.5f}'
    ...:         .format( JB, p ))
    ...:
JB-statistic: 0.90421,  p-value: 0.63629
```

e) Condition Number

The *condition number* measures the sensitivity of a function's output to its input. When two predictor variables are highly correlated, which is called *multicollinearity*, the coefficients or factors of those predictor variables can fluctuate erratically for small changes in the data or the model. Ideally, similar models should be similar, i.e., have approximately equal coefficients. Multicollinearity can cause numerical matrix inversion to crap out, or produce inaccurate results (see Kaplan 2009). One approach to this problem in regression is the technique of *ridge regression*, which is available in the *Python* package *sklearn*.

We calculate the condition number by taking the eigenvalues of the product of the predictor variables (including the constant vector of ones) and then taking the square root of the ratio of the largest eigenvalue to the smallest eigenvalue. If the condition number is greater than 30, then the regression may have multicollinearity.

```
In [17]: X = np.matrix( X )
    ...: EV = np.linalg.eig( X * X.T )
    ...: print(EV)
    ...:
(array([   0.18412885,  136.51527115]),
matrix([[-0.96332746, -0.26832855],
        [ 0.26832855, -0.96332746]]))
```

Note that $X.T * X$ should be $(P+1)\times(P+1)$, where P is the number of degrees of freedom of the model (the number of predictors) and the +1 represents the addition of the constant vector of ones for the intercept term. In our case, the product should be a 2×2 matrix, so we will have two eigenvalues. Then our condition number is given by

```
In [18]: CN = np.sqrt( EV[0].max() / EV[0].min() )
    ...: print('Condition No.: {:.5f}'.format( CN ))
    ...:
Condition No.: 27.22887
```

Our condition number is just below 30 (weak!), so we can sort of sleep okay.

11.5.7 Outliers

Now that we have seen an example of linear regression with a reasonable degree of linearity, compare that with an example of one with a significant outlier. In practice, outliers should be understood before they are discarded, because they might turn out to be very important. They might signify a new trend, or some possibly catastrophic event.

```
In [19]: X = df[['Tobacco','Ones']]
    ...: Y = df.Alcohol
    ...: result = sm.OLS( Y, X ).fit()
    ...: result.summary()
    ...:
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Alcohol   R-squared:                       0.050
Model:                            OLS   Adj. R-squared:                 -0.056
Method:                 Least Squares   F-statistic:                    0.4735
Date:                Sun, 27 Apr 2014   Prob (F-statistic):              0.509
Time:                        12:58:27   Log-Likelihood:                -12.317
No. Observations:                  11   AIC:                             28.63
Df Residuals:                       9   BIC:                             29.43
Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      4.3512      1.607      2.708      0.024         0.717     7.986
Tobacco        0.3019      0.439      0.688      0.509        -0.691     1.295
==============================================================================
Omnibus:                        3.123   Durbin-Watson:                   1.655
Prob(Omnibus):                  0.210   Jarque-Bera (JB):                1.397
Skew:                          -0.873   Prob(JB):                        0.497
Kurtosis:                       3.022   Cond. No.                         25.5
==============================================================================
```

11.5.8 Regression Using Sklearn

scikit-learn is arguably the most advanced open source machine learning package available (http://scikit-learn.org). It provides simple and efficient tools for data mining and data analysis, covering supervised as well as unsupervised learning.

It provides tools for

- **Classification** Identifying to which set of categories a new observation belongs to.
- **Regression** Predicting a continuous value for a new example.
- **Clustering** Automatic grouping of similar objects into sets.
- **Dimensionality reduction** Reducing the number of random variables to consider.
- **Model selection** Comparing, validating, and choosing parameters and models.
- **Preprocessing** Feature extraction and normalization.

Here we use it for the simple case of a regression analysis.

In order to use *sklearn*, we need to input our data in the form of vertical vectors. Therefore, in our case, we will cast the DataFrame to `np.matrix` so that vertical arrays stay vertical once they are sliced off the data set. (This is made necessary by the awkward *Python* property that a one-dimensional slice of a *numpy* array is a vector, and so by definition is always horizontally oriented.)

```
In [20]: data = np.matrix( df )
```

Next, we create the regression objects, and fit the data to them. In this case, we will consider a clean set, which will fit a linear regression better, which consists of the data for all of the regions except Northern Ireland, and an original set consisting of the original data

```
In [21]: cln = LinearRegression()
    ...: org = LinearRegression()
    ...:
    ...: X, Y = data[:,2], data[:,1]
    ...: cln.fit( X[:-1], Y[:-1] )
    ...: org.fit( X, Y )
    ...:
    ...: clean_score    = '{0:.3f}'.format(
                                  cln.score( X[:-1], Y[:-1] ) )
    ...: original_score = '{0:.3f}'.format( org.score( X, Y ) )
    ...:
```

The next piece of code produces a scatter plot of the regions, with all of the regions plotted as empty blue circles, except for Northern Ireland, which is depicted as a red star.

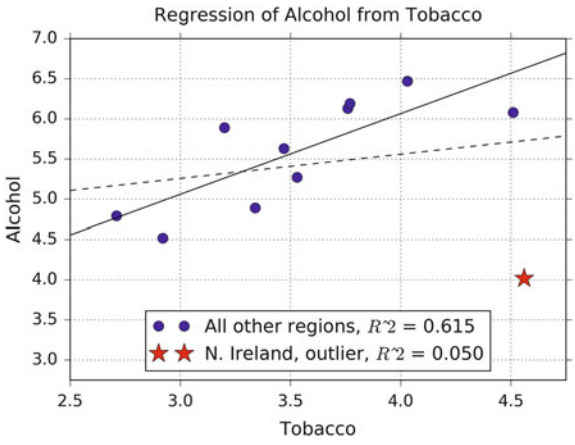
```
In [22]: mpl.rcParams['font.size']=16
    ...:
    ...: plt.plot( df.Tobacco[:-1], df.Alcohol[:-1], 'bo',
    ...:     markersize=10, label='All other regions,
    ...:     $R^$ = '+clean_score )
    ...:
    ...: plt.hold(True)
    ...: plt.plot( df.Tobacco[-1:], df.Alcohol[-1:], 'r*',
    ...:     ms=20, lw=10, label='N. Ireland, outlier,
    ...:     $R^2$ = '+original_score)
    ...:
```

The next part generates a set of points from 2.5 to 4.85, and then predicts the response of those points using the linear regression object trained on the clean and original sets, respectively.

```
In [23]: test = np.c_[np.arange(2.5, 4.85, 0.1)]
    ...: plt.plot( test, cln.predict( test ), 'k' )
    ...: plt.plot( test, org.predict( test ), 'k--' )
    ...:
```

Finally, we limit and label the axes, add a title, overlay a grid, place the legend at the bottom, and then save the figure.

```
In [24]: xlabel('Tobacco') ; xlim(2.5,4.75)
    ...: ylabel('Alcohol') ; ylim(2.75,7.0)
    ...: title('Regression of Alcohol from Tobacco')
    ...: grid()
    ...: legend(loc='lower center')
    ...: plt.show()
    ...:
```



11.5.9 Conclusion

Before you do anything, visualize your data. If your data is highly dimensional, then at least examine a few slices using boxplots. At the end of the day, use your own judgement about a model based on your knowledge of your domain. Statistical tests should guide your reasoning, but they should not dominate it. In most cases, your data will not align itself with the assumptions made by most of the available tests. A very interesting, openly accessible article on classical hypothesis testing has been written by Nuzzo (2014). A more intuitive approach to hypothesis testing is Bayesian analysis (see Chap. 14).

11.6 Assumptions of Linear Regression Models

Standard linear regression models with standard estimation techniques make a number of assumptions about the predictor variables, the response variables, and their relationship. Numerous extensions have been developed that allow each of these assumptions to be relaxed (i.e., reduced to a weaker form), and in some cases eliminated entirely. Some methods are general enough that they can relax

multiple assumptions at once, and in other cases this can be achieved by combining different extensions. Generally these extensions make the estimation procedure more complex and time-consuming, and may also require more data in order to get an accurate model.

The following are the major assumptions made by standard linear regression models with standard estimation techniques (e.g., ordinary least squares):

1. The independent variables (i.e., x) are exactly known.
2. Validity: Most importantly, the data you are analyzing should map to the research question you are trying to answer. This sounds obvious but is often overlooked or ignored because it can be inconvenient. For example, a linear regression does not properly describe a quadratic curve. A valid model should also result in the normality of errors.
3. Additivity and linearity: The most important mathematical assumption of the regression model is that its deterministic component is a linear function of the separate predictors.
4. Equal variance of errors.
5. Independence of errors from the values of the independent variables.
6. Independence of the independent variables.

Code: "ISP_anscombe.py"[6]: Code for the generation of Anscombe's Quartet.

Let me discuss each of them in some more detail[7]:

1. **Weak exogeneity**. This essentially means that the predictor variables x can be treated as fixed values, rather than random variables. This means, for example, that the predictor variables are assumed to be error-free, that is they are not contaminated with measurement errors. Although not realistic in many settings, dropping this assumption leads to significantly more difficult errors-in-variables models.
2. **Validity**. Figure 11.8 (*Anscombe's quartet*) shows how a linear fit can be meaningless if the wrong model is chosen, or if some of the assumptions are not met.
3. **Linearity**. This means that the mean of the response variable is a linear combination of the parameters (regression coefficients) and the predictor variables. Note that this assumption is much less restrictive than it may at first seem. Because the predictor variables are treated as fixed values (see above), linearity is really only a restriction on the parameters. The predictor variables themselves can

[6] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/anscombe.

[7] This section and the next chapter are based on Wikipedia https://en.wikipedia.org/wiki/Linear_regression, last accessed 21 Oct 2015.

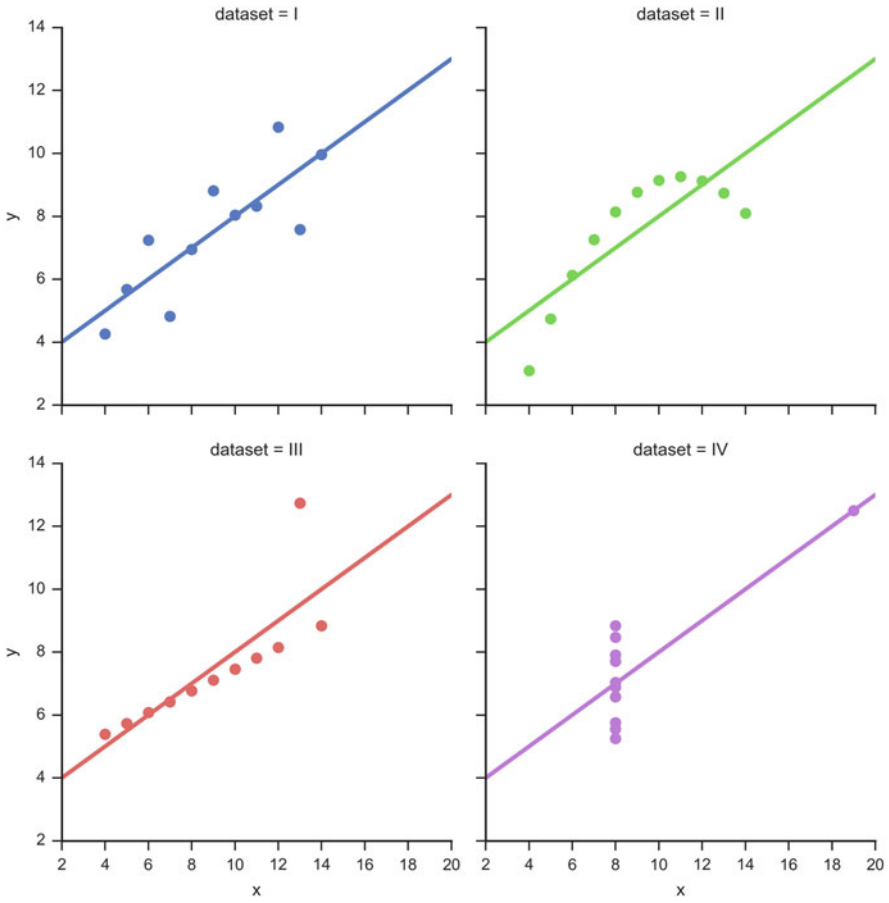


Fig. 11.8 The sets in the "Anscombe's quartet" have the same linear regression line but are themselves very different

be arbitrarily transformed, and in fact multiple copies of the same underlying predictor variable can be added, each one transformed differently. This trick is used, for example, in polynomial regression, which uses linear regression to fit the response variable as an arbitrary polynomial function (up to a given rank) of a predictor variable (see Sect. 11.2.2). This makes linear regression an extremely powerful inference method. In fact, models such as polynomial regression are often "too powerful," in that they tend to overfit the data. As a result, some kind of regularization must typically be used to prevent unreasonable solutions coming out of the estimation process. Common examples are *ridge regression* and *lasso regression*. *Bayesian linear regression* can also be used, which by its nature is more or less immune to the problem of overfitting. (In fact, ridge regression and lasso regression can both be viewed as special cases of Bayesian linear

regression, with particular types of prior distributions placed on the regression coefficients.)

4. **Constant variance** (aka *homoscedasticity*). This means that different response variables have the same variance in their errors, regardless of the values of the predictor variables. In practice this assumption is invalid (i.e., the errors are heteroscedastic) if the response variables can vary over a wide scale. In order to determine for heterogeneous error variance, or when a pattern of residuals violates model assumptions of homoscedasticity (error is equally variable around the 'best-fitting line' for all points of x), it is prudent to look for a "fanning effect" between residual error and predicted values. This is to say there will be a systematic change in the absolute or squared residuals when plotted against the predicting outcome. Error will not be evenly distributed across the regression line. Heteroscedasticity will result in the averaging over of distinguishable variances around the points to get a single variance that is inaccurately representing all the variances of the line. In effect, residuals appear clustered and spread apart on their predicted plots for larger and smaller values for points along the linear regression line, and the mean squared error for the model will be wrong. Typically, for example, a response variable whose mean is large will have a greater variance than one whose mean is small. For example, a given person whose income is predicted to be $100,000 may easily have an actual income of $80,000 or $120,000 (a standard deviation]] of around $20,000), while another person with a predicted income of $10,000 is unlikely to have the same $20,000 standard deviation, which would imply their actual income would vary anywhere between −$10,000 and $30,000. (In fact, as this shows, in many cases—often the same cases where the assumption of normally distributed errors fails—the variance or standard deviation should be predicted to be proportional to the mean, rather than constant.) Simple linear regression estimation methods give less precise parameter estimates and misleading inferential quantities such as standard errors when substantial heteroscedasticity is present. However, various estimation techniques (e.g., weighted least squares and heteroscedasticity-consistent standard errors) can handle heteroscedasticity in a quite general way. Bayesian linear regression techniques can also be used when the variance is assumed to be a function of the mean. It is also possible in some cases to fix the problem by applying a transformation to the response variable (e.g., fit the logarithm of the response variable using a linear regression model, which implies that the response variable has a log-normal distribution rather than a normal distribution).
5. **Independence of errors**. This assumes that the errors of the response variables are uncorrelated with each other. (Actual statistical independence is a stronger condition than mere lack of correlation and is often not needed, although it can be exploited if it is known to hold.) Some methods (e.g., generalized least squares) are capable of handling correlated errors, although they typically require significantly more data unless some sort of regularization is used to bias the model towards assuming uncorrelated errors. Bayesian linear regression is a general way of handling this issue.

6. **Lack of multicollinearity in the predictors**. For standard least squares estimation methods, the design matrix $\mathbf{X}$ must have full column rank p; otherwise, we have a condition known as *multicollinearity* in the predictor variables. (This problem is analyzed clearly and in detail by Kaplan (2009).) It can be triggered by having two or more perfectly correlated predictor variables (e.g., if the same predictor variable is mistakenly given twice, either without transforming one of the copies or by transforming one of the copies linearly). It can also happen if there is too little data available compared to the number of parameters to be estimated (e.g., fewer data points than regression coefficients). In the case of multicollinearity, the parameter vector $\boldsymbol{\beta}$ will be non-identifiable, it has no unique solution. At most we will be able to identify some of the parameters, i.e., narrow down its value to some linear subspace of R^p. Methods for fitting linear models with multicollinearity have been developed. Note that the more computationally expensive iterated algorithms for parameter estimation, such as those used in generalized linear models, do not suffer from this problem—and in fact it's quite normal when handling categorically valued predictors to introduce a separate indicator-variable predictor for each possible category, which inevitably introduces multicollinearity.

Beyond these assumptions, several other statistical properties of the data strongly influence the performance of different estimation methods:

- The statistical relationship between the error terms and the regressors plays an important role in determining whether an estimation procedure has desirable sampling properties such as being unbiased and consistent.
- The arrangement or probability distribution of the predictor variables in $\mathbf{X}$ has a major influence on the precision of estimates of $\boldsymbol{\beta}$. Sampling and design of experiments are highly developed subfields of statistics that provide guidance for collecting data in such a way to achieve a precise estimate of $\boldsymbol{\beta}$.

11.7 Interpreting the Results of Linear Regression Models

A fitted linear regression model can be used to identify the relationship between a single predictor variable x_j and the response variable y when all the other predictor variables in the model are "held fixed." Specifically, the interpretation of β_j is the expected change in y for a one-unit change in x_j when the other covariates are held fixed—that is, the expected value of the partial derivative of y with respect to x_j. This is sometimes called the *unique effect* of x_j on y. In contrast, the *marginal effect* of x_j on y can be assessed using a correlation coefficient or simple linear regression model relating x_j to y; this effect is the total derivative of y with respect to x_j.

Care must be taken when interpreting regression results, as some of the regressors may not allow for marginal changes (such as dummy variables, or the intercept term), while others cannot be held fixed. (Recall the example from the polynomial

fit in Sect. 11.2.2: it would be impossible to "hold t_j fixed" and at the same time change the value of t_i^2.)

It is possible that the unique effect can be nearly zero even when the marginal effect is large. This may imply that some other covariate captures all the information in x_j, so that once that variable is in the model, there is no contribution of x_j to the variation in y. Conversely, the unique effect of x_j can be large while its marginal effect is nearly zero. This would happen if the other covariates explained a great deal of the variation of y, but they mainly explain variation in a way that is complementary to what is captured by x_j. In this case, including the other variables in the model reduces the part of the variability of y that is unrelated to x_j, thereby strengthening the apparent relationship with x_j.

The meaning of the expression "held fixed" may depend on how the values of the predictor variables arise. If the experimenter directly sets the values of the predictor variables according to a study design, the comparisons of interest may literally correspond to comparisons among units whose predictor variables have been "held fixed" by the experimenter. Alternatively, the expression "held fixed" can refer to a selection that takes place in the context of data analysis. In this case, we "hold a variable fixed" by restricting our attention to the subsets of the data that happen to have a common value for the given predictor variable. This is the only interpretation of "held fixed" that can be used in an observational study.

The notion of a "unique effect" is appealing when studying a complex system where multiple interrelated components influence the response variable. In some cases, it can literally be interpreted as the causal effect of an intervention that is linked to the value of a predictor variable. However, it has been argued that in many cases multiple regression analysis fails to clarify the relationships between the predictor variables and the response variable when the predictors are correlated with each other and are not assigned following a study design.

Code: "ISP_simpleModels.py"[8] shows an example.

11.8 Bootstrapping

Another type of modeling is *bootstrapping*. Sometimes we have data describing a distribution, but do not know what type of distribution it is. So what can we do if we want to find out, e.g., confidence values for the mean?

[8] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/simpleModels.

The answer is bootstrapping. Bootstrapping is a scheme of *resampling*, i.e., taking additional samples repeatedly from the initial sample, to provide estimates of its variability. In a case where the distribution of the initial sample is unknown, bootstrapping is of special help in that it provides information about the distribution.

The application of bootstrapping in *Python* is much facilitated by the package *scikits.bootstrap* by Constantine Evans (http://github.org/cgevans/scikits-bootstrap).

Code: "ISP_bootstrapDemo.py"[9]: Example of bootstrapping the confidence interval for the mean of a sample distribution.

11.9 Exercises

11.1 Correlation

First read in the data for the average yearly temperature at the Sonnblick, Austria's highest meteorological observatory, from the file *Data/data_others/AvgTemp.xls*. Then calculate the Pearson and Spearman correlation, and Kendall's tau, for the temperature vs. year.

11.2 Regression

For the same data, calculate the yearly increase in temperature, assuming a linear increase with time. Is this increase significant?

11.3 Normality Check

For the data from the regression model, check if the model is ok by testing if the residuals are normally distributed (e.g., by using the Kolmogorov–Smirnov test).

[9] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/11_LinearModels/bootstrapDemo.

Chapter 12
Multivariate Data Analysis

When moving from two to many variables, the correlation coefficient gets replaced by the *correlation matrix*. And if we want to and predict the value of many other variables, linear regression has to be replaced by *multilinear regression*, sometimes also referred to as *multiple linear regression*.

However, many pitfalls loom when working with many variables! Consider the following example: golf tends to be played by richer people; and it is also known that on average the number of children goes down with increasing income. In other words, we have a fairly strong negative correlation between playing golf and the number of children, and one could be tempted to (falsely) draw the conclusion that playing golf reduces the fertility. But in reality it is the higher income which causes both effects. Kaplan (2009) nicely describes where those problems come from, and how best to avoid them.

12.1 Visualizing Multivariate Correlations

12.1.1 Scatterplot Matrix

If we have three to six variables that may be related to each other, we can use a *scatterplot matrix* to visualize the correlations between the different variables (Fig. 12.1):

```
import seaborn as sns
sns.set()

df = sns.load_dataset("iris")
sns.pairplot(df, hue="species", size=2.5)
```


T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_12

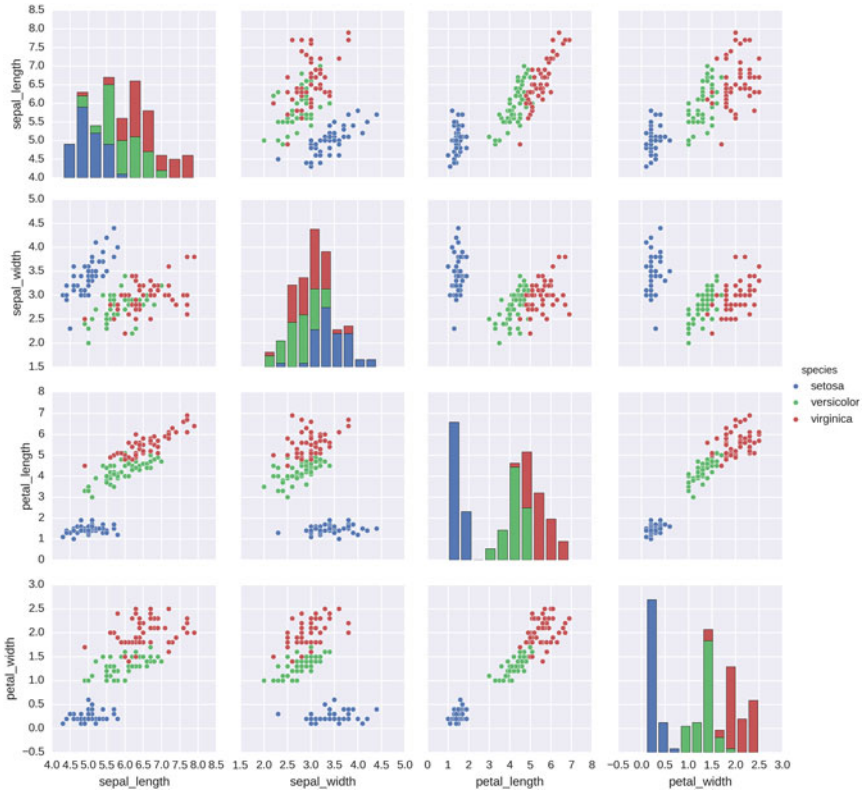


Fig. 12.1 Scatterplot matrix

12.1.2 Correlation Matrix

An elegant way to visualize the correlation between a large number of variables is the *correlation matrix*. Using *seaborn*, the following example shows how to implement a correlation matrix. In the example, the parameter for `np.random.RandomState` is the seed for the random number generation. The data are normally distributed dummy data, simulating 100 recordings from 30 different variables. The command `sns.corrplot` calculates and visualizes the cross correlation between each possible combination of variables (Fig.12.2):

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style="darkgrid")

rs = np.random.RandomState(33)
d = rs.normal(size=(100, 30))
```

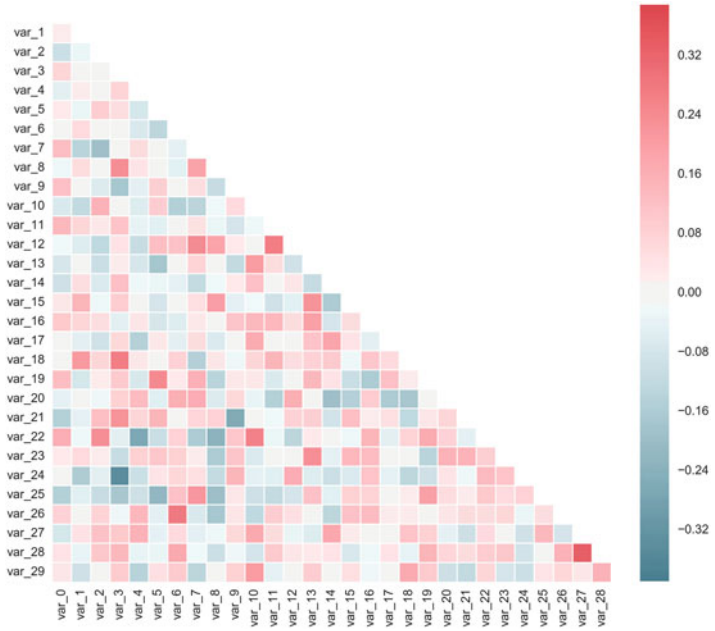


Fig. 12.2 Visualization of the correlation matrix

```
f, ax = plt.subplots(figsize=(9, 9))
cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.corrplot(d, annot=False, sig_stars=False,
             diag_names=False, cmap=cmap, ax=ax)
f.tight_layout()
```

12.2 Multilinear Regression

If we have truly independent variables, *multilinear regression* (or *multiple regression*) is a straightforward extension of the simple linear regression.

As an example, let us look at a *multiple regression* with covariates (i.e., independent variables) w_i and x_i. Suppose that the data are seven observations, and for each observed value to be predicted (y_i) there are two covariates that were also observed, w_i and x_i. The model to be considered is

$$y_i = \beta_0 + \beta_1 w_i + \beta_2 x_i + \epsilon_i \tag{12.1}$$

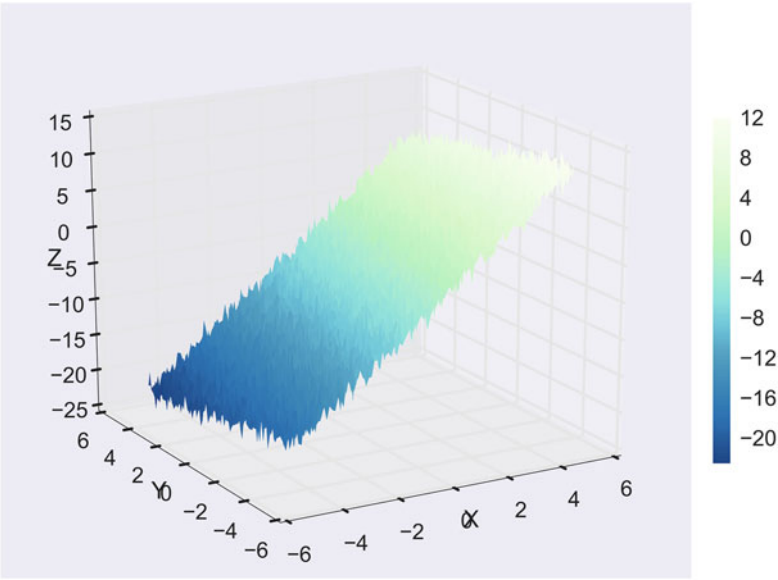


Fig. 12.3 Visualization of a multilinear regression

This model can be written in matrix terms as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & w_1 & x_1 \\ 1 & w_2 & x_2 \\ 1 & w_3 & x_3 \\ 1 & w_4 & x_4 \\ 1 & w_5 & x_5 \\ 1 & w_6 & x_6 \\ 1 & w_7 & x_7 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix} \tag{12.2}$$

Figure 12.3 shows how a data set with two covariates can be visualized as a three-dimensional surface. The code example in `C12_2_multipleRegression.py` shows how to generate this 3D plot, and how to find the corresponding regression coefficients.

Code: "ISP_multipleRegression.py"[1]: Multiple regression example, including solution for the regression coefficients and visualization.

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/12_Multivariate/multipleRegression.

Chapter 13
Tests on Discrete Data

Data can be discrete for different reasons. One is that they were acquired in a discrete way (e.g., levels in a questionnaire). Another one is that the paradigm only gives discrete results (e.g., rolling dice). For the analysis of such data, we can build on the tools for the analysis of *ranked data* that have already been covered in the previous chapters. Extending this analysis to statistical models of ranked data requires the introduction of *Generalized Linear Models* (*GLMs*). This chapter shows how to implement *logistic regression*, one frequently used application of GLMs, with the tools provided by *Python*.

13.1 Comparing Groups of Ranked Data

Ordinal data have clear rankings, e.g., "none–little–some–much–very much." However they are not continuous. For the analysis of such *rank ordered data* we can use the rank order methods described in Chap. 8:

Two groups	When comparing two rank ordered groups, we can use the *Mann–Whitney test* (Sect. 8.2.3)
Three or more groups	When comparing three or more rank ordered groups, we can use the *Kruskal–Wallis test* (Sect. 8.3.3)

Hypothesis tests allow to state quantitative probabilities on the likelihood of a hypothesis. *Linear Regression Modeling* allows to make predictions and gives confidence intervals for output variables that depend linearly on given inputs. But a large class of problems exceeds these requirements. For example, suppose we want to calculate the probability that a patient survives an operation, based on the amount of anesthetic he/she receives, and we want to find out how much anesthetic we can give the patient so that the chance of survival is at least 95 %.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_13

The answer to this question involves statistical modeling, and the tool of *logistic regression*. If more than two ordinal (i.e., naturally ranked) levels are involved, the so-called *ordinal logistic regression* is used.

To cover such questions *Generalized Linear Models* (*GLMs*) have been introduced, which extend the technique of linear regression to a wide range of other problems. A general coverage of GLMs is beyond the goals of this book, but can be found in the excellent book by Dobson and Barnett (2008). While Dobson only gives solutions in *R* and *Stata*, I have developed *Python* solutions for almost all examples in that book (https://github.com/thomas-haslwanter/dobson).

In the following chapter I want to cover one commonly used case, *logistic regression*, and its extension to *ordinal logistic regression*. The *Python* solutions presented should allow the readers to solve similar problems on their own, and should give a brief insight into Generalized Linear Models.

13.2 Logistic Regression

So far we have been dealing with linear models, where a linear change on the input leads to a corresponding linear change on the output (Fig. 11.2):

$$y = k * x + d + \epsilon \tag{13.1}$$

However, for many applications this model is not suitable. Suppose we want to calculate the probability that a patient survives an operation, based on the amount of anesthetic he/she receives. This probability is bounded on both ends, since it has to be a value between 0 and 1.

We can achieve such a bounded relationship, though, if we don't use the output of Eq. 13.1 directly, but wrap it by another function. Here we achieve this with the frequently used *logistic function*

$$p(y) = \frac{1}{1 + e^{\alpha+\beta y}} \tag{13.2}$$

13.2.1 Example: The Challenger Disaster

A good example for logistic regression is the simulation of the probability of an O-ring failure as a function of temperature for space shuttle launches. Here we analyze it with a logistic regression model, whereas in the next chapter we will look at it with the tools of Bayesian modeling.

On January 28, 1986, the twenty-fifth flight of the U.S. space shuttle program ended in disaster when one of the rocket boosters of the Shuttle Challenger exploded shortly after lift-off, killing all seven crew members. The presidential commission

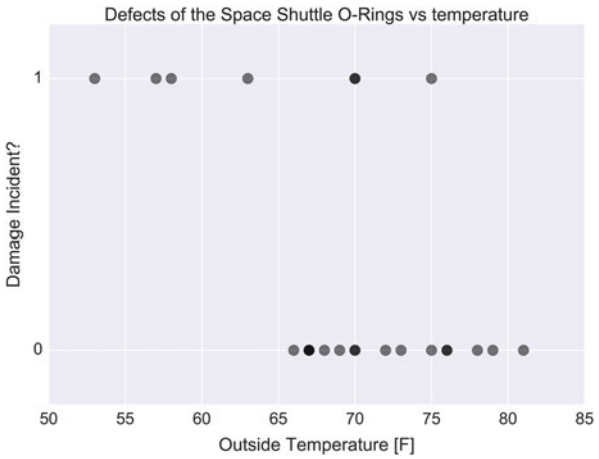


Fig. 13.1 Failure of O-rings during space shuttle launches, as a function of temperature

on the accident concluded that it was caused by the failure of an O-ring in a field joint on the rocket booster, and that this failure was due to a faulty design that made the O-ring unacceptably sensitive to a number of factors including outside temperature. Of the previous 24 flights, data were available on failures of O-rings on 23 (one was lost at sea), and these data were discussed on the evening preceding the Challenger launch, but unfortunately only the data corresponding to the 7 flights on which there was a damage incident were considered important and these were thought to show no obvious trend (top row points in Fig. 13.1). However, the full set of data indicates a trend to O-ring failure with lower temperatures. The full data set is shown in Fig. 13.1.

To simulate the probability of the O-rings failing, we can use the *logistic function* (Eq. 13.2):

With a given p-value, the *binomial distribution* (Sect. 6.2.2) determines the probability-mass-function for a given number n of shuttle launches. This tells us how likely it is to have $0, 1, 2, \ldots$ failures during those n launches.

Listing 13.1 L13_1_logitShort.py

```
# Import standard packages
import numpy as np
import os
import pandas as pd

# additional packages
from statsmodels.formula.api import glm
from statsmodels.genmod.families import Binomial

# Get the data
inFile = 'challenger_data.csv'
challenger_data = np.genfromtxt(inFile, skip_header=1,
                    usecols=[1, 2], missing_values='NA',
                    delimiter=',')
```

```
# Eliminate NaNs
challenger_data = challenger_data[~np.isnan(challenger_data
    [:, 1])]

# Create a dataframe, with suitable columns for the fit
df = pd.DataFrame()
df['temp'] = np.unique(challenger_data[:,0])
df['failed'] = 0
df['ok'] = 0
df['total'] = 0
df.index = df.temp.values

# Count the number of starts and failures
for ii in range(challenger_data.shape[0]):
    curTemp = challenger_data[ii,0]
    curVal  = challenger_data[ii,1]
    df.loc[curTemp,'total'] += 1
    if curVal == 1:
        df.loc[curTemp, 'failed'] += 1
    else:
        df.loc[curTemp, 'ok'] += 1

# fit the model

# --- >>> START stats <<< ---
model = glm('ok + failed ~ temp', data=df, family=Binomial())
    .fit()
# --- >>> STOP stats <<< ---

print(model.summary())
```

Code: "ISP_logisticRegression.py"[1] shows the full code for Fig. 13.2.

To summarize, we have three elements in our model

1. A probability distribution, which determines the probability of the outcome for a given trial (here the binomial distribution).
2. A linear model that relates the covariates (here the temperature) to the variates (the failure/success of an O-ring).
3. A *link-function* that wraps the linear model to produce the parameter for the probability distribution (here the logistic function).

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/13_LogisticRegression/LogisticRegression.

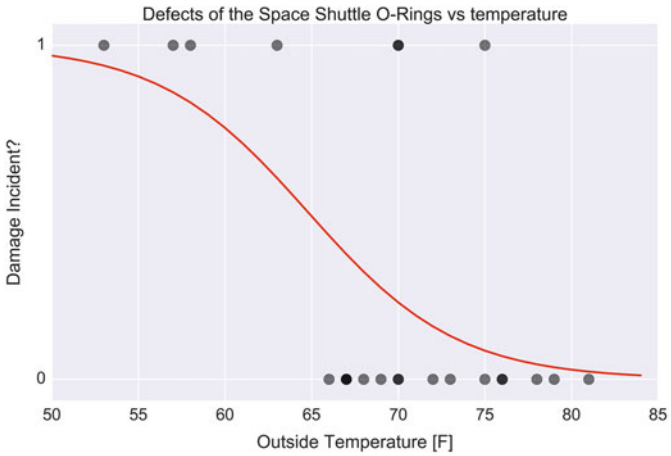


Fig. 13.2 Probability for O-ring failure

13.3 Generalized Linear Models

The model above is an example of a *Generalized Linear Model* (*GLM*), a powerful tool for the analysis of wide range of statistical models. Here only the general principles will be described. For details please refer to the excellent book by Dobson and Barnett (2008).

A GLM consists of three elements:

1. A probability distribution from the *exponential family*.
2. A linear predictor $\eta = \mathbf{X} \cdot \boldsymbol{\beta}$.
3. A link function g such that $E(Y) = \mu = g^{-1}(\eta)$.

13.3.1 Exponential Family of Distributions

The exponential family is a set of probability distributions of a certain form, specified below. This special form is chosen for mathematical convenience, on account of some useful algebraic properties, as well as for generality, as exponential families are in a sense very natural sets of distributions to consider. The exponential families include many of the most common distributions, including the normal, exponential, chi-squared, Bernoulli, Poisson distribution, and many others. (A common distribution that is not from the exponential family is the t-distribution.)

In mathematical terms, a distribution from the exponential family has the general form

$$f_X(x|\theta) = h(x)g(\theta)\exp\left(\eta(\theta) \cdot T(x)\right) \tag{13.3}$$

where $T(x), h(x), g(\theta)$, and $\eta(\theta)$ are known functions. While Eq. 13.3 is dauntingly abstract, it provides the theocratical basis for a common consistent treatment of a large number of different statistical models.

13.3.2 Linear Predictor and Link Function

The linear predictor for GLM is the same as the one used for *linear models*. The resulting terminology is unfortunately fairly confusing:

General Linear Models are models of the form $y = \mathbf{X} \cdot \beta + \epsilon$, where ϵ is normally distributed (see Chap. 11).

Generalized Linear Models encompass a much wider class of models, including all distributions from the exponential family *and* a link function. The *linear predictor* $\eta = \mathbf{X} \cdot \beta$ is now "only" an element of the distribution function any more, which provides the flexibility of GLMs.

The *link function* is an arbitrary function, with the only requirements that it is continuous and invertible.

13.4 Ordinal Logistic Regression

13.4.1 Problem Definition

Section 13.2 has shown one example of a GLM, logistic regression. In this section I want to show how further generalization, from a *yes/no* decision to a decision for one-of-many groups ($Group_1/Group_2/Group_3$), leads into the area of numerical optimization.

The *logistic ordinal regression* model,[2] also known as the proportional odds model, was introduced in the early 1980s by McCullagh (1980; McCullagh and Nelder 1989) and is a generalized linear model specially tailored for the case of predicting ordinal variables, that is, variables that are discrete (as in classification) but which can be ordered (as in regression). It can be seen as an extension of the logistic regression model described above to the ordinal setting (Fig. 13.3).

$$P(y \leq j|X_i) = \phi(\theta_j - w^T X_i) = \frac{1}{1 + \exp(w^T X_i - \theta_j)} \tag{13.4}$$

[2]This section has been taken with permission from Fabian Pedregosa's blog on ordinal logistic regression, http://fa.bianp.net/blog/2013/logistic-ordinal-regression/.

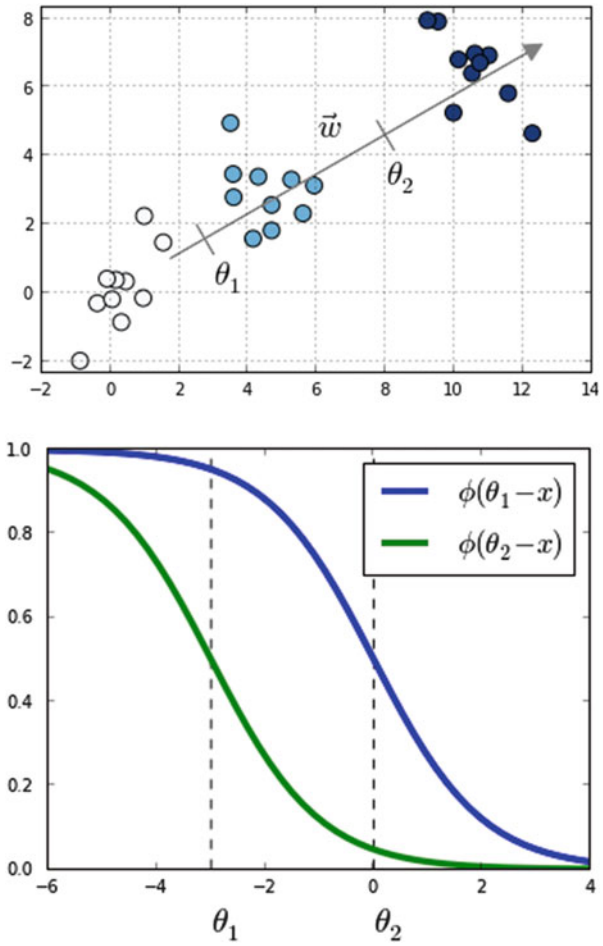


Fig. 13.3 Toy example with three classes denoted in different *colors*. Also shown are the vector of coefficients **w** and the thresholds θ_0 and θ_1. (Figure from Fabian Pedregosa, with permission.)

where w and θ are vectors to be estimated from the data, and ϕ is the logistic function defined as $\phi(t) = \frac{1}{1+exp(-t)}$.

Compared to multiclass logistic regression, we have added the constraint that the hyperplanes that separate the different classes are *parallel* for all classes, that is, the vector w is common across classes. To decide to which class will X_i be predicted we make use of the vector of thresholds θ. If there are K different classes, θ is a nondecreasing vector (that is, $\theta_1 \leq \theta_2 \leq \ldots \leq \theta_{K-1}$) of size $K-1$. We will then assign the class j if the prediction $w^T X$ (recall that it's a linear model) lies in the interval $[\theta_{j-1}, \theta_j[$. In order to keep the same definition for extremal classes, we define $\theta_0 = -\infty$ and $\theta_K = +\infty$.

The intuition is that we are seeking a vector w such that $X \cdot w$ produces a set of values that are well separated into the different classes by the different thresholds θ_i. We choose a logistic function to model the probability $P(y \leq j|X_i)$, but other choices are also possible. In the proportional hazards model (McCullagh 1980) the probability is modeled as

$$-\log(1 - P(y \leq j|X_i)) = \exp(\theta_j - w^T X_i) \tag{13.5}$$

Other link functions are possible, where the link function satisfies $link(P(y \leq j|X_i)) = \theta_j - w^T X_i$. Under this framework, the logistic ordinal regression model has a logistic link function, and the proportional hazards model has a log-log link function.

The logistic ordinal regression model is also known as the *proportional odds model*, because the ratio of corresponding odds for two different samples X_1 and X_2 is $\exp(w^T(X_1 - X_2))$ and so does not depend on the class j but only on the difference between the samples X_1 and X_2.

13.4.2 Optimization

Model estimation can be posed as an optimization problem. Here, we minimize the loss function for the model, defined as minus the log-likelihood:

$$\mathscr{L}(w, \theta) = -\sum_{i=1}^{n} \log(\phi(\theta_{y_i} - w^T X_i) - \phi(\theta_{y_i - 1} - w^T X_i)) \tag{13.6}$$

In this sum all terms are convex on w, thus the loss function is convex over w. We use the function `fmin_slsqp` in `scipy.optimize` to optimize $\mathscr{L}$ under the constraint that θ is a nondecreasing vector.

Using the formula $\log(\phi(t))' = (1 - \phi(t))$, we can compute the gradient of the loss function as

$$\nabla_w \mathscr{L}(w, \theta) = \sum_{i=1}^{n} X_i(1 - \phi(\theta_{y_i} - w^T X_i) - \phi(\theta_{y_i - 1} - w^T X_i))$$

$$\nabla_\theta \mathscr{L}(w, \theta) = \sum_{i=1}^{n} e_{y_i} \left(1 - \phi(\theta_{y_i} - w^T X_i) - \frac{1}{1 - \exp(\theta_{y_i - 1} - \theta_{y_i})}\right)$$

$$+ e_{y_i - 1} \left(1 - \phi(\theta_{y_i - 1} - w^T X_i) - \frac{1}{1 - \exp(-(\theta_{y_i - 1} - \theta_{y_i}))}\right)$$

where e_i is the ith canonical vector.

13.4.3 Code

The attached code sample implements a *Python* version of this algorithm using *scipy*'s `optimize.fmin_slsqp` function. This takes as arguments the loss function, the gradient denoted before and a function that is >0 when the inequalities on θ are satisfied.

13.4.4 Performance

Fabian Pedregosa has compared the prediction accuracy of this model in the sense of mean absolute error on the Boston house-prices data set. To have an ordinal variable, he rounded the values to the closest integer, which resulted in a problem of size $506 * 13$ with 46 different target values (Fig. 13.4). Although not a huge increase in accuracy, this model did give better results on this particular data set:

Here, ordinal logistic regression is the best-performing model, followed by a Linear Regression model and a One-versus-All Logistic regression model as implemented in scikit-learn.

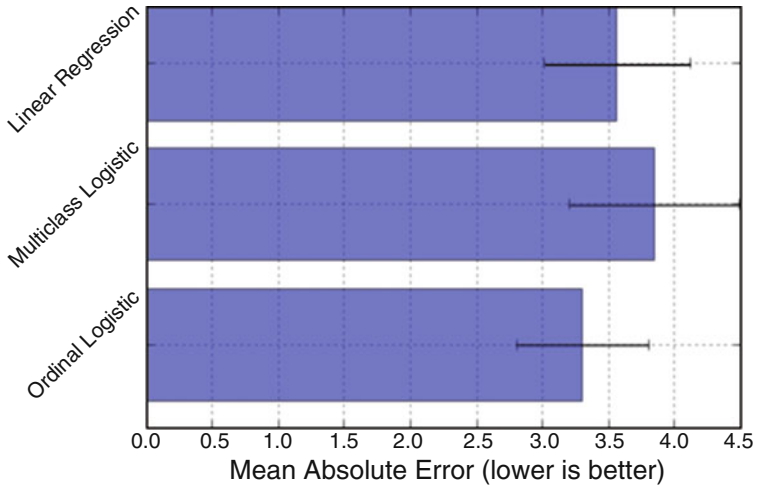


Fig. 13.4 Errors in predicting house-prices in Boston, for three different types of models. Ordinal logistic regression produces the best predictions. Figure from Fabian Pedregosa, with permission

Code: "ISP_ordinalLogisticRegression.py"[3] corresponding code by Fabian Pedregosa.

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/13_LogisticRegression/OrdinalLogisticRegression.

Chapter 14
Bayesian Statistics

Calculating probabilities is only one part of statistics. Another is the interpretation of them—and the consequences that come with different interpretations.

So far we have restricted ourselves to the *frequentist interpretation*, which interprets p as the *frequency of an occurrence*: if an outcome of an experiment has the probability p, it means that if that experiment is repeated N times (where N is a large number), then we observe this specific outcome $N * p$ times. Or in other words: given a certain model, we look at the likelihood to find the observed set of data.

The *Bayesian interpretation* of p is quite different, and interprets p as our *belief of the likelihood* of a certain outcome. Here we take the observed data as fixed, and look at the likelihood to find certain model parameters. For some events, this makes a lot more sense. For example, a presidential election is a one-time event, and we will never have a large number of N repetitions.

14.1 Bayesian vs. Frequentist Interpretation

In addition to this difference in interpretation, the Bayesian approach has another advantage: it lets us bring in *prior knowledge* into the calculation of the probability p, through the application of *Bayes' Theorem*:

In its most common form, it is:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}. \tag{14.1}$$

In the Bayesian interpretation, probability measures a degree of belief. Bayes' theorem then links the degree of belief in a proposition before and after accounting for evidence. For example, suppose it is believed with 50 % certainty that a coin is twice as likely to land heads than tails. If the coin is flipped a number of times

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6_14

and the outcomes observed, that degree of belief may rise, fall, or remain the same depending on the results.

John Maynard Keynes, a great economist and thinker, said "When the facts change, I change my mind. What do you do, sir?" This quote reflects the way a Bayesian updates his or her beliefs after seeing evidence.

For proposition A and evidence B,

- $P(A)$, the *prior probability*, is the initial degree of belief in A.
- $P(A|B)$, the *posterior probability*, is the degree of belief having accounted for B. It can be read as *"the probability of A, given that B is the case"*.
- the quotient $P(B|A)/P(B)$ represents the support B provides for A.

If the number of available data points is large, the difference in interpretation does typically not change the result significantly. If the number of data points is small, however, the possibility to bring in external knowledge may lead to a significantly improved estimation of p.

14.1.1 Bayesian Example

Suppose a man told you he had a nice conversation with someone on the train. Not knowing anything about this conversation, the probability that he was speaking to a woman is 50 % (assuming the speaker was as likely to strike up a conversation with a man as with a woman). Now suppose he also told you that his conversational partner had long hair. It is now more likely he was speaking to a woman, since women are more likely to have long hair than men. Bayes' theorem can be used to calculate the probability that the person was a woman.

To see how this is done, let W represent the event that the conversation was held with a woman, and L denote the event that the conversation was held with a long-haired person. It can be assumed that women constitute half the population for this example. So, not knowing anything else, the probability that W occurs is $P(W) = 0.5$.

Suppose it is also known that 75 % of women have long hair, which we denote as $P(L|W) = 0.75$ (read: the probability of event L given event W is 0.75, meaning that the probability of a person having long hair (event "L"), given that we already know that the person is a woman ("event W") is 75 %). Likewise, suppose it is known that 15 % of men have long hair, or $P(L|M) = 0.15$, where M is the complementary event of W, i.e., the event that the conversation was held with a man (assuming that every human is either a man or a woman).

Our goal is to calculate the probability that the conversation was held with a woman, given the fact that the person had long hair, or, in our notation, $P(W|L)$. Using the formula for Bayes' theorem, we have:

$$P(W|L) = \frac{P(L|W)P(W)}{P(L)} = \frac{P(L|W)P(W)}{P(L|W)P(W) + P(L|M)P(M)}$$

where we have used the *law of total probability* to expand $P(L)$. The numeric answer can be obtained by substituting the above values into this formula (the algebraic multiplication is annotated using "$*$"). This yields

$$P(W|L) = \frac{0.75 * 0.50}{0.75 * 0.50 + 0.15 * 0.50} = \frac{5}{6} \approx 0.83,$$

i.e., the probability that the conversation was held with a woman, given that the person had long hair, is about 83 %.

Another way to do this calculation is as follows. Initially, it is equally likely that the conversation is held with a woman as with a man, so the prior odds are 1:1. The respective chances that a man and a woman have long hair are 15 and 75 %. It is five times more likely that a woman has long hair than that a man has long hair. We say that the likelihood ratio or Bayes factor is 5:1. Bayes' theorem in odds form, also known as *Bayes' rule*, tells us that the posterior odds that the person was a woman is also 5:1 (the prior odds, 1:1, times the likelihood ratio, 5:1). In a formula:

$$\frac{P(W|L)}{P(M|L)} = \frac{P(W)}{P(M)} \cdot \frac{P(L|W)}{P(L|M)}.$$

14.2 The Bayesian Approach in the Age of Computers

Bayes' theorem was named after the Reverend Thomas Bayes (1701–1761), who studied how to compute a distribution for the probability parameter of a binomial distribution. So it has been around for a long time. The reason Bayes' theorem has become so popular in statistics in recent years is the cheap availability of massive computational power. This allows the empirical calculation of posterior probabilities, one-by-one, for each new piece of evidence. This, combined with statistical approaches like Markov-chain–Monte-Carlo simulations, has allowed radically new statistical analysis procedures, and has led to what may be called "statistical trench warfare" between the followers of the different philosophies. If you don't believe me, check the corresponding discussions on the WWW.

For more information on that topic, check out (in order of rising complexity)

- Wikipedia, which has some nice explanations under *"Bayes . . ."*
- Bayesian Methods for Hackers (http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/), a nice, free ebook, providing a practical introduction to the use of PyMC (see below).
- The PyMC User Guide (http://pymc-devs.github.io/pymc/): PyMC is a very powerful *Python* package which makes the application of MCMC techniques very simple.

- *Pattern Classification*, does not avoid the mathematics, but uses it in a practical manner to help you gain a deeper understanding of the most important machine learning techniques (Duda 2004).
- *Pattern Recognition and Machine Learning*, a comprehensive, but often quite technical book by Bishop (2007).

14.3 Example: Analysis of the Challenger Disaster with a Markov-Chain–Monte-Carlo Simulation

In the following we will reanalyze the data from the Challenger disaster already used in the previous chapter, but this time with a Markov-chain–Monte-Carlo (MCMC) simulation. [This chapter is an excerpt of the excellent 'Bayesian Methods for Hackers' (Pilon 2015, with permission from the author).]

The data are again from the Challenger disaster (see Sect. 13.2.1). To perform the simulation, we are going to use *PyMC*, a *Python* module that implements Bayesian statistical models and fitting algorithms, including MCMC-simulations (http://pymc-devs.github.io/pymc/). Its flexibility and extensibility make it applicable to a large suite of problems. Along with core sampling functionality, PyMC includes methods for summarizing output, plotting, goodness-of-fit, and convergence diagnostics.

PyMC provides functionalities to make Bayesian analysis as painless as possible. Here is a short list of some of its features:

- Fits Bayesian statistical models with Markov chain Monte Carlo and other algorithms.
- Includes a large suite of well-documented statistical distributions.
- Includes a module for modeling Gaussian processes.
- Creates summaries including tables and plots.
- Traces can be saved to the disk as plain text, *Python* pickles, SQLite or MySQL database, or hdf5 archives.
- Extensible: easily incorporates custom step methods and unusual probability distributions.
- MCMC loops can be embedded in larger programs, and results can be analyzed with the full power of *Python*.

To simulate the probability of the O-rings failing, we need a function that goes from one to zero. We again apply the logistic function:

$$p(t) = \frac{1}{1 + e^{\alpha+\beta t}}$$

In this model, the variable β that describes how quickly the function changes from 1 to 0, and α indicates the location of this change.

Using the *Python* package PyMC, a Monte-Carlo simulation of this model can be done remarkably easily:

```
# --- Perform the MCMC-simulations ---
temperature = challenger_data[:, 0]
D = challenger_data[:, 1]  # defect or not?

# Define the prior distributions for alpha and beta
# 'value' sets the start parameter for the simulation
# The second parameter for the normal distributions is the
# "precision", i.e. the inverse of the standard deviation
beta = pm.Normal("beta", 0, 0.001, value=0)
alpha = pm.Normal("alpha", 0, 0.001, value=0)

# Define the model-function for the temperature
@pm.deterministic
def p(t=temperature, alpha=alpha, beta=beta):
    return 1.0 / (1. + np.exp(beta * t + alpha))

# connect the probabilities in `p` with our observations
# through a Bernoulli random variable.
observed = pm.Bernoulli("bernoulli_obs", p, value=D,
                        observed=True)

# Combine the values to a model
model = pm.Model([observed, beta, alpha])

# Perform the simulations
map_ = pm.MAP(model)
map_.fit()
mcmc = pm.MCMC(model)
mcmc.sample(120000, 100000, 2)
```

From this simulation, we obtain not only our best estimate for α and β, but also information about our uncertainty about these values (Fig. 14.1).

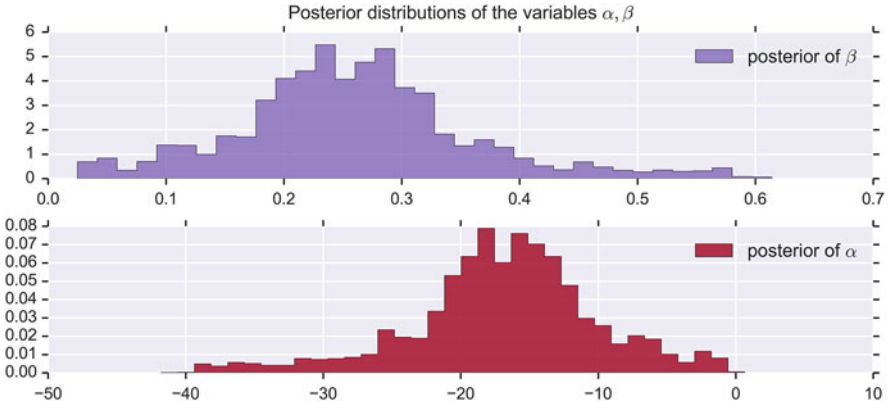


Fig. 14.1 Probabilities for alpha and beta, from the MCMC simulations

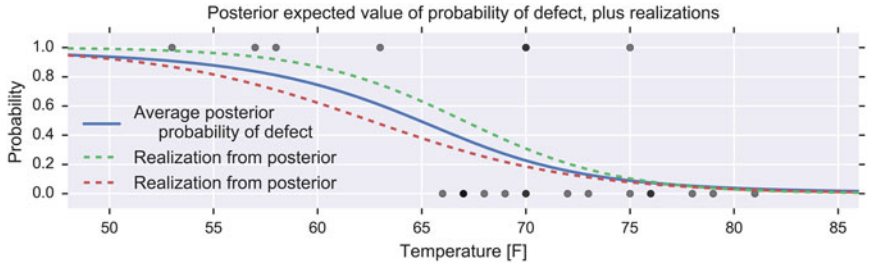


Fig. 14.2 Probability for an O-ring failure, as a function of temperature

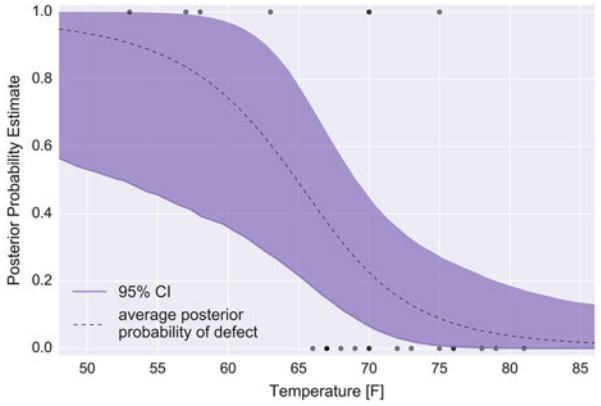


Fig. 14.3 Ninety-five percent confidence intervals for the probability for an O-ring failure

The probability curve for an O-ring failure thus looks as shown in Fig. 14.2.

One advantage of the MCMC simulation is that it also provides confidence intervals for the probability (Fig. 14.3).

On the day of the Challenger disaster, the outside temperature was 31 °F. The posterior distribution of a defect occurring, given this temperature, almost guaranteed that the Challenger was going to be subject to defective O-rings.

Code: "ISP_bayesianStats.py"[1]: Full implementation of the MCMC simulation.

[1] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/14_Bayesian/bayesianStats.

14.4 Summing Up

The Bayesian approach offers a natural framework to deal with parameter and model uncertainty, and has become very popular, especially in areas like machine learning. However, it is computationally much more intensive, and therefore typically implemented with the help of existing tools like *PyMC* or *scikit-learn*.

This also shows one of the advantages of free and open languages like *Python*: they provide a way to build on the existing work of the scientific community, and only require your enthusiasm and dedication. Therefore I would like to finish this book by thanking the *Python* community for the incredible amount of work that they have put into the development of core *Python* and *Python* packages. And I hope that this book allows you to share some of my enthusiasm for this vibrant language.

Solutions[1]

Problems of Chap. 2

2.1 Data Input

Listing 14.1 S2_python.py

```
''' Solution to Exercise "Data Input" '''

# author: Thomas Haslwanter, date: Oct-2015

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import urllib
import io
import zipfile

def getDataDobson(url, inFile):
    '''Extract data from a zipped-archive'''

    # get the zip-archive
    GLM_archive = urllib.request.urlopen(url).read()

    # make the archive available as a byte-stream
    zipdata = io.BytesIO()
    zipdata.write(GLM_archive)

    # extract the requested file from the archive, as a
        pandas XLS-file
```

[1]Published with the kind permission of © Thomas Haslwanter 2015. Published under the Creative Commons Attribution-ShareAlike International License (CC BY-SA 4.0).

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6

```
    myzipfile = zipfile.ZipFile(zipdata)
    xlsfile = myzipfile.open(inFile)

    # read the xls-file into Python, using Pandas, and return
        the extracted data
    xls = pd.ExcelFile(xlsfile)
    df  = xls.parse('Sheet1', skiprows=2)

    return df

if __name__ == '__main__':
    # 1.1 Numpy --------------------
    # Read in a CSV file, and show the top:
    inFile1 = 'swim100m.csv'
    data = pd.read_csv(inFile1)
    print(data.head())

    # Read in an excel file, and show the bottom
    inFile2 = 'Table 2.8 Waist loss.xls'

    xls = pd.ExcelFile(inFile2)
    data = xls.parse('Sheet1', skiprows=2)
    print(data.tail())

    # Read in a zipped data-file from the WWW
    url = r'http://cdn.crcpress.com/downloads/C9500/GLM_data.
        zip'
    inFile = r'GLM_data/Table 2.8 Waist loss.xls'

    df = getDataDobson(url, inFile)
    print(df.tail())
```

2.2 First Steps with *pandas*

Listing 14.2 S2_pandas.py

```
''' Solution to Exercise "First Steps with pandas":
Generate a sine and cosine wave using pandas' DataFrames,
and write them to an out-file.
'''

# author: Thomas Haslwanter, date: Sept-2015

import numpy as np
import pandas as pd

# Set the parameters
rate = 10
dt = 1/rate
freq = 1.5

# Derived quantities
omega = 2*np.pi*freq
```

```
# Generate the data
t = np.arange(0,10,dt)
y = np.sin(omega*t)
z = np.cos(omega*t)

# Assemble them in a DataFrame
df = pd.DataFrame({'Time':t, 'YVals':y, 'ZVals':z})

# Show the top 5 values
print(df.head())

# Save lines 10-15 of the y- and z-values to an outfile
outfile = 'out.txt'
df[10:16][['YVals', 'ZVals']].to_csv(outfile)
print('Data written to {0}'.format(outfile))
input('Done')
```

Problems of Chap. 4

4.1 Displaying Data

Listing 14.3 S4_display.py

```
''' Solution for Exercise "Data Display"
Read in weight-data recorded from newborns, and analyze the
data based on the gender of the baby.'''

# author: Thomas Haslwanter, date: Oct-2015

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd
import seaborn as sns
import os

def getData():
    '''Read in data from a text-file, and return them as
        labelled DataFrame'''

    # Set directory and infile
    dataDir = '.'
    inFile = 'babyboom.dat.txt'

    # Read and label the data
    os.chdir(dataDir)
    data = pd.read_csv(inFile, sep='[ ]*', header=None,
        engine='python',
                       names= ['TOB', 'sex', 'Weight', '
                           Minutes'])

    # Eliminate "Minutes", since this is redundant
    df = data[['Minutes', 'sex', 'Weight']]

    return(df)

def showData(df):
    '''Graphical data display'''

    # Show the data: first all of them ....
    plt.plot(df.Weight, 'o')

    plt.title('All data')
    plt.xlabel('Subject-Nr')
    plt.ylabel('Weight [g]')
    plt.show()

    # To make the plots easier to read, replace "1/2" with "
        female/male"
```

```
        df.sex = df.sex.replace([1,2], ['female', 'male'])

        # ... then show the grouped plots
        df.boxplot(by='sex')
        plt.show()

        # Display statistical information numerically
        grouped = df.groupby('sex')
        print(grouped.describe())

        # This is a bit fancier: scatter plots, with labels and
            individual symbols
        symbols = ['o', 'D']
        colors = ['r', 'b']

        fig = plt.figure()
        ax = fig.add_subplot(111)

        # "enumerate" provides a counter, and variables can be
            assigned names in one step if
        # the "for"-loop uses a tuple as input for each loop:
        for (ii, (sex, group)) in enumerate(grouped):
            ax.plot(group['Weight'], marker = symbols[ii],
                linewidth=0, color = colors[ii], label=sex)

        ax.legend()
        ax.set_ylabel('Weight [g]')
        plt.show()

        # Fancy finish: a kde-plot
        df.Weight = np.double(df.Weight)    # kdeplot requires
            doubles

        sns.kdeplot(grouped.get_group('male').Weight, color='b',
            label='male')
        plt.hold(True)
        sns.kdeplot(grouped.get_group('female').Weight, color='r'
            , label='female')

        plt.xlabel('Weight [g]')
        plt.ylabel('PDF(Weight)')
        plt.show()

    # Statistics: are the data normally distributed?
    def isNormal(data, dataType):
        '''Check if the data are normally distributed'''
        alpha = 0.05
        (k2, pVal) = stats.normaltest(data)
        if pVal < alpha:
            print('{0} are NOT normally distributed.'.format(
                dataType))
        else:
            print('{0} are normally distributed.'.format(dataType
                ))
```

```
def checkNormality(df):
    '''Check selected data vlaues for normality'''

    grouped = df.groupby('sex')

    # Run the check for male and female groups
    isNormal(grouped.get_group('male').Weight, 'male')
    isNormal(grouped.get_group('female').Weight, 'female')

if __name__ == '__main__':
    '''Main Program'''

    df = getData()
    showData(df)
    checkNormality(df)

    # Wait for an input before exiting
    input('Done - Hit any key to continue')
```

Problems of Chap. 6

6.1 Sample Standard Deviation

Listing 14.4 S6_sd.py

```
''' Solution to Exercise "Sample Standard Deviation" '''

# author: Thomas Haslwanter, date: Sept-2015

import numpy as np

x = np.arange(1,11)
print('The standard deviation of the numbers from 1 to 10 is
    {0:4.2f}'.format(np.std(x, ddof=1)))
```

6.2 Normal Distribution

Listing 14.5 S6_normDist.py

```
''' Solution to Exercise "Normal Distribution" '''

# author: Thomas Haslwanter, date: Sept-2015

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd
import seaborn as sns


# Generate a PDF, with a mean of 5 and a standard deviation
    of 3
nd = stats.norm(5,3)

# Generate 1000 data from this distribution
data = nd.rvs(1000)

# Standard error
se = np.std(data, ddof=1)/np.sqrt(1000)
print('The standard error is {0}'.format(se))

# Histogram
plt.hist(data)
plt.show()

# 95% confidence interval
print('95% Confidence interval: {0:4.2f} - {1:4.2f}'.format(
    nd.ppf(0.025), nd.ppf(0.975)))

# SD for hip implants
```

```
nd = stats.norm()
numSDs = nd.isf(0.0005)
tolerance = 1/numSDs
print('The required SD to fulfill both requirements = {0:6.4f
    } mm'.format(tolerance))
```

6.3 Other Continuous Distributions

Listing 14.6 S6_continuous.py

```
'''Solution for Exercise "Continuous Distribution Functions"
    '''

# author: Thomas Haslwanter, date: Oct-2015

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# T-distibution
    ----------------------------------------------------
# Enter the data
x = [52, 70, 65, 85, 62, 83, 59]
''' Note that "x" is a Python "list", not an array!
 Arrays come with the numpy package, and have to contain all
     elements of the same type.
 Lists can mix different types, e.g. "x = [1, 'a', 2]"
 '''

# Generate the t-distribution: note that the degrees of
    freedom is the
# length of the data minus 1.
# In Python, the length of an object x is given by "len(x)"
td = stats.t(len(x)-1)
alpha = 0.01

# From the t-distribution, you use the "PPF" function and
    multiply it with the standard error
tval = abs( td.ppf(alpha/2)*stats.sem(x) )
print('mean +/- 99%CI = {0:3.1f} +/- {1:3.1f}'.format(np.mean
    (x),tval))

# Chi2-distribution, with 3 DOF
    ------------------------------------------------
# Define the normal distribution
nd = stats.norm()

# Generate three sets of random variates from this
    distribution
numData = 1000
data1 = nd.rvs(numData)
data2 = nd.rvs(numData)
data3 = nd.rvs(numData)
```

```
# Show a histogram of the sum of the squares of these random
    data
plt.hist(data1**2+data2**2 +data3**2, 100)
plt.show()

# F-distribution
    ------------------------------------------------
apples1 = [110, 121, 143]
apples2 = [88, 93, 105, 124]
fval = np.std(apples1, ddof=1)/np.std(apples2, ddof=1)
fd = stats.distributions.f(len(apples1),len(apples2))
pval = fd.cdf(fval)
print('The p-value of the F-distribution = {0}.'.format(pval)
    )
if pval>0.025 and pval<0.975:
    print('The variances are equal.')
```

6.4 Discrete Distributions

Listing 14.7 S6_discrete.py

```
'''Solution for Exercise "Continuous Distribution Functions"
    '''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats

# Binomial distribution
    ------------------------------------------------
# Generate the distribution
p = 0.37
n = 15
bd = stats.binom(n, p)

# Select the interesting numbers, and calculate the "
    Probability Mass Function" (PMF)
x = [3,6,10]
y = bd.pmf(x)

# To print the result, we use the "zip" function to generate
    pairs of numbers
for num, solution in zip(x,y):
    print('The chance of finding {0} students with blue eyes
        is {1:4.1f}%.'.format(num, solution*100))

# Poisson distribution
    ------------------------------------------------
# Generate the distribution.
# Watch out NOT to divide integers, as "3/4" gives "0" in
    Python 2.x!
prob = 62./(365./7)
pd = stats.poisson(prob)
```

```
# Select the interesting numbers, calculate the PMF, and
    print the results
x = [0,2,5]
y = pd.pmf(x)*100
for num, solution in zip(x,y):
    print('The chance of haveing {0} fatal accidents in one
        week is {1:4.1f}%.'.format(num,solution))

# The last line just makes sure that the program does not
    close, when it is run from the commandline.
input('Done! Thanks for using programs by thomas.')
```

Problems of Chap. 8

8.1 Comparing One or Two Groups

Listing 14.8 S8_twoGroups.py

```
'''Solution for Exercise "Comparing Groups" '''

# author: Thomas Haslwanter, date: Sept-2015

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import scipy as sp
import os

def oneGroup():
    '''Test of mean value of a single set of data'''

    print('Single group of data
        ========================================')

    # First get the data
    data = np.array([5260, 5470, 5640, 6180, 6390, 6515,
        6805, 7515, 7515, 8230, 8770], dtype=np.float)
    checkValue = 7725   # value to compare the data to

    # 4.1.1. Normality test
    # We don't need the first parameter, so we just assign
        the output to the dummy variable "_"
    (_, p) = stats.normaltest(data)
    if p > 0.05:
        print('Data are distributed normally, p = {0}'.format
            (p))

    # 4.1.2. Do the onesample t-test
    t, prob = stats.ttest_1samp(data, checkValue)
    if prob < 0.05:
        print('With the one-sample t-test, {0:4.2f} is
            significantly different from the mean (p={1:5.3f})
            .'.\
        format(checkValue, prob))
    else:
        print('No difference from reference value with
            onesample t-test.')

    # 4.1.3. This implementation of the Wilcoxon test checks
        for the "difference" of one vector of data from zero
    (_,p) = stats.wilcoxon(data-checkValue)
    if p < 0.05:
```

```
            print('With the Wilcoxon test, {0:4.2f} is
                significantly different from the mean (p={1:5.3f})
                .'.\
            format(checkValue, p))
        else:
            print('No difference from reference value with
                Wilcoxon rank sum test.')

    def twoGroups():
        '''Compare the mean of two groups'''

        print('Two groups of data
            ========================================')

        # Enter the data
        data1 = [76., 101., 66., 72., 88., 82., 79., 73., 76.,
            85., 75., 64., 76., 81., 86.]
        data2 = [64., 65., 56., 62., 59., 76., 66., 82., 91.,
            57., 92., 80., 82., 67., 54.]

        # Normality test
        print('\n Normality test
            ----------------------------------------------------')

        # To do the test for both data-sets, make a tuple with
            "(... , ...)", add a counter with
        # "enumerate", and and iterate over the set:
        for ii, data in enumerate((data1, data2)):
            (_, pval) = stats.normaltest(data)
            if pval > 0.05:
                print('Dataset # {0} is normally distributed'.
                    format(ii))

        # T-test of independent samples
        print('\n T-test of independent samples
            ----------------------------------------')

        # Do the t-test for independent samples
        t, pval = stats.ttest_ind(data1, data2)
        if pval < 0.05:
            print('With the T-test, data1 and data2 are
                significantly different (p = {0:5.3f})'.format(
                pval))
        else:
            print('No difference between data1 and data2 with T-
                test.')

        # Mann-Whitney test
            ----------------------------------------------------------
        print('\n Mann-Whitney test
            ----------------------------------------------------------
            ')
        # Watch out: the keyword "alternative" was introduced in
            scipy 0.17, with default"two-sided";
```

```
        if np.int(sp.__version__.split('.')[1]) > 16:
            u, pval = stats.mannwhitneyu(data1, data2,
                alternative='two-sided')
        else:
            u, pval = stats.mannwhitneyu(data1, data2)
            pval *= 2    # because the default was a one-sided p-
                value
        if pval < 0.05:
            print('With the Mann-Whitney test, data1 and data2
                are significantly different(p = {0:5.3f})'.format(
                pval))
        else:
            print('No difference between data1 and data2 with
                Mann-Whitney test.')

    if __name__ == '__main__':
        oneGroup()
        twoGroups()

        # And at the end, make sure the results are displayed,
            even if the program is run from the commandline
        input('\nDone! Thanks for using programs by thomas.\nHit
            any key to finish.')
```

8.2 Comparing Multiple Groups

Listing 14.9 S8_multipleGroups.py

```
''' Solution for Exercise "Comparing Multiple Groups" '''

# author: Thomas Haslwanter, date: Sept-2015

# Load the required modules
    #-----------------------------------------------------------
# Standard modules
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd

# Modules for data-analysis
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
from statsmodels.stats import multicomp

# Module for working with Excel-files
import xlrd

def get_ANOVA_data():
    '''Get the data for the ANOVA'''

    # First we have to get the Excel-data into Python. This
        can be done e.g. with the package "xlrd"
```

```
    # You have to make sure that you select a valid location
        on your computer!
    inFile = 'Table 6.6 Plant experiment.xls'
    book = xlrd.open_workbook(inFile)
    # We assume that the data are in the first sheet. This
        avoids the language problem "Tabelle/Sheet"
    sheet = book.sheet_by_index(0)

    # Select the columns and rows that you want:
    # The "treatment" information is in column "E", i.e. you
        have to skip the first 4 columns
    # The "weight" information is in column "F", i.e. you
        have to skip the first 5 columns
    treatment = sheet.col_values(4)
    weight = sheet.col_values(5)

    # The data start in line 4, i.e. you have to skip the
        first 3
    # I use a "pandas" DataFrame, so that I can assign names
        to the variables.
    data = pd.DataFrame({'group':treatment[3:], 'weight':
        weight[3:]})

    return data

def do_ANOVA(data):
    '''4.3.2. Perform an ANOVA on the data'''

    print('ANOVA:
        ----------------------------------------------')

    # First, I fit a statistical "ordinary least square (ols)
        "-model to the data, using the
    # formula language from "patsy". The formula 'weight ~ C(
        group)' says:
    # "weight" is a function of the categorical value "group"
    # and the data are taken from the DataFrame "data", which
         contains "weight" and "group"
    model = ols('weight ~ C(group)', data).fit()

    # "anova_lm" (where "lm" stands for "linear model")
        extracts the ANOVA-parameters
    # from the fitted model.
    anovaResults = anova_lm(model)
    print(anovaResults)

    if anovaResults['PR(>F)'][0] < 0.05:
        print('One of the groups is different.')

def compare_many(data):
    '''Multiple comparisons: Which one is different? '''

    print('\n MultComp:
        ---------------------------------------')
```

```
# An ANOVA is a hypothesis test, and only answers the
    question: "Are all the groups
# from the same distribution?" It does not tell you which
     one is different.
# Since we now compare many different groups to each
    other, we have to adjust the
# p-values to make sure that we don't get a Type I error.
     For this, we use the
# statscom module "multicomp"
mc = multicomp.MultiComparison(data['weight'], data['
    group'])

# There are many ways to do multiple comparisons. Here,
    we choose
# "Tukeys Honest Significant Difference" test
# The first element of the output ("0") is a table
    containing the results
print(mc.tukeyhsd().summary())

# Show the group names
print(mc.groupsunique)

# Generate a print ----------------

res2 = mc.tukeyhsd()      # Get the data

simple = False
if simple:
    # You can do the plot with a one-liner, but then this
         does not - yet - look that great
    res2.plot_simultaneous()
else:
    # Or you can do it the hard way, i.e. by hand:

    # Plot values and errorbars
    xvals = np.arange(3)
    plt.plot(xvals, res2.meandiffs, 'o')
    errors = np.ravel(np.diff(res2.confint)/2)
    plt.errorbar(xvals, res2.meandiffs, yerr=errors, fmt=
        'o')

    # Set the x-limits
    xlim = -0.5, 2.5
    # The "*xlim" passes the elements of the variable "
        xlim" elementwise into the function "hlines"
    plt.hlines(0, *xlim)
    plt.xlim(*xlim)

    # Plot labels (this is a bit tricky):
    # First, "np.array(mc.groupsunique)" makes an array
        with the names of the groups;
    # and then, "np.column_stack(res2[1][0])]" puts the
        correct groups together
```

```
            pair_labels = mc.groupsunique[np.column_stack(res2.
                _multicomp.pairindices)]
            plt.xticks(xvals, pair_labels)

            plt.title('Multiple Comparison of Means - Tukey HSD,
                FWER=0.05' +
            '\n Pairwise Mean Differences')

        plt.show()

    def KruskalWallis(data):
        '''Non-parametric comparison between the groups'''

        print('\n Kruskal-Wallis test
            ----------------------------------------------------')

        # First, I get the values from the dataframe
        g_a = data['weight'][data['group']=='TreatmentA']
        g_b = data['weight'][data['group']=='TreatmentB']
        g_c = data['weight'][data['group']=='Control']

        #Note: this could also be accomplished with the "groupby"
             function from pandas
        #groups = pd.groupby(data, 'group')
        #g_a = groups.get_group('TreatmentA').values[:,1]
        #g_c = groups.get_group('Control').values[:,1]
        #g_b = groups.get_group('TreatmentB').values[:,1]

        # Then do the Kruskal-Wallis test
        h, p = stats.kruskal(g_c, g_a, g_b)
        print('Result from Kruskal-Wallis test: p = {0}'.format(p
            ))

    if __name__ == '__main__':
        data = get_ANOVA_data()
        do_ANOVA(data)
        compare_many(data)
        KruskalWallis(data)

        input('\nThanks for using programs by Thomas!\nHit any
            key to finish')
```

Problems of Chap. 9

9.1 A Lady Drinking Tea

Listing 14.10 S9_fisherExact.py

```
'''Solution for Exercise "Categorical Data"
"A Lady Tasting Tea"
'''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats
obs = [[3,1], [1,3]]
_, p = stats.fisher_exact(obs, alternative='greater')

#obs2 = [[4,0], [0,4]]
#stats.fisher_exact(obs2, alternative='greater')

print('\n--- A Lady Tasting Tea (Fisher Exact Test) ---')
print('The chance that the lady selects 3 or more cups
    correctly by chance is {0:5.3f}'.format(p))
```

9.2 Chi2 Contingency Test

Listing 14.11 S9_chi2Contingency.py

```
'''Solution for Exercise "Categorical Data":
Chi2-test with frequency tables
'''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats

obs = [[36,14], [30,25]]
chi2, p, dof, expected = stats.chi2_contingency(obs)

print('--- Contingency Test ---')
if p < 0.05:
    print('p={0:6.4f}: the drug affects the heart rate.'.
        format(p))
else:
    print('p={0:6.4f}: the drug does NOT affect the heart
        rate.'.format(p))

obs2 = [[36,14], [29,26]]
chi2, p, dof, expected = stats.chi2_contingency(obs2)
chi2, p2, dof, expected = stats.chi2_contingency(obs2,
    correction=False)
```

```
print('If the response in 1 non-treated person were different
    , \n we would get p={0:6.4f} with Yates correction, and p
    ={1:6.4f} without.'.format(p, p2))
```

9.3 Chi2 Oneway Test

Listing 14.12 S9_chi2OneWay.py

```
'''Solution for Exercise "Categorical Data" '''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats

# Chi2-oneway-test
obs = [4,6,14,10,16]
_, p = stats.chisquare(obs)

print('\n--- Chi2-oneway ---')
if p < 0.05:
    print('The difference in opinion between the different
        age groups is significant (p={0:6.4f})'.format(p))
else:
    print('The difference in opinion between the different
        age groups is NOT significant (p={0:6.4f})'.format(p))

print('DOF={0:3d}'.format(len(obs)-1))
```

9.4 McNemar Test

Listing 14.13 S9_mcNemar.py

```
'''Solution for Exercise "Categorical Data"
McNemar's Test
'''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats
from statsmodels.sandbox.stats.runs import mcnemar

obs = [[19,1], [6, 14]]
obs2 = [[20,0], [6, 14]]

_, p = mcnemar(obs)
_, p2 = mcnemar(obs2)

print('\n--- McNemar Test ---')
if p < 0.05:
    print('The results from the neurologist are significanlty
         different from the questionnaire (p={0:5.3f}).'.
        format(p))
```

```
else:
    print('The results from the neurologist are NOT
        significanlty different from the questionnaire (p
        ={0:5.3f}).'.format(p))

if (p<0.05 == p2<0.05):
    print('The results would NOT change if the expert had
        diagnosed all "sane" people correctly.')
else:
    print('The results would change if the expert had
        diagnosed all "sane" people correctly.')
```

Problems of Chap. 11

11.1 Correlation

Listing 14.14 S11_correlation.py

```
'''Solution for Exercise "Correlation" in Chapter 11 '''

# author: Thomas Haslwanter, date: Oct-2015

import numpy as np
import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns

def getModelData(show=True):
    ''' Get the data from an Excel-file '''

    # First, define the in-file and get the data
    in_file = 'AvgTemp.xls'

    # When the data are neatly organized, they can be read in
        directly with the pandas-function:
    # with "ExcelFile" you open the file
    xls = pd.ExcelFile(in_file)

    # and with "parse" you get get the data from the file,
       from the specified Excel-sheet
    data = xls.parse('Tabelle1')

    if show:
        data.plot('year', 'AvgTmp')
        plt.xlabel('Year')
        plt.ylabel('Average Temperature')
        plt.show()

    return data

if __name__=='__main__':
    data = getModelData()

    # Correlation
        ----------------------------------------------------
    # Calculate and show the different correlation
       coefficients
    print('Pearson correlation coefficient: {0:5.3f}'.format(
         data['year'].corr(data['AvgTmp'], method = 'pearson')
         ))
    print('Spearman correlation coefficient: {0:5.3f}'.format
        ( data['year'].corr(data['AvgTmp'], method = 'spearman
        ') ))
```

```
    print('Kendall tau: {0:5.3f}'.format( data['year'].corr(
        data['AvgTmp'], method = 'kendall') ))
```

11.1 Regression

Listing 14.15 S11_regression.py

```
'''Solution for Exercise "Regression" '''

# author: Thomas Haslwanter, date: Sept-2015

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
import statsmodels.formula.api as sm

# We don't need to invent the wheel twice ;)
from S11_correlation import getModelData

if __name__== '__main__':

    # get the data
    data = getModelData(show=False)

    # Regression
        ------------------------------------------------------
    # For "ordinary least square" models, you can do the
        model directly with pandas
    #model = pd.ols(x=data['year'], y=data['AvgTmp'])

    # or you can use the formula-approach from statsmodels:
    # offsets are automatically included in the model
    model = sm.ols('AvgTmp ~ year', data)
    results = model.fit()
    print(results.summary())

    # Visually, the confidence intervals can be shown using
        seaborn
    sns.lmplot('year', 'AvgTmp', data)
    plt.show()

    # Is the inclination significant?
    ci = results.conf_int()

    # This line is a bit tricky: if both are above or both
        below zero, the product is positive:
    # we look at the coefficient that describes the
        correlation with "year"
    if np.prod(ci.loc['year'])>0:
        print('The slope is significant')
```

11.1 Normality Check

Listing 14.16 S11_normCheck.py

```
'''Solution for Exercise "Normality Check" in Chapter 11 '''

# author: Thomas Haslwanter, date: Sept-2015

from scipy import stats
import matplotlib.pyplot as plt
import statsmodels.formula.api as sm
import seaborn as sns

# We don't need to invent the wheel twice ;)
from S11_correlation import getModelData

if __name__== '__main__':

    # get the data
    data = getModelData(show=False)

    # Fit the model
    model = sm.ols('AvgTmp ~ year', data)
    results = model.fit()

    # Normality check
        ------------------------------------------------------
    res_data = results.resid    # Get the values for the
        residuals

    # QQ-plot, for a visual check
    stats.probplot(res_data, plot=plt)
    plt.show()

    # Normality test, for a quantitative check:
    _, pVal = stats.normaltest(res_data)
    if pVal < 0.05:
        print('WARNING: The data are not normally distributed
             (p = {0})'.format(pVal))
    else:
        print('Data are normally distributed.')
```

Glossary

bias Systematic deviation of a sample statistic from the corresponding population statistic. Often caused by poor selection of subjects.

blocking To reduce the variability of a variable that cannot be randomized by fixating it.

box-plot A common visualization of the distribution of data, expressed by a box with a line inside that box, and whiskers at the top and bottom. The box indicates the first and third quartile, and the line the median value of the data sample. The whiskers can indicate either the range of the data or the most extreme value within 1.5*the inner-quartile-range.

case–control study A type of observational study in which two existing groups differing in outcome are identified and compared on the basis of some supposed causal attribute. ("First treat, then select.")

categorical data Data that can take on one of a limited, and usually fixed, number of possible values, with no natural order. (If a "mean value" makes no sense.)

cohort study A type of observational study, where you first select the patients, and then follow their development. For instance, in medicine a cohort study starts with an analysis of risk factors. Then the study follows a group of people who do not have the disease. Finally, correlations are used to determine the absolute risk of subject contraction. ("First select, then treat.")

confidence interval Interval estimate of a population parameter, which contains the true value of the parameter with a defined percent-likelihood (e.g., 95 %-CI).

correlation Any departure of two or more random variables from independence.

crossover study A longitudinal study in which all subjects receive a sequence of different treatments.

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6

cumulative distribution function The probability to find a random variable with a value lower than the given one.

degrees of freedom The number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary.

density A continuous function that describes the relative likelihood for a random variable to take on a given value. For example *kernel-density* estimation (KDE), or *probability-density* function (PDF).

design matrix The data matrix $\mathbf{X}$ in the regression model $y = \mathbf{X} \cdot \boldsymbol{\beta} + \epsilon$.

distribution A function which assigns a probability to each measurable subset of the possible outcomes of a random experiment.

experimental study Study where the selection of the subjects as well as the conditions of the study are under the control of the investigator.

factor Also called *treatment* or *independent variable*, is an explanatory variable manipulated by the experimenter.

function A *Python* object that accepts input data, executes commands and calculations with them, and can return one or more return objects.

hypothesis test A method of statistical inference used for testing a statistical hypothesis.

IPython The *IPython* kernel is an interactive shell that allows the immediate execution of *Python* commands, and extends these with "magic functions" (e.g., "%cd," or "%whos") which facilitate the interactive work. It has been so successful that its development has been split into two separate projects: *IPython* now handles the computational kernel; and *Jupyter* provides the frontend. This allows *IPython* to be executed in a terminal modus (which is rarely used on its own, but allows it to embed *IPython* in other applications), in a *Jupyter QtConsole* which allows the display of help and the graphical outputs in the command window, and in a *Jupyter notebook*, a browser-based implementation with support for code, rich text, mathematical expressions, inline plots, and other rich media.

kurtosis Measure of the peakedness of a distribution. It is approximately 3 for normally distributed data. "Excess kurtosis" is the kurtosis relative to the normal distribution.

linear regression Modeling a scalar variable (*dependent variable*) using a linear predictor function. The unknown model parameters are estimated from the data.

Simple linear regression: $y = k * x + d$.
Multiple linear regression: $y = k_1 * x_1 + k_2 * x_2 + \ldots + k_n * x_n + d$.

location Parameter that shifts the mean of a probability distribution.

logistic regression Also called *logit regression*.The probabilities describing the possible outcomes of a single trial are modeled, as a function of the explanatory (predictor) variables, using a logistic function: $f(x) = \frac{L}{1+e^{-k(x-x_0)}}$.

Markov Chain Stochastic model of a process where the probability of each state only depends on the previous state.

Matplotlib A *Python* package which provides the ability to generate 2D and 3D plots. This includes the plotting commands, as well as the functionality of different output media, also called *backends*: an output can for example be into the *Jupyter Notebook*, into a PDF file, or into a separate graphics window.

maximum likelihood For a fixed set of data and an underlying statistical model, the method of maximum likelihood selects the set of values of the model parameters that maximizes the likelihood function. Intuitively, this maximizes the "agreement" of the selected model with the observed data, and for discrete random variables it indeed maximizes the probability of the observed data under the resulting distribution.

median value The value separating the higher half of the data sample from the lower half.

minimization Closely related to *randomization*. Thereby one takes whichever treatment has the smallest number of subjects, and allocates this treatment with a probability greater than 0.5 to the next patient.

mode value The highest value in a discrete or continuous probability distribution.

module A file containing *Python* variables and function definitions.

Monte Carlo Simulation Repeated simulation of the behavior of some parameter based on repeated sampling of a random variable.

numerical data Data that can be expressed by a (continuous or discrete) number.

numpy *Python* package for numerical data manipulation. Provides efficient handling of mathematical manipulations on vectors and on arrays of two or more dimensions.

observational study Study where the assignment of subjects into a treated group versus a control group is outside the control of the investigator.

paired test Two data sets are *paired* when the following one-to-one relationship exists between values in the two data sets: (1) Each data set has the same number of data points. (2) Each data point in one data set is related to one, and only one, data point in the other data set.

percentile Also called *centile*. Value that is larger or equal than $p\%$ of the data, where $1 <= p < 100$.

population Includes all of the elements from a set of data.

power analysis Calculation of the minimum sample size required so that one can be reasonably likely to detect an effect of a given size.

power Same as *sensitivity*. Denoted by $1 - \beta$, with β the probability of Type II errors.

probability density function (PDF) A continuous function which defines the likelihood to find a random variable with a certain value. The probability to find the value of the random variable in a given interval is the integral of the probability density function over that interval.

probability mass function (PMF) A discrete function which defines the probability to obtain a given number of events in an experiment or observation.

prospective study A prospective study watches for outcomes, such as the development of a disease, during the study period and relates this to other factors such as suspected risk or protection factor(s).

pylab pylab is a convenience *Python* module that bulk imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space.

package A folder containing one or more *Python* modules and an ".ini"-file.

quantile Value that is larger or equal than $p*100\,\%$ of the data, where $0 < p <= 1$.

quartile Value that is larger or equal than $25\,\%/50\,\%/75\,\%$ of the data (first/second/third quartile). The 2nd quartile is equivalent to the median.

randomization A method to eliminate bias from research results, by dividing a homogeneous group of subjects into a *control group* (which receives no treatment) and a *treatment group*.

ranked data Numbered data where the number corresponds to the rank of the data, i.e., the number in the sequence of the sorted data, and not to a continuous value.

regular expression A sequence of characters that define a search pattern, mainly for use in pattern matching with strings. Available for Unix, and for *Python*, *Perl*, *Java*, *C++*, and many other programming languages.

residual Difference between the observed value and the estimated function value.

retrospective study A retrospective study looks backwards and examines exposures to suspected risk or protection factors in relation to an outcome that is established at the start of the study.

sample One or more observations from a population.

scale Parameter that controls the variance of a probability distribution.

scipy *Python* package for scientific computation. Based on *numpy*.

sensitivity Proportion of actual positives which are correctly identified as such (e.g., the percentage of sick people who are correctly identified as having the condition).

shape parameter Parameters beyond *location* and *scale* which control the shape of a probability distribution. Rarely used.

skewness Measure of the asymmetry of a distribution.

specificity Proportion of actual negatives which are correctly identified as such (e.g., the percentage of healthy people who are correctly identified as not having the condition).

standard deviation Square root of variance.

standard error Often short for *standard error of the mean*. Square root of the variance of a statistic.

treatment Same as *factor*.

type I error A type I error occurs when one rejects the null hypothesis when it is true. The probability of a type I error is the level of significance of the hypothesis test, and is commonly denoted by α.

type II error A type II error occurs when one rejects the alternative hypothesis (fails to reject the null hypothesis) when the alternative hypothesis is true. Therefore it is dependent on an alternative hypothesis. The probability of a type II error is commonly denoted by β.

unpaired test Test with two sets of independent data.

variance Measure of how far a set of numbers is spread out. Mathematically, it is the expected value of the squared deviation from the mean: $Var(X) = E[(X - \mu)^2]$. The variance of a sample gives an estimate of the population variance that is biased by a factor of $\frac{n-1}{n}$. The best unbiased estimate of the population variance is therefore given by $s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2$ which is called *(unbiased) sample variance*.

References

Altman, D. G. (1999). *Practical statistics for medical research*. New York: Chapman & Hall/CRC.

Amess, J. A., Burman, J. F., Rees, G. M., Nancekievill, D. G., & Mollin, D. L. (1978). Megaloblastic haemopoiesis in patients receiving nitrous oxide. *Lancet*, *2*(8085):339–342.

Bishop, C. M. (2007). *Pattern recognition and machine learning*. New York: Springer.

Box, J. F. (1978). *R. A. Fisher: The life of a scientist.* New York: Wiley.

Dobson, A. J., & Barnett, A. (2008). *An introduction to generalized linear models* (3rd ed.). Boca Raton: CRC Press.

Duda, R. O., Hart, P. E., & Stork, D. G. (2004). *Pattern classification* (2nd ed.). Hoboken: Wiley-Interscience.

Ghasemi, A., & Zahediasl, S. (2012). Normality tests for statistical analysis: a guide for non-statisticians. *International Journal of Endocrinology and Metabolism, 10*(2):486–489. doi:10.5812/ijem.3505. http://dx.doi.org/10.5812/ijem.3505

Harms, D., & McDonald, K. (2010). *The quick python book* (2nd ed.). Greenwich: Manning Publications Co.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics, 6*:65–70.

Kaplan, D. (2009). *Statistical modeling: A fresh approach*. St Paul: Macalester College.

Kaplan, R. M., & Irvin, V. L. (2015). Likelihood of null effects of large nhlbi clinical trials has increased over time. *PLoS One, 10*(8):e0132382. doi:10.1371/journal.pone.0132382. http://dx.doi.org/10.1371/journal.pone.0132382

Klamroth-Marganska, V., Blanco, J., Campen, K., Curt, A., Dietz, V., Ettlin, T., Felder, M., Fellinghauer, B., Guidali, M., Kollmar, A., Luft, A., Nef, T., Schuster-Amft, C., Stahel, W., & Riener, R. (2014). Three-dimensional, task-specific robot therapy of the arm after stroke: a multicentre, parallel-group randomised trial. *The Lancet Neurology, 13*(2):159–166. doi:10.1016/S1474-4422(13)70305-3. http://dx.doi.org/10.1016/S1474-4422(13)70305-3

McCullagh, P. (1980). Regression models for ordinal data. *Journal of the Royal Statistical Society. Series B (Methodological), 42*(2):109–142.

McCullagh, P. & Nelder, J. A. (1989). *Generalized linear models* (2nd ed.). New York: Springer.

Nuzzo, R. (2014). Scientific method: Statistical errors. *Nature, 506*(7487):150–152. doi:10.1038/506150a. http://www.nature.com/news/scientific-method-statistical-errors-1.14700

Olson, R. (2012). *Statistical analysis made easy in python*. http://www.randalolson.com/2012/08/06/statistical-analysis-made-easy-in-python/

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6

Open Science Collaboration (OSC). (2015). Psychology. Estimating the reproducibility of psychological science. *Science, 349*(6251):aac4716. doi:10.1126/science.aac4716. http://dx.doi.org/10.1126/science.aac4716

Pilon, C. D. (2015). *Probabilistic programming and Bayesian methods for hackers*. http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/

Riffenburgh, R. H. (2012). *Statistics in medicine* (3rd ed.). Amsterdam: Academic Press.

Rosenbaum, P. R., & Rubin, D. B. (1983). The central role of the propensity score in observational studies for causal effects. *Biometrika, 70*(1):41–55.

Scopatz, A., & Huff, K. D. (2015). *Effective computation in physics*. Sebastopol: O'Reilly Media.

Sellke, T., Bayarri, M. J., & Berger, J. O. (2001). Calibration of p values for testing precise null hypotheses. *The American Statistician, 55*:62–71.

Sheppard, K. (2015). *Introduction to python for econometrics, statistics and data analysis*. http://www.kevinsheppard.com/images/0/09/Python_introduction.pdf

Wilkinson, G. N., & Rogers, C. E. (1973). Symbolic description of factorial models for analysis of variance. *Applied Statistics, 22*:, 392–399.

Index

T. Haslwanter, *An Introduction to Statistics with Python*, Statistics and Computing,
DOI 10.1007/978-3-319-28316-6