Alan Agresti and Maria Kateri

Python-Web-Appendix of Foundations of Statistics for Data Scientists

Contents

0

CHAPTER 0: BASICS OF `PYTHON`

This Appendix provides implementation in `Python` of examples that are worked out in the chapters of this book in `R`. A familiarity with the open source language `Python` is assumed. For beginners, there is a variety of very good introductions to the `Python` language available in the internet. For general information on `Python`, documentation and tutorials, please visit https://www.python.org/. We use the version `Python 3.7` (https://www.python.org/). A helpful overview is provided in the tutorial https://docs.python.org/3.7/tutorial/index.html.

To install `Python` the *Python distributions* can be used, which collect and install simultaneously major libraries required. Among the *Python distributions*, probably the most handy is the *Individual Edition* of *Anaconda* (https://www.anaconda.com/products/individual); see also the Anaconda documentation (https://docs.anaconda.com/anaconda/user-guide/). We further recommend to run `Python` in an IDE (integrated development environment), such as *Spyder*, which is distributed with *Anaconda* and used in this Appendix.

B0.1 `Python` Preliminaries

The most popular `Python` libraries that are also relevant in this book are listed below.

Library	Description
`glmnet`	Lasso and Elastic-Net Regularized GLMs
`ipython`	Interactive computing
`mathplotlib`	Tools for data visualization
`numpy`	Numeric Python: operations for multidimensional arrays and linear algebra functions
`os`	Operating system interfaces: for better programs' portability between different platforms
`pandas`	Python Data Analysis Library: creation of data frames and data manipulation
`pymc3`	Probabilistic Programming in Python: used for Bayesian modelling and MCMC algorithms
`rp2`	Python interface to the R language: runs an embedded `R`, providing access to it from Python
`scikit-learn`	Machine Learnig in Python (see https://scikit-learn.org/stable)
`scipy`	Scientific Computing: essential scientific algorithms, incl. statistical functions (scipy.stats) (see http://scipy-lectures.org/intro/)
`seaborn`	Statistical Data Visualization (based on `matplotlib`)
`statsmodels`	Classes and functions for statistical modeling and statistical tests

For some statistical functions, model fitting procedures and graphs, there exist more

than one options, provided in different libraries. All libraries (and their functions) have very good documentation and helpful examples. For example, statistical functions in `python` are descibed in https://docs.python.org/3/library/statistics.html, while for the `statsmodels` library see https://www.statsmodels.org/stable/index.html.

Getting started, we activate the `Skyper`–console. In Figure B0.1 a screen–shot is provided illustrating a first trial with some simple values assignments, computing and use of the function `type()`. Code can be typed in the spyder editor console on the left, which can be run (line-wise, selection of lines or a whole script). The output occurs then in the IPython console on the bottom right. You can give in commands also directly in the IPython console. The up right console provides the list of the active variables, help, plots and list of files.

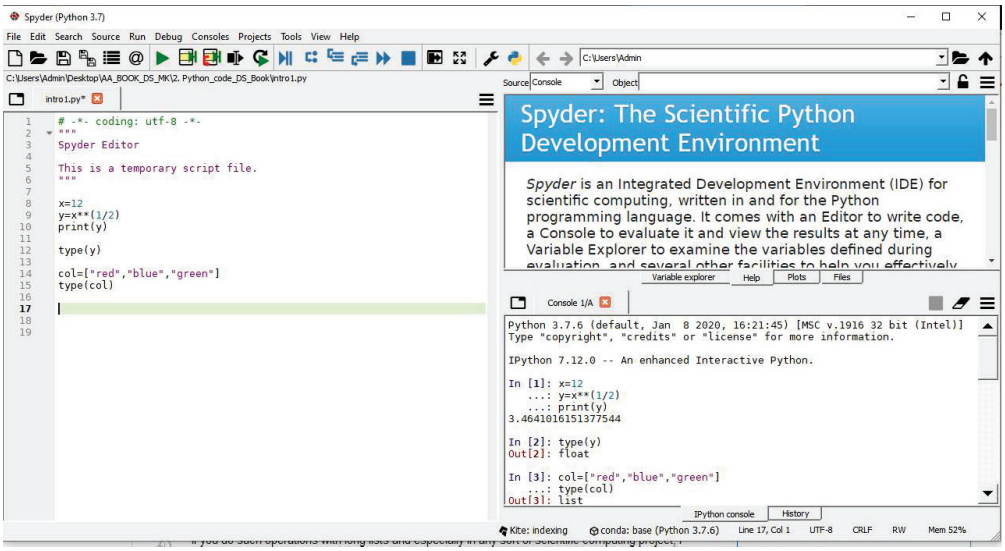


FIGURE B0.1: `Skyper`–console screen–shot.

There is the option in `Spyder` to select whether graphs appear in a separate window (`%matplotlib qt`) or inline (`%matplotlib inline`). Note that such commands (the so called *magic commands*) do work in the `IPython` console but not within a script (left window in `Spyder`)

Also in `Python`, as in `R`, comments follow the # symbol. It is important to load at the beginning of a `Python` session the required libraries. For example, for dividing the entries of a vector `a=[1,2,3,4]` by a scalar (here 2), the `numpy` library is required, as shown below. The code is shown in this appendix as it appears in the `IPython` console, with the input/output (In []:/Out []:) indication, which of course has to be omitted when typing the code.

```
IPython 7.12.0 -- An enhanced Interactive Python.

In [1]: a=[1,2,3,4]
   ...: b=a/2           # see error message below for 'list'/'int' operation
Traceback (most recent call last):

  File "<ipython-input-1-ca95076dec07>", line 2, in <module>
    b=a/2

TypeError: unsupported operand type(s) for /: 'list' and 'int'

In [2]: import numpy          # numpy is required for computing with vectors
```

```
    ...: a=numpy.array([1,2,3,4])          # a has to be defined as an array
    ...: b=a/2
    ...: print(b)
[0.5 1.  1.5 2. ]
```

B0.2 Data Structures and Data Input

Analogously to `R`, `python` offers a variety of data structures.

- *Lists* for grouping objects of the same type: There exist several methods for handling and working with lists, as for example `list.append()` or `list.count()`. Lists can be nested.
- *Tuples* for grouping objects of different type: Tuples can be also nested.
- *Sets*: A set is an unordered collection of elements with no duplicate elements, mainly used for membership testing and eliminating duplicate entries.
- *Dictionaries*: Dictionaries are 'associative arrays', indexed by keys (instead of a range of numbers).
- Arrays: An array can consist of basic values: characters, integers, floating point numbers.
- Data frames: Convenient for data sets and statistical data analysis.

The first five are data types of `python` while data frames is of the `pandas` library. The `NumPy` library defines another array type appropriate for multivariate numerical data (numerical and scientific functions apply on such arrays). For a detailed description of the powerful options of the data types and examples, see in https://docs.python.org/3/tutorial/datastructures.html, https://docs.python.org/3/library/array.html, https://numpy.org/doc/stable/reference/generated/numpy.array.html and https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

Data files of various formats are easily read in `pandas`. Commonly data are saved in CSV files and the data are given in a form such that rows correspond to cases and columns to variables measured. Such files (or other separated value files, such as `R .dat` files) can be easily read, either directly from a url or from a file providing the corresponding path. If the data separator is ',', then a data file is read by `pd.read_csv('...')`. In case of different separator, the separator has to be given. For example, values in data files used in this book are separated by spaces. Such data files are read by `pd.read_csv('...', sep='\s+')`, see for example in Section B1.2.1.

The `pandas` library has also functions for reading files of different formats such as Excel (`read_excel()`), HDF5 (`read_hdf()`), in table format (`read_table()`) or from the clipboard (`read_clipboard()`).

1

CHAPTER 1: PYTHON FOR DESCRIPTIVE STATISTICS

B1.1 Random Number Generation

The example of random number generation in Section 1.3.1 can be implemented in Python as shown below.

```
In [1]: import random
   ...: randomlist = random.sample(range(1,60), 5) # randomly sample 5
   ...: print(randomlist)                          # integers from 1 to 60
[37, 31, 34, 3, 17]                                # without replacement
In [2]: import numpy as np
   ...: y = list(range(1, 60))
   ...: randomlist2 = np.random.choice(y, 5)       # sample with replacement
   ...: print(randomlist2)
[ 5 35  1 35 18]
```

Python can also randomly generate values from various distributions, such as shown in Sections B1.5, B2.1 and B2.3.

B1.2 Summary Statistics and Graphs for Quantitative Variables

B1.2.1 Descriptive statistics for carbon dioxide emissions

The following code reads the data file on per-capita carbon dioxide emissions for 31 European nations analyzed in Chapter 1 (starting in Section 1.4.1) and finds some descriptive statistics:

```
In [1]: import os
   ...: import pandas as pd   # used to read data file
   ...: Carbon = pd.read_csv('http://stat4ds.rwth-aachen.de/data/
                Carbon.dat', sep='\s+') # values separator is a space
In [2]: Carbon               # prints the data file (not shown here)
In [3]: Carbon.shape         # dimensions of the array
Out[3]: (31, 2)              # (31 rows with 2 columns)
In [4]: Carbon.columns       # variables (columns) in the file
Out[4]: Index(['Nation', 'CO2'], dtype='object')
In [5]: Carbon.head()        # first 5 observations (starts numbering with 0)
Out[5]:
     Nation  CO2
0   Albania  2.0
1   Austria  6.9
2   Belgium  8.3
3    Bosnia  6.2
```

```
4  Bulgaria  5.9
In [6]: Carbon.tail ()      # last 5 observations (not shown here)
In [7]: Carbon.describe()   # n, mean, std. dev., and five-number
Out[7]:                     # summary for numerical variables
             CO2
count  31.000000 # sample size n
mean    5.819355 # mean of observations in the data file
std     1.964929 # standard deviation
min     2.000000 # minimum value
25%     4.350000 # lower quartile (25th percentile)
50%     5.400000 # median (50th percentile)
75%     6.700000 # upper quartile (75th percentile)
max     9.900000 # maximum value
In [8]: Carbon['CO2'].mean()    # mean
Out[8]: 5.819354838709677
In [9]: Carbon['CO2'].std()     # standard deviation
Out[9]: 1.9649290665464592
In [10]: Carbon['CO2'].median() # median
Out[10]: 5.4
```

The histogram of CO2 with 8 bins of equal length can be derived in `Python` (see Figure B1.1), as shown below:

```
# 'density=False' would use counts for the y-axis
In [11]: import matplotlib.pyplot as plt
    ...: plt.hist(carbonDF['CO2'], density=True, bins=8)
    ...: plt.ylabel('Proportion')
    ...: plt.xlabel('CO2');
    ...: plt.title('Histogram of carbonDF[CO2]')
%Out[2]: Text(0.5, 1.0, 'Histogram of carbonDF[CO2]')
```

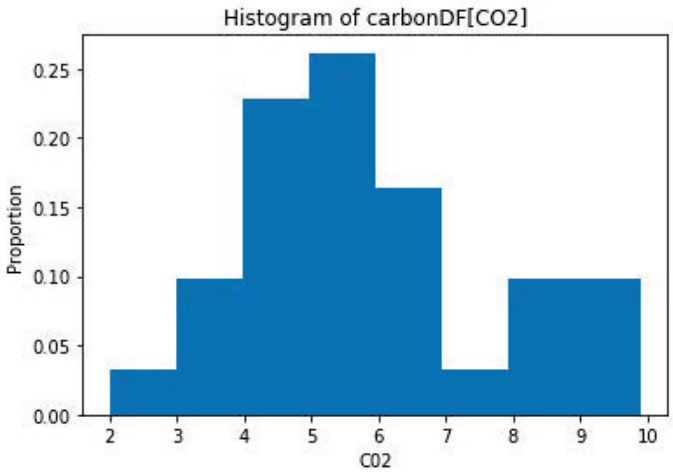


FIGURE B1.1: Histogram for frequency distribution of European CO2 values.

The box plot of CO2 (see Figure B1.2) is produced as follows:

```
In [12]: fig1, ax1 = plt.subplots()     # creates common layouts of subplots,
    ...:            # including the enclosing figure object, in a single call
    ...: plt.xlabel('CO2 values')
    ...: ax1.boxplot(carbonDF['CO2'],vert=False) # creates horizontal box plots
```

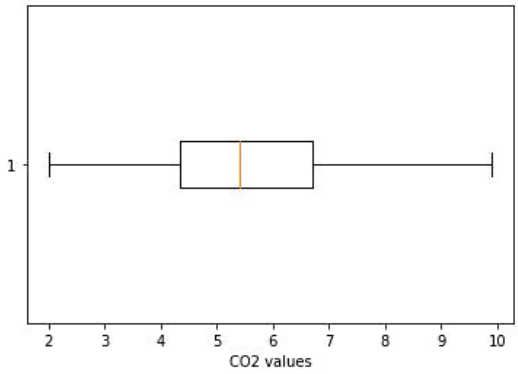


FIGURE B1.2: Box plot of CO2 values for European nations.

B1.2.2 Side-by-side box plots for U.S. and Canadian murder rates

Side-by-side box plots are illustrated in Section 1.4.5 for comparing the murder rates in U.S. and Canada. Here is `Python` code for this figure, shown in Figure B1.3, and for using the `groupby` command to report summary statistics by nation:

```
In [1]: import pandas as pd
   ...: import seaborn as sns
   ...: Crime = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Murder2.dat', sep='\s+')
   ...: sns.boxplot(x='murder', y='nation', data=Crime, orient='h')
In [2]: Crime.groupby('nation')['murder'].describe()
Out[2]:
        count      mean       std  min   25%    50%    75%    max
nation
Canada   10.0  1.673000  1.184437  0.0  1.03  1.735  1.875   4.07
US       51.0  5.252941  3.725391  1.0  2.65  5.000  6.450  24.20
```

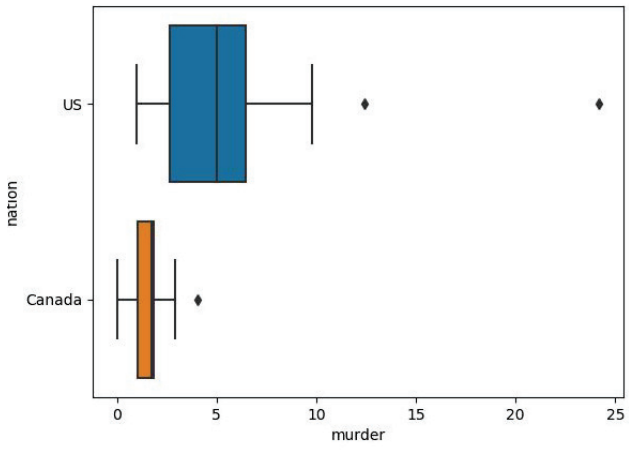


FIGURE B1.3: Side-by-side box plots for U.S. and Canadian murder rates.

B1.3 Descriptive Statistics for Bivariate Quantitative Data

Fr the example in Section 1.5.1 relating statewide suicide rates in U.S. to the percentage of people who own guns, we show next code to construct the scatter plot in Figure B1.4 (left) as well as the scatter plot with the fitted regression line (see Figure B1.4, right). The associated Pearson's correlation coefficient and the simple linear regression model fit are also derived:

```
In [1]: import pandas as pd
   ...: import seaborn as sns
   ...: import matplotlib.pyplot as plt         # use for scatter plot
   ...: GS = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Guns_Suicide.dat',
                              sep='\s+')
In [2]: GS.info()                  # number of non-missing values per variable
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 51 entries, 0 to 50
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   state    51 non-null     object
 1   guns     51 non-null     float64
 2   suicide  51 non-null     float64

# scatterplot:
In [3]: GS.plot(kind='scatter', x='guns', y='suicide', color='blue', figsize=(10, 7))
   ...: plt.xlabel('guns', size=14)
   ...: plt.ylabel('suicide', size=14)
Out[3]: Text(0, 0.5, 'suicide')

In [4]: GS.corr()            # correlation matrix for pairs of
Out[4]:                      # variables in GS data file
             guns   suicide
guns     1.000000  0.738667  # corr. = 0.739 between guns and suicide
suicide  0.738667  1.000000

In [5]: import numpy as np  # scatter plot with linear regression line
   ...: coef = np.polyfit(GS['guns'], GS['suicide'], 1)
   ...: LR_fn = np.poly1d(coef)               # LR_fn: returns fitted y values
   ...: fig = plt.figure(figsize=(10,7))      # submit next 4 lines together
   ...: plt.plot(GS['guns'],GS['suicide'],'o',GS['guns'],LR_fn(GS['guns']))
   ...: plt.xlabel('guns', size=14)
   ...: plt.ylabel('suicide', size=14)
Out[5]:
[<matplotlib.lines.Line2D at 0x109535b0>,
 <matplotlib.lines.Line2D at 0x10953710>]

In [6]: import statsmodels.formula.api as sm  # fit linear regression
   ...: mod = sm.ols(formula='suicide ~ guns', data=GS).fit()
   ...: print(mod.params)  # model parameter estimates
Intercept    7.390080
guns         0.193565      # slope estimate for effect of guns on suicide
dtype: float64

In [7]: print(mod.summary()) # summary of fit
                            OLS Regression Results
==============================================================================
Dep. Variable:                suicide   R-squared:                       0.546
Model:                            OLS   Adj. R-squared:                  0.536
Method:                 Least Squares   F-statistic:                     58.84
Date:                Fri, 10 Jul 2020   Prob (F-statistic):           6.11e-10
Time:                        11:49:09   Log-Likelihood:                -121.12
```

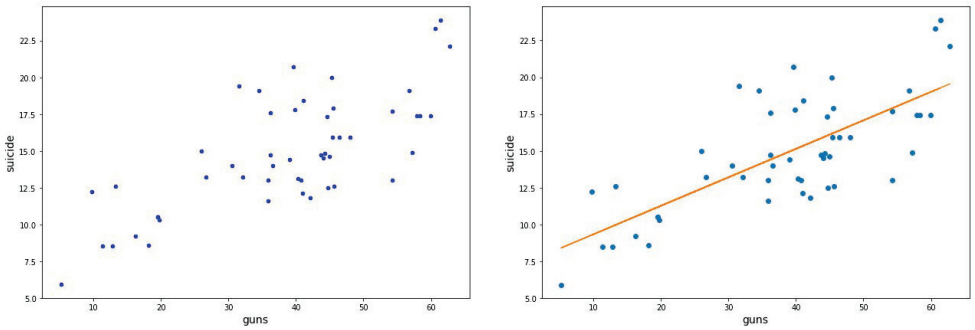


FIGURE B1.4: Scatterplot relating state-level data in the U.S. on percent gun ownership and suicide rate

```
No. Observations:                  51   AIC:                             246.2
Df Residuals:                      49   BIC:                             250.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      7.3901      1.037      7.125      0.000       5.306       9.474
guns           0.1936      0.025      7.671      0.000       0.143       0.244
==============================================================================
Omnibus:                        2.959   Durbin-Watson:                   2.314
Prob(Omnibus):                  0.228   Jarque-Bera (JB):                2.820
Skew:                           0.525   Prob(JB):                        0.244
Kurtosis:                       2.527   Cond. No.                         115.
==============================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

B1.4 Descriptive Statistics for Bivariate Categorical Data

We next show code for forming a contingency table and mosaic plot in `Python` for the example in Section 1.5.2 of cross-classifying race and political party identification for data from the 2018 General Subject Survey.

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: import matplotlib as plt
   ...: PID = pd.read_csv('http://stat4ds.rwth-aachen.de/data/
                          PartyID.dat', sep='\s+')
   ...: PID_table = pd.crosstab(PID['race'], PID['id'], margins=False)
   ...: PID_table
Out[1]:
id     Democrat  Independent  Republican
race
black       281           65          30
other       124           77          52
white       633          272         704
```

```
In [2]: from scipy.stats.contingency import margins # find marginal
   ...: mr, mc = margins(PID_table)                 # dist. counts
   ...: print(mr)    # row marginal counts (output not shown)
   ...: print(mc)    # column marginal counts (output not shown)
# derivation of joint probability table:
In [3]: asarray = np.array(PID_crosstab)/sum(sum(np.array(PID_crosstab)))
   ...: probtable=pd.DataFrame(asarray, columns=["Democrat",
                                    "Independent","Republican"])
   ...: probtable.index=["black", "white", "other"]
   ...: probtable
Out[3]:
       Democrat  Independent  Republican
black  0.125559     0.029044    0.013405
white  0.055407     0.034406    0.023235
other  0.282842     0.121537    0.314567
# derivation of conditional row probabilities (within row) tables:
# (for within column probability table replace in the next line mr by mc )
In [4]: asarray1 = np.array(PID_crosstab)/mr
   ...: probtable1=pd.DataFrame(asarray1, columns=["Democrat",
                                    "Independent","Republican"])
   ...: probtable1.index=["black", "white", "other"]
   ...: probtable1
Out[4]:
       Democrat  Independent  Republican
black  0.747340     0.172872    0.079787
white  0.490119     0.304348    0.205534
other  0.393412     0.169049    0.437539
# alternatively contingency table in statsmodels:
In [5]: import statsmodels.api as sm
   ...: PIDtable = sm.stats.Table.from_data(PID)

In [6]: from statsmodels.graphics.mosaicplot import mosaic
   ...: fig, _ = mosaic(PID, index=["race", "id"])               # mosaic plot
```

The mosaic plot is given in Figure B1.5. The areas of the rectangles corresponding to each cell are analogous to the respective cell probabilities (given in `Out[3]` above). If the conditional row probabilities (see `Out[4]` above) were equal, then the heights of the rectangles (races) would be equal across the columns (PatyID), which is here not the case.

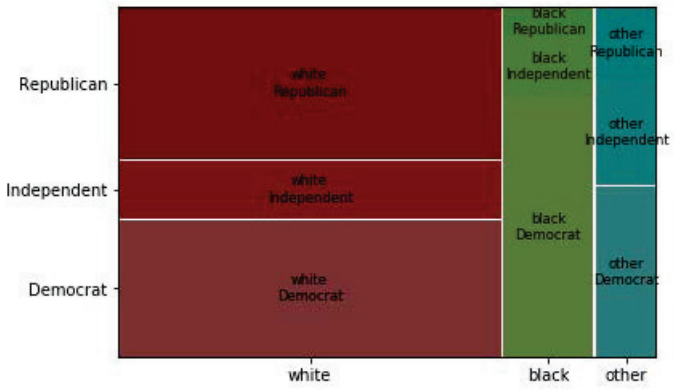


FIGURE B1.5: Mosaic plot for the cross–classification of GSS2018 responders by their race and party ID.

B1.5 Simulating Samples from a Bell-Shaped Population

The simulation example in Section 1.5.3 took two random samples of size $n = 30$ each from a bell-shaped population (specifically, the normal distribution introduced in Section 2.5.1) with a mean of 100 and a standard deviation of 16. The following code performs the simulation, finds sample means and standard deviations, and constructs histograms:

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
In [2]: mu, sigma = 100, 16

In [3]: y1 = np.random.normal(mu, sigma, 30)
   ...: y1.mean(), y1.std()
Out[3]: (101.46071134287304, 16.14904095192038)

In [4]: plt.hist(y1, bins='auto')
Out[4]:
(array([ 1.,  5.,  5., 11.,  6.,  2.]),
 array([ 60.51205673,  73.24661343,  85.98117012,  98.71572682,
        111.45028351, 124.1848402 , 136.9193969 ]),
 <a list of 6 Patch objects>)

In [5]: y2 = np.random.normal(mu, sigma, 30)
   ...: y2.mean(), y2.std()
Out[5]: (100.29079228919267, 16.792865510807726)

In [6]: plt.hist(y2, bins='auto')
Out[6]:
(array([5., 5., 7., 7., 3., 3.]),
 array([ 71.54756049,  82.06008145,  92.57260241, 103.08512337,
        113.59764433, 124.11016529, 134.62268625]),
 <a list of 6 Patch objects>)
```

The histograms are shown in Figure B1.6 (upper part). Compare to the corresponding results for samples of size 1000 (see Figure B1.6, lower part).

This example illustrates that descriptive statistics such as the sample mean can themselves be regarded as variables, their values varying from sample to sample. Chapter 3 provides results about the nature of that variation.

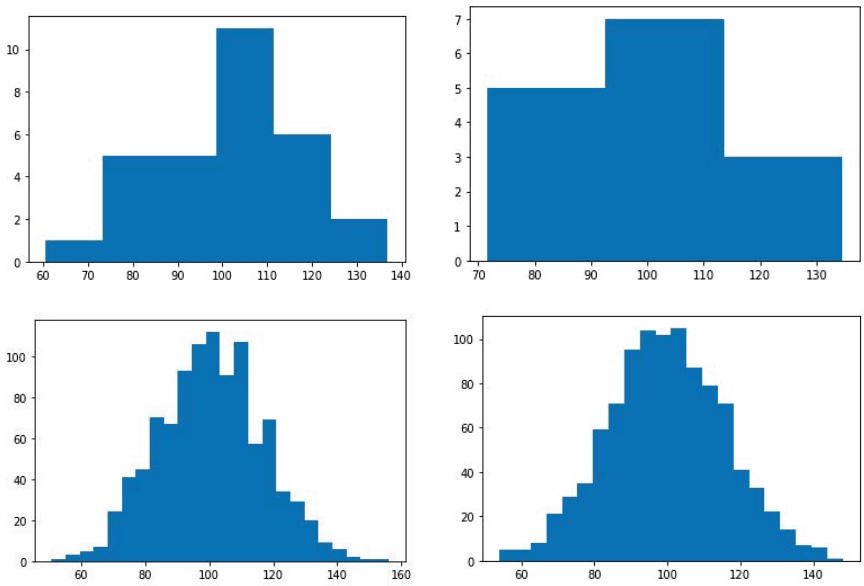


FIGURE B1.6: Histograms of two simulated data sets from $\mathrm{N}(100, 16^2)$ of size $n = 30$ (upper) and $n = 1000$ (lower).

2

CHAPTER 2: PYTHON FOR PROBABILITY DISTRIBUTIONS

B2.1 Simulating a Probability as a Long-Run Relative Frequency

The examples of random number generation in Section 2.1.1, based on uniform and binomial distribution, can be implemented in Python as shown below:

```
In [1]: import numpy as np
In [2]: y = list(range(0, 10))             # list integers from 0 to 9
   ...: randomlist = np.random.choice(y,7) # sample n=7 observations
   ...: print(randomlist)                  # with replacement
[1 4 6 8 3 2 2] # 0 and 1 represent rain, so rain occurs only on day 1
In [3]: np.random.binomial(7, 0.2, size=1) # 1 simulation of 7 flips
Out[3]: array([2])                         # obtain 2 heads in 7 flips
In [4]: n, p = 1, 0.2 # no. of flips (trials), prob(success) in each
   ...: s = np.random.binomial(n, p, 7)    # 7 simulations of n flips
   ...: print(s)
[0 0 0 1 1 0 0] # heads on flips 4 and 5 simulate rain on days 4 and 5
```

Next we illustrate the definition of the probability of an outcome as the long-run relative frequency of that outcome in n observations, with n taking values 100, 1000, 10000, 100000, 1000000, and with probability 0.20 for each observation:

```
# proportion of "heads" in 100, 1000, 10000, 100000, 1000000 flips:
In [1]: import numpy as np
   ...: x1 =np.random.binomial(100, 0.2, 1); print(x1/100)
   ...: x2=np.random.binomial(1000, 0.2, 1); print(x2/1000)
   ...: x3=np.random.binomial(10000, 0.2, 1); print(x3/10000)
         ...: x4=np.random.binomial(100000, 0.2, 1); print(x4/100000)
   ...: x5=np.random.binomial(1000000, 0.2, 1); print(x5/1000000)
[0.18]        # n=100     or:   sum(np.random.binomial(1, 0.2, 100) == 1)/100
[0.206]       # n=1000
[0.2023]      # n=10000
[0.20037]     # n=100000
[0.199933]    # n=1000000
```

The Python code to derive a figure similar to Figure 2.1 in Chapter 2 is provided next:

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: n = 1001  # Number of independent experiments in each trial
   ...: p = 0.2  # Probability of success for each experiment
   ...: def run_binom(n, p):  # Function that runs binomials
   ...:         phat = []
   ...:         for i in range(1,n):
   ...:                 phat.append(np.random.binomial(i,p,1)/i)
   ...:         return phat
   ...: phat = run_binom(n, p)  # run the function
   ...: len(phat)               # check the length of the created list
```

```
    ...: fig = plt.figure(figsize=(10, 7))
    ...: plt.scatter(range(1,n), phat,s=10)  # plot scatterplot
    ...: plt.xlabel("n", size=14)
    ...: plt.ylabel("proportion", size=14)
```

B2.2 Python Functions for Discrete Probability Distributions

Many discrete probability distributions have objects available in the `scipy.stats` module of the `scipy` library, including the binomial (`binom`), geometric (`geom`), multinomial (`multinomial`), Poisson (`poisson`), and negative binomial (`nbinom`). Each has arguments for the parameter values and for options such as displaying the *pmf*.

B2.2.1 Binomial Distribution

The `binom` object has several options for binomial distributions, including calculation of *pmf* or *cdf* values and random number generation. The following shows code for computing binomial probabilities and plotting a binomial *pmf* for the example in Section 2.4.2 about the Hispanic composition of a jury list, which has $n = 12$ and $\pi = 0.20$:

```
In [1]: import numpy as np
   ...: from scipy.stats import binom
   ...: import matplotlib.pyplot as plt
In [2]: binom.pmf(1, 12, 0.20)           # binomial P(Y=1) when n=12, pi=0.20
Out[2]: 0.2061584302079996
In [3]: fig, ax = plt.subplots(1, 1)
   ...: n, pi = 12, 0.2           # following creates plot of bin(12,0.2) pmf
   ...: y=list(range(0,13))       # y values between 0 and 12
   ...: ax.vlines(y, 0, binom.pmf(y, n, pi), colors='b', lw=5, alpha=0.5)
   ...: plt.xlabel("y")
   ...: plt.ylabel("P(y)")
   ...: plt.xticks(np.arange(min(y), max(y)+1, 1.0))
In [4]: print(list(binom.pmf(y, n, pi))) # displays binomial probabilities
[0.06871947673599997, 0.2061584302079996, 0.28346784153599947,
0.23622320128000002, 0.1328755507199998, 0.05315022028799997,
0.01550214758399999, 0.003321888767999998, 0.0005190451199999995,
5.767168000000002e-05, 4.3253759999999935e-06, 1.9660799999999964e-07,
4.096000000000008e-09]
In [5]: mean, variance, skewness = binom.stats(n, pi, moments='mvs')
   ...: mean, variance, skewness   # compare to: print(mean, variance, skewness)
Out[5]: (array(2.4), array(1.92), array(0.4330127))
```

The following shows code to construct a figure similar to Figure 2.5, for a survey about legalized marijuana, with $n = 3$ and $\pi = 0.50$.

```
In [1]: from scipy.stats import binom
   ...: import matplotlib.pyplot as plt
   ...: fig, ax = plt.subplots(1, 1)
         ...: n, p = 3, 0.5
   ...: x=[0,1,2,3]     # or: x=list(range(0,4))
   ...: ax.vlines(x, 0, binom.pmf(x, n, p), colors='b', lw=5, alpha=0.5)
   ...: plt.xlabel("y")
   ...: plt.ylabel("P(y)")
   ...: plt.xticks(np.arange(min(x), max(x)+1, 1.0))
```

The figure is to be shown in Figure B2.1.

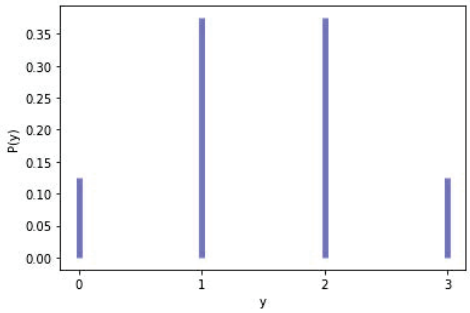


FIGURE B2.1: The probability distribution of Y in Table 2.2 of the book.

B2.2.2 Poisson Distribution

For a Poisson distribution, here is how to find probabilities of individual values using the *pmf* or of a range of values using the *cdf*, such as in the example in Section 2.4.7:

```
In [1]: from scipy.stats import poisson
In [2]: poisson.pmf(0, 2.3) # P(Y=0) if Poisson mean = 2.3
Out[2]: 0.10025884372280375
  # Difference of cdf values at 130 and 69 for Poisson with mean = 100:
In [3]: poisson.cdf(130, 100) - poisson.cdf(69, 100)
Out[3]: 0.9976322764993413
  # Probability within 2 standard deviations of mean (from 80 to 120):
In [4]: poisson.cdf(120, 100) - poisson.cdf(79, 100)
Out[4]: 0.9598793484053718
```

B2.3 `Python` Functions for Continuous Probability Distributions

Many continuous probability distributions are available in the `scipy.stats` module of the `scipy` library, including the beta (`beta`), chi-squared (`chi2`), exponential (`expon`), F (`f`), gamma (`gamma`), logistic (`logistic`), log-normal (`lognorm`), normal (`norm`) and multivariate normal (`multivariate_normal`), t (`t`), and uniform (`uniform`). Each has arguments for the parameter values and for options such as displaying the *pdf*.

B2.3.1 Uniform Distribution

The pdf of a uniform random variable (see Figure B2.2) over the interval [0,1] can be plotted in `Python` as shown below:

```
In [1]: import numpy as np
   ...: from scipy.stats import uniform
   ...: import matplotlib.pyplot as plt
   ...: fig, ax = plt.subplots(1, 1)
   ...: x = np.linspace(uniform.ppf(0.01), uniform.ppf(0.99), 100)
   ...: rv = uniform()
   ...: ax.plot(x, rv.pdf(x), lw=2, color='blue')
   ...: plt.plot([-0.3, 0], [0, 0], lw=2, color='blue')
   ...: plt.plot([1,1.3], [0, 0], lw=2, color='blue')
   ...: plt.xticks(np.arange(0, 1.2, 0.2))
```

```
In [2]: uniform.rvs(0,100,1)  # a single uniform random number in [0,100]
Out[2]: array([76.36941394])
```

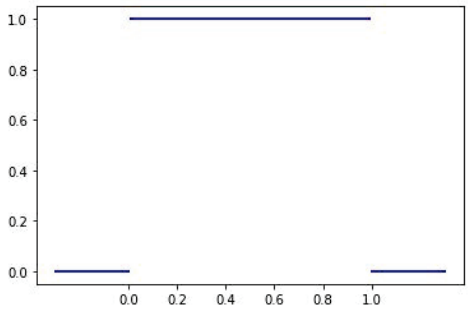


FIGURE B2.2: Probability density function of a uniform random variable over the interval [0, 1].

The plot of a cdf of a uniform random variable can be derived as follows:

```
In [1]: import numpy as np
   ...: from scipy.stats import uniform
   ...: import matplotlib.pyplot as plt
   ...: fig, ax = plt.subplots(1, 1)
   ...: x = np.linspace(uniform.ppf(0.01), uniform.ppf(0.99), 100)
   ...: rv = uniform()
   ...: ax.plot(x, rv.cdf(x), lw=2, color='blue')
   ...: plt.plot([-0.3, 0], [0, 0], lw=2, color='blue')
   ...: plt.plot([1,1.3], [1, 1], lw=2, color='blue')
   ...: plt.xticks(np.arange(0, 1.2, 0.2))
```

B2.3.2 Exponential and Gamma Distributions

The *pdf* of an exponential distribution can be plotted using the `expon` object of the `scipy.stats` module. Next we show how to plot an exponential *pdf* with $\lambda = 1$ (shown in Figure B2.3), analogous to Figure 2.8:

```
In [1]: import numpy as np
   ...: import scipy.stats as ss
   ...: from scipy.stats import expon
   ...: import matplotlib.pyplot as plt
   ...: x = np.linspace(0, 10, 5000)
   ...: th = 1
   ...: y = ss.expon.pdf(x, 0, th)
   ...: plt.plot(x,y, lw=2, color="blue") # see Figure B2.3
   ...: plt.xlabel("y")
   ...: plt.ylabel("f(y)")
```

The following code also shows how to find the 0.05 and 0.95 quantiles of an exponential distribution, such as done with `R` in Section 2.5.6:

```
In [2]: expon.ppf(0.05, scale=1), expon.ppf(0.95, scale=1) # scale is lambda parameter
Out[2]: (0.05129329438755, 2.9957322736)                   # of exponential distribution
```

The probability integral transformation for generating random numbers from an exponential distribution can easily be implemented in `Python`. Below is given an example for $\lambda = 0.5$.

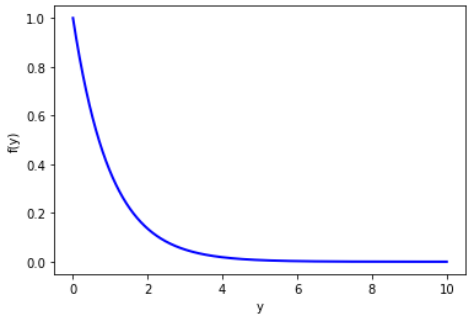


FIGURE B2.3: Probability density function of an exponential random variable with $\lambda = 1$.

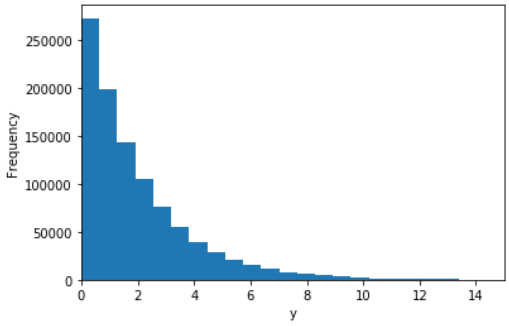


FIGURE B2.4: Randomly generated values from an exponential distribution with $\lambda = 0.5$.

```
In [1]: import numpy as np
   ...: import statistics
   ...: import matplotlib.pyplot as plt
   ...: X=np.random.uniform(0, 1, 1000000)
   ...: Y = -np.log(1 - X)/(0.50)                  # Y has expon.dist., lambda = 0.50
   ...: statistics.mean(Y), statistics.stdev(Y)
Out[1]: (1.9986466750756648, 1.9999145800376117)  # E(Y) = std.dev(Y) = 2.0
In [2]: plt.hist(Y, bins=50)
   ...: plt.xlabel('Y')
   ...: plt.ylabel('Frequency')
   ...: plt.xlim(0, 15)
```

Figure 2.12 in the book portrays gamma distributions with $\mu = 10$ and shape parameters $k = 1, 2$ and 10. Such a plot can be derived in `Python` as shown below:

```
In [1]: import numpy as np
   ...: from scipy.stats import gamma
   ...: from matplotlib import pyplot as plt
   ...: fig, ax = plt.subplots(1, 1)
   ...: a=np.array([1,2,10])   # our k
   ...: sc =10/a               # scale=1/lambda=mu/k
   ...: x = np.linspace(0,40, 100)
   ...: def gamma_pdfs():
   ...:             fig, ax = plt.subplots(1, 1, figsize=(10, 7))
   ...:             for i in range(3):
   ...:                 ax.plot(x, gamma.pdf(x, a[i], 0, sc[i]), lw=2)
   ...:                 ax.legend(['k=1', 'k=2', 'k=10'], loc='upper right')
   ...: gamma_pdfs()
```

```
    ...: plt.xlabel("y")
    ...: plt.ylabel("pdf f(y)")
```

B2.3.3 Normal Distribution

We use the *cdf* of a normal distribution to find tail probabilities or central probabilities. Next, using the *cdf* of the standard normal, we find the probabilities falling within 1, 2, and 3 standard deviations of the mean, as in the R code in Section 2.5.2:

```
In [1]: from scipy.stats import norm
In [2]: norm.cdf(1) - norm.cdf(-1) # probability within 1 standard deviation of mean
Out[2]: 0.6826894921370859
In [3]: norm.cdf(2) - norm.cdf(-2) # probability within 2 standard deviation of mean
Out[3]: 0.9544997361036416
In [4]: norm.cdf(3) - norm.cdf(-3) # probability within 3 standard deviation of mean
Out[4]: 0.9973002039367398
```

Next we use Python for the Section 2.5.3 examples of finding probabilities and quantiles, such as finding the proportion of the self-employed who work between 50 and 70 hours a week, when the times have a $N(45, 15^2)$ distribution. We can apply normal distributions other than the standard normal by specifying μ and σ:

```
In [1]: from scipy.stats import norm
In [2]: norm.cdf(70,45,15) - norm.cdf(50,45,15) # mean = 45, standard dev. = 15
Out[2]: 0.32165098790894897     # probability between 50 and 70
In [3]: norm.ppf(0.99)          # 0.99 quantile of standard normal
Out[3]: 2.3263478740408408
In [4]: norm.ppf(0.99, 100, 16) # 0.99 normal quantile for IQ's
Out[4]: 137.22156598465347      # when mean = 100, standard deviation = 16
In [5]: norm.cdf(550, 500, 100) # SAT = 550 is 69th percentile
Out[5]: 0.6914624612740131      # when SAT mean = 500, standard deviation = 100
In [6]: norm.cdf(30, 18, 6)     # ACT =  30 is 97.7 percentile
Out[6]: 0.9772498680518208      # when ACT mean = 18, standard deviation = 6
```

The code for plotting the *pmf* of a Poisson distribution along with the *pdf* of a normal with $\mu = 100$ and $\sigma = 10$ (see Figure B2.5) is shown next:

```
In [5]: from scipy.stats import poisson
   ...: import numpy as np               # additional imports to the 'poisson'
   ...: import matplotlib.pyplot as plt
In [6]: fig, ax = plt.subplots(1, 1)
   ...: # creation of a plot of a poisson(100) pmf follows:
   ...: y=list(range(60,141)) # y values between 60 and 140 with increment of 1
   ...: ax.vlines(y, 0, poisson.pmf(y, 100), colors='b', lw=1, alpha=0.5)
   ...: ax.plot(y, norm.pdf(y,100,10),lw=2, color='r',alpha=0.5)  # normal pdf
   ...: plt.xlabel("y")
   ...: plt.ylabel("P(y)")
   ...: plt.xticks(np.arange(min(y), max(y)+1, 10))
```

B2.3.4 *Q-Q* Plots and the Normal Quantile Plot

Exercise 2.67 in Chapter 2 and Section A.2.2 in the R Appendix introduced the *Q-Q plot* (*quantile-quantile plot*) as a graphical comparison of an observed sample data distribution with a theoretical distribution. With the standard normal distribution for the theoretical quantiles, the Q-Q plot is called a *normal quantile plot*. If the observed data also come from a normal distribution, the points on the normal quantile plot should follow approximately

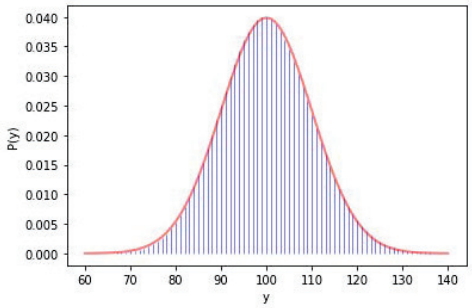


FIGURE B2.5: Probability mass function of a Poisson random variable when $\mu = 100$ and pdf of a $N(100, 10^2)$ in red.

a straight line, the deviations from a straight line reflecting random sampling variability. To illustrate how `Python` can construct this plot, we construct one for the carbon dioxide emissions values for the 31 European nations in the `Carbon` data file:

```
In [1]: import pandas as pd
   ...: from scipy.stats import probplot
   ...: from matplotlib import pyplot
In [2]: Carbon = pd.read_csv('http://stat4ds.rwth-aachen.de/data/
                Carbon.dat', sep='\s+')
In [3]: probplot(Carbon['CO2'], dist = 'norm', plot = pyplot)
In [4]: Carbon2 = pd.read_csv('http://stat4ds.rwth-aachen.de/data/
                Carbon_West.dat', sep='\s+')
In [5]: probplot(Carbon2['CO2'], dist = 'norm', plot = pyplot)
```

Figure B2.6 shows the plot, on the left. It does not show any clear departure from normality, as with such a small n, the deviations from the straight line could merely be due to ordinary sampling variability. Figure B2.6 (right) also shows the normal quantile plot for the `Carbon_West` data file at the book's website that adds four Western nations to the data file for Europe. The values are quite large for three nations (Australia, Canada, U.S.), and these appear on the upper-right part of the plot as values that are larger than expected for observations in the right tail of a normal distribution. A couple of points on the lower-left part of the plot are not as small as expected for observations in the left tail of a normal distribution. This is a typical normal quantile plot display when the sample-data distribution is skewed to the right.

B2.4 Expectations of Random Variables

B2.4.1 Binomial distribution

For a sufficiently large number of simulations, the sample mean of a random sample from a binomial distribution is close to its expected value. Section 2.3.1 illustrated this for the example shown next with `Python`:

```
In [1]: import numpy as np # randomly generate 10000000 bin(3,0.5) rv's
In [2]: y = np.random.binomial(3, 0.5, 10000000)
In [3]: list(y[0: 10])     # first 10 of 10 million generated
Out[3]: [1, 2, 2, 3, 1, 2, 1, 1, 2, 1]
```

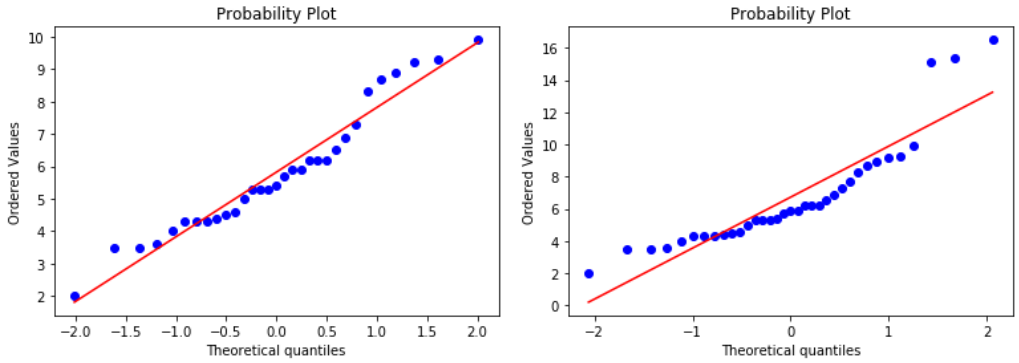


FIGURE B2.6: Normal quantile plots for carbon dioxide emissions, for 31 European nations (left) and also including four other Western nations (right).

```
In [4]: sum(y)/10000000    # sample mean of 10000000 binomial outcomes
Out[4]: 1.4999395          # binomial expected value n(pi) = 3(0.5) = 1.5
```

For the example in Section 2.4.4 of gauging the popularity of a prime minister, using a sample survey with $n = 1500$ when $\pi = 0.60$, we use `Python` to find the mean and standard deviation of the relevant binomial distribution and find the probability within 2 and within 3 standard deviations of the mean:

```
In [1]: from scipy.stats import binom
   ...: n, p = 1500, 0.60
   ...: mu = binom.mean(n, p)    # mean of binomial(1500, 0.60)
   ...: sigma = binom.std(n, p)  # standard deviation of binomial(1500, 0.60)
   ...: mu, sigma
Out[1]: (900.0, 18.973665961010276)
In [2]: binom.cdf(mu + 2*sigma, n, p) - binom.cdf(mu - 2*sigma, n, p)
Out[2]: 0.9519324392528513       # probability within 2 standard dev's of mean
In [3]: binom.cdf(mu + 3*sigma, n,p) - binom.cdf(mu - 3*sigma, n,p)
Out[3]: 0.9971083299488276       # probability within 3 standard dev's of mean
```

Since this binomial distribution is approximately normal, the probabilities are close to the normal probabilities of 0.9545 and 0.9973.

B2.4.2 Uniform Distribution

Section 2.3.3 showed that a uniform random variable over the interval $[0, U]$ has $\mu = U/2$ and $\sigma = U/\sqrt{12}$. Here we use `Python` fo find the mean and standard deviation of a simulated sample of 10 million random outcomes from a uniform [0, 100] distribution, for which $\mu = 50.0$ and $\sigma = 28.8675$:

```
In [1]: import numpy as np
   ...: n=10000000
   ...: y=np.random.uniform(0, 100, n)
In [2]: list(y[0:5])                          # first 5 simulated values
Out[2]:
[27.34765205743761,
 20.216650993789067,
 10.371009047647906,
 78.2854396004128,
 62.198820513759124]
```

```
In [3]: ymean=sum(y)/n                        # mean of values in list y
   ...: ysd=np.sqrt(sum((y-ymean)**2)/(n-1))  # standard deviation of values in list y
In [2]: ymean, ysd
Out[2]: (49.999411632715706, 28.86646306368665)
# alternatively:
In [3]: import statistics                      # required for functions for mean and st.dev.
   ...: statistics.mean(y), statistics.stdev(y)
```

B2.4.3 Finding the Correlation For a Joint Probability Distribution

For a particular joint probability distribution, we can find the correlation using equation (2.16). We illustrate for the correlation between income and happiness for the joint distribution in Table 2.5, using the fact that the covariance between a random variable and itself is the variance:

```
In [9]: import numpy as np
   ...: prob =[0.2, 0.1, 0.0, 0.1, 0.2, 0.1, 0.0, 0.1, 0.2]
   ...: x=[1,1,1,2,2,2,3,3,3]
   ...: y=[1,2,3,1,2,3,1,2,3]
   ...: covxy = np.cov(x, y, rowvar=False, aweights=prob)
   ...: covx  = np.cov(x, x, rowvar=False, aweights=prob)
   ...: covy  = np.cov(y, y, rowvar=False, aweights=prob)
   ...: r = covxy/(np.sqrt(varx*vary)) # 2x2 correlation matrix
   ...: print(round(r[0,1], 5))        # x-y correlation: element r[0,1]=r[1,0]
0.66667
```

3

CHAPTER 3: PYTHON FOR SAMPLING DISTRIBUTIONS

B3.1 Simulation to Illustrate a Sampling Distribution

To explain the concept of a sampling distribution, Section 3.1.1 used simulation to illustrate results of an exit poll in a U.S. Presidential election, when the probability is $\pi = 0.50$ of voting for Joe Biden. Here we use Python to do this for a random sample of 2271 voters:

```
In [1]: import numpy as np
   ...: n, p = 2271, 0.50                # values for binomial n, pi
   ...: x = np.random.binomial(n, p, 1)  # 1 binomial experiment
   ...: print(x); print(x/n)
[1128]         # binomial random variable = 1128 Biden votes
[0.49669749]   # simulated proportion of Biden votes = 0.497
```

The above process is repeated a million times next, aiming at investigating the variability in the results of the simulated proportion voting for Biden, when half of the population voted for him. Also shown is the code for deriving the histogram of the million simulated proportions. The histogram is pictured in Figure B3.1 (compare to Figure 3.1).

```
In [2]: import matplotlib.pyplot as plt
   ...: import statistics     # use for mean and standard deviation functions
In [3]: results = np.random.binomial(n, p, 1000000)/n
   ...: statistics.mean(results)
Out[3]: 0.4999967811536768    # mean of million sample proportion values
In [4]: statistics.stdev(results)
Out[4]: 0.010503525956680382  # standard deviation of million sample proportions
In [5]: plt.hist(results, bins=14, edgecolor='k') # histogram
   ...: plt.xlabel('Sample proportion'); plt.ylabel('Frequency')
```

B3.2 Law of Large Numbers

The simulation discussed in Section 3.2.5, to illustrate the law of large numbers, is performed here in Python, using the code for uniform random number generation already seen in Section B2.4.2:

```
In [1]: import numpy as np
   ...: n1, n2, n3 = 10, 1000, 10000000
   ...: y1=np.random.uniform(0, 100, n1); mean1=sum(y1)/n1
   ...: print(mean1)        # sample mean for random sample of
27.12067089376516           # n=10  from uniform [0,100]
In [2]: y2=np.random.uniform(0, 100, n2); mean2=sum(y2)/n2
   ...: print(mean2)        # (population mean = 50.0)
```

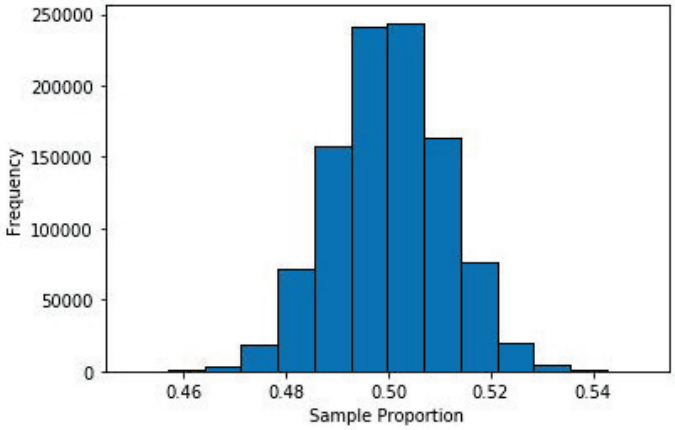


FIGURE B3.1: Histogram of one million simulations of the sample proportion favoring Biden, for simple random samples of 2271 subjects from a population in which exactly half voted for Biden.

```
49.955368338199484          # sample mean for n=1000
In [3]: y3=np.random.uniform(0, 100, n3); mean3=sum(y3)/n3
   ...: print(mean3)
49.99943168403832           # sample mean for n=10000000
```

4

CHAPTER 4: PYTHON FOR ESTIMATION

B4.1 Confidence Intervals for Proportions

The $(1-\alpha)100\%$ asymptotic confidence intervals (CIs) for a binomial proportion can be easily derived in the `statsmodels` library of `python`, as shown below for the example of Section 4.3.4. Of the 1497 respondents in The Netherlands, 778 reported being atheists or agnostics:

```
In [1]: from statsmodels.stats.proportion import proportion_confint
   ...: proportion_confint(count=778,               # Number of successes
   ...:                    nobs=1497,               # Number of trials
   ...:                    alpha=0.05, method="normal")       # default
Out[1]: (0.4943973906940667, 0.545014766954564)               # 95% Wald CI

In [2]: proportion_confint(778,  1497, method="wilson")
Out[2]: (0.4943793119474541, 0.5449319688365669)              # 95% Score CI
```

Notice that the score CI is called Wilson after the name of the statistician who originally proposed it. `Python` does not seem to currently have a function for finding a likelihood-ratio test-based CI for a proportion.

B4.2 The t Distribution

The t distribution is used for the construction of confidence intervals for the mean of normal populations when the variances are unknown (see Section 4.4.2). In Section 4.4.1 the properties of t distribution are discussed and t cumulative probabilities as well as t-quantiles, which are required for the derivation of asymptotic CIs are calculated. The same calculations are carried out in `Python` below. Also the code for deriving the plot of pdf's for t distributions of various degrees of freedom is provided (analogue to Figure 4.5):

```
In [1]: import numpy as np
   ...: from scipy.stats import t
   ...: from scipy.stats import norm
   ...: from matplotlib import pyplot as plt
   ...: fig, ax = plt.subplots(1, 1)
In [2]: df=np.array([1,3,8,30])    # degrees of freedom
   ...: y = np.linspace(-4,4, 100)
   ...: def t_pdfs():              # function that creates plot as in Figure 4.5
   ...:            fig, ax = plt.subplots(1, 1, figsize=(10, 7))
   ...:            for i in range(4):
   ...:                ax.plot(y, t.pdf(y, df[i]), lw=2)
   ...:            ax.plot(y, norm.pdf(y), lw=2, linestyle='dashed')
   ...:            ax.legend(['df=1', 'df=3', 'df=8', 'df=30', 'normal'],
                             loc='upper right')
```

```
   ...: t_pdfs()                    # runs the function
   ...: plt.xlabel("y")
   ...: plt.ylabel("Probability density function")
In [3]: df=np.array([1,10,30,100,1000,10000])
   ...: t.ppf(0.975, df)            # 0.975-quantiles for corresponding df
Out[3]:
array([12.70620474,  2.22813885,  2.04227246,  1.98397152,  1.96233908,
       1.96020124])
In [4]: t.cdf(1.960201, 10000)
Out[4]: 0.9749999859839616          # cumulative prob. at t=1.960201 when df=10000
```

B4.3 Confidence Intervals for Means

We next find descriptive statistics and CIs for the Section 4.4.3 example of analyzing weight changes of anorexic girls who are undergoing a cognitive behavioral therapy:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
In [2]: Anor = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Anorexia.dat', sep='\s+')
In [3]: Anor.head(3)
Out[3]:
   subject therapy  before  after
0        1      cb    80.5   82.2
1        2      cb    84.9   85.6
2        3      cb    81.5   81.4
In [4]: change = Anor['after'] - Anor['before']
   ...: Anor['change'] = change     # add new variable to the data frame
   ...: Anor.loc[Anor['therapy'] == 'cb']['change'].describe()
Out[4]:
count    29.000000
mean      3.006897
std       7.308504
min      -9.100000
25%      -0.700000
50%       1.400000
75%       3.900000
max      20.900000
Name: change, dtype: float64
In [5]: bins=list(range(-10,30,5))   # histogram with pre-specified bins:
   ...: plt.hist(Anor.loc[Anor['therapy']=='cb']['change'],
                 bins, edgecolor='k')
   ...: plt.xlabel('Weight change'); plt.ylabel('Frequency')
In [6]: changeCB = Anor.loc[Anor['therapy'] == 'cb']['change']
In [7]: import statsmodels.stats.api as sms
   ...: sms.DescrStatsW(changeCB).tconfint_mean() # default alpha=0.05
Out[7]: (0.2268901583588, 5.78690294509)      # 95% CI for mean change
In [8]: sms.DescrStatsW(changeCB).tconfint_mean(alpha=0.01)
Out[8]: (-0.743279444048, 6.75707254750)      # 99% CI for mean change
```

B4.4 Confidence Intervals Comparing Means and Comparing Proportions

In `R` the function for t tests for comparing means also provides the corresponding CI for the difference of means (Section 4.5.3). However, the functions in the `statsmodels` and `scipy` libraries for t tests do not also provide CIs. To construct these CIs, assuming equal or unequal population variances for the two groups, we provide the following[1] function:

```
In [1]: from scipy.stats import t
In [2]: def t2ind_confint(y1, y2, equal_var=True, alpha = 0.05):
   ...:
   ...:    # y1, y2 :   vectors or data frames of values for group A and B
   ...:    # returns:   mean_diff: mean(A)-mean(B) (float)
   ...:    #            confint: CI for mu_A - mu_B  (1d ndarray)
   ...:    #            conf: confidence level of the CI (float)
   ...:    #            df (float)
   ...:
   ...:    n1 = len(y1); n2=len(y2)
   ...:    var1 = np.var(y1)*n1/(n1-1); var2 = np.var(y2)*n2/(n2-1)
   ...:
   ...:    if equal_var:
   ...:      df=n1+n2-2
   ...:      vardiff=((n1-1)*var1+(n2-1)*var2)/(n1+n2-2)*(1/n1+1/n2)
   ...:    else:
   ...:      df= (var1/n1+var2/n2)**2/(var1**2/(n1**2*(n1-1))+var2**2/(n2**2*(n2-1)))
   ...:      vardiff= var1/n1+var2/n2
   ...:
   ...:    se = np.sqrt(vardiff)
   ...:    qt = t.ppf(1-alpha / 2,df)    # t quantile for 100(1-alpha)% CI
   ...:    mean_diff = np.mean(y1) - np.mean(y2)
   ...:    confint = mean_diff + np.array([-1, 1]) * qt * se
   ...:    conf= 1-alpha
   ...:    return mean_diff, confint, conf, df
# returns: mean(A) - mean(B), CI for mu_A - mu_B, confidence level, df
```

Next we implement this function for computing a 95% CI for the average difference in weight change between the therapy and the control groups in the anorexia study:

```
In [1]: import pandas as pd
In [2]: Anor = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Anorexia.dat', sep='\s+')
In [3]: cogbehav = Anor.loc[Anor['therapy']=='cb']['change']
   ...: control = Anor.loc[Anor['therapy']=='c']['change']
In [5]: mean_diff, confint, conf, df = t2ind_confint(cogbehav,control)  # call the function above
   ...: print('mean1-mean2 =', mean_diff)   # assume equal variances
   ...: print(conf, 'CI:', confint)
   ...: print('df =', df)
mean1-mean2 = 3.456896551724137
0.95 CI: [-0.68013704  7.59393014]
df = 53
In [6]: mean_diff, confint, conf, df = t2ind_confint(cogbehav,control, equal_var=False)
                                            # permit unequal variances
   ...: print('mean1-mean2 =', mean_diff)
   ...: print(conf, 'CI:', confint)
   ...: print('df =', df)
mean1-mean2 = 3.456896551724137
0.95 CI: [-0.70446319  7.61825629]
df = 50.97065330426786
```

[1] We introduce *v1* and *v2* because `np.var()` uses n instead of $n-1$ in the denominator.

Asymptotic CIs for the difference of two proportions are not provided directly in the standard libraries of python. Next, we provide a function that computes the $(1-\alpha)100\%$ Wald CI and implement it on the example analyzed in R in Section 4.5.5 about whether prayer helps coronary surgery patients:

```
In [1]: import numpy as np
   ...: from scipy.stats import norm
In [2]: def prop2_confint(y1, n1, y2, n2, alpha = 0.05):
   ...:     # y1, y2 : Number of successes in group A and B (int)
   ...:     # n1, n2 : Number of trials in group A and B (int)
   ...:     # returns:   prop_diff for A-B (float), comfint (1d ndarray)
   ...:
   ...:     prop1 = y1 / n1; prop2 = y2 / n2
   ...:     var = prop1 * (1 - prop1) / n1 + prop2 * (1 - prop2) / n2
   ...:     se = np.sqrt(var)
   ...:     qz = norm.ppf(1-alpha / 2)        # standard normal quantile
   ...:
   ...:     prop_diff = prop1 - prop2
   ...:     confint = prop_diff + np.array([-1, 1]) * qz * se
   ...:     conf= 1-alpha
   ...:     return prop_diff, confint, conf   # returns diff, CI, level
         ...:
# call the function for data on prayers and coronary surgery:
In [3]: prop_diff, confint, conf = prop2_confint(315, 604, 304, 597)
   ...: print('prop1-prop2 =', prop_diff)
   ...: print(conf, 'CI:', confint)
prop1-prop2 = 0.012310448489689096
0.95 CI: [-0.04421536  0.06883625]
```

B4.5 Bootstrap Confidence Intervals

Next, we repeat in Python the analysis of Section 4.6.2. There bootstrap CIs for the median and standard deviation were constructed. The following code uses the bootstrapped package to construct a percentile-based CI, for the variable P in the Library data file giving the number of years since publication of the book. It also provides summary statistics and two options for constructing a box plot, with and without outliers:

```
In [1]: import os
   ...: import pandas as pd
         ...: import numpy as np
   ...: import matplotlib.pyplot as plt
In [2]: Books = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Library.dat', sep='\s+')
   ...: Books.head(3)
Out[2]:
   C  P
0  1  3
1  9  9
2  4  4
In [3]: Books.describe()
Out[3]:
                C           P
count  53.000000   53.000000
mean    8.490566   21.981132
std    15.191879   25.793179
min     0.000000    3.000000
25%     1.000000    9.000000
50%     4.000000   17.000000
```

```
75%     9.000000   19.000000
max    92.000000  140.000000
In [4]: np.median(Books['C'])
Out[4]: 4.0
In [5]: np.median(Books['P'])
Out[5]: 17.0
In [6]: plt.boxplot(Books["P"], vert=False)   # Box plot of 'P'
   ...: plt.xlabel("Years since publication")
# Box plot of 'P' without outliers:
In [7]: plt.boxplot(Books["P"], vert=False, showfliers=False)
   ...: plt.xlabel("Years since publication")
# --------  BOOTSTRAP by bootstrapped: ----------------------------
In [8]: pip install bootstrapped    # needs to be done once!
In [9]: import bootstrapped.bootstrap as bs
   ...: import bootstrapped.stats_functions as bs_stats
In [10]: population = Books["P"]
    ...: samples = population[:10000]
In [11]: print(bs.bootstrap(samples, stat_func=bs_stats.median))
17.0    (15.0, 23.0)  # bootstrap CI for median of P
In [12]: print(bs.bootstrap(samples, stat_func=bs_stats.std))
25.548688675899356    (15.742691539935805, 37.91811859395936) # for st.dev.
```

B4.6 Bayesian Posterior Intervals for Proportions and Means

We next show a Python implementation of the posterior interval for a proportion using the Jeffreys prior, which is the beta(0.5, 0.5) distribution, for the example in Section 4.7.3 about the proportion believing in hell. The Jeffreys posterior interval is provided as an option in the `proportion_confint()` function of `statsmodels.stats.proportion`. HPD regions can be derived in `pymc3`, which has to be installed before being imported.

```
In [1]: from statsmodels.stats.proportion import proportion_confint
   ...: proportion_confint(814, 1142, method='jeffreys')
Out[1]: (0.686028505, 0.738463665)  # 95% Jeffreys posterior interval
In [2]: import pymc3
   ...: from scipy.stats import beta
In [3]: beta_dist = beta.rvs(size = 5000000, a = 814.5, b = 328.5)
   ...: print(pymc3.stats.hpd(beta_dist, alpha=0.05))
[0.68727542 0.73758295]                        # 95% HPD interval when use Jeffreys prior
In [4]: import numpy as np
   ...: print('[',np.quantile(beta_dist, 0.025),',', np.quantile(beta_dist, 0.975),']')
[ 0.6860454783123715 , 0.7384521768118637 ]    # ordinary 95% posterior interval
```

The HPD posterior interval for the mean weight change of anorexic girls is derived in `pymc3` with exactly the same approach as in Section 4.8.2 using `R`:

```
# continue analysis from Section B.4.3 with Anor data file
# (required is the variable: changeCB )
In [1]: import numpy as np
In [2]: from pymc3 import  *
   ...: data = dict(y = changeCB)
   ...: B0=10**(-7)                        # using priors: inverse gamma,
   ...: with Model() as model:
   ...:   # define highly disperse priors for variance and mean
   ...:   sigma = InverseGamma('sigma', B0, B0, testval=1.)
   ...:   intercept = Normal('Intercept', 0, sigma=1/B0)
   ...:   # define likelihood function for normal responses
   ...:   likelihood = Normal('y',mu=intercept,sigma=sigma,observed=changeCB)
```

```
    ...:   trace = sample(50000, cores=2)  # 100000 posterior samples
In [3]: np.mean(trace['Intercept'])
Out[4]: 3.007279525692707                  # mean of posterior distribution
In [4]: np.std(trace['Intercept'])
Out[4]: 1.413687215567763                  # standard deviation of posterior dist.
In [5]: pymc3.stats.hpd(trace['Intercept'], alpha=0.05)
Out[5]: array([0.31450337, 5.61027393])   # 95% posterior interval
```

For comparison, the classical 95% CI of (0.227, 5.787) for the population mean weight change gives similar substantive conclusions.

Note that in `pymc3` the standard choice for a weakly informative prior [2] for σ^2 is a half Cauchy distribution, following the suggestion by Gelman [3]. A half Cauchy distribution has probability density function

$$f(y;\mu,\sigma) = \begin{cases} \frac{2}{\pi\sigma}\frac{1}{1+(y-\mu)^2/\sigma^2} & y \geq \mu \\ 0 & \text{otherwise} \end{cases},$$

i.e., it is a truncated Cauchy distribution with support $[\mu, \infty)$. Repeating the analysis above with the half Cauchy prior, i.e., replacing

```
sigma = InverseGamma ('sigma', B0, B0, testval=1.)
```

by

```
sigma = HalfCauchy('sigma', beta=25, testval=1.)
```

we get the following results:

```
In [5]: np.mean(trace['Intercept'])
Out[5]: 3.0113614298800147
In [6]: np.std(trace['Intercept'])
Out[6]: 1.4595045114786154
In [7]: pymc3.stats.hpd(trace['Intercept'], alpha=0.05)
Out[7]: array([0.2359793 , 5.69659903])
```

[2] A prior that includes less information than we actually have for regularization or stabilization purposes.

[3] Gelman, A. (2006) Prior distributions for variance parameters in hierarchical models, *Bayesian Analysis*, 1, 515–533

5

CHAPTER 5: PYTHON FOR SIGNIFICANCE TESTING

B5.1 Significance Tests for Proportions

The asymptotic test for a binomial proportion, based on the asymptotic normal distribution of the test statistic can be implemented in `statsmodels` as shown below for the example of Section 5.2.2 about climate change. Asymptotic confidence intervals, Wald and score, are also derived:

```
In [1]: import numpy as np
   ...: from statsmodels.stats.proportion import proportions_ztest
In [2]: stat, pval = proportions_ztest(524, 1008, 0.5)
   ...: print('{0:0.4f},'.format(stat),'{0:0.4f}'.format(pval))
1.2609, 0.2074                    # z test statistic and two-sided P-value
In [3]: from statsmodels.stats.proportion import proportion_confint
   ...: proportion_confint(524, 1008)
Out[3]: (0.48899905080191974, 0.55068348888062)              # Wald 95% CI
In [4]: proportion_confint(524, 1008, method='wilson')
Out[1]: (0.48898223316199607, 0.5505496516518761)            # score 95% CI
```

The same `Python` function can test equality of two population proportions. Here is the code for the example of Section 5.4.2 comparing proportions suffering complications after heart surgery for prayer and non-prayer groups:

```
In [1]: import numpy as np
   ...: from statsmodels.stats.proportion import proportions_ztest
In [2]: count = np.array([315,304])       # group 'success' counts
   ...: nobs = np.array([604,597])        # group sample sizes
   ...: stat, pval = proportions_ztest(count, nobs)
   ...: print('{0:0.4f},'.format(stat),'{0:0.4f}'.format(pval))
0.4268, 0.6695           # z test statistic and two-sided P-value
```

B5.2 Chi-Squared Tests Comparing Multiple Proportions in Contingency Tables

The analysis of two-way contingency tables presented in Section 5.4.4 and illustrated on the Happiness example of Section 5.4.5 can be implemented in `Python` as follows:

```
In [1]: import pandas as pd
   ...: Happy = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Happy.dat', sep='\s+')
   ...: rowlabel=['Married', 'Divorced/Separated', 'Never married']
   ...: collabel=['Very happy', 'Pretty happy', 'Not too happy']
```

```
   ...: table = pd.crosstab(Happy['marital'], Happy['happiness'], margins = False)
   ...: table.index=rowlabel
   ...: table.columns=collabel
   ...: table
Out[1]:
                    Very happy  Pretty happy  Not too happy
Married                    432           504             61
Divorced/Separated          92           282            103
Never married              124           409            135

# conditional distributions on happiness (proportions within rows):
In [2]: proptable = pd.crosstab(Happy['marital'], Happy['happiness'], normalize='index')
   ...: proptable.index=rowlabel
   ...: proptable.columns=collabel
   ...: proptable
Out[2]:
                    Very happy  Pretty happy  Not too happy
Married               0.433300      0.505517       0.061184
Divorced/Separated    0.192872      0.591195       0.215933
Never married         0.185629      0.612275       0.202096

In [3]: import statsmodels.api as sm   # expected frequencies under H0: independence
   ...: table = sm.stats.Table(table)
   ...: print(table.fittedvalues)
                    Very happy  Pretty happy  Not too happy
Married             301.613445    556.216153     139.170401
Divorced/Separated  144.302521    266.113445      66.584034
Never married       202.084034    372.670401      93.245565

In [4]: X2 = table.test_nominal_association()     # chi-squared test of independence
   ...: print(X2)
df          4
pvalue      0.0
statistic   197.407019249992
In [5]: table.standardized_resids
Out[5]:
                    Very happy  Pretty happy  Not too happy
Married              12.295576     -4.554333      -9.770639
Divorced/Separated   -5.913202      1.661245       5.457032
Never married        -7.928512      3.411881       5.619486
```

For the derivation of the mosaic plot, we first transform our data from numeric to string, since Python has no way to directly assign labels to the categories of the classification variables within the mosaic command. Coloring the cells according to the values of the standardized residuals is possible by setting the argument `statistic=True` (the default value is `False`), as done below. The mosaic plot is shown in Figure B5.1.

```
In [5]: Happy.loc[Happy['happiness'] == 1, 'happiness'] = 'Very'
   ...: Happy.loc[Happy['happiness'] == 2, 'happiness'] = 'Pretty'
   ...: Happy.loc[Happy['happiness'] == 3, 'happiness'] = 'Not too'
   ...: Happy.loc[Happy['marital'] == 1, 'marital'] = 'Married'
   ...: Happy.loc[Happy['marital'] == 2, 'marital'] = 'Div/Sep'
   ...: Happy.loc[Happy['marital'] == 3, 'marital'] = 'Never'
   ...:
   ...: from statsmodels.graphics.mosaicplot import mosaic
   ...: fig, _ = mosaic(Happy, ['marital','happiness'], statistic=True)
```

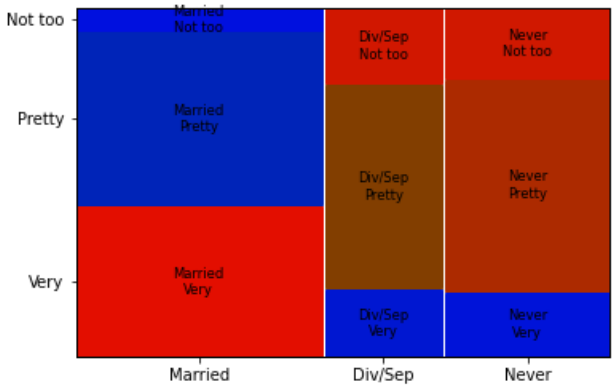


FIGURE B5.1: Mosaic plot for the contingency table cross-classifying the variables 'Happiness' and 'Marital Status' of the data file `Happy`.

B5.3 Significance Tests for Means

To illustrate the t test for a mean, Section 5.2.5 tests whether the Hispanics in a GSS sample have population mean political ideology differing from the moderate value of 4.0, on an ordinal scale from 1 to 7. Here is this test in `Python`:

```
In [1]: import pandas as pd
   ...: Polid = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Polid.dat', sep='\s+')
In [2]: Polid.head(2)
Out[2]:
       race  ideology
1  hispanic         1
2  hispanic         1
In [3]: from scipy import stats
   ...: stats.ttest_1samp(Polid.loc[Polid['race']=='hispanic']['ideology'], 4.0)
                                                          # H_0 mean value is 4.0
Out[3]: Ttest_1sampResult(statistic=1.2827341281592484, pvalue=0.20039257254280335)
In [4]: import statsmodels.stats.api as sms
   ...: sms.DescrStatsW(Polid.loc[Polid['race']=='hispanic']['ideology']).tconfint_mean()
Out[4]: (3.9523333438892, 4.2265284447287) # 95% CI for population mean
```

In discussing the impact of sample size on significance tests, namely that large n can result in statistical significance without practical significance, Section 5.6.2 conducted the test of H_0: $\mu = 4.0$ against H_a: $\mu \neq 4.0$ for political ideology of the entire sample. Here are the results and a corresponding CI:

```
In [5]: from scipy import stats
   ...: stats.ttest_1samp(Polid['ideology'], 4.0)
Out[5]: Ttest_1sampResult(statistic=3.8455584366605935, pvalue=0.00012319510560068636)
In [6]: Iimport statsmodels.stats.api as sms
   ...: sms.DescrStatsW(Polid['ideology']). tconfint_mean()
Out[6]: (4.05291076215289, 4.163011567944197)  # 95% CI for mu
```

B5.4 Significance Tests Comparing Means

In order to test whether the mean weight change in the population differentiates between the therapy and the treatment group, the asymptotic t test was applied assuming that the two groups have equal variances or not (see Section 5.3.2. Here, we perform these tests in `scipy`:

```
In [1]: import pandas as pd
   ...: Anor = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Anorexia.dat', sep='\s+')
In [2]: cogbehav = Anor.loc[Ano['therapy']=='cb']['after']-Anor.loc[Ano['therapy']
                    =='cb']['before']
   ...: control = Anor.loc[Ano['therapy']=='c']['after']-Anor.loc[Ano['therapy']
                           =='c']['before']

In [3]: from scipy import stats
   ...: stats.ttest_ind(cogbehav,control, equal_var = True)
Out[3]: Ttest_indResult(statistic=1.6759971255662465, pvalue=0.09962901351492101)

In [4]: stats.ttest_ind(cogbehav,control, equal_var = False)
Out[4]: Ttest_indResult(statistic=1.667749691844813, pvalue=0.1014985956595161)
```

The CIs for the mean difference are not parts of the output. For these, we constructed the `t2ind_confint()` function (see Section B4.4).

B5.4.1 `Anorexia` Example: Comparison of Therapy and Control Groups

To compare two means with independent samples, Section 5.3.5 showed how to use the modeling approach of Chapter 6, in which an indicator variable represents the two groups being compared. As in that section, we first show the classical analysis and then the Bayesian analysis, for the anorexia study comparing the mean weight change between the cognitive behavioral therapy and control groups:

```
In [1]: import pandas as pd
   ...: Anor = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Anorexia.dat', sep='\s+')
In [5]: import numpy as np
   ...: import statsmodels.formula.api as sm
   ...:
   ...: Anor2 = Anor.loc[Anor['therapy'] != 'f']
   ...: change = Anor2['after'] - Anor2['before']; Anor2['change']=change
   ...: mod = sm.ols(formula="change ~ C(therapy)", data=Anor2).fit()
   ...: print(mod.summary())
==============================================================================================
                       coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------------------
Intercept           -0.4500      1.498     -0.300      0.765      -3.454       2.554
C(therapy)[T.cb]     3.4569      2.063      1.676      0.100      -0.680       7.594
==============================================================================
```

The analysis is equivalent to the one shown in the previous output that assumed equal variances for the groups. The estimated difference between the means of 3.457 has a standard error of 2.063 and a P-value of 0.10 for testing H_0: $\mu_1 = \mu_2$ against H_a: $\mu_1 \neq \mu_2$.

Next the Bayesian analysis follows, verifying the results obtained in `R` (see discussion in Section 5.3.5). We transform first the string variable `therapy` to a binary `x`.

```
In [10]: from sklearn.preprocessing import LabelEncoder
   ...: LE = LabelEncoder()
   ...: Anor2['x'] = LE.fit_transform(Anor2['therapy'])
```

```
   ...: x = np.array(Anor2['x']) # x=0 controls (c), x=1 therapy (cb)
In [11]: from pymc3 import  *
   ...: data = dict(x = x, y = change)
   ...: B0=10**(-8)
   ...: with Model() as model:
   ...:     # define very disperse prior distributions
   ...:     sigma = InverseGamma ('sigma', B0, B0, testval=1.)
         ...:     # alternative prior for sigma (see footnote in Section B4.6):
         ...:     # sigma = HalfCauchy('sigma', beta=25, testval=1.)
   ...:     intercept = Normal('Intercept', 0, sigma = 1/B0)
   ...:     x_coeff = Normal('x', 0, sigma = 1/B0)
   ...:     # define likelihood function for normal response variable
   ...:     likelihood = Normal('y', mu = intercept + x_coeff * x,
                                sigma = sigma, observed = change)
   ...:     fit = sample(50000, cores=2) # posterior samples
In [12]:  summary(fit)
Out[12]:
            mean     sd  hdi_3%  hdi_97% # actually 2.5% and 97.5%
Intercept -0.444  1.543  -3.438    2.366
x          3.452  2.124  -0.517    7.449
sigma      7.749  0.781   6.333    9.220
In [13]:  np.mean(fit['x'] < 0)
Out[13]: 0.05217
```

The posterior mean estimated difference of 3.45 has a posterior standard deviation of 2.12. The 95% posterior interval infers that the population mean difference falls between −0.52 and 7.45. This interval includes the value of 0, indicating it is plausible that $\mu_1 = \mu_2$. As an analog of a one-sided P-value, the Bayesian analysis reports that the posterior probability is 0.052 that the population mean weight change is smaller for the cognitive behavioral group than for the control group.

B5.5 The Power of a Test in `Python`

The `statsmodels` library provide functions for the calculation of powers for basic statistical tests. For example, for the calculation of the power of a test for a proportion with $H_0 : \pi = 1/3 = \pi_0$ and $H_1 : \pi > 1/3$, the power at $\pi_1 = 0.5$ is given below (compare to Section 5.5.6.). Note that the `normal_power()` functions requires as input not the difference in probabilities but the effect size, which for proportions is $2\arcsin(\sqrt{\pi_1}) - 2\arcsin(\sqrt{\pi_0})$.

```
In [1]: import numpy as np
   ...: from statsmodels.stats.power import normal_power
   ...: normal_power(2*(np.arcsin(np.sqrt(0.5))-np.arcsin(np.sqrt(1/3))), 116, 0.05,
                             alternative='larger', sigma=1.)
Out[1]: 0.9780634871667955
```

B5.6 Nonparametric Statistics: Permutation Test and Wilcoxon Test

Limited permutation tests are available of the hypothesis that two populations have identical distributions. We next show one that uses the difference of means as the test statistic to

order all the possible samples, for the example of Section 5.8.2 about petting versus praise of dogs:

```
In [1]: pip install mlxtend
In [2]: from mlxtend.evaluate import permutation_test
   ...: data1 = [114, 203, 217, 254, 256, 284, 296]  # petting observations
   ...: data2 = [4, 7, 24, 25, 48, 71, 294]          # praise observations
   ...: p_value = permutation_test(data1, data2)
   ...: print(p_value)
0.006993006993006993         # P-value for default two-sided alternative
In [3]: p_value = permutation_test(data1, data2, func='x_mean > y_mean')
print(p_value)               # one-sided test of greater mean for petting
0.0034965034965034965        # classical t one-sided P-value is 0.0017
```

Simulations can approximate the P-value (e.g., `'method = approximate', num_rounds = 10000`) when it is infeasible to generate all permutations. This permutation test does not have the option of the test statistic being the difference between the sample medians, which the example in Section 5.8.2 used because of the potentially highly skewed distributions.

More options are provided in the `permute` package (see http://statlab.github.io/permute/). The permutation test for the one-sided alternative for the same example follows, using both available options for test statistic, the difference of the means and the corresponding t statistic. Again, the difference of medians is not an option to order the samples.

```
In [1]: pip install permute
   ...: pip install permute
   ...: pip install pycrypto
   ...: pip install cryptorandom
In [2]: import numpy as np
   ...: from scipy import stats
   ...: from permute.core import two_sample
In [3]: data1 = [114, 203, 217, 254, 256, 284, 296]
   ...: data2 = [4, 7, 24, 25, 48, 71, 294]
   ...: p, t = two_sample(data1, data2, stat='mean', alternative='greater')
In [4]: print('Test statistic:', np.round(t, 5))
   ...: print('P-value (two-sided):', np.round(p, 5))
Test statistic: 164.42857
P-value (two-sided): 0.00354
In [5]: p, t = two_sample(data1, data2, stat='t',alternative='greater', seed=20)
   ...: print('Test statistic:', np.round(t, 5))
   ...: print('P-value (two-sided):', np.round(p, 5))
Test statistic: 3.63509
P-value (two-sided): 0.00352
```

The Wilcoxon test of the hypothesis that two populations have identical distributions, based on comparing the mean ranks of the two samples, is equivalent to the *Mann–Whitney test*. That test is performed by the `mannwhitneyu` function of `scipy.stats`. However, that function uses only the large-sample normal approximation for the distribution of the test statistic rather than an exact permutation analysis. The example of comparing petting with praise for dogs has very small samples ($n_1 = n_2 = 7$), so we show this analysis only for illustration:

```
In [4]: from scipy.stats import mannwhitneyu
   ...: stat, p = mannwhitneyu(data1, data2, use_continuity = False,
                               alternative = 'greater')
In [5]: print(stat,p)        # approximate, based on large-sample distribution
43.0 0.009043230657843692    # exact one-sided P-value is 0.0087
```

B5.7 Kaplan–Meier Estimation of Survival Functions

The analysis of the survival times of the example in Section 5.8.4 can be implemented in `Python` using the `KaplanMeierFitter` function of `lifelines`. The plot of Kaplan—Meier estimators of survival functions for the drug and control groups as a function of time (see Figure 5.10) can be derived as shown below. The produced figure is given in Figure B5.2.

```
In [1]: import pandas as pd
   ...: Survival=pd.read_csv('http://stat4ds.rwth-aachen.de/data/Survival.dat',sep='\s+')
In [2]: import pandas as pd
   ...: from lifelines import KaplanMeierFitter
   ...: kmf1 = KaplanMeierFitter()  # creates class to create an object
   ...: kmf2 = KaplanMeierFitter()
   ...: groups = Survival['group']
   ...: i1 = (groups == 1)          # group i1: drug
   ...: i2 = (groups == 0)          # group i2: control
   ...: T = Survival['time']
   ...: E = Survival['status']      # Event occured (=1)
   ...: kmf1.fit(T[i1], E[i1], label='drug')     # fits model for 1st group
   ...: a1 = kmf1.plot(ci_show=False)
   ...: a1.set_ylabel('Estimated P(survival)')
   ...: kmf2.fit(T[i2], E[i2], label='control')  # fits model for 2nd group
   ...: a2 = kmf2.plot(ax=a1,ci_show=False)
        ...: a2.set_xlabel('Time')
```

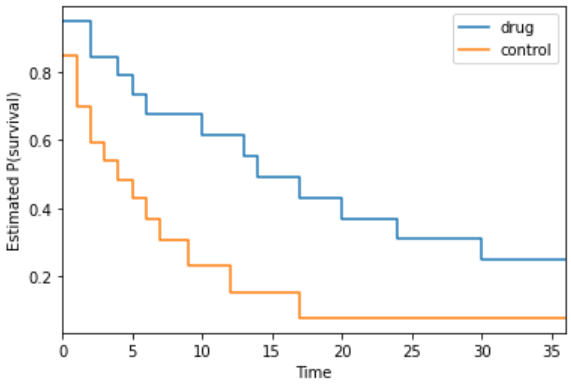


FIGURE B5.2: Kaplan–Meier estimators of survival functions for drug and control groups, giving estimated probabilities of survival as a function of time.

The chi-squared test for comparing the survival distributions with censored times is in `Python` referred as the *log-rank test* and conducted as shown next.

```
In [3]: from lifelines.statistics import logrank_test
   ...: results = logrank_test(T[i1],T[i2],event_observed_A=E[i1],event_observed_B=E[i2])
   ...: results.print_summary()
<lifelines.StatisticalResult: logrank_test>
               t_0 = -1
 null_distribution = chi squared
degrees_of_freedom = 1
         test_name = logrank_test
---
 test_statistic    p  -log2(p)
           6.25 0.01      6.33
```

6

CHAPTER 6: PYTHON FOR LINEAR MODELS

B6.1 Fitting Linear Models

We illustrate Python fitting of linear models with the Scottish hill races data set analyzed in Section 6.1.4. The following code produces a scatterplot matrix for the variables timeW (record time for women), distance, and climb:

```
In [1]: import pandas as pd
   ...: Races = pd.read_csv('http://stats4ds.rwth-aachen.de/data/ScotsRaces.dat', sep='\s+')
In [2]: Races.head(3)
Out[2]:
              race  distance  climb   timeM   timeW
0       AnTeallach      10.6  1.062   74.68   89.72
1     ArrocharAlps      25.0  2.400  187.32  222.03
2  BaddinsgillRound     16.4  0.650   87.18  102.48
# create a data frame containing only the variables for the scatterplot matrix:
In [3]: Races2 = Races.drop(['timeM'], axis=1)
In [4]: import seaborn as sns
   ...: sns.set(style="ticks")
   ...: sns.pairplot(Races2)
```

Figure B6.1 shows the scatterplot matrix. Each diagonal entry of the matrix portays a histogram of the corresponding variable.

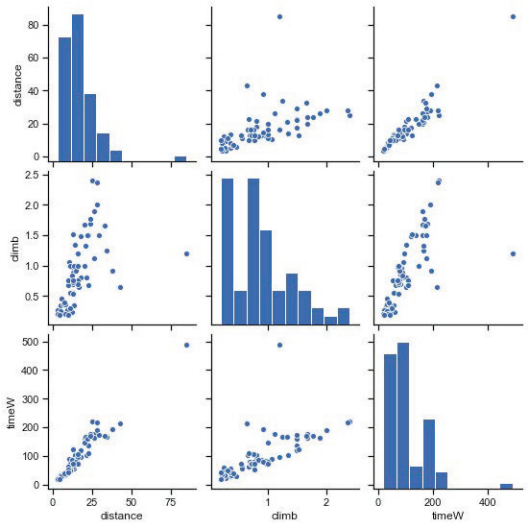


FIGURE B6.1: Scatterplot matrix for record time for women, distance, and climb, for Scottish hill races data.

A linear regression model for predicting the women record time with distance as the sole explanatory variable is fitted by the function `sm.ols()` of `statsmodels`. The associated code is given below, providing part of the derived output:

```
In [5]: import statsmodels.formula.api as smf
   ...: fitd = smf.ols(formula='timeW ~ distance', data=Races).fit()
   ...: print(fitd.summary()) # edited output
              coef  std err         t    P>|t|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept    3.1076    4.537     0.685   0.496      -5.950    12.165
distance     5.8684    0.223    26.330   0.000       5.423     6.313
==============================================================================
```

The following provides just the parameter estimates of the above fitted model:

```
print(fitd.params)
Intercept    3.107563
distance     5.868443
dtype: float64
```

Next, the linear regression model for women's record time with two explanatory variables (`distance` and `climb`) is fitted (compare to the `R` code in Section 6.2.2):

```
In [6]: fitdc = smf.ols(formula="timeW ~ distance + climb", data=Races).fit()
   ...: print(fitdc.summary())
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -14.5997      3.468     -4.210      0.000     -21.526      -7.674
distance       5.0362      0.168     29.919      0.000       4.700       5.372
climb         35.5610      3.700      9.610      0.000      28.171      42.951
==============================================================================
```

To permit interaction, we place a colon between an interacting pair, as in `R`. For the Scottish hill races, the linear regression model with added interaction term between distance and climb (see Section 6.2.7) is fitted as shown below:

```
In [7]: fitdc_int = smf.ols(formula='timeW ~ distance + climb + distance:climb',
                            data=Races).fit()
   ...: print(fitdc_int.summary())
                  coef  std err        t    P>|t|     [0.025   0.975]
------------------------------------------------------------------------------
Intercept      -5.0162    6.683   -0.751    0.456    -18.367    8.335
distance        4.3682    0.433   10.083    0.000      3.503    5.234
climb          23.9446    7.858    3.047    0.003      8.247   39.643
distance:climb  0.6582    0.394    1.669    0.100     -0.129    1.446
==============================================================================
```

B6.2 The Correlation and R-Squared

The correlation matrix with all pairwise correlations among the variables in the reduced data frame (without `timeM`, record time for men) is given next (`pandas` is required). We also do this by excluding the outlying case 41 for the extremely long race, repeating the analysis of Section 6.1.5.

```
In [8]: Races2.corr() # Races2 is data frame without timeM
Out[8]:               # correlation matrix for variables in Races2
          distance     climb     timeW
distance  1.000000  0.514471  0.955549
climb     0.514471  1.000000  0.685292
timeW     0.955549  0.685292  1.000000

In [9]: Races3 = Races2.loc[Races2.index != 40] # row 41 has index=40
   ...: Races3.corr()                           # (indices start at 0)
Out[9]:
          distance     climb     timeW
distance  1.000000  0.661714  0.920539
climb     0.661714  1.000000  0.851599
timeW     0.920539  0.851599  1.000000
```

We next find R^2, the multiple correlation, and the residual standard error and variance and marginal variance of the `timeW` response variable, using the model applied to the data file without the outlying observation:

```
# Race3 excludes case 41:
In [10]: fitdc2 = smf.ols(formula = 'timeW ~ distance + climb', data=Races3).fit()
    ...: print(fitdc2.summary()) # not shown here
    ...: print (fitdc2.params)   # parameter estimates without case 41
----------------------------------------------------------------------
Intercept    -8.931466
distance      4.172074          # 5.036 when include case 41
climb        43.852096          # 35.561 when include case 41
----------------------------------------------------------------------
In [11]: print ('R-Squared:', fitdc2.rsquared)
    ...: print ('adjusted R-Squared:', fitdc2.rsquared_adj)
R-Squared: 0.9519750513925197
adjusted R-Squared: 0.9504742717485359
In [12]: fitted = fitdc2.predict()               # model fitted values for timeW
In [13]: np.corrcoef(Races3.timeW, fitted)[0,1]  # multiple correlation
Out[13]: 0.9756920884134088
In [14]: residuals = fitdc2.resid
    ...: n=len(Races3.index); p=2                # p = number of explanatory var's
    ...: res_se = np.std(residuals)*np.sqrt(n/(n-(p+1)))
    ...: print ('residual standard error:', res_se)
residual standard error: 12.225327029914554
In [15]: res_se**2    # estimated error variance = squared residual standard error
Out[15]: 149.45862098835943
In [16]: np.var(Races3.timeW)*n/(n-1)            # estimated marginal variance
Out[16]: 3017.7975421076458                      # of women's record times
```

B6.3 Diagnostics: Residuals and Cook's Distances for Linear Models

We next show how `Python` can use diagnostics to check model assumptions and detect influential observations. Section A6.2 in the `R` webappendix discussed how plots of the residuals can detect violations of model assumptions. We next show some residual plots that are available in `Python`. We first produce a histogram and a normal quantile plot [1] of the residuals, for the linear model for the complete Scottish hill races data with explanatory variables distance and climb:

[1] Sections A2.2 and B2.3.4 introduced Q-Q plots and the normal quantile plot that uses the standard normal distribution for the theoretical quantiles.

```
In [17]: import matplotlib.pyplot as plt
    ...: import statsmodels.api as sm
In [18]: fitted = fitdc.predict()               # fitted values that predict timeW
    ...: residuals = fitdc.resid                # observed timeW - fitted value
    ...: residuals.head()                       # first five residual values
Out[18]:
0    13.170403
1    25.378848
2    11.371690
3     6.504777
4     4.410875
In [19]: plt.hist(residuals, density=False)     # histogram (Figure B6.2, left)
    ...: plt.xlabel('residuals'); plt.ylabel('frequencies')
In [20]: import scipy.stats as stats
# qqplot of residuals:   (Figure B6.2, right)
In [21]: fig = sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True)
```

These two derived plots, shown in Figure B6.2, check whether the conditional distribution of the response variable is approximately normal, which is an assumption for making statistical inference with a linear model. Here, you can check that the histogram of the residuals is approximately bell-shaped and that the normal quantile plot shows a few outliers at the low and high ends, suggesting that the conditional distribution of *timeW* is approximately normal for this model. Check of normality is here presented only for illustration, as inference is not relevant for the Scottish hill data.

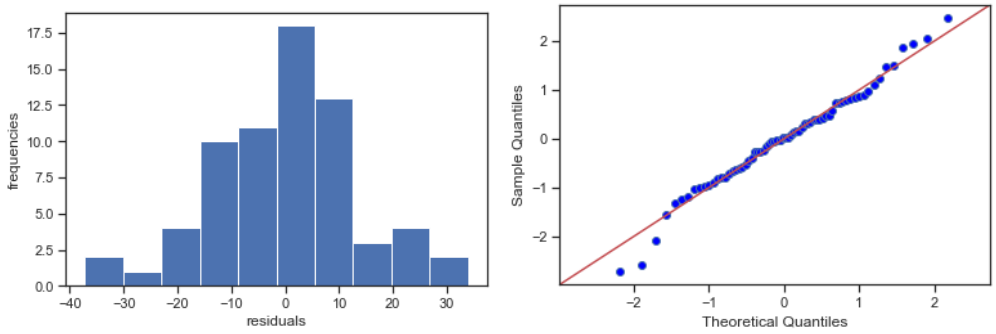


FIGURE B6.2: Histogram and QQ-plot of the residuals for the linear model for the complete Scottish hill races data with explanatory variables distance and climb.

The other plots we consider *are* relevant, relating to the adequacy of the linear model itself. The following code constructs plots of the residuals against the observation index number and against the fitted values:

```
In [22]: index =  list(range(1, len(residuals) + 1))
In [23]: plt.scatter(index, residuals)    # residuals vs. observation index
    ...: plt.title(' ')
    ...: plt.xlabel('Index'); plt.ylabel('Residuals')
In [24]: plt.scatter(fitted, residuals)   # residuals vs. fitted values
    ...: plt.title(' ')
    ...: plt.xlabel('Fitted values'); plt.ylabel('Residuals')
```

Figure B6.3 shows these two residual plots. The plot of the residuals against the index does not show any extreme values. The plot of the residuals against the fitted values shows the large residual for the observation that has much larger fitted value than the others, which is the outlying observation 41 that is a very long race.

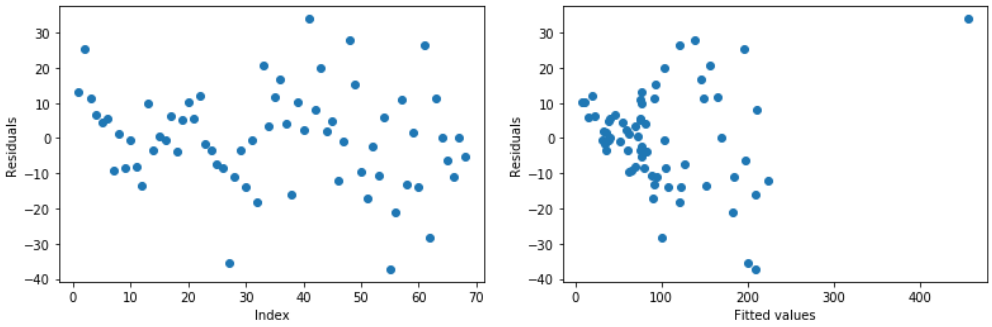


FIGURE B6.3: Plots of residuals against observation index number and against fitted values for linear model for record times with distance and climb explanatory variables, in Scottish hill races.

Residuals plotted against each explanatory variable can highlight possible nonlinearity in an effect or severely nonconstant variance. A *partial regression plot* displays the relationship between a response variable and an explanatory variable after removing the effects of the other explanatory variables that are in the model. It does this by plotting the residuals from models using these two variables as responses and the other explanatory variable(s) as predictors. The least squares slope for the points in this plot is necessarily the same as the estimated partial slope for the multiple regression model. A further diagnostic plot is that of the *partial residuals*, also known as Component-Component plus Residual (CCPR) plot. This plot, for a specific explanatory variable, say X_1 =*distance*, plots the residuals plus the linear estimated effect of X_1 against X_1. It shows the relationship between X_1 and the response variable accounting for the remaining explanatory variables in the model. Partial residual plots should be used with caution, since in case X_1 is highly correlated with any of the other independent variables, then the variance shown in the partial residual plot is underestimated. The following code plots for each explanatory variable (i) observed and fitted response values against the explanatory variable, including prediction intervals, (ii) residuals against the explanatory variable, (iii) partial regression plots, and (iv) the CCPR plot:

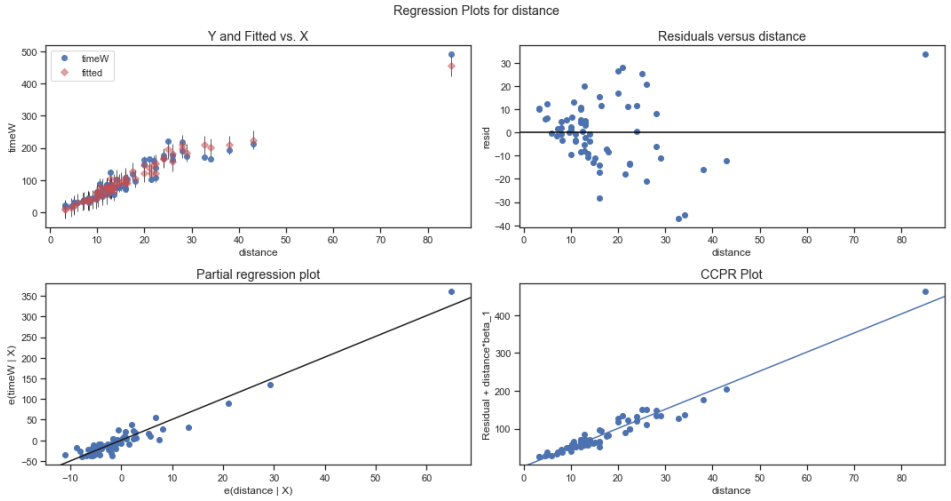
```
# diagnostic plots for each explanatory variable (see Figure B6.4):
In [25]: sm.graphics.plot_regress_exog(fitdc, 'distance', fig=plt.figure(figsize=(15, 8)))
In [26]: fig= sm.graphics.plot_regress_exog(fitdc,'climb',fig=plt.figure(figsize=(15, 8)))
```

The derived residuals against explanatory variables plots (see upper right plots in Figure B6.4 (a) and (b)) reveal that the residuals tend to be small in absolute values at low values of distance and climb, suggesting (not surprisingly) that *timeW* tends to vary less at those low values. The partial regression plots, shown for *distance* and *climb* in the lower left plots in Figure B6.4 (a) and (b), suggest that the partial effects of distance and climb are approximately linear and positive.

Solely he partial regression plots for each explanatory variable can be constructed as follows:

```
# partial regression plot for each explanatory variable, adjusting for other:
In [27]: fig_dis = sm.graphics.plot_partregress('timeW','distance', ['climb'],
                   data = Races, obs_labels = False)
In [28]:fig_climb = sm.graphics.plot_partregress('timeW','climb', ['distance'],
                   data = Races, obs_labels = False)
```

(a)



(b)

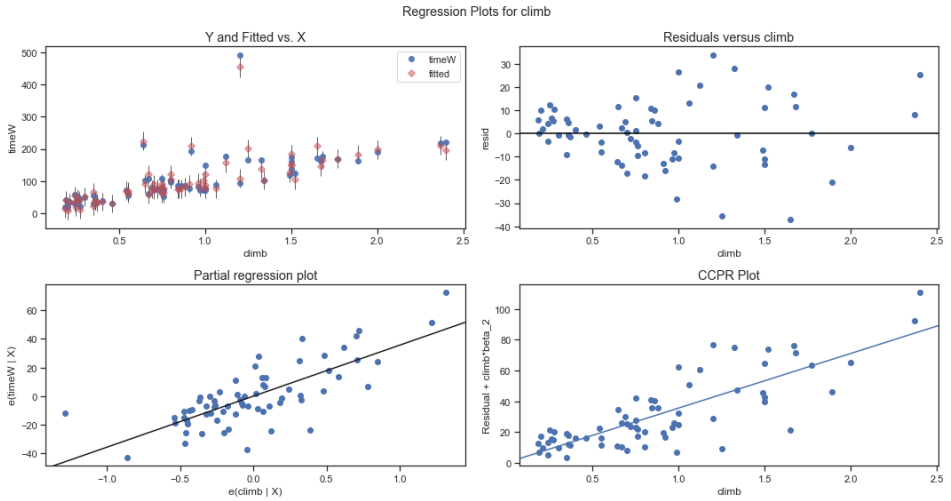


FIGURE B6.4: Residual diagnostic plots against each explanatory variable for the linear model for the complete Scottish hill races data with explanatory variables distance and climb.

We next repeat with `Python` the analysis performed with `R` in Section 6.2.8 to use Cook's distances to detect potentially influential observations. Cook's distance is large when an observation has a large residual and a large leverage. The following code requests a plot of squared normalized residuals against the leverage:

```
In [29]: sm.graphics.plot_leverage_resid2(fitdc)
```

Figure B6.5 shows the plot. We've seen in Figure B6.3 that observation 41 (index 40 in `Python`) has a large residual, and Figure B6.5 shows it also has a large leverage, highlighting it as potentially problematic.

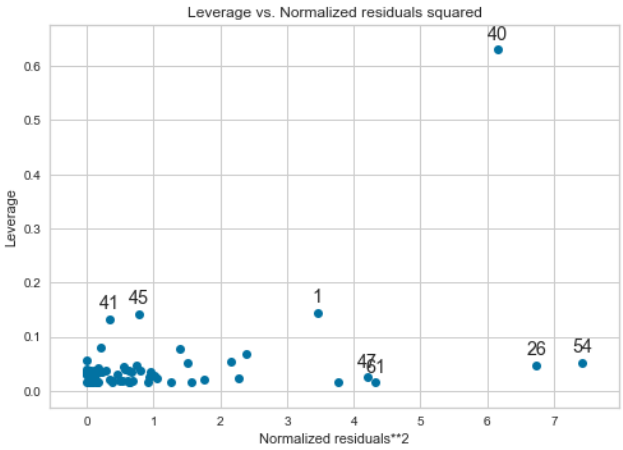


FIGURE B6.5: Plot of leverage against squared normalized residuals for linear model for Scottish hill races.

Diagnostics for influential results can be saved in `fitdc.get_influence`, as shown in the following code. The function `OLSInfluence` of `statsmodels.stats.outliers_influence` [2], which has to be imported, also provides them. To plot Cook's distances, we install the `yellowbrick` library (in the console window). To create the plot, you need to define two new data frames, one containing the explanatory variables in the model and the other containing the response variable. This plot, shown in Figure B6.6(left), and the following code detect the extremely large Cook's distance for observation 41 (race: Highland Fling):

```
In [30]: pip install yellowbrick
In [31]: influence = fitdc.get_influence()
    ...: leverage = influence.hat_matrix_diag   # hat values
    ...: cooks_d = influence.cooks_distance     # Cook's distances
In [32]: cooks_df = pd.DataFrame(cooks_d,index=['CooksDist','p-value'])
    ...: cooks_df = pd.DataFrame.transpose(cooks_df)
    ...: cooks_df.head(3)
Out[32]:
   CooksDist   p-value
0   0.007997  0.999008
1   0.216293  0.884759
2   0.004241  0.999615
In [33]: max(cooks_df. CooksDist)
Out[33]: 9.068276652414221
In [34]: cooks_df.CooksDist.idxmax(axis = 0)
Out[34]: 40 # case 41 has maximum Cook's Distance (index starts at 0)
In [35]: from yellowbrick.regressor import CooksDistance
    ...: X = Races.drop(['race', 'timeM', 'timeW'], axis=1)
    ...: y = Races['timeW']                     # y only has response variable
    ...: visualizer = CooksDistance()           # plot Cook's distances
    ...: visualizer.fit(X, y)                   # Figure B6.6(left)
```

Next follows the code for plotting the Cook's distances for the model fitted on the data set excluding observation 41, shown in Figure B6.6(right):

```
In [36]: X1 = X.loc[X.index != 40]              # 41st case has index=40
```

[2] You may find information of the available options in https://www.statsmodels.org/stable/generated/statsmodels.stats.outliers_influence.OLSInfluence.html.

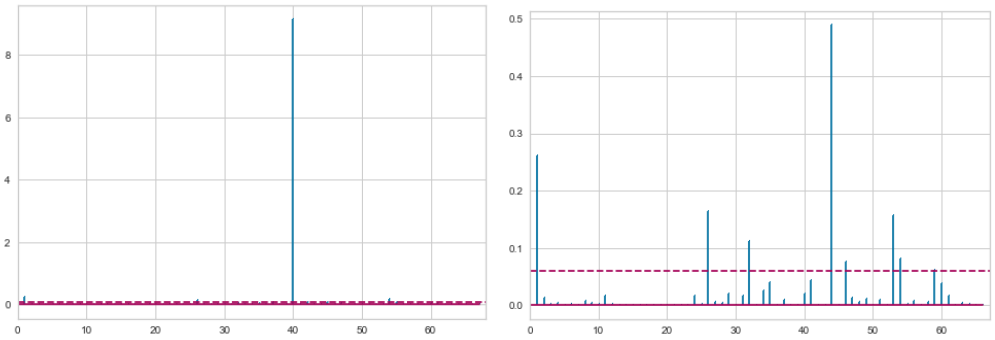


FIGURE B6.6: Plot of the Cook's distances for the linear model fitted on the Scottish hill races data with explanatory variables distance and climb. The plot on the left corresponds to the model fitted on the complete data while that on the right excluding observation 41.

```
   ...: y1 = y.loc[y.index != 40]
   ...: visualizer = CooksDistance()          # plot Cook's distances without
   ...: visualizer.fit(X1, y1)                # influential case (not shown)
```

Notice that the vertical scales on the two plots in Figure B6.6 are very different, thus the highs of their bars are not comparable.

B6.4 Statistical Inference and Prediction for Linear Models

To illustrate statistical inference for linear models, Section 6.4.2 used a study of mental impairment, with life events and SES as explanatory variables. Here are inferential analyses for those data using `Python`, showing the global F test and individual t tests and CIs:

```
In [1]: import pandas as pd
   ...: Mental = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Mental.dat', sep='\s+')
In [2]: Mental.head(1)    # 1st case in the sample
Out[2]:
   impair  life  ses
0      17    46   84
In [3]: Mental.tail(1)    # last (40th) case
Out[3]:
39      41    89   75
In [4]: Mental.corr()     # correlation matrix
Out[4]:
          impair      life       ses
impair  1.000000  0.372221 -0.398568
life    0.372221  1.000000  0.123337
ses    -0.398568  0.123337  1.000000
In [5]: import statsmodels.formula.api as smf
   ...: fit = smf.ols(formula="impair ~ life + ses", data=Mental).fit()
   ...: print(fit.summary())              # showing some of the output
==============================================================================
DF Residuals: 37    F-statistic:           9.495  # F statistic tests
Df Model:      2    Prob (F-statistic): 0.000470  # H_0: beta1=beta2=0
==============================================================================
              coef  std err          t    P>|t|      [0.025    0.975]
------------------------------------------------------------------------------
```

```
Intercept  28.2298    2.174     12.984    0.000      23.824    32.635
life        0.1033    0.032      3.177    0.003       0.037     0.169
ses        -0.0975    0.029     -3.351    0.002      -0.156    -0.039
```

We next find a CI for the regression line value $E(Y)$, a predicted point estimation and prediction interval (PI) for a new observation Y_0, at fixed values of the explanatory variables. At their mean values (44.42 for *life* and 56.6 for *ses*), we find the 95% CI of (25.84, 28.76) for the mean mental impairment and 95% prediction interval of (17.95, 36.65) for a new observation Y_0. The following code is analogous to the `R` code in Section 6.4.5:

```
In [6]: newdata = pd.DataFrame({'life':[44.42], 'ses':[56.60]})
In [7]: predictions = fit.get_prediction(newdata)
   ...: pd.options.display.width = 0    # detection and adjustment of the display size
        ...: pd.set_option('display.max_columns', 7) # number of columns to be displayed in the output
   ...: predictions.summary_frame(alpha=0.05)
Out[7]:
   mean      mean_se   mean_ci_lower mean_ci_upper obs_ci_lower  obs_ci_upper
0  27.299484 0.720436  25.839742     28.759226     17.952574     36.646394
```

If one asks for the predictions summary above without the `pd.options.display.width` command, then not all columns of the output are visible (the inner columns are omitted). Alternatively, one can control the number of columns to be printed, e.g., set equal to 10, by the command `pd.set_option('display.max_columns', 10)`.

The same results can be obtained by the function given below, which provides the prediction, $(1-\alpha)$% CI and $(1-\alpha)$% PI at a given data point, based on the formulas in Section 6.4.2. Notice that this function works for a single data point while the `predictions.summary_frame` above applies also for data sets with more than one data point.

```
In [8]: import numpy as np
   ...: from scipy.stats import t
   ...:
   ...: def regr_intervals(newdata, fit, alpha = 0.05):
   ...:         # input: newdata (a data point)
        ...:         #        fit (the results of a model fit)
        ...:         # output: prediction, (1-alpha) CI, PI
   ...:         y_pred = np.array(fit.predict(newdata))
   ...:         residuals = fit.resid
   ...:         n= len(residuals)
   ...:         dof = n-len(fit.params)
   ...:         res_se = np.std(residuals)*np.sqrt(n/dof)   # s
   ...:         mean_se = res_se*np.sqrt(1/n)
   ...:         mean_se_pred = res_se*np.sqrt(1+1/n)
   ...:         qt = t.ppf(1-alpha / 2, dof)                # t quantile
   ...:         CI = y_pred + np.array([-1, 1]) * qt * mean_se
   ...:         PI = y_pred + np.array([-1, 1]) * qt * mean_se_pred
   ...:         conf= 1-alpha
   ...:         return y_pred, CI, PI, conf
   ...:
# Application:
In [9]: predict, CI, PI, conf = regr_intervals(newdata,fit)  # calls the function
   ...: print('prediction =', predict)
   ...: print(conf, 'CI:', CI)
   ...: print(conf, 'PI:', PI)
prediction = [27.2994837]
0.95 CI: [25.8397416  28.75922581]
0.95 PI: [17.95257365 36.64639375]
```

CIs and PIs can be calculated on the values of the sample on which the model estimation is based by the `summary_table` of the `statsmodels.stats.outliers_influence` function

as shown below. The `data` of the output of this function contains the fitted values (column 2), the lower and upper CI limits (columns 4,5) and the lower and upper PI limits (columns 6,7).

```
In [9]: from statsmodels.stats.outliers_influence import summary_table
   ...: # summary_table returns:
   ...: # st: simple table, data: raw data of table, ss2: labels of tables' columns
   ...: st, data, ss2 = summary_table(fit, alpha=0.05)
   ...: fittedvalues = data[:, 2]
   ...: predict_mean_se  = data[:, 3]
   ...: predict_mean_ci_low, predict_mean_ci_up = data[:, 4:6].T  # CI
   ...: predict_ci_low, predict_ci_up = data[:, 6:8].T            # PI
```

Next we construct a plot (see Figure B6.7) where the scatterplot of the data is shown (in terms of a specific explanatory variable) along with the CI and PI at each data point.

```
In [10]: fig, ax = plt.subplots()
    ...: plt.plot(x, y, 'o', markersize=5)
    ...: n=len(fittedvalues)
    ...: for i in range (1,n+1):
    ...:   plt.plot((x[i-1],x[i-1]), (predict_ci_low[i-1],predict_ci_up[i-1]),
                              '--', color='r', linewidth=1)
    ...:   plt.plot((x[i-1],x[i-1]), (predict_mean_ci_low[i-1],
                             predict_mean_ci_up[i-1]), color='b', linewidth=1)
    ...: plt.ylim(predict_ci_low.min()-1,predict_ci_up.max()+1)
    ...: plt.xlabel("ses")
    ...: plt.ylabel("impair")
    ...: plt.show()
```

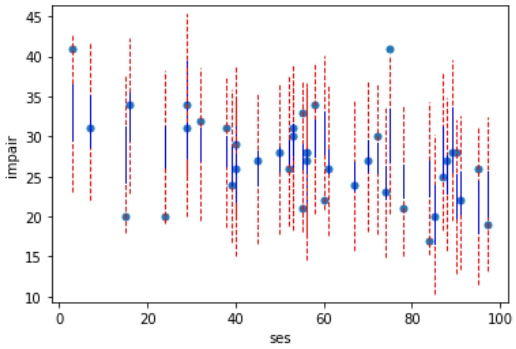


FIGURE B6.7: Scatterplot for impair and ses, for the mental impairment data, with the CI (solid blue) and PI (dashed red) at each data point.

B6.5 Categorical Explanatory Variables in Linear Models

The analysis of variance (ANOVA) model in Section 6.5.2, for comparing the mean incomes among three racial-ethnic groups is implemented here in `Python`. Denote that in `statsmodels` a categorical explanatory variable is transformed to a factor by `C()`:

```
In [1]: import pandas as pd
   ...: import statsmodels.formula.api as smf
   ...: Income = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Income.dat', sep='\s+')
In [2]: Income.head(1)
Out[2]:
   income  education race                  # race is categorical explanatory variable
0      16         10    B
In [3]: fit = smf.ols(formula = 'income ~ C(race)', data=Income).fit()
   ...: print(fit.summary())               # showing some output; F stat. tests
 F-statistic: 4.244    Prob (F-statistic): 0.0178            # H_0: equal means
==============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      27.7500      4.968      5.586      0.000      17.857      37.643
C(race)[T.H]    3.2500      7.273      0.447      0.656     -11.232      17.732
C(race)[T.W]   14.7300      5.708      2.581      0.012       3.364      26.096
==============================================================================
```

B6.5.1 Multiple Comparisons of Means: Bonferroni and Tukey Methods

For the Income example above, the multiple comparisons of the means discussed in Section 6.5.4 are given below:

```
In [4]: import statsmodels.api as sm
   ...: aov_table = sm.stats.anova_lm(fit)               # ANOVA table
   ...: aov_table
Out[4]:
            df    sum_sq      mean_sq         F   PR(>F)
C(race)    2.0   3352.47  1676.235000  4.244403  0.01784
Residual  77.0  30409.48   394.928312       NaN      NaN

In [5]: import statsmodels.stats.multicomp as mc
   ...: comp = mc.MultiComparison(Inc['inc'], Inc['race'])
   ...: post_hoc_res = comp.tukeyhsd()
   ...: print(post_hoc_res.summary())
 Multiple Comparison of Means - Tukey HSD, FWER=0.05
====================================================
group1 group2 meandiff p-adj   lower    upper  reject
----------------------------------------------------
     B      H     3.25 0.8882 -14.1311 20.6311  False
     B      W    14.73 0.0312   1.0884 28.3716   True
     H      W    11.48 0.1426  -2.8809 25.8409  False
----------------------------------------------------
# Plot Tukey intervals (Figure B6.8):
In [6]: post_hoc_res.plot_simultaneous(ylabel= "race", xlabel= "mean income")
In [7]: import statsmodels.stats.api as sms
   ...: bonf, a1, a2 = comp.allpairtest(sms.ttest_ind, method= "bonf")
   ...: print(bonf)                    # Bonferroni
Test Multiple Comparison ttest_ind
FWER=0.05 method=bonf
alphacSidak=0.02, alphacBonf=0.017
================================================
group1 group2   stat   pval  pval_corr reject
------------------------------------------------
     B      H -0.6796 0.5023       1.0  False
     B      W -2.4398 0.0175    0.0524  False
     H      W -1.7942 0.0777     0.233  False
================================================
```

The *reject* column of the multiple comparisons table indicates whether one would reject

the pair of groups having identical means, so based on the Tukey HSD comparisons, we can conclude only that black and white racial-ethnic groups have differing population mean annual incomes. The graphical display of the Tukey HSD comparisons is shown in Figure B6.8.

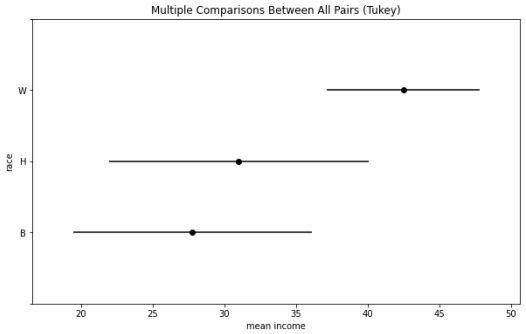


FIGURE B6.8: Tukey intervals for the means by race for the income data.

B6.5.2 Models with Categorical and Quantitative Explanatory Variables

We expand the model fitted in Section B6.5 on the income data by adding the quantitative explanatory variable education (educ), following the analysis of Section 6.5.5.

```
In [8]: fit2 = smf.ols(formula="inc ~ C(race) + educ", data=Inc).fit()
   ...: print(fit2.summary())                              # part of the output
F-statistic:       21.75                  Prob (F-statistic):   2.85e-10
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      -26.5379      8.512     -3.118      0.003     -43.492      -9.584
C(race)[T.H]     5.9407      5.670      1.048      0.298      -5.352      17.234
C(race)[T.W]    10.8744      4.473      2.431      0.017       1.966      19.783
educ             4.4317      0.619      7.158      0.000       3.199       5.665
==============================================================================
In [9]: sm.stats.anova_lm(fit2)
Out[9]:
            df        sum_sq       mean_sq         F        PR(>F)
C(race)    2.0   3352.470000   1676.235000   7.01344  1.602408e-03
educ       1.0  12245.231928  12245.231928  51.23458  4.422192e-10
Residual  76.0  18164.248072    239.003264       NaN           NaN

In [10]: sm.stats.anova_lm(fit2, typ=2)                 # Type II sums of squares
Out[10]:
               sum_sq    df          F        PR(>F)
C(race)    1460.583947   2.0   3.055573  5.292198e-02   # F test for each variable,
educ      12245.231928   1.0  51.234580  4.422192e-10   # adjusted for other variable
Residual  18164.248072  76.0        NaN           NaN
```

The bottom of the output shows tests for the effect of each variable, adjusted for the other one. For example, the test that racial-ethnic status has no effect on mean annual income, adjusting for years of experience, has a test statistic of $F = 3.06$ and a P-value of 0.053.

B6.5.3 Interaction with Categorical and Quantitative Explanatory Variables

Continuing the analysis of the income data, the results presented in Section 6.5.7 for the model with interaction between race and education are derived here in `Python`.

```
In [11]: fit3 = smf.ols(formula="inc ~ C(race) + educ + C(race):educ", data=Inc).fit()
    ...: print(fit3.summary())                          # part of the output
F-statistic:       13.80                  Prob (F-statistic):   1.62e-09
==============================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------------------
Intercept            -6.5355     14.980     -0.436      0.664     -36.385      23.314
C(race)[T.H]        -10.0692     26.527     -0.380      0.705     -62.926      42.788
C(race)[T.W]        -19.3333     18.293     -1.057      0.294     -55.782      17.116
educ                  2.7988      1.182      2.368      0.021       0.444       5.154
C(race)[T.H]:educ     1.2899      2.193      0.588      0.558      -3.079       5.659
C(race)[T.W]:educ     2.4107      1.418      1.700      0.093      -0.414       5.236
==============================================================================================
In [12]: sm.stats.anova_lm(fit3, typ=2)
Out[12]:
                     sum_sq    df          F        PR(>F)
C(race)         1460.583947   2.0   3.092968  5.127908e-02
educ           12245.231928   1.0  51.861597  4.131357e-10
C(race):educ     691.836568   2.0   1.465050  2.376899e-01
Residual       17472.411504  74.0        NaN           NaN
```

B6.6 Bayesian Fitting of Linear Models

For the mental impairment data, we proceed to a Bayesian analysis, implementing the approach described in Section 6.6.2. The Bayesian analysis is carried out in the `pymc3` package, which needs to be imported first. The code shown next uses two MCMC chains of length 100,000 each, for which the MCMC approximation to the true fit seems to be reasonably good:

```
In [1]: import pandas as pd
   ...: Mental = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Mental.dat', sep='\s+')
%matplotlib inline   # in the IPython console
In [3]: import matplotlib.pyplot as plt
   ...: from pymc3 import  *
         ...: import pymc3
   ...: import statsmodels.api as sm
   ...: from pandas.plotting import scatter_matrix

In [3]: y=Mental.impair
   ...: life=Mental.life
   ...: ses=Mental.ses
In [4]: B0=10**(-20); C0=10**(-10)
# model specifications in PyMC3 are wrapped in a with-statement:
   ...: with Model() as model:
   ...:     # Define priors
   ...:     sigma = InverseGamma ('sigma', C0, C0, testval=1.)
   ...:     Intercept = Normal('Intercept', 0, sigma=1/B0)
   ...:     beta1 = Normal('beta1', 0, sigma=1/B0)
   ...:     beta2 = Normal('beta2', 0, sigma=1/B0)
   ...:     # Define the likelihood function
   ...:     likelihood = Normal('y', mu=Intercept + beta1 * life + beta2 * ses,
```

```
   ...:                              sigma=sigma, observed=Mental.impair)
   ...:    # Inference!
   ...:    trace = sample(100000, cores=2)        # 2x100000 posterior samples
Sampling 2 chains for 1_000 tune and 100_000 draw iterations (2_000 + 200_000
draws total) took 4418 seconds
In [5]: scatter_matrix(trace_to_dataframe(trace),figsize=(12,12))    # produces Figure B6.3
In [6]: summary(trace)
Out[6]:
             mean     sd  hdi_2.5%  hdi_97.5%
Intercept  28.222  2.250  24.025   32.509
beta1       0.103  0.034   0.040    0.167
beta2      -0.097  0.030  -0.154   -0.040
sigma       4.653  0.561   3.659    5.715
In [7]: pm.stats.hpd(trace['Intercept'], alpha=0.05) # 95% HPD Intervals
Out[7]: array([24.02491087, 32.50900145])
In [8]: pm.stats.hpd(trace['beta1'], alpha=0.05)
Out[8]: array([0.04017495, 0.16693835])
In [9]: pm.stats.hpd(trace['beta2'], alpha=0.05)
Out[9]: array([-0.15374007, -0.04024125])
In [10]: pm.stats.hpd(trace['sigma'], alpha=0.05)
Out[10]: array([3.65905699, 5.71531038])
In [11]: np.median(trace['Intercept'])               # median
Out[11]: 28.22698834083754
In [12]: np.median(trace['beta1'])
Out[12]: 0.10334276150835833
In [13]: np.median(trace['beta2'])
Out[13]: -0.0973951399724454
In [14]: sum(trace['beta1']<0)/200000   # analog of one-sided P-value
Out[14]: 0.001405                       # for H_a: life events beta > 0
In [15]: sum(trace['beta2']>0)/200000   # analog of one-sided P-value
Out[15]: 0.000995                       # for H_a: SES beta < 0
```

Figure B6.2 portrays the posterior distributions of the model parameters. Results are similar to the Bayesian results obtained with `R` and similar to the classical results in Section 6.4.2, which is implemented in `Python` as follows:

```
In [2]: import statsmodels.formula.api as smf   # standard frequentis analysis
   ...: fit = smf.ols(formula="impair ~ life + ses", data=Mental).fit()
   ...: print(fit.summary())  # part of the output
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     28.2298      2.174     12.984      0.000      23.824      32.635
life           0.1033      0.032      3.177      0.003       0.037       0.169
ses           -0.0975      0.029     -3.351      0.002      -0.156      -0.039
==============================================================================
```

Final Remark

In this chapter we have fitted linear regression models in `statsmodels`. For this, we called either `statsmodels.formula.api` or `statsmodels.api`. The former accepts `formula` and `df` (pandas data frames) arguments in a manner similar to `R` while the latter only takes `endog` (endogenous, i.e. response) variables and design matrices (`exog`: exogenous for independent variables). In order to view a list with the names of the available models under `statsmodels.formula.api`, type `dir(smf)`.

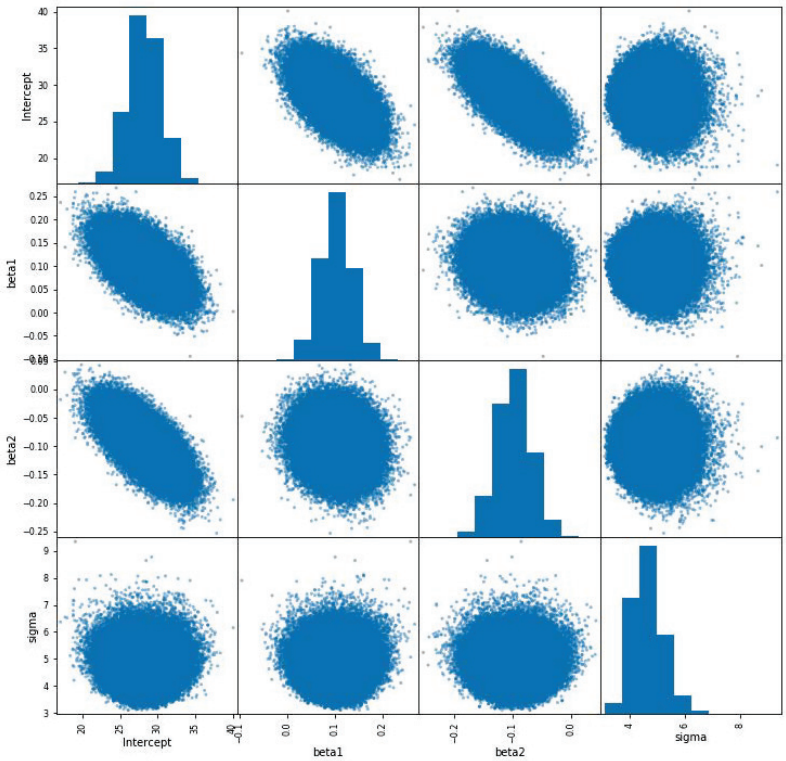


FIGURE B6.9: Posterior distributions of the parameters for the model fitted on the Mental impairment data.

7

CHAPTER 7: PYTHON FOR GENERALIZED LINEAR MODELS

Linear models are special cases of GLMs, so models fitted to the Scottish hill races data in Section B6.1 by least squares using `smf.ols` can equivalently be fitted as GLMs:

```
In [1]: import pandas as pd
   ...: Races=pd.read_csv('http://stat4ds.rwth-aachen.de/data/ScotsRaces.dat',sep='\s+')
In [2]: import statsmodels.formula.api as smf
In [3]: fitdc=smf.ols(formula='timeW ~ distance + climb', data=Races).fit()
   ...: print(fitdc.summary())    # edited output of least squares fit
==============================================================================
                 coef  std err        t   P>|t|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept  -14.5997    3.468   -4.210   0.000     -21.526    -7.674
distance     5.0362    0.168   29.919   0.000       4.700     5.372
climb       35.5610    3.700    9.610   0.000      28.171    42.951
------------------------------------------------------------------------------
In [4]: fitdc.glm = smf.glm(formula='timeW ~ distance + climb', data=Races).fit()
    ...: print(fitdc.glm.summary())         # edited output of GLM fit
==============================================================================
                 coef  std err        z    P>|z|     [0.025   0.975]
------------------------------------------------------------------------------
Intercept   -14.5997   3.468   -4.210    0.000    -21.397   -7.802
distance      5.0362   0.168   29.919    0.000      4.706    5.366
climb        35.5610   3.700    9.610    0.000     28.309   42.813
==============================================================================
```

The default for `smf.glm` is the Gaussian (normal) distribution family with identity link function. The parameter estimates are identical to the least squares fit, but inference about individual coefficients differs slightly because the *glm* function uses normal distributions for the sampling distributions (regardless of the assumed distribution for Y), whereas the *ols* function uses the t distribution, which applies only with normal responses.

B7.1 GLMs with Identity Link

Section 7.1.3 used GLMs with identity link function to model house selling prices. The scatterplot of the selling prices by size of home and whether it is new, such as shown in Figure 7.1, can be obtained in `Python` as follows:

```
In [1]: import pandas as pd
   ...: import matplotlib.pyplot as plt
In [2]: Houses = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Houses.dat', sep='\s+')
In [3]: import seaborn as sns
   ...: Houses['house'] = Houses['new'].apply(lambda x: 'old' if x==0 else 'new')
   ...: sns.pairplot(x_vars=['size'], y_vars=['price'], data=Houses, hue='house', size=5)
```

We next fit the GLMs with identity link function discussed in Section 7.1.3, assuming normal (for which the identity link is the default) or gamma distributions for house selling price:

```
In [4]: import statsmodels.formula.api as smf
   ...: import statsmodels.api as sm
# GLM assuming normal response, identity link, permitting interaction
In [5]: fit1 = smf.glm(formula = 'price ~ size + new + size:new', data = Houses,
                       family = sm.families.Gaussian()).fit()
   ...: print(fit1.summary())                          # edited output
                 Generalized Linear Model Regression Results
==============================================================================
               coef  std err          z    P>|z|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept  -33.3417   23.282    -1.432    0.152     -78.973    12.290
size         0.1567    0.014    11.082    0.000       0.129     0.184
new       -117.7913   76.511    -1.540    0.124    -267.751    32.168
size:new     0.0929    0.033     2.855    0.004       0.029     0.157
==============================================================================
# GLM assuming gamma response, identity link, permitting interaction:
In [6]: gamma_mod = smf.glm(formula = 'price ~ size + new + size:new', data = Houses,
                       family = sm.families.Gamma(link = sm.families.links.identity))
   ...: fit2 = gamma_mod.fit()
   ...: print(fit2.summary())                          # edited output
                 Generalized Linear Model Regression Results
==============================================================================
               coef  std err          z    P>|z|      [0.025   0.975]
------------------------------------------------------------------------------
Intercept  -11.1764   19.461    -0.574    0.566     -49.320   26.967
size         0.1417    0.015     9.396    0.000       0.112    0.171
new       -116.8569   96.873    -1.206    0.228    -306.725   73.011
size:new     0.0974    0.055     1.769    0.077      -0.011    0.205
==============================================================================
```

The interaction term is not needed for the gamma GLM, reflecting the greater variability in the response as the mean increases for that GLM.

B7.1.1 Example: Normal and Gamma GLMs for Covid-19 Data

We next conduct the analyses of the Covid-19 data set in Section 7.1.8 using normal and gamma GLMs. We fit a (1) normal linear model for the log counts (i.e., assuming a log-normal distribution for the response),(2) GLM using the log link for a normal response, (3) GLM using the log link for a gamma response:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
In [2]: Covid = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Covid19.dat', sep='\s+')
In [3]: fit1 = smf.ols(formula='np.log(cases) ~ day', data=Covid).fit()
   ...: print(fit1.summary())          # normal linear model for log-counts
==============================================================================
               coef  std err          t    P>|t|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept    2.8439    0.084    33.850    0.000       2.672     3.016
day          0.3088    0.005    67.377    0.000       0.299     0.318
==============================================================================
In [4]: fit2 = smf.glm(formula = 'cases ~ day', family = sm.families.Gaussian
                   (link = sm.families.links.log), data = Covid).fit()
   ...: print(fit2.summary())          # normal GLM with log link
==============================================================================
               coef  std err          z    P>|z|      [0.025    0.975]
```

```
------------------------------------------------------------------------------
Intercept      5.3159    0.168     31.703    0.000       4.987     5.645
day            0.2129    0.006     37.090    0.000       0.202     0.224
==============================================================================
In [5]: fit2.aic
Out[5]: 594.1435463614453
In [6]: fit3 = smf.glm(formula='cases ~ day', family = sm.families.Gamma
                       (link = sm.families.links.log), data = Covid).fit()
   ...: print(fit3.summary())          # gamma GLM with log link
==============================================================================
                 coef   std err          z      P>|z|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept      2.8572    0.077     36.972    0.000       2.706     3.009
day            0.3094    0.004     73.388    0.000       0.301     0.318
==============================================================================
In [7]: fit3.aic
Out[7]: 479.3853756004412              # better fit that normal GLM with log link
```

All three models assume an exponential relationship for the response over time, but results are similar with models (1) and (3) because they both permit the variability of the response to grow with its mean.

B7.2 Logistic Regression: Logit Link with Binary Data

To illustrate logistic regression, Section 7.2.3 models the probability of death for flour beetles after five hours of exposure to various log-dosages of gaseous carbon disulfide (in mg/liter). The response variable is binary with $y = 1$ for death and $y = 0$ for survival. We can fit the model with *grouped* data (Section 7.2.3) or with *ungrouped* data (i.e. using a data file having a separate row for each beetle in the sample; see Section 7.2.4). We first present the ungrouped-data analysis:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
   ...: import matplotlib.pyplot as plt
In [2]: Beetles = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Beetles_ungrouped.dat',
                              sep='\s+')
In [3]: Beetles.head(2)
Out[3]:
       x  y
0  1.691  1
1  1.691  1
In [4]: Beetles.tail(1)
Out[4]:
480  1.884  1         # 481 observations in ungrouped data file
                      # logit link is binomial default with smf.glm
In [5]: fit = smf.glm('y ~ x', family = sm.families.Binomial(), data=Beetles).fit()
   ...: print(fit.summary())                              # edited output
                Generalized Linear Model Regression Results
Deviance:    372.35       Df Residuals:     479
==============================================================================
                 coef   std err          z      P>|z|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept    -60.7401    5.182   -11.722    0.000     -70.896   -50.584
x             34.2859    2.913    11.769    0.000      28.576    39.996
==============================================================================
```

The following code shows how to plot the proportion of dead beetles versus the log dosage of gaseous carbon disulfide, showing also the fit of the logistic regression model:

```
In [6]: logdose = Beetles.x.unique()    # vector of unique values of x
   ...: yx=pd.crosstab(Beetles['y'],Beetles['x'], normalize='columns')
   ...: y_prop=yx.iloc[1]               # vector of sample  proportions of y=1
   ...: def f(t):
   ...:     return np.exp(fit.params[0] + fit.params[1]*t)/
                      (1 + np.exp(fit.params[0] + fit.params[1]*t))
   ...: t1 = np.arange(1.65, 1.95, 0.0001)
   ...: fig, ax = plt.subplots()
   ...: ax.plot(t1, f(t1),'blue')
   ...: ax.scatter(logdose, y_prop, s=5, color='red')
   ...: ax.set(xlabel='x', ylabel='P(Y=1)')
        ...: plt.show()
```

The plot itself is shown in Figure B6.1 (compare to Figure 7.5 of the book).

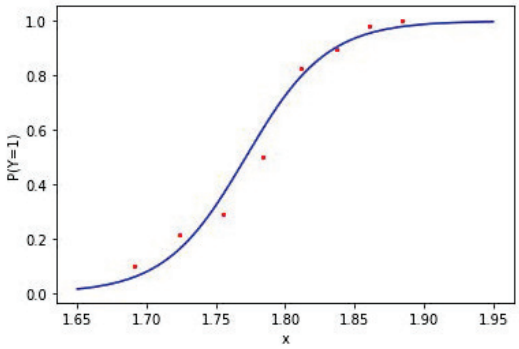


FIGURE B6.1: Proportion of dead beetles versus log dosage of gaseous carbon disulfide, with fit of logistic regression model.

The code for predicting the probability of death for new values of x (log-dosages), say 1.7 and 1.8, is given below:

```
In [7]: x_new = pd.DataFrame({'x': [1.7,1.8]})
   ...: y_pred = fit.predict(x_new); print(y_pred)
0    0.079143
1    0.726023
```

Next we fit the logistic regression model to the grouped-data file for the beetles:

```
In [1]: import pandas as pd
   ...: from statsmodels.formula.api import glm
   ...: import statsmodels.api as sm
In [2]: Beetles2=pd.read_csv('http://stat4ds.rwth-aachen.de/data/Beetles.dat',sep='\s+')
In [3]: Beetles2
Out[3]:
   logdose  live  dead   n
0    1.691    53     6  59
1    1.724    47    13  60
2    1.755    44    18  62
3    1.784    28    28  56
4    1.811    11    52  63
5    1.837     6    53  59
6    1.861     1    61  62
7    1.884     0    60  60      # 8 observations in grouped data file
```

```
In [4]: fit = glm('dead + live ~ logdose', data = Beetles2,
                  family = sm.families.Binomial()).fit()
In [5]: print(fit.summary())    # same results as with ungrouped data
                 Generalized Linear Model Regression Results
Deviance:   11.116       Df Residuals:    6
==========================================================================
               coef  std err         z   P>|z|     [0.025   0.975]
--------------------------------------------------------------------------
Intercept  -60.7401    5.182   -11.722   0.000    -70.896  -50.584
logdose     34.2859    2.913    11.769   0.000     28.576   39.996
==========================================================================
```

The deviance differs, since now the data file has 8 observations instead of 481, but the ML estimates and standard errors are identical.

Sections B7.3 and B8.1 show other examples of fitting logistic regression models using `Python`.

B7.3 Separation and Bayesian Fitting in Logistic Regression

When the explanatory variable values satisfy *complete separation*, the *glm* function of `statsmodels` reports a separation error and does not provide results. We illustrate for the toy example of Section 7.2.6:

```
In [1]: import pandas as pd
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
In [2]: dat = pd.DataFrame({'x' : [1,2,3,4,5,6], 'y' : [0,0,0,1,1,1]})
   ...: fit = smf.glm('y ~ x', family=sm.families.Binomial(), data=dat).fit()
   ...: print(fit.summary())
PerfectSeparationError: Perfect separation detected, results not available
```

With *quasi-complete separation*, results are reported, but truly infinite estimates have enormous standard errors. We illustrate with the endometrial cancer example from Section 7.3.2, for which the ML estimate of the NV effect is truly infinite. First of all we verify in `Python` that a quasi-complete separation is present. Next, before fitting the logistic regression model, we standardize first the quantitative explanatory variables (PI and EH), so we can compare the magnitudes of their estimated effects (which are truly finite). For this, we using the `preprocessing` object of the `sklearn` library:

```
In [1]: import pandas as pd
In [2]: Endo =pd.read_csv('http://stat4ds.rwth-aachen.de/data/Endometrial.dat',sep='\s+')
In [3]: pd.crosstab(Endo.NV, Endo.HG)
Out[3]:
HG   0   1
NV
0   49  17        # quasi-complete separation: no 'HG=0 and NV=1' cases
1    0  13
In [4]: from sklearn import preprocessing      # standardize PI and EH:
  ...: Endo['PI2'] = preprocessing.scale(Endo.PI)
  ...: Endo['EH2'] = preprocessing.scale(Endo.EH)
In [5]: Endo['NV2'] = Endo['NV'] - 0.5     # centers NV around 0
In [6]: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
   ...: fit = smf.glm('HG ~ NV2 + PI2 + EH2',
                      family = sm.families.Binomial(), data = Endo).fit()
   ...: print(fit.summary())      # true ML estimate for NV is infinite
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      9.8411   6338.889      0.002      0.999   -1.24e+04    1.24e+04
NV2           22.1856   1.27e+04      0.002      0.999   -2.48e+04    2.49e+04
PI2           -0.4191      0.440     -0.952      0.341      -1.282       0.444
EH2           -1.9097      0.556     -3.433      0.001      -3.000      -0.819
==============================================================================
```

Python does not yet seem to have capability of Firth's penalized likelihood method (Section 7.7.1) or profile-likelihood CIs for the logistic regression model. However, Bayesian analysis is possible, which is especially useful when some ML estimates are infinite. For this, we use the function for GLMs of pymc3, which is simpler to apply and has code similar to standard GLM code. We illustrate for the endometrial cancer example,[1] using highly disperse normal prior distributions for the model parameters with $\mu = 0$ and $\sigma = 10$:

```
In [7]: import pymc3 as pm
   ...: from pymc3 import  *
   ...: import numpy as np
In [7]: priors = {'Intercept': pm.Normal.dist(mu = 0, sd = 10),
   ...:           'Regressor': pm.Normal.dist(mu = 0, sd = 10)
   ...:           }
   ...: with pm.Model() as fit:
   ...:     pm.glm.GLM.from_formula('HG ~ NV2 + PI2 + EH2',
               Endo, family = pm.glm.families.Binomial(), priors = priors)
   ...:     trace_fit = pm.sample(10000)
In [8]: summary(trace_fit)
Out[8]:
            mean     sd  hdi_2.5%  hdi_97.5%
Intercept  3.219  2.529  -0.649    8.149
NV2        9.123  5.036   1.270   18.618    # compare to infinite ML estimate
PI2       -0.475  0.453  -1.349    0.335
EH2       -2.131  0.594  -3.267   -1.054

In [9]: sum(trace_fit['NV2'] < 0)/20000     # posterior probability of negative NV effect
Out[9]: 0.0002
In [10]: np.median(trace_fit['Intercept'])  # median of the posterior distribution
Out[10]: 2.740539578438713                  # for each model parameter
In [11]: np.median(trace_fit['NV2'])
Out[11]: 8.156850630693388
In [12]: np.median(trace_fit['PI2'])
Out[12]: -0.4566611821769522
In [13]: np.median(trace_fit['EH2'])
Out[13]: -2.0932345181467547
```

Results are similar to those using R in Section 7.3.2. The posterior distributions of the model parameters (see Figure B6.2) can be plotted as shown below:

```
In [14]: import matplotlib.pyplot as plt              # Figure B7.2
    ...: from pandas.plotting import scatter_matrix
    ...: scatter_matrix(trace_to_dataframe(trace_fit), figsize=(12,12))
```

B7.4 Poisson Loglinear Model for Counts

To illustrate the modeling of count data, Section 7.4.2 used Poisson loglinear models for data on female horseshoe crabs, in which the response variable is the number of male satellites

[1] Warning: This is very slow compared with R.

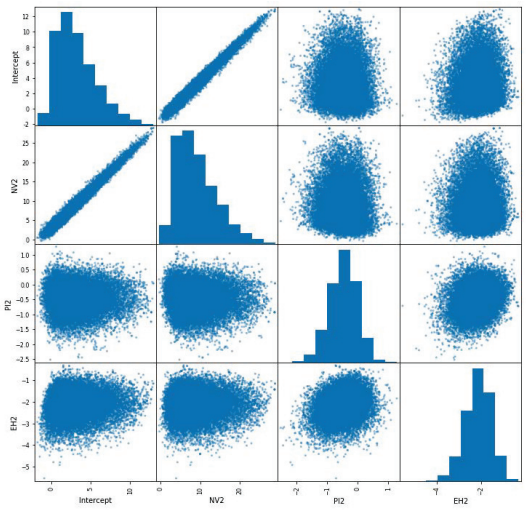


FIGURE B6.2: Posterior distributions of the parameters for the model fitted on the endometrial cancer study data.

during a mating season. Here is `Python` code for the model with explanatory variables weight and color, treating color as a factor:

```
In [1]: import pandas as pd
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
In [2]: Crabs = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Crabs.dat', sep='\s+')
In [3]: fit = smf.glm('sat ~ weight + C(color)',        # default log link
                      family=sm.families.Poisson(), data=Crabs).fit()
   ...: print(fit.summary())                            # edited output
                 Generalized Linear Model Regression Results
Deviance:  551.80     Df Residuals: 168
==============================================================================
                  coef  std err         z   P>|z|     [0.025   0.975]
------------------------------------------------------------------------------
Intercept      -0.0498    0.233    -0.214   0.831     -0.507    0.407
C(color)[T.2]  -0.2051    0.154    -1.334   0.182     -0.506    0.096
C(color)[T.3]  -0.4498    0.176    -2.560   0.010     -0.794   -0.105
C(color)[T.4]  -0.4520    0.208    -2.169   0.030     -0.861   -0.044
weight          0.5462    0.068     8.019   0.000      0.413    0.680
==============================================================================
In [4]: fit.aic              # AIC indicates the fit is poorer than
Out[4]: 917.1026114781453    # negative binomial model in Section B.7.5
```

The output includes the residual deviance and its *df*. We can obtain the null deviance and its *df* by fitting the null model, i.e. the model containing only the intercept (in `R` the null deviance is part of the standard output):

```
In [5]: fit0 = smf.glm('sat ~ 1', family = sm.families.Poisson(), data=Crabs).fit()
In [6]: fit0.deviance, fit0.df_resid
Out[6]: (632.791659200811, 172)
```

`Python` produces an analysis of variance (ANOVA) table only for linear models. For a GLM, we can construct an analogous analysis of deviance table, as shown next for summarizing likelihood-ratio tests for the explanatory variables in the loglinear model for horseshoe crab satellite counts:

```
In [7]: fitW = smf.glm('sat ~ weight', family = sm.families.Poisson(),
                       data=Crabs).fit()             # weight the sole predictor
   ...: fitC = smf.glm('sat ~ C(color)', family=sm.families.Poisson(),
                        data=Crabs).fit()            # color the sole predictor
   ...: D = fit.deviance; D1 = fitW.deviance; D2 = fitC.deviance
In [8]: df = fit.df_resid             # residual degrees of freedom of the models
   ...: dfW = fitW.df_resid
   ...: dfC = fitC.df_resid
In [9]: from scipy import stats       # P-values for likelihood-ratio tests
   ...: P_weight = 1 - stats.chi2.cdf(D2 - D, dfC - df)
   ...: P_color = 1 - stats.chi2.cdf(D1 - D, dfW - df)
In [10]: pd.DataFrame({'Variable': ['weight','C(color)'],
                       'LR Chisq': [round(D2 - D, 3), round(D1 - D, 3)],
                       'df': [dfC - df, dfW - df],
                       'Pr(>Chisq)': [P_weight, P_color]})
   Variable  LR Chisq  df  Pr(>Chisq)
0    weight    57.334   1  3.6748e-14
1  C(color)     9.061   3  2.8485e-02  # color effect P-value = 0.028
```

Recall that the analysis of deviance table of a GLM in R can be obtained by the `Anova` function (see Section 7.4.2).

B7.4.1 Modeling Rates

To model a rate with a Poisson loglinear model, we use an offset. Here is code for an expanded version of the analysis in Section 7.4.3 of lung cancer survival, including histology as an additional prognostic factor:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
In [2]: Cancer = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Cancer2.dat', sep='\s+')
In [3]: Cancer.head(2)
Out[3]:
   time  histology  stage  count  risktime
0     1          1      1      9       157
1     1          2      1      5        77
In [4]: logrisktime = np.log(Cancer.risktime)
   ...: fit = smf.glm('count ~ C(histology) + C(stage) + C(time)',
              family = sm.families.Poisson(), offset = logrisktime, data = Cancer).fit()
   ...: print(fit.summary())                                          # edited output
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -3.0093      0.167    -18.073      0.000      -3.336      -2.683
C(histology)[T.2]     0.1624      0.122      1.332      0.183      -0.077       0.401
C(histology)[T.3]     0.1075      0.147      0.729      0.466      -0.181       0.397
C(stage)[T.2]         0.4700      0.174      2.694      0.007       0.128       0.812
C(stage)[T.3]         1.3243      0.152      8.709      0.000       1.026       1.622
C(time)[T.2]         -0.1275      0.149     -0.855      0.393      -0.420       0.165
C(time)[T.3]         -0.0797      0.164     -0.488      0.626      -0.400       0.241
C(time)[T.4]          0.1189      0.171      0.695      0.487      -0.216       0.454
C(time)[T.5]         -0.6651      0.261     -2.552      0.011      -1.176      -0.154
C(time)[T.6]         -0.3502      0.243     -1.438      0.150      -0.827       0.127
C(time)[T.7]         -0.1752      0.250     -0.701      0.483      -0.665       0.315
==============================================================================
```

After fitting the model, we can construct an analysis of deviance analog of an ANOVA table showing results of likelihood-ratio tests for the effects of individual explanatory variables, and construct CIs for effects (here, comparing stages 2 and 3 to stage 1):

```
In [5]: fit1 = smf.glm('count ~ C(stage) + C(time)',
               family = sm.families.Poisson(), offset = logrisktime,
               data = Cancer).fit()
   ...: fit2 = smf.glm('count ~ C(histology) + C(time)',
               family = sm.families.Poisson(), offset = logrisktime,
               data = Cancer).fit()
   ...: fit3 = smf.glm('count ~ C(histology) + C(stage)',
               family = sm.families.Poisson(), offset = logrisktime,
               data = Cancer).fit()
In [6]: D = fit.deviance; D1 = fit1.deviance
   ...: D2 = fit2.deviance; D3 = fit3.deviance
   ...: df = fit.df_resid; df1 = fit1.df_resid          # residual df values
   ...: df2 = fit2.df_resid; df3 = fit3.df_resid
   ...: from scipy import stats
   ...: P_hist = 1 - stats.chi2.cdf(D1 - D, df1 - df)   # like.-ratio
   ...: P_stage = 1 - stats.chi2.cdf(D2 - D, df2 - df)  # test P-values
   ...: P_time = 1 - stats.chi2.cdf(D3 - D, df3 - df)
In [7]: pd.DataFrame({'Variable':['C(histology)', 'C(stage)', 'C(time)'],
            'LR Chisq':[round(D1-D,3), round(D2-D,3), round(D3-D,3)],
            'df': [df1 - df, df2 - df, df3 - df],
            'Pr(>Chisq)': [P_hist, P_stage, P_time]})
       Variable  LR Chisq  df  Pr(>Chisq)
0  C(histology)     1.876   2    0.391317
1      C(stage)    99.155   2    0.000000
2       C(time)    11.383   6    0.077237

In [8]: CI = np.exp(fit.conf_int(alpha = 0.05))
   ...: CI.iloc[[3]]              # 95% CI for multiplicative effect of
Out[8]:                           # stage 2, compared to stage 1
                        0         1
C(stage)[T.2]  1.136683  2.252211
In [9]: CI.iloc[[4]]              # 95% CI for multiplicative effect of
Out[9]:                           # stage 3, compared to stage 1
                        0        1
C(stage)[T.3]  2.790695  5.06488
```

B7.5 Negative Binomial Modeling of Count Data

Negative binomial modeling of count data has more flexibility than Poisson modeling, because the response variance can exceed the mean, permitting overdispersion. As in Section 7.5.3, we first show that the marginal distribution of the number of satellites for the female horseshoe crabs exhibits more variability than the Poisson permits and then plot a histogram of the satellite counts (not shown here):

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import statsmodels.api as sm
   ...: import statsmodels.formula.api as smf
   ...: import matplotlib.pyplot as plt
In [2]: Crabs = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Crabs.dat', sep='\s+')
In [3]: round(np.mean(Crabs.sat), 4)      # mean number of satellites
Out[3]: 2.9191
In [4]: round(np.var(Crabs.sat), 4)       # variance of number of satellites
Out[4]: 9.8547 # overdispersion: for Poisson, true variance = mean
In [5]: plt.hist(Crabs['sat'], density=True, bins=16, edgecolor='k')
   ...: plt.ylabel('Proportion'); plt.xlabel('Satellites');
   ...: plt.show()
```

Negative binomial GLMs can be fitted in `statsmodels`. However, the dispersion parameter ($1/k$ in formula (7.7), called *alpha* in the `Python` output) is not estimated (as in the `glm.nb()` function of the `MASS` package in `R`) and needs to be specified. Thus, in practice a negative binomial GLM has to be fitted in two steps. First, we need to estimate the dispersion parameter and then fit the model, as illustrated below for our example. The estimation of the dispersion parameter can be done by a negative binomial model fitting in `statsmodels.discrete.discrete_model`. In order to estimate α correctly, we need to specify the design matrix of the GLM we want to fit. This stage yields correct estimates for the model parameters but not for their standard errors. In a second step, we re-fit the negative binomial GLM, setting the dispersion parameter equal to its estimate, derived in the first step.

```
# Step 1:
In [6]: from statsmodels.discrete.discrete_model import NegativeBinomial
In [7]: formula = 'sat ~ weight+C(color)'
   ...: model=NegativeBinomial.from_formula(formula, data=Crabs, loglike_method='nb2')
In [8]: fit_dispersion=model.fit()
   ...: print(fit_dispersiona.summary())
Out[8]: # correct estimates but wrong std. errors; alpha = dispersion est.
==============================================================================
                 coef  std err        z   P>|z|    [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.4263    0.559   -0.762   0.446    -1.522     0.670
C(color)[T.2] -0.2528    0.351   -0.721   0.471    -0.940     0.435
C(color)[T.3] -0.5219    0.379   -1.376   0.169    -1.265     0.222
C(color)[T.4] -0.4804    0.428   -1.124   0.261    -1.319     0.358
weight         0.7121    0.178    4.005   0.000     0.364     1.061
alpha          1.0420    0.190    5.489   0.000     0.670     1.414
# Step 2:
In [9]: a = fit_dispersion.params[5]                    # a set equal to alpha
   ...: fit = smf.glm('sat ~ weight + C(color)', family =
                      sm.families.NegativeBinomial(alpha = a), data = Crabs).fit()
In [10]: print(fit.summary())                           # edited output
                 Generalized Linear Model Regression Results
              Deviance:   196.56             Df Residuals: 168
==============================================================================
                  coef  std err        z   P>|z|     [0.025   0.975]
------------------------------------------------------------------------------
Intercept      -0.4263    0.538   -0.792   0.428     -1.481    0.629
C(color)[T.2]  -0.2527    0.349   -0.725   0.468     -0.936    0.430
C(color)[T.3]  -0.5218    0.380   -1.373   0.170     -1.266    0.223
C(color)[T.4]  -0.4804    0.428   -1.122   0.262     -1.320    0.359
weight          0.7121    0.161    4.410   0.000      0.396    1.029
==============================================================================
In [9]: print('AIC:', round(fit.aic, 3))
AIC: 755.935      # better fit than Poisson loglinear model in Section B.7.4
```

An analysis of deviance table can be constructed for this model, analogous to the tables constructed for the Poisson loglinear model in Section B7.4.1.

B7.6 Regularization: Penalized Logistic Regression Using the Lasso

The lasso regularization method is available with `glmnet`, which first has to be installed.[2] We illustrate it for the logistic regression model fitted on the student survey data, with response variable the opinion about whether abortion should be legal in the first three months of a pregnancy (1 = yes, 0 = no) and with 14 explanatory variables. The response and the explanatory variables in `glmnet` must be in separate arrays of type 'float64', using only the variables to be employed in the model. In our case, we therefore drop the variables `subject`, `abor` and `life` from the `Students` data frame. Using cross validation, the smoothing parameter that minimizes the mean prediction error is saved as *lambda_max_*, and the value from the one-standard-error rule is saved as *lambda_best_*. The latter is used for estimates and predictions unless the user selects an alternative value. The results shown next utilize it:

```
In [1]: conda install -c conda-forge glmnet
In [2]: import pandas as pd
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: from glmnet import LogitNet
In [3]: Students=pd.read_csv('http://stat4ds.rwth-aachen.de/data/Students.dat',sep='\s+')
   ...: y = Students.abor
   ...: x = Students.drop(['subject', 'abor', 'life'], axis = 1).astype('float64')
In [4]: fit = LogitNet()                  # ElasticNet() for regularized linear model
   ...: fit = fit.fit(x, y)
In [5]: print(fit.intercept_)
2.3671347098687052
In [6]: print(fit.coef_) # default: one std. error rule for smoothing.
[[ 0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
  -0.25995012 -0.18310286  0.        ]]   # only ideol and relig have
                                          # non-zero estimates
In [7]: print(fit.lambda_best_)           # smoothing parameter for one se rule
[0.12677869]
In [8]: print(fit.lambda_max_)            # smoothing to minimize prediction error
0.037826288644684083                      # less smoothing
In [9]: p = fit.predict(x)                # predict abortion response category
   ...: prob = fit.predict_proba(x)       # probability estimates
```

The plot of lasso model parameter estimates as function of the smoothing parameter $\log(\lambda)$, shown in Figure B6.3, is similar to Figure 7.9 of the book and is derived as follows:

```
In [10]: lambdas = fit.lambda_path_
    ...: betas = fit.coef_path_
    ...: n=betas.shape; n
Out[10]: (1, 15, 84)
In [11]: fig, ax = plt.subplots()
    ...: for i in range(1,n[1]):
    ...:      ax.plot(np.log10(lambdas),betas[0,i,])
    ...: # ax.set_xscale('log')
    ...: # ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
    ...: plt.axvline(x=np.log10(fit.lambda_best_), ymin=betas.min(), ymax=
                      betas.max(), linestyle='dashed', color='black', linewidth=0.7)
    ...: plt.axvline(x=np.log10(fit.lambda_max_), ymin=betas.min(), ymax=
                       betas.max(), linestyle='dotted', color='red', linewidth=0.7)
    ...: plt.xlabel('lambda')
```

[2] See https://pypi.org/project/glmnet

```
    ...: plt.ylabel('coefficients')
    ...: plt.axis('tight')
    ...: plt.show()
```

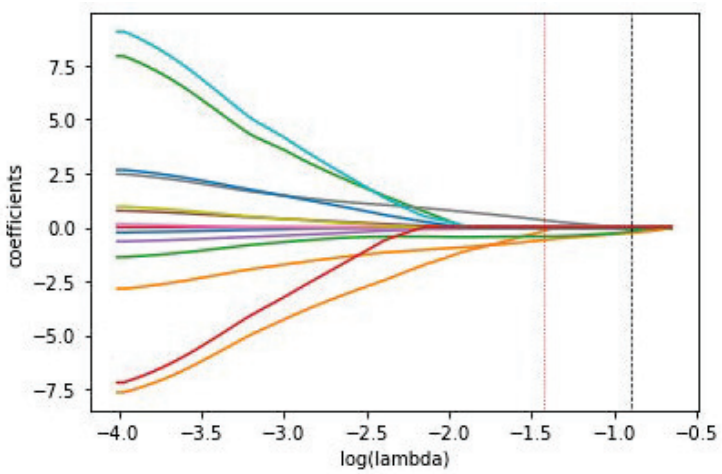


FIGURE B6.3: Plot of lasso model parameter estimates for predicting opinion on legalized abortion for the student survey, as function of the smoothing parameter $\log(\lambda)$. The black dashed and dotted red lines indicate the λ value suggested by one-st.dev. error rule and by 3-fold cross-validation, respectively.

By default the `LogitNet()` function applies 3-fold cross-validation. Changing this to 10-fold, we obtain the lasso parameter estimates using the best λ by 10-fold cross-validation.

```
In [12]: fit = LogitNet(n_splits=10)
    ...: fit = fit.fit(x, y)
In [13]: print(fit.intercept_)
2.6606674571327993
In [14]: print(fit.coef_)
[[ 0.          0.          0.          0.          0.          0.
   0.          0.          0.08637701  0.          0.          0.
  -0.37772893 -0.32340613  0.        ]]
In [15]: print(fit.lambda_best_)
[0.07962071]
In [16]: print(fit.lambda_max_)
0.0661025136734736
In [16]: fit = LogitNet(n_splits=10, cut_point=0.0661025136734736)
    ...: fit = fit.fit(x, y)
In [17]: print(fit.intercept_)
2.696921632490962
In [18]: print(fit.coef_)
[[ 0.          0.          0.          0.          0.          0.
   0.          0.          0.13995022  0.          0.          0.
  -0.42388451 -0.36032689  0.        ]]
In [19]: print(fit.lambda_best_)
[0.06610251]
In [20]: print(fit.lambda_max_)
0.0661025136734736
```

Alternatively, penalized logistic regression can be fitted in the `scikit-learn` library, which is built on `NumPy`, `SciPy`, and `matplotlib`. It is designed for machine learning applications of classification, clustering and penalized regression problems.

8

CHAPTER 8: PYTHON FOR CLASSIFICATION AND CLUSTERING

B8.1 Linear Discriminant Analysis

We illustrate a linear discriminant analysis (LDA) for the example in Section 8.1.2 of whether female horseshoe crabs have male satellites (y = 1, yes; y = 0, no). That section notes that it is sufficient to use weight or carapace width together with color as explanatory variables, and here we use width and color. In `python`, LDA can be implemented in `sklearn`. Notice that `sklearn` requires the the response variable to be a `numpy array` (`y`) and the explanatory variables to be another `numpy array` (`X`). We find the linear discriminant function,[1] the prediction for Y and the posterior probabilities for the two categories of Y at a particular setting of the explanatory variables, and then show how to find these for the horseshoe crabs in the sample:

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
In [2]: Crabs = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Crabs.dat', sep='\s+')
In [3]: y = np.asarray(Crabs['y'])       # we form X so it has only width and color
   ...: X = np.asarray(Crabs.drop(['crab','sat','y','weight','spine'], axis=1))
   ...: from sklearn.discriminant_analysis  import LinearDiscriminantAnalysis
   ...: lda = LinearDiscriminantAnalysis(priors=None)
   ...: lda.fit(X, y)
In [4]: lda.coef_
Out[4]: array([[ 0.429729, -0.552606]])  # coefficients of linear discriminant function
In [5]: lda.intercept_
Out[5]: array([-9.22893006])
In [6]: lda.score(X, y)                  # mean classification error
Out[6]: 0.7283236994219653
In [7]: print(lda.predict([[30, 3]]))    # predict y at x1 = 30, x2 = 3
[1]     # predicted y = 1 (yes for satellites) at width = 30, color = 3
In [8]: print(lda.predict_proba([[30, 3]]))
[[0.11866592 0.88133408]]                # posterior probabilities for y=0 and y=1
# Fisher's Discriminant Function value at the point X=(30,3):
In [9]: print(lda.decision_function([[30, 3]]))
[2.00512458]
# category prediction and associated probabilities for the sample:
In [10]: y_pred = lda.predict(X)
    ...: y_pred_prob = lda.predict_proba(X)
In [11]: print(y_pred)
[1 0 1 0 1 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1
 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1
 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1
 0 1 1 1 0 0 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1]
```

[1] The linear discriminant coefficients provided by the default singular value decomposition solver (*solver='svd'*) differ slightly from those found by R, but the estimated probabilities are the same.

```
In [12]: print(y_pred_prob) # estimates of prob. [P(Y=0),P(Y=1)], shown only for two crabs
 [0.1385732  0.8614268 ]                # crab 1 had width x1 = 28.3, color x2 = 2
 [0.77168388 0.22831612]                # crab 2 had width x1 = 22.5, color x2 = 3
```

The reported predictions and posterior probabilities take the prior probability π_0 for $P(Y = 1)$ to be the default value, which is the sample proportion with $y = 1$. You can set an alternative, such as by replacing *priors=None* with *priors = (0.5, 0.5)*.

A scatterplot of the explanatory variable values can show the actual y values and use a background color to show the predicted value. This plot is shown in Figure B8.1 and is derived by the following code which is based on functions provided in the `scikit learn` web-sides:

```
In [13]: from matplotlib import colors
    ...: def plot_data(lda, X, y, y_pred):
    ...:     splot = plt.subplot()
    ...:     plt.title(' ')
    ...:     plt.ylabel('color')
    ...:     plt.xlabel('width')
    ...:
    ...:     tp = (y == y_pred)  # True Positive
    ...:     tp0, tp1 = tp[y == 0], tp[y == 1]
    ...:     X0, X1 = X[y == 0], X[y == 1]
    ...:     X0_tp, X0_fp = X0[tp0], X0[~tp0]
    ...:     X1_tp, X1_fp = X1[tp1], X1[~tp1]
    ...:
    ...:     # class 0: dots
    ...:     plt.scatter(X0_tp[:, 0], X0_tp[:, 1], marker='.', color='red')
    ...:     plt.scatter(X0_fp[:, 0], X0_fp[:, 1], marker='x',
    ...:                 s=20, color='#990000')  # dark red
    ...:
    ...:     # class 1: dots
    ...:     plt.scatter(X1_tp[:, 0], X1_tp[:, 1], marker='.', color='blue')
    ...:     plt.scatter(X1_fp[:, 0], X1_fp[:, 1], marker='x',
    ...:                 s=20, color='#000099')  # dark blue
    ...:
    ...:     # class 0 and 1 : areas
    ...:     nx, ny = 200, 100
    ...:     x_min, x_max = plt.xlim()
    ...:     y_min, y_max = plt.ylim()
    ...:     xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
    ...:                          np.linspace(y_min, y_max, ny))
    ...:     Z = lda.predict_proba(np.c_[xx.ravel(), yy.ravel()])
    ...:     Z = Z[:, 1].reshape(xx.shape)
    ...:     plt.pcolormesh(xx, yy, Z, cmap='coolwarm_r',
    ...:                    norm=colors.Normalize(0., 1.), zorder=0)
    ...:     plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='white')
    ...:
    ...:     # means
    ...:     plt.plot(lda.means_[0][0], lda.means_[0][1],
    ...:              '<', color='red', markersize=10, markeredgecolor='red')
    ...:     plt.plot(lda.means_[1][0], lda.means_[1][1],
    ...:              '>', color='blue', markersize=10, markeredgecolor='blue')
    ...:
    ...:     return splot
    ...:
In [14]: plot_data(lda, X, y, y_pred)  # call of the function
```

B8.1.1 Predictive Power

Evaluation of the quality of prediction achieved by a LDA or a logistic regression model is most frequently summarized in the classification table and visualized by the ROC curves

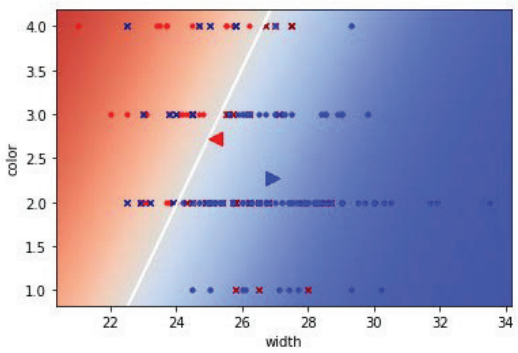


FIGURE B8.1: Scatterplot matrix for width and color of the female crabs data with the prediction areas marked (0:red, 1: blue). Blue (red) points indicate the correct predicted `y=1` (`y=0`) cases, while `x` indicate the corresponding false predicted cases.

(see Section 8.1.3). Measures for evaluation of the predicted classification are provided in the `sklearn.metrics` module, as illustrated below for the crabs classification problem, analyzed either by LDA or logistic regression:

```
In [15]: from sklearn.metrics import confusion_matrix
    ...: confusion_matrix = confusion_matrix(y, y_pred)
    ...: print(confusion_matrix)
[[29 33]           # mean classification error lda.score(X,y) = (29+97)/173
 [14 97]]
In [16]: from sklearn.model_selection import cross_val_score
    ...: crv = cross_val_score(lda, X,y, cv=10); crv
Out[16]:  # score for each of the 10 "folds"
array([0.78947368, 0.77777778, 0.70588235, 0.70588235, 0.76470588,
       0.76470588, 0.76470588, 0.52941176, 0.76470588, 0.70588235])
In [17]: classif_accuracy = np.mean(crv); classif_accuracy
Out[17]: 0.7273133814929481
In [18]: import warnings
    ...: warnings.filterwarnings("ignore")
    ...: from sklearn import metrics
    ...: print(metrics.classification_report(y, y_pred, digits=4))
              precision    recall  f1-score   support

           0     0.6744    0.4677    0.5524        62
           1     0.7462    0.8739    0.8050       111

    accuracy                         0.7283       173
   macro avg     0.7103    0.6708    0.6787       173
weighted avg     0.7204    0.7283    0.7145       173
```

The *precision* for each response category i, $i = 0, 1$, is the ratio $\frac{tp_i}{tp_i+fp_i}$ where tp_i and fp_i is the number of true positives and false positives, respectively, for the i-th category. The *recall* is the ratio $\frac{tp_i}{tp_i+fn_i}$, where fn_i is the number of false negatives for the i-th category, i.e. it expresses the ability of detecting all `y=1` cases. The *F-beta* (f1) score is a weighted harmonic mean of precision and recall and lies in the interval $[0, 1]$, with 1 corresponding to its best value.

The ROC curve can be plotted using the `roc_curve` function of the `sklearn.metrics` module. The required arguments are the observed category membership for our data (`y`) and the predicted probabilities for `y=1` (`y_pred_prob[:,1]`). Next, we provide code for con-

structing the ROC curve and finding the area under the curve to summarize the predictive power:

```
In [19]: def plot_roc(fpr, tpr):
    ...:     splot= plt.subplot()
    ...:     roc_auc = auc(fpr, tpr)
    ...:     plt.figure(1, figsize=(12,6))
    ...:     plt.plot(fpr, tpr, lw=2, alpha=0.7, color="red",
                              label='AUC = %0.4f' % (roc_auc))
    ...:     plt.plot([0, 1], [0, 1], linestyle='--',lw=0.7,color='k',alpha=.4)
    ...:     plt.xlim([-0.05, 1.05])
    ...:     plt.ylim([-0.05, 1.05])
    ...:     plt.xlabel('1-Specificity')
    ...:     plt.ylabel('Sensitivity')
    ...:     plt.legend(loc="lower right")
    ...:     # plt.grid()     # to add grid to the plot (here commented out)
    ...:     return splot
```

This function is called next for the predicted probabilities of the (i) LDA and (ii) logistic regression models fitted above:

```
In [20]: from sklearn.metrics import roc_curve
    ...: from sklearn.metrics import auc
In [21]: fpr, tpr, thresholds = roc_curve(y, y_pred_prob[:,1])
    ...: roc_auc = auc(fpr, tpr); roc_auc        # AUC (LDA)
Out[21]: 0.7640947399011915
In [22]: fpr1, tpr1, thresholds1 = roc_curve(y, y_pred_lgr_prob[:,1])
    ...: roc_auc1 = auc(fpr1, tpr1); roc_auc1    # AUC (logistic regression)
Out[22]: 0.7242807323452484
In [23]: plot_roc(fpr, tpr)    # call of plot function for LDA predictions
In [24]: plot_roc(fpr1, tpr1)  # call for logistic regression predictions
```

The ROC curves for LDA and logistic regression are shown in Figure B8.2.

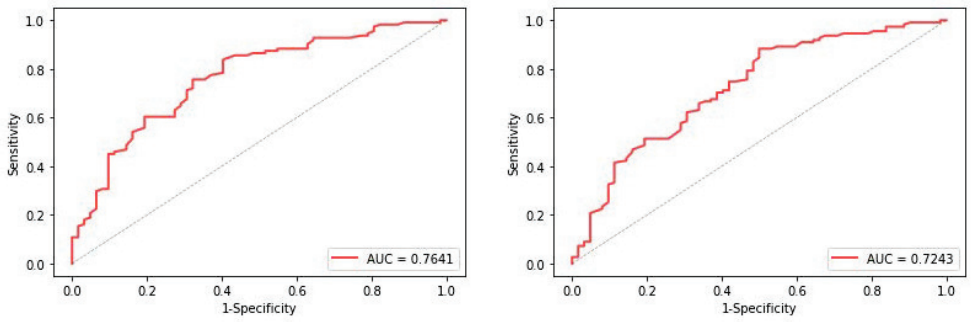


FIGURE B8.2: ROC curves for the female crabs category membership prediction (have satellites or not), based on the variables `width` and `color`, for LDA (left) and logistic regression (right).

The ROC curve is also relevant for predictions made using logistic regression. Next we fit that model to these data and find the ROC curve and the area under it:

```
In [25]: import statsmodels.api as sm
    ...: import statsmodels.formula.api as smf
In [26]: fit = smf.glm('y ~ width + color', family = sm.families.Binomial(),
                       data=Crabs).fit()
In [27]: print(fit.summary())
```

```
                 Generalized Linear Model Regression Results
                coef  std err         z   P>|z|    [0.025    0.975]
------------------------------------------------------------------------
Intercept   -10.0708    2.807    -3.588   0.000   -15.572    -4.569
width         0.4583    0.104     4.406   0.000     0.254     0.662
color        -0.5090    0.224    -2.276   0.023    -0.947    -0.071
========================================================================
In [28]: predictions = fit.predict()
In [29]: fpr, tpr, thresholds = roc_curve(y, predictions)
In [30]: auc(fpr, tpr)                  # area under ROC curve
Out[30]: 0.762205754141238
In [31]: plot_roc(fpr, tpr)             # plots the ROC curve
```

B8.2 Classification Trees and Neural Networks for Prediction

Classification trees can be obtained by the class `DecisionTreeClassifier` of `sklearn.trees`, which also takes as input an array X of size (n, p), holding the explanatory variables (training sample), where n is the sample size and p the number of explanatory variables (features) and an one-dimensional array y of integers, of size n, holding the response categories (class labels of the training sample). Similar to other classification methods, class membership predictions and measures of accuracy can be derived. The default criterion used for measuring the quality of split is the Gini imputity (`criterion='gini'`) or an entropy measure corresponding to maximizing the binomial log-likelihood function (`criterion='entropy'`). There are further parameters that control the pruning of a classification tree. For example, for the crabs data below, we ask for a tree with maximal number of leaf (terminal) nodes equal to 3:

```
# continuing with Crabs file and X and y arrays previously formed
In [32]: from sklearn import tree
   ...: clf = tree.DecisionTreeClassifier(criterion = 'gini', max_leaf_nodes=3).fit(X, y)
In [33]: X_names=['width', 'color']
   ...: y_names=['no', 'yes']
   ...: fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (4,4), dpi=300)
   ...: # classification tree shown in figure:
   ...: tree.plot_tree(clf, feature_names = X_names, class_names = y_names, filled=True)
In [34]: from sklearn import metrics
   ...: y_pred = clf.predict(X)
   ...: print('Accuracy:', metrics.accuracy_score(y, y_pred))
Accuracy: 0.716763006              # proportion correct = (44 + 75 + 5)/173 = 0.717
```

Figure B8.3 shows the classification tree obtained, which surprisingly differs somewhat from the one in Section 8.1.4 using `R` with the same (*gini*) criterion. It predicts that the crabs that have satellites are the ones of width greater than 25.85 *cm* that are in color classes 1, 2, and 3, of which the terminal node tells us that 13 did not have satellites and 75 did. For these data, the overall proportion of correct predictions with this tree is $(44 + 75 + 5)/173 = 0.717$. The tree constructed with three terminal nodes by `R`, shown in Figure 8.2, also predicted satellites for horseshoe crabs of color 4 with width above 25.85 *cm* and those of colors 1 and 2 with width less than 25.85 *cm*.

Another option is to use the classifier `DecisionTreeRegressor` that uses as criterion the mean square error and otherwise has further parameters, analogous to those of `DecisionTreeClassifier`.

A neural network can be fitted by the `MLPClassifier` classifier of `sklearn.neural_network`, as shown below for the crabs data set. We repeat the analysis of Section 8.1.6, resulting to

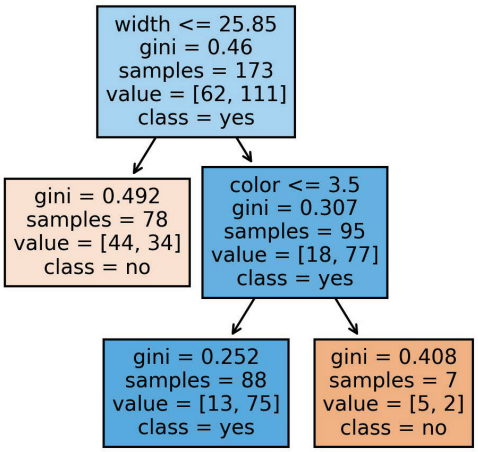


FIGURE B8.3: Classification tree for whether female crabs have satellites ($\hat{y} = 1$, in blue) or not ($\hat{y} = 0$, in orange).

classifications of comparable accuracy. We provide the classification tables for two threshold probabilities, the standard (0.5) and 0.64. Notice that in `sklearn` there exist no parameter controlling the threshold. For this, we provide next a function for adjusting it. The default solver used in `MLPClassifier` is 'adam' (a stochastic gradient-based optimizer). We use for this application the 'lbfgs' optimizer (a quasi-Newton type), since it is more adequate for small data sets in terms of convergence and performance.

```
In [35]: from sklearn.neural_network import MLPClassifier
    ...: clf = MLPClassifier(solver='lbfgs')
In [36]: from sklearn.model_selection import train_test_split
    ...: # use 2/3 to train, 1/3 to test
    ...: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
In [37]: clf.fit(X_train, y_train)
    ...: y_pred = clf.predict(X_test)     # y predictions for test data
    ...: y_pred_prob = clf.predict_proba(X_test)
In [38]: from sklearn.metrics import confusion_matrix
    ...: confusion_matrix = confusion_matrix(y_test, y_pred)
    ...: print('classification table:')
    ...: print(confusion_matrix)
classification table:
[[10  7]            # accuracy proportion (10 + 38)/58 = 0.828
 [ 3 38]]           # this is random, depending on which 2/3 chosen for training

# function to change the threshold from p=0.5 to t:
In [39]: def prob_threshold(y_pred_prob, t):
    ...:     n=len(y_pred_prob)
    ...:     y_pred_adj=[]
    ...:     y_pred_adj= [0 for x in range(n)]
    ...:     for i in range(1,n):
    ...:         if y_pred_prob[i-1,1] >= t:
    ...:               y_pred_adj[i-1]=1
    ...:     return y_pred_adj

# analysis using the sample proportion with y = 1 (0.64) as the threshold
In [40]: y_pred_adj = prob_threshold(y_pred_prob, 0.64)       # y=0 for prob<0.64
In [41]: from sklearn import metrics
```

```
    ...: print('Accuracy:', metrics.accuracy_score(y_test, y_pred))
    ...: print('Accuracy (adj. threshold):',
                    metrics.accuracy_score(y_test, y_pred_adj))
Accuracy: 0.8275862068965517
Accuracy (adj. threshold): 0.7068965517241379

In [42]: confusion_matrix_adj = confusion_matrix(y_test, y_pred_adj)
    ...: print('classification table (adjusted threshold):')
    ...: print(confusion_matrix_adj)
classification table (adjusted threshold):
[[15  2]
 [15 26]]
```

B8.3 Cluster Analysis

Hierarchical clustering can be performed by the `Aglomerative Clustering` object of `sklearn.cluster`, offering the standard linkage methods ('ward', 'complete', 'average', 'single'). The norm used to compute the linkage can be chosen among 'euclidean', 'l1', 'l2', 'manhattan', 'cosine', and 'precomputed', the last requiring a distance matrix as input for the fit method. For the 'ward' linkage, which is the default, only 'euclidean' is accepted. For the criterion to stop the clustering, one can specify the number of desired clusters or specity a linkage distance threshold. K-means clustering is berformed by the `KMeans` object.

We illustrate hierarchical clustering with 'average linkage' and 'manhattan' distance for the U.S. Presidential elections example of Section 8.2.3. The dendrogram is constructed in `scipy.cluster.hierarchy`.

```
In [1]: import pandas as pd
   ...: from sklearn.cluster import AgglomerativeClustering
   ...: Elections = pd.read_csv('http://stat4ds.rwth-aachen.de/data/Elections2.dat',
                                    sep='\s+')
   ...: Elections.head(2)
Out[1]:
   number       state  e1  e2  e3  e4  e5  e6  e7  e8  e9  e10
0       1     Arizona   0   0   0   0   1   0   0   0   0    0
1       2  California   0   0   0   1   1   1   1   1   1    1
In [2]: y = np.array(Elections['state'])
   ...: state = ['Arizona','California','Colorado','Florida','Illinois','Massachusetts',
            'Minnesota','Missouri','NewMexico','NewYork','Ohio','Texas','Virginia','Wyoming']
   ...: X = np.array(Elections.drop(['number','state'], axis=1))
   ...: clustering = AgglomerativeClustering(n_clusters = None, affinity = 'manhattan',
                          linkage = 'average', distance_threshold = 0.01).fit(X)
In [3]: cluster = clustering.labels_
In [4]: %matplotlib inline
   ...: import matplotlib.pyplot as plt
   ...: import scipy.cluster.hierarchy as sch
   ...: # Calculate the distance matrix
   ...: Z = sch.linkage(X, method='average', metric='cityblock')
   ...: plt.figure(figsize=(10, 7))
   ...: sch.dendrogram(Z, orientation='top',           # dendrogram
   ...:         labels=state, leaf_rotation=80,
   ...:         distance_sort='descending',show_leaf_counts=False)
```

The heatmap with the dendrogram for cluster analysis can be constructed by `clustermap` of `seaborn`. Though the graph is produced when applied on the `np.array` X, it is better to apply it on a data frame, so that labels (here, the states) can be added on the plot. For this, we define next a data frame, X1, which has the states as index:

```
In [6]: import seaborn as sns
   ...: sns.set(color_codes=True)
In [7]: X1=Elections.drop(['number','state'], axis=1)
   ...: X1.index=state
In [8]: g = sns.clustermap(X1, metric='cityblock', method='average',
   ...:                    figsize=(7, 5), col_cluster=False,
   ...:                    dendrogram_ratio=(.1, .2), cbar_pos=(0, .2, .03, .4),
   ...:                    xticklabels=True,           # adds variables as labels
   ...:                    yticklabels=True,           # adds states as labels
   ...:                    cmap='coolwarm')            # heatmap with dendrogram
```

The above derived dendrogram and heatmap with dentrogram are shown in Figure B8.4.

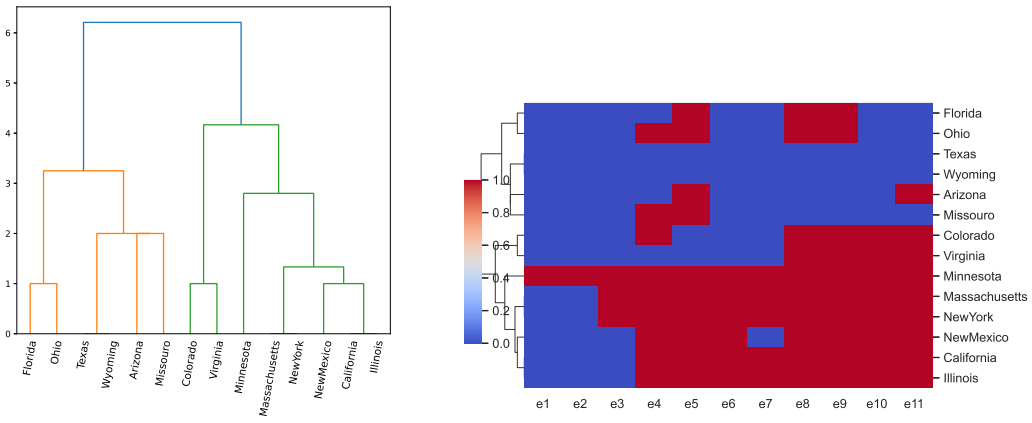


FIGURE B8.4: Dendrogram and heatmap with dendrogram for the cluster analysis of the election data.