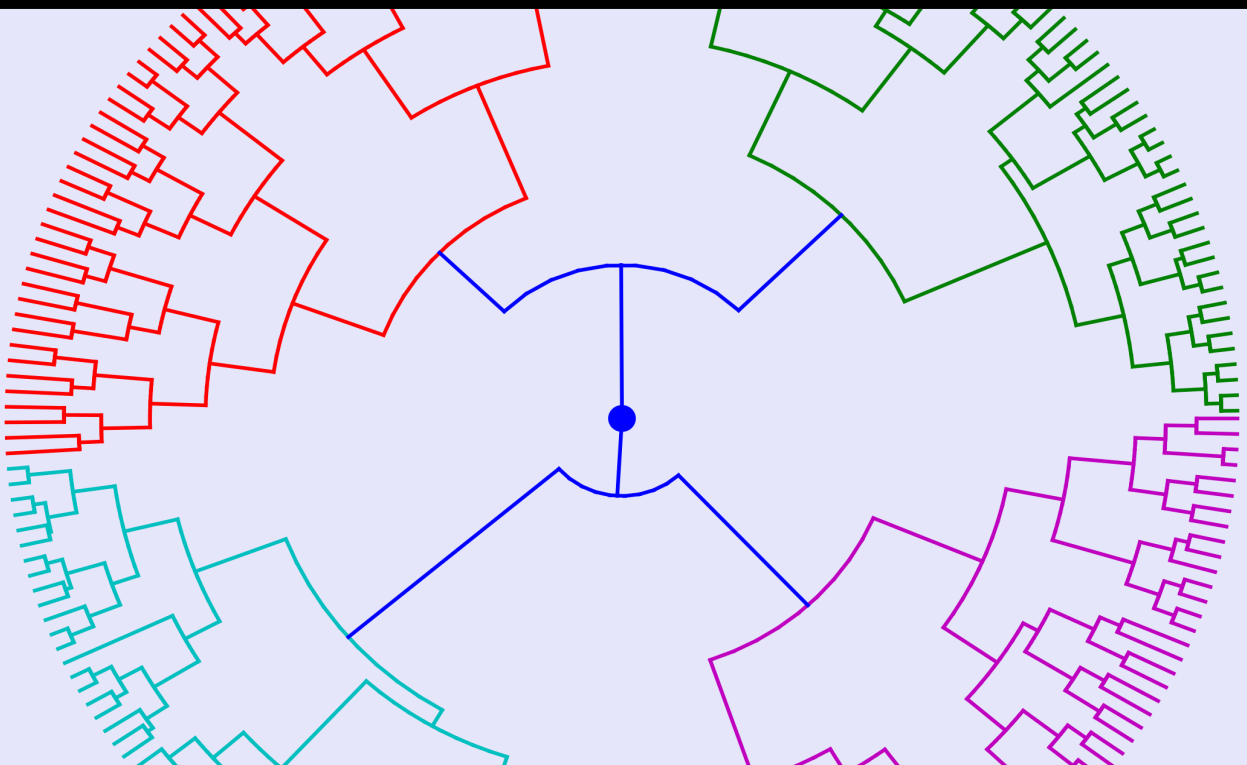
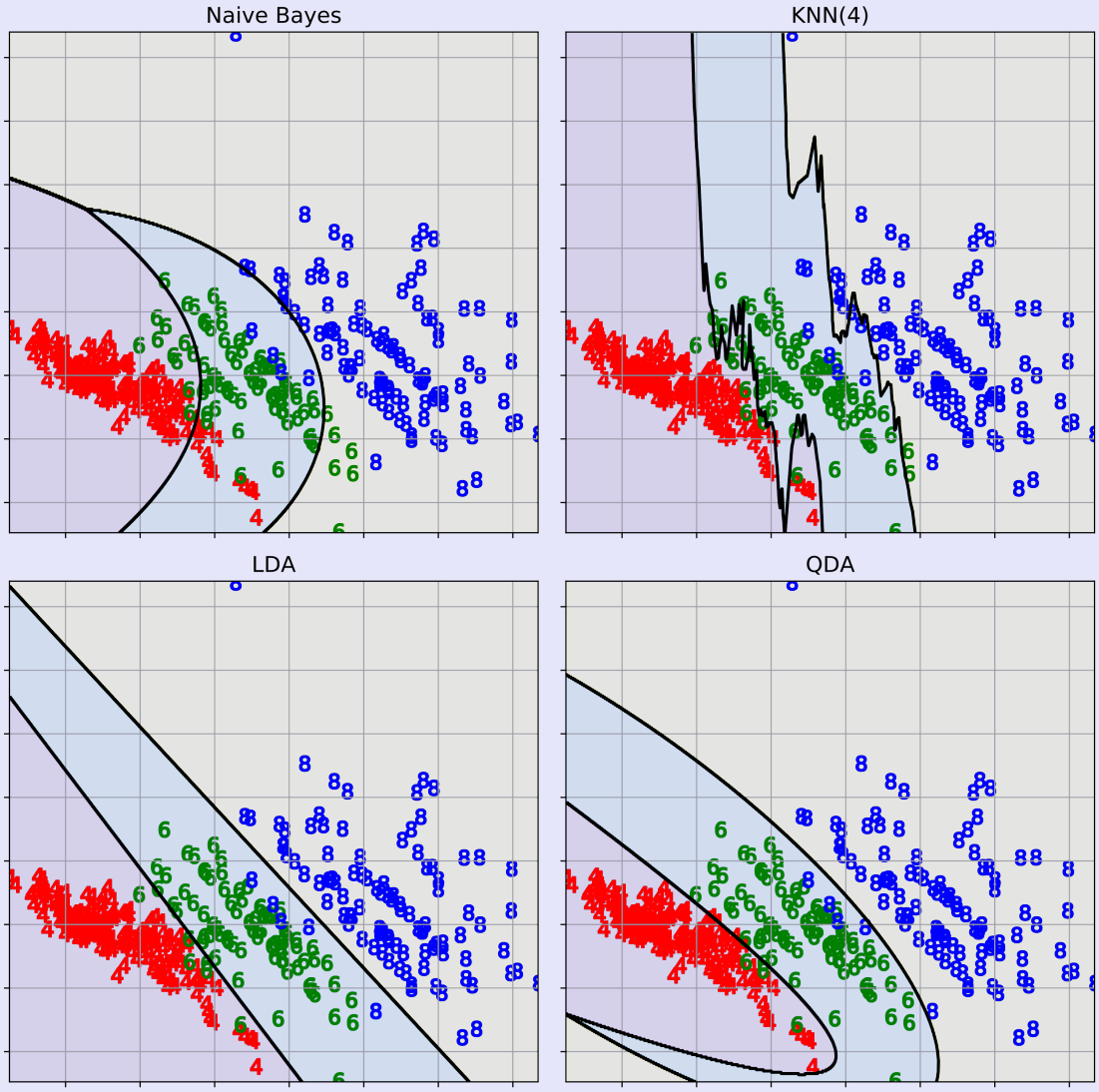
Getting Started in Machine Learning

Easy Recipes for Python 3, Scikit-Learn, and Jupyter

Isabella Romeo





Bella is a blue nose American Staffordshire Terrier. Bella's hobbies include barking at trees, chasing grasshoppers, and sleeping on the couch. Since she does not have opposable thumbs, this book was dictated to her human for subsequent typesetting. Said human teaches math at a California State University, and in his past lives has worked at Caltech, JPL and NASA/GSFC.

Getting Started in Machine Learning

Easy Recipes for Python 3, Scikit-Learn, and Jupyter

Isabella Romeo

Zeroth Edition (2019)

Sherwood Forest

calculuscastle.com

Getting Started in Machine Learning: Easy Recipes for Python 3, Scikit-Learn, and Jupyter, Zeroth Edition, by Bella Romeo
Last Revised on: 5/18/19 (Version 0.1.5075)
Sherwood Forest Books, Los Angeles, CA, USA

ISBN-13: 978-0-9966860-7-5
ISBN-10: 0-9966860-7-X

© 2019 B. E. Shapiro. All Rights Reserved. No part of this document may be reproduced, stored electronically, or transmitted by any means without prior written permission of the author.

THIS DOCUMENT IS PROVIDED IN THE HOPE THAT IT WILL BE USEFUL BUT WITHOUT ANY WARRANTY, WITHOUT EVEN THE IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THE DOCUMENT IS PROVIDED ON AN "AS IS" BASIS AND THE AUTHOR HAS NO OBLIGATIONS TO PROVIDE CORRECTIONS OR MODIFICATIONS. THE AUTHOR MAKES NO CLAIMS AS TO THE ACCURACY OF THIS DOCUMENT. IN NO EVENT SHALL THE AUTHOR BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, UNSATISFACTORY CLASS PERFORMANCE, POOR GRADES, CONFUSION, MISUNDERSTANDING, EMOTIONAL DISTURBANCE OR OTHER GENERAL MALAISE ARISING OUT OF THE USE OF THIS DOCUMENT OR ANY SOFTWARE DESCRIBED HEREIN, EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. IT MAY CONTAIN TYPOGRAPHICAL ERRORS. WHILE NO FACTUAL ERRORS ARE INTENDED THERE IS NO SURETY OF THEIR ABSENCE.

First editions suck. Especially ones without editors. Don't expect this one to be any better. Please report any errors, omissions, or suggestions for improvements through the form at `https://github.com/biomathman/machine-learning-in-python-book/issues`. A list of known typographical errors is maintained on the wiki at `https://github.com/biomathman/machine-learning-in-python-book/wiki`.

Full color paperback copies of this book may be purchased from `barnesandnoble.com` at `https://www.barnesandnoble.com/w/getting-started-in-machine-learning-isabella-romeo/1130341765`.

Electronic copies of this book may be purchased through Gumroad at `https://gumroad.com/l/FPxDo`. If you have obtained a copy from any other source your copy is unauthorized and in violation of international copyright law. Please go online and pay for your copy.

Contents

Contents iii

Preface iii

1 Linear Regression 1

2 Multilinear Regression . . 27

3 Polynomial Regression . . 37

4 Stepwise Regression 51

5 Ridge Regression 61

6 Lasso Regression 73

7 Spline Regression 79

8 Nonlinear Regression . . . 85

9 Regression Trees 97

10 Logistic Regression 107

11 Evaluating Binary Classification 115

12 Artificial Neural Networks 125

13 Naive Bayes 145

14 K-Nearest Neighbors . . . 159

15 Discriminant Analysis . . 165

16 Principal Component Analysis 171

17 Support Vector Machines 185

18 Decision Trees 191

19 Tree Ensemble Methods . 197

20 Comparing Classifiers . . 209

21 Clustering Methods 219

A Review of Probability . . 240

B Review of Linear Algebra 259

Literature Citations 284

Index 288

Preface

What is Machine Learning?

Ask any two data scientists questions like **"What is machine learning?"** or **"What is Data Science?"** and you are likely to get two different answers. This section will address these questions for the purposes used in this textbook.

The term **machine learning** is widely used to encompass a wide variety of data analysis techniques. Many of the techniques were invented and in common practice long before the invention of computers and and well before the subject now called machine learning began to coalesce as a subject of inquiry.

The term **machine learning** usually encompasses techniques that will help user perform one or more of the following overlapping functions:

- Curve and model fitting, such as regression;
- Classification, such as grouping items into categories;
- Identification, such is figuring out which category an item fits into;
- Decision making, such as figuring how to schedule all the meetings in different conference rooms during a large convention;
- Recognizing patterns such as peoples faces, words, or handwriting;
- Prediction or extrapolation.

Sometimes **Machine Learning** tasks are categorized into **regression** tasks and **classification** tasks, but this is something of a simplification. In a regression task, we explore how one variable changes with respect to another variable. In a classification task, we explore how things are classified into different groups. Often we find ourselves doing a little bit of both to solve a problem, and we can find that the technique to do one is equivalent to solving the other.

Another sub-classification of machine learning methods describes how models

are trained. In a **supervised learning** technique, a model is trained with exemplars that indicate the correct answer. For example, a model that finds the location of a fellow named Waldo from pictures will be trained with a large number of pictures with Waldo circled in red. An **unsupervised model** will similarly be trained with a large number of pictures, but will not have Waldo circled.

Related subjects are **statistical learning**, **computational statistics**, **data mining**, and **data science**. These are all different subjects and should not be confused with one another. **Data science** might be thought of as the broad field of gleaning meaningful information from data. To perform **data science**, a data set must already exist. **Data mining** is a sub-discipline of **data science** that involves actively seeking interesting or useful data or data sets that are related to a particular subject or question. Thus both **data science** and **data mining** are **data driven**. **Statistics**, on the other hand, is **hypothesis driven**. A hypothesis must be made before data is obtained. Inferential **statistics** and **data science** thus represent two different philosophical approaches to the same problem. While many of the methods and techniques of **statistics** are used by a **data-driven** analysis; this does not make it **statistics**.

About this Book

This book is intended for an advanced community college or middle-level college student in computer science, engineering, math, or the sciences who is interested in learning methods in machine learning.

Examples of every method are provided in Python 3, and Jupyter notebooks for each chapter can be found in the accompanying github site.[1] Use of the standard libraries like **`sklearn`** is taught for every method. Wherever possible, student should learn to use these methods. However, as this is a textbook, where possible, students are taught how to implement their own method, as well as to use the standard libraries. Extensive footnotes provide links to online code documentation to encourage students to use the original source.

While some calculus, linear algebra, and probability is used throughout the text, deeper understanding of these subjects is not required. Appendices provide some review material in probability and linear algebra that should be sufficient for most students. An underlying mathematical justification for each machine learning method is provided at the level of the target audience; consequently the depth of the description will vary by chapter, often leaving details and specifics to references.

[1] `https://github.com/biomathman/machine-learning-in-python-book`

Prerequisites

All of the examples in this book are in Python 3. Directions for installing Python on any standard desktop or laptop operating system (Windows, Mac, or Linux) can be found at the Python download site.[2] Most modern versions of MacOS and and Linux will already have Python 3 installed. Some users may prefer to use an integrated environment such as Anaconda.[3] Pydroid and Pythonista are available for Android and iOS operating systems, respectively. The linux versions of Python can be installed on some Chromebooks in developer mode.[4]

This is not a book in Python, so students who are new to the language will probably find a stiff learning curve. There is extensive documentation of the Python 3 language online including at the Python web site.[5] Students should familiarize themselves with this type of documentation as it is the standard reference material on Python. When new functions, particularly in the machine learning package **`sklearn`** are introduced the first time, a footnote to the full documentation is given. Some students may prefer to refer to one of the texts in the bibliography as they are going through this material.[6]

All of the code in this book assumes the user is writing Python in Jupyter notebooks. Jupyter is an interface that allows you to combine code, narrative text, graphics, input and output in a single, seamless stream, similar to a lab notebook. The name Jupyter is a combination of three computer languages that Jupyter was originally designed to interface with: Julia, Python, and R. At the time of this writing, over 100 computer languages supported Jupyter. Installation instructions are given on the Jupyter web set.[7] While in principle, the material herein could be transcribed using a general IDE and interpreted via the command line environment, that is not the intent of this book. Students are encouraged to learn to use the notebook interface.

Expressions are written in blocks and are intended to convey a single cell in a notebook, as such:

```
# This represents a single
# cell in a Jupyter Notebook
```

```
# This represents a different
# cell in the Jupyter notebook
```

[2]See `https://www.python.org`.
[3]Available at `https://www.anaconda.com/download`).
[4]See `https://wsvincent.com/install-python3-chromebook/`
[5]See `https://www.python.org/doc/`.
[6]Such as the ones by Lutz; Martelli; Summerfield; or Zelle
[7]See `https://jupyter.org/install`.

In most cases the commands do not have to be included together in the same cell (although plots are usually an exception).

Using sklearn

While not everything in this book uses **sklearn**,[8] much of it does. Many (though not all) of the methods in **sklearn** follow a common pattern. In general, the pattern for both classification and regression is the same.

1. Let **X** be a feature matrix (page 28). For supervised methods, let **Y** be exemplar values, and separate the **(X,Y)** pairs into a training set **XTRAIN**, **YTRAIN** and a test set **XTEST**, **YTEST**.
2. Load the method:

```
from sklearn.somepackage import classifier
clf = classifier(options)
```

3. Fit training set (supervised methods) using the classifier:

```
clf.fit(XTRAIN,YTRAIN)
```

   or for an unsupervised method,

```
clf.fit(X)
```

4. Make a prediction:

```
YPREDICT=clf.predict(XTEST)
```

   Unsupervised methods may not have a separate **predict** method and instead use something like:

```
clusters = clf.fit_predict(X)
```

   The difference is that clustering methods only predict cluster membership for the original data set, not for additional data points. With supervised methods, once a model is fit, it can be applied to any data set of the appropriate dimensionality.
5. Score or test the results by comparing **YPREDICT** with **YTEST**, or examining the nature of the clusters as a function of the complexity of the parameters. There are many different methods that we will talk about for this.

[8] Pedregosa, F et al (2011) "Scikit-learn: Machine Learning in Python." *JMLR.* **12**:2825-2830. Full documentation is available online at `https://scikit-learn.org/stable/`.

Chapter 1

Linear Regression

Data and Tuples

Suppose we are faced with a set of n **data pairs**

$$(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1}) \tag{1.1}$$

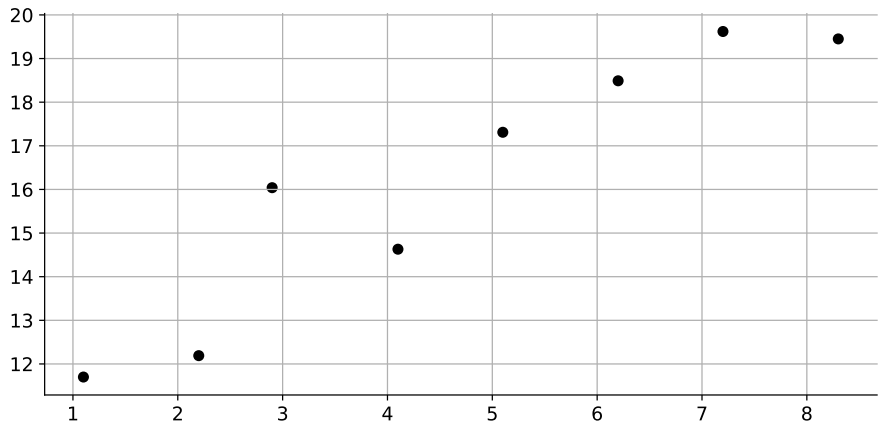
We will call each pair of numbers (x_i, y_i) a **data point**. Typically, we might such a data point by drawing a small black dot (or other **marker** or **symbol**) on a **scatter plot** (such as figure 1.1). We will refer to the complete collection of all data points that we are interested in studying at any given time as a **data set**.

An example of a data set is given in table 1.1. Here the x values give the year, and the y values give the population, in thousands of individuals. Thus $(x_0, y_0) = (1790, 3929)$, $(x_1, y_1) = (1800, 5308)$, and so forth. Since there are 21 data pairs in the table, the value $n = 21$, and the final entry is $(x_{20}, y_{20}) = (1990, 248709)$. Data sets are frequently stored comma-separated-value files. These are text files that contain columns of numbers separated by commas, and typically have file extensions of **`csv`** or **`txt`**.[1] The data set illustrated in figure 1.1, for example, is contained in the file **`first-csv-file.csv`**.[2] The contents of this **`csv`** file are illustrated below.

[1] Note that on some operating systems, such as Windows, the visibility of the file extension in the file name is suppressed by default.

[2] Unless they are obtained from other sources, data files can be found in the **`data`** folder of github repository that accompanies this book.

Figure 1.1: Sample scatter plot of the data points (1.1, 11.7), (2.2, 12.19), (2.9, 16.04), (4.1, 14.63), (5.1, 17.31), (6.2, 18.49), (7.2, 19.62), (8.3, 19.45).




```
x, y
1.1, 11.7
2.2, 12.19
2.9, 16.04
4.1, 14.63
5.1, 17.31
6.2, 18.49
7.2, 19.62
8.3, 19.45
```

We can easily input the contents of a **csv** file into a Jupyter notebook[3] for Python 3 using the **pandas read_csv** function.[4] This function reads a **csv** file into a **pandas DataFrame**.

```
import pandas as pd # Load pandas library
data = pd.read_csv("first-csv-file.csv")
```

Using **pandas** we can print out the **DataFrame** so that it looks like a spreadsheet. The lines are numbered and the columns are neatly aligned:

```
print(data)
```

[3]See http://jupyter.org/ for details on Jupyter notebooks.
[4]For details on **pandas**, including tutorials and full reference manuals on all functions, including **read_csv**, see https://pandas.pydata.org/.

```
     x      y
0  1.1  11.70
1  2.2  12.19
2  2.9  16.04
3  4.1  14.63
4  5.1  17.31
5  6.2  18.49
6  7.2  19.62
7  8.3  19.45
```

A **pandas DataFrame** can be directly converted to a **numpy array**.

```
xyvals=np.array(data)
```

All of the standard array manipulation functions available in **numpy** are then available for processing the data: columns in the **DataFrame** become columns in the **array** and rows in the **DataFrame** become rows in the **array**. So if you already know how to manipulate **numpy array**'s, then you can use **pandas** with zero learning curve.

We say that the data set described in equation 1.1 is **two-dimensional** because each data point consists of two values: an x-coordinate value, and a y-coordinate value. In general, data points may contain more than two values. For example, a bank may determine an applicants credit score by calculating their total income, bank account balance, monthly home expenses, and current debt. We can assign a tuple to a particular individual applicant, and let

$$x_i = \text{Monthly Income} \tag{1.2}$$

$$y_i = \text{Bank Balance} \tag{1.3}$$

$$z_i = \text{Monthly Home Expense} \tag{1.4}$$

$$w_i = \text{Monthly Payment on Current Debt} \tag{1.5}$$

Then we would define a **tuple** (x_i, y_i, z_i, w_i) to represent the ith applicant, and the data set would consist of the collection of n tuples, each tuple containing four items. In mathematics, a tuple is analogous to an ordered pair, but contains more than two items. If a data tuple contains more than numbers we say the its dimension is equal to the number of data items in the in the data tuple. We will provide examples of such higher dimensional data sets later in this book.

Scatter Plots

When the data is two-dimensional, it is convenient to represent the data points on a scatter plot, such as the black dots in figure 1.1. Even when the data has

Table 1.1: Population of the United States every ten years from 1790 through 1990. A copy of this data is in the csv file file **us-population.txt**.

x	y	x	y	x	y	x	y
1790	3929	1850	23191	1910	92228	1970	203302
1800	5308	1860	31433	1920	106021	1980	226542
1810	7239	1870	38558	1930	123202	1990	248709
1820	9638	1880	50189	1940	132164		
1830	12866	1890	62979	1950	151325		
1840	17069	1900	76212	1960	179323		

multiple dimensions, it is still sometimes convenient to do this, by plotting only two columns of the data, such as total monthly debt payment as a function of total monthly income. However, when we do this, it is important to remember that we are not illustrating the entire data set, and that some information is missing from the picture.

The easiest way to draw a scatter plot is using **matplotlib.pyplot**. This is the standard plotting library in Python. We typically import this library as **plt** to be consistent with other Python programmers. This allows us to refer to the library using the shorthand name **plot**. We also follow this line with the Jupyter **magic command**[5] "**% matplotlib inline**" (all magic commands begin with either one or two percent signs). This forces all plots to appear inline[6] instead of appearing in a pop-up window.

```
import matplotlib.pyplot
%matplotlib inline
```

The scatter plot in figure 1.2 was made with the following code. First, we extract the data into an array of x-values and an array of y-values.

```
xyvals=np.array(data)
xvals=xyvals[:,0]
yvals=xyvals[:,1]
```

It is not strictly necessary to do this as a separate step, but we will do it here for clarity. The reason we want to do this is because the standard function for plotting a scatter plot is

[5]Jupyter magic commands allow you to talk directly to Jupyter when you are using an iPython kernel. They are described at `https://ipython.readthedocs.io/en/stable/interactive/magics.html`.

[6]When we do this, we never need to use the **show** command, as we would in Python shell.

```
plt.scatter(xarray, yarray, ...)
```

where **xarray** and **yarray** are arrays that contain, respectively, all of the x and y values to be plotted. In other words, the first data point is (**xarray[0]**, **yarray[0]**); the second point is (**xarray[1]**, **yarray[1]**); and so forth. Both arrays must have the same length, or you will get an error. The ellipsis (...) represents the additional options that you can send to the **scatter** function. These allow you to control things like marker size, color, and shape.[7] Once the data has been defined in our separate arrays, we can generate the plot as shown with

```
plt.scatter(xvals,yvals,c="k")
plt.grid() # draw gridlines
# remove top and right line of frame
ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# set the font size of the tick labels
ax.tick_params(axis = 'both', which = 'major', labelsize = 12)
```

The **spines** command was used to make the top and right axes invisible. We did this by first accessing the current **axes** object in **pyplot**. This object is created whenever you make a plot, and can be accessed with **plot.gca()**.[8]

The code given so far is sufficient to draw the plot in the Jupyter notebook. In order to work properly, all of the **plt** commands must be in the same notebook cell. If you also want to save the plot as a file, additional commands are required. For example, to set the default figure size to 8 inches (wide) by 4 inches (tall), trim the extra white space, and save the figure to the file **myplotfile.pdf** file, use the code.

```
f=plt.gcf()
f.set_size_inches(8,4)
f.tight_layout()
f.savefig("myplotfile.pdf")
```

The above commands must be in the same cell as the **plt** commands that produce the plot. The plot will be drawn in the notebook and save to the fill **myplotfile.pdf**.

It is not necessary to convert the **DataFrame** to an **array** before making a plot.

[7] For a detailed list of all available options, see https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html.

[8] For an introduction to using **spines** see the *Ticks and Spines Demo* at https://matplotlib.org/examples/ticks_and_spines/spines_demo.html.

In fact, you can refer to the columns in the **DataFrame** by their names. It is possible to make a scatter plot with a single line:

```
plt.scatter(data["x"],data["y"])
```

You can follow this up with any other options you like, such as the ones we used previously. A word of caution is in order here: the variable names in a **csv** file can be deceptive. Any extra white space between the commas and the variable names in the first row of the file will be included in the variable name. When you display the columns the variable in the second column will look like it has a name '**"y"**, whereas really it might have a column name of **" y"**. To determine the column names you can use the function **DF.columns**, where **DF** is the name of your **DataFrame** variable.

Method of Least Squares

Our goal is to determine a **model** that describes the data. In this chapter we will use the **method of least squares** to find a best fit model. In the method of least squares we assume that the data points given by equation 1.1 are described by some function and that we know the form of the function up to some unknown **constants**, or **parameters**. We will write the model function as

$$y = f(x) \tag{1.6}$$

Then the **predicted** (or **modeled**) value $\hat{y}_i$ at each point x_i is given by

$$\hat{y}_i = f(x_i) \tag{1.7}$$

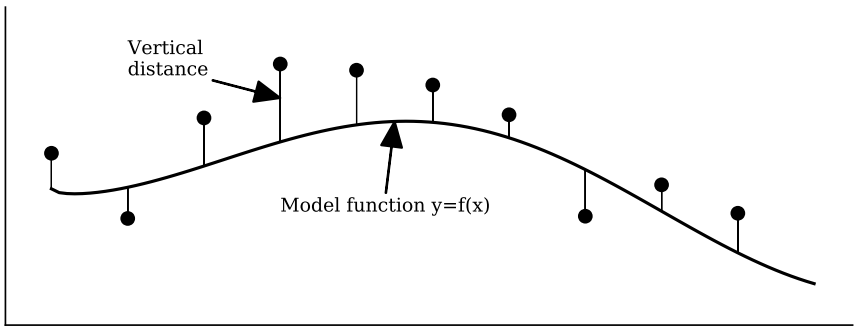
In general our observed data point $y_i \neq \hat{y}_i$. We call this difference the **residual error** Δy:

$$\Delta y = (\text{observed}) - (\text{predicted}) = y_i - f(x_i) \tag{1.8}$$

The residual errors are the signed lengths of the vertical lines in figure 1.2. If $y_i > f(x_i)$, then the residual error at x_i is positive; if $y_i < f(x_i)$, then the residual error at x_i is negative. If we sum the squared lengths of all of the residual errors we can define an **objective function** or **total energy function** as

$$\mathcal{E} = \frac{1}{2}\sum_{i=0}^{n-1}(f(x_i) - y_i)^2 \tag{1.9}$$

Figure 1.2: The method of least squares. Data points (x_i, y_i) are illustrated by black dots by the model $y = f(x)$ is illustrated by a smooth curve. Residual errors are indicated by the vertical lines joining the black dots with the model curve.



While we could could do a summation here with an absolute value of the error, rather than the squared error, it is generally more convenient to use the squared error.[9]

The sum is taken over all data points and is equal to one half of the **residual sum-squared** (RSS) modeling error:

$$\text{RSS error} = \sum_{i=0}^{n-1} (f(x_i) - y_i)^2 = 2\mathcal{E} \tag{1.10}$$

The factor of 1/2 in equation 1.9 is for convenience in deriving the equations for **least squared linear regression**. We could just have well omitted it and used the RSS error to define $\mathcal{E}$. We call $\mathcal{E}$ and objective or energy function because we want to find the best possible function $f(x)$, subject to some sort of constraints (to be determined later). The basic idea is this. First, we pick the general format of $f(x)$. It will depend on some sort of constants (also called parameters). We don't know the values of these parameters. The choice of function will depend on what we know about the problem domain. Some examples of such functions include the following:

[9]This is related to the mathematical question of which p-norm we want to use, and usually we want to use the 2-norm. It is also related to the fact that it is easier to differentiate a square than an absolute value.

$$\text{Linear: } f(x) = a + bx \tag{1.11}$$

$$\text{Polynomial: } f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k \tag{1.12}$$

$$\text{Hill: } f(x) = \frac{x^k}{a^k + x^k} \tag{1.13}$$

$$\text{Logistic: } f(x) = \frac{1}{1 + e^{-(a+bx)}} \tag{1.14}$$

The objective function depends on parameters in the model function. For example, if our model function is a Hill function (eq. 1.13), the objective function depends on the cooperativity k and half-activation strength a, and is $\mathcal{E}(a, k)$. If our model is linear (eq. 1.11), the objective function depends on the intercept and slope a and b as $\mathcal{E}(a, b)$. In the **method of least squares**, our goal is to find the parameters that minimize the total error. This is equivalent to minimizing the objective function. This is similar to an idea taken from physics, where an energy function is minimized to solved problems in dynamics.

In general, we are faced with two types of problems in the method of least squares:

1. If the objective function depends linearly on the constants in the problem, such as equations 1.11 or 1.12, it is possible to actually solve exactly for the best solution. The exact solution for the best fit to a straight line is usually discussed in introductory statistics books. This is a problem that is often solved in linear algebra or multivariate calculus books. It is a matter of setting the partial derivatives of the objective function taken with respect to each of the parameters equal to zero and solving the remaining system of linear equation for the unknown parameters.

2. If the objective function does not depend linearly on the constants in the problem, such as equations 1.13 or 1.14, in general it is not possible to find explicit solutions. This is because the system of equations obtained after solving for the partial derivatives is, in general, not linear. This means that some sort of iterative process must be used to find the optimum solution. In many cases, the implementation for solving these nonlinear equations already includes the iteration, so you don't even know that it is happening.

Mathematics of Linear Regression: Straight Lines

If the goal is to fit a straight line to our data then

$$f(x) = a + bx \tag{1.15}$$

where a, the y-intercept, and b, the slope, are undetermined parameters. The objective function is

$$\mathcal{E} = \frac{1}{2}\sum_{i=0}^{n-1}(a + bx_i - y_i)^2 \tag{1.16}$$

where (x_i, y_i) are the known data values as given by equation 1.1.

We can find an exact solution for the coefficients a and b. To do this, we need to set each of the partial derivatives $\partial\mathcal{E}/\partial a$ and $\partial\mathcal{E}/\partial b$ equal to zero to obtain a set of two equations in two unknowns. Differentiating with respect to a,

$$0 = \frac{\partial\mathcal{E}}{\partial a} = \frac{\partial}{\partial a}\frac{1}{2}\sum_{i=1}^{n}(a + bx_i - y_i)^2 \tag{1.17}$$

$$= \sum_{i=1}^{n}(a + bx_i - y_i) \tag{1.18}$$

$$= \sum_{i=1}^{n} a + \sum_{i=1}^{n} bx_i - \sum_{i=1}^{n} y_i \tag{1.19}$$

$$= na + b\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i \tag{1.20}$$

Similarly, if we differentiate with respect to b

$$0 = \frac{\partial\mathcal{E}}{\partial b} = \frac{\partial}{\partial b}\frac{1}{2}\sum_{i=1}^{n}(a + bx_i - y_i)^2 \tag{1.21}$$

$$= \sum_{i=1}^{n} x_i(a + bx_i - y_i) \tag{1.22}$$

$$= a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i y_i \tag{1.23}$$

Equations 1.20 and 1.23 give two two linear equations in two unknowns a and b. We can re-arrange them into a more standard form, as follows:

$$na + b\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \tag{1.24}$$

$$a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i \tag{1.25}$$

The algebra becomes a little bit easier to follow if make the following substitutions:

$$X = \sum_{i=1}^{n} x_i \tag{1.26}$$

$$Y = \sum_{i=1}^{n} y_i \tag{1.27}$$

$$A = \sum_{i=1}^{n} x_i^2 \tag{1.28}$$

$$C = \sum_{i=1}^{n} x_i y_i \tag{1.29}$$

Since everything on the right hand side of equations 1.26 through 1.29 is calculated from known data points (eq. 1.1), the values of X, Y, and AC, and C are fully determined constants that are already known and can be computed in advance. Using them in equations 1.24 and 1.25,

$$na + bX = Y \tag{1.30}$$

$$aX + bA = C \tag{1.31}$$

In matrix form,

$$\begin{bmatrix} n & X \\ X & A \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} Y \\ C \end{bmatrix} \tag{1.32}$$

Defining $M = \begin{bmatrix} n & X \\ X & A \end{bmatrix}$ then $M^{-1} = \frac{1}{\Delta}\begin{bmatrix} A & -X \\ -X & n \end{bmatrix}$ where $\Delta = \det A = nA - X^2$. The solution is then

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\Delta}\begin{bmatrix} A & -X \\ -X & n \end{bmatrix} \begin{bmatrix} Y \\ C \end{bmatrix} = \frac{1}{\Delta}\begin{bmatrix} AY - XC \\ -XY + nC \end{bmatrix} \tag{1.33}$$

Therefore the coefficients a and b are completely determined as:

$$a = \frac{AY - XC}{nA - X^2} = \frac{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i y_i \sum_{i=1}^{n} x_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{1.34}$$

$$b = \frac{XY - nC}{X^2 - nA} = \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i - n\sum_{i=1}^{n} x_i y_i}{\left(\sum_{i=1}^{n} x_i\right)^2 - n\sum_{i=1}^{n} x_i^2} \tag{1.35}$$

Linear Regression in Python

These equations are simple enough that we don't even have to use any special Python library to solve them (even though there is a Python library to solve this problem). We can define a function **linear_regression(x, y)** that takes as input two **numpy** arrays of the same length, **x** and **y**, and returns an order pair values **(intercept, slope)** representing the values of a and b. The array **x** must contain the values $x_0, x_1, \ldots, x_{n-1}$ from equation 1.1, and the array **y** must contain the corresponding values for $y_0, y_1, \ldots, y_{n-1}$. If the determinant is singular (i.e., if $\Delta = 0$), we return slope and intercept both equal to zero. Otherwise we return a pair of floating point numbers.

```
def linear_regression(x, y):
    X=sum(x)
    Y=sum(y)
    A=sum(x*x)
    C=sum(x*y)
    n=len(x)
    Del=n*A-X*X
    if Del==0:
        print("System is singular!")
        return((0,0))
    a=(A*Y-X*C)/Del
    b=(n*C-X*Y)/Del
    return((a,b))
```

We can run this function on the data arrays such as **xvals** and **yvals**, print out the results, and plot the resulting line along with the original scatter plot with the original scatter plot using the following four lines of code.

```
intercept, slope=linear_regression(xvals,yvals)
print(intercept, slope)
plt.scatter(data["x"],data["y"], c="k")
plt.plot([1,8],[intercept + slope*1, intercept+slope*8], c="k")
```

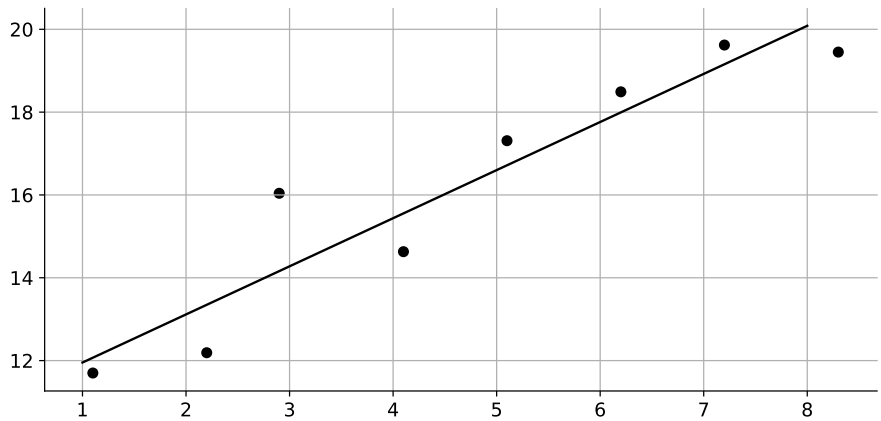
```
10.791852248092578  1.1615952025676408
```

Many different methods are built in to Python to perform linear regression. One of the easiest is **scipy.stats.linregress**.

```
from scipy.stats import linregress as LR
LR(xvals,yvals)
```

```
LinregressResult(slope=1.1615952025676397,
   intercept=10.791852248092571, rvalue=0.941717832188376,
   pvalue=0.00047355160326094863, stderr=0.16940229837546086)
```

Figure 1.3: Least squares fit and scatter plot of the same data shown in figure 1.1. The equation of the line is $y = 10.79 + 1.16x$ (rounding to 2 decimal places).



Besides the intercept and slope, the values returned include the following numbers that are sometimes use to evaluate the goodness of fit:

- the **r-value** of the fit. The r-value is related to the slope b as $b = r(s_y)/(s_x)$, where s_x and s_y are the standard deviations of the arrays of x values and y values, respectively. The value of r^2 gives the fraction of the variability that is described by a straight line, assuming that a straight line is the correct model, and that the data is distributed normally about the line. An r value of 0.94 is sometimes interpreted to mean that a fraction of $(0.94)^2 = 0.88$ or 88 percent of the variation in the data is described by the line.
- the **p-value** of a two sided hypothesis test $H_0 : b = 0$ vs $H_0 : b \neq 0$ (where b is the slope), as determined using a two-sided T-Test.
- the **standard error** of the slope as determined by the fit. This is sometimes interpreted as saying that the error in the slope is roughly $\pm 2(se)$. The factor of 2 is usually added to account for a 95% variability.

Linear Regression: Statistical Formulation

In is not unusual to see data sets such as equation 1.1 described by measures such as the **mean** μ, the **standard deviation** σ, and the **Pearson correla-**

tion coefficient r. Normally, the symbols μ and σ are actually **parameters** that describe large **populations** of single variables. In inferential statistics, symbols such as $\overline{x}$ and s are used to represent the mean and standard deviation of **random samples of observed data**, and extreme care is taken to distinguish between the statistics ($\overline{x}$, s) and the parameters (μ, σ). The goal of inferential statistics is to make educated guesses of the values of parameters (μ, σ) that describe a population (e.g., the height of all males in the United States between ages 21 and 29) from statistics ($\overline{x}$, s) taken from samples (e.g., a random selection of 700 American males in the same age group).

In a data-driven analysis we do not start with a sample. Instead, we are usually faced with a very large data set. This data set will not (necessarily) represent the entire population but we will, in general, pretend that it does. So we will most often use the symbols that represent populations (μ, σ) rather samples ($\overline{x}$, s) when describing our data set. In fact, if the data set is large enough it is sometimes treated as census of the population. In this case, traditional inferential statistics can be performed by drawing random samples from data a performing hypothesis testing or calculating confidence intervals as described in introductory statistics textbooks.

In the multi-variate (or bi-variate) case it is not uncommon to use a subscript symbols as μ_x, μ_y, σ_x, and σ_y to refer to the mean and standard deviation of the x and y values, respectively. The means of the x and y values of equation 1.1, respectively, are defined as

$$\mu_x = \frac{1}{n}\sum_{i=0}^{n-1} x_i, \qquad \mu_y = \frac{1}{n}\sum_{i=0}^{n-1} y_i \tag{1.36}$$

The standard deviation is the positive square root of the **variance**. The variances σ_x^2 and σ_y^2 are given by

$$\sigma_x^2 = \frac{1}{n-1}\sum_{i=0}^{n-1}(x_i - \overline{x})^2, \qquad \sigma_y^2 = \frac{1}{n-1}\sum_{i=0}^{n-1}(y_i - \overline{y})^2 \tag{1.37}$$

The Pearson correlation is given by

$$r = \frac{1}{(n-1)\sigma_x\sigma_y}\sum_{i=0}^{n-1}(x_i - \overline{x})(y_i - \overline{y}) \tag{1.38}$$

Our goal is to re-write the equations for the linear regression of y on x as $y = a + bx$, whose solution is given by equations 1.34 and 1.35, in terms of the statistics μ_x, μ_y, σ_x, σ_y, and r. In the following derivations we will assume that all summations are over i and they are over the indices $i = 0$ through $i = n - 1$.

If there is an x or y on the inside of a summation, we will assume that it is a shorthand for x_i. Consider the common denominators of equations 1.34 and 1.35:

$$
\begin{aligned}
n\sum(x^2) - \left(\sum x\right)^2 &= n\sum(x^2) - n^2\mu_x^2 && (1.39)\\
&= n\left[\sum(x^2) - n\mu_x^2\right] && (1.40)\\
&= n\left[\sum(x^2) - 2n\mu_x^2 + n\mu_x^2\right] && (1.41)\\
&= n\left[\sum(x^2) - \sum(2x\mu_x) + \sum\mu_x^2\right] && (1.42)\\
&= n\sum(x-\mu_x)^2 && (1.43)\\
&= n(n-1)\sigma_x^2 && (1.44)
\end{aligned}
$$

Consider the numerator of 1.35, the equation for the slope of b:

$$
\begin{aligned}
\sum x\sum y - n\sum(xy) &= -n\left(\sum(xy) - n\mu_y\mu_x\right) && (1.45)\\
&= -n\left(\sum(xy) - n\mu_y\mu_x - n\mu_x\mu_y + n\mu_x\mu_y\right) && (1.46)\\
&= -n\left(\sum(xy) - \mu_y\sum x - \mu_x\sum y + n\mu_x\mu_y\right) && (1.47)\\
&= -n\left(\sum(xy - x\mu_y - y\mu_x + \mu_x\mu_y)\right) && (1.48)\\
&= -n\sum(x-\mu_x)(y-\mu_y) && (1.49)\\
&= -n(n-1)\sigma_x\sigma_y r && (1.50)
\end{aligned}
$$

Substituting equations 1.44 and 1.50 into equation 1.35, the slope b is given by

$$
\begin{aligned}
b &= \frac{\sum x\sum y - n\sum(xy)}{\left(\sum x\right)^2 - n\sum x^2} && (1.51)\\
&= \frac{-n(n-1)\sigma_x\sigma_y r}{-n(n-1)\sigma_x^2} && (1.52)\\
&= \frac{\sigma_y}{\sigma_x}r && (1.53)
\end{aligned}
$$

Next, consider the numerator of equation 1.34 for the intercept a. Here we first substitute equation 1.44 into the denominator:

$$
a = \frac{\sum x^2\sum y - \sum xy\sum x}{n(n-1)\sigma_x^2} \qquad (1.54)
$$

First we will expand the first factor in the numerator in equation 1.54:

$$\sum x^2 = \sum (x-\mu_x)^2 + \sum 2\mu_x x - \sum \mu_x^2 \tag{1.55}$$
$$= (n-1)\sigma_x^2 + 2n\mu_x^2 - n\mu_x^2 \tag{1.56}$$
$$= (n-1)\sigma_x^2 + n\mu_x^2 \tag{1.57}$$

Using this result in equation 1.54 and setting $\sum_y = n\mu_y$ in the second factor in the numerator, gives

$$a = \frac{\big((n-1)\sigma_x^2 + n\mu_x^2\big)n\mu_y - \sum xy \sum x}{n(n-1)\sigma_x^2} \tag{1.58}$$

Expanding the $\sum xy$ into $\sum (x-\mu_x)(y-\mu_y)$ (as we did previously) gives

$$\sum xy = (n-1)r\sigma_x\sigma_y + n\mu_x\mu_y \tag{1.59}$$

Using this result and setting $\sum x = n\mu_x$ in equation 1.58 we obtain

$$a = \frac{\big((n-1)\sigma_x^2 + n\mu_x^2\big)n\mu_y - \big((n-1)r\sigma_x\sigma_y + n\mu_x\mu_y\big)(n\mu_x)}{n(n-1)\sigma_x^2} \tag{1.60}$$
$$= \frac{n(n-1)\sigma_x^2\mu_y - n(n-1)r\sigma_x\sigma_y\mu_x}{n(n-1)\sigma_x^2} \tag{1.61}$$
$$= \mu_y - \frac{r\sigma_y\mu_x}{\sigma_x} = \mu_y - b\mu_x \tag{1.62}$$

Thus we have the **statistical formulation of linear regression**, which can be expressed very concisely as:

$$b = r\sigma_y/\sigma_x \tag{1.63}$$
$$a = \mu_y - b\mu_x \tag{1.64}$$

We could use these formulas to define an alternative version of our linear regression function that also prints out the correlation:

```
def linear_regression2(x,y):
    mux=np.mean(x)
    muy=np.mean(y)
    sx=np.std(x,ddof=1)
    sy=np.std(y,ddof=1)
    r=np.corrcoef(x,y)[1,0]
    b=r*sy/sx
    a=muy-b*mux
    print("intercept a=",a)
    print("slope b=    ",b)
    print("correlation=",r)
    return(a,b)
linear_regression2(xvals,yvals)
```

```
intercept a= 10.791852248092571
slope b=     1.1615952025676397
correlation= 0.9417178321883759

(10.791852248092571, 1.1615952025676397)
```

Linear Regression: Evaluation

When we send a model – such as linear regression – out into the world to be used, it will be applied to data that we have never seen. This data that we have never seen is called **test data** or a **test set**. The data that we use to build a model is then called **training data** or a **training set**. Since the data in the test set is not used to build the model there is no guarantee that a model will perform very well when it is used with the test set. Ideally, we would like to check a model out on this test data before we release it. Since we don't have the test data, we can't do this.

There are, in general, two different methods that are used to get around this problem. Both involve putting aside some of the training data and using it as a test set.

- Split your data set up into two non-overlapping subsets. The most common split is 80%/20%. Designate the larger subset **training subset** and the smaller subset as a **test subset**. Build your model with only the training subset. Then test it with the test subset. This will simulate using real world data, because nothing in the test subset was used to generate the model.
- Split your data set into N non-overlapping subsets. Typically either $N = 5$ or $N = 10$ subsets are used. Each of these subsets are called **folds**, e.g., 5-fold or 10-fold splitting. Use $N - 1$ folds as the training set and the Nth fold as the test set. Repeat this process, using each fold as a test set, and compute the average of the $N - 1$ errors. This is sometimes called **leave out one cross validation** (LOOCV).[10]

The most commonly used measure of fit for linear regression is the **mean sum of squares** (MSS). This is the residual sum of squares(RSS, eq. 1.10) divided

[10] We focus on train/test splits for the remainder of this text and leave LOOCV to the student to explore on their own.

by the number of data points.

$$\text{MSS} = \frac{1}{n}\sum_{i=0}^{n-1}(f(x_i) - y_i)^2 = \frac{1}{n}\sum_{i=0}^{n-1}(a + bx_i - y_i)^2 \tag{1.65}$$

The mean sum of squares defined in this manner is calculated using the entire data set. However, it is more common (and better practice) to separate your original data into a **test data set** and a **training data set** (see figure 1.4). Ideally these two data sets should be non-intersecting. If the original data set set is large enough you can then do the following:

1. Set aside approximately 20% to 25% of you data as testing data. These points should be selected randomly throughout the data set. The actual number can vary; the numbers 20% to 25% are arbitrary.

2. Designate the remaining data as your training set.

3. Calculate you fit parameters as if your training set were your entire data set. In other words, do not use the entire data set to calculate a and b. Just use the data points in the training set.

   In the *theoretical* formulation this might required a replacement of variables such as n with something like $n_{training}$, the size of the training set; and i, the index into the data set, with j, an index into the training set.

   In the *software implementation* it is crucial to implement this (and virtually everything else we talk about in this book) as a function. Then (e.g., in a function such as **`linear_regression`**, **`linear_regression2`**, **`scipy.stats.linregress`**) the only change we need to make is that we would need to pass in the names of arrays that contain the training data rather than the arrays that contain the entire data set.

4. Using the parameters found in step 3, calculate the value of the fit on each point in the test data set. Use these calculations to determine the MSS.

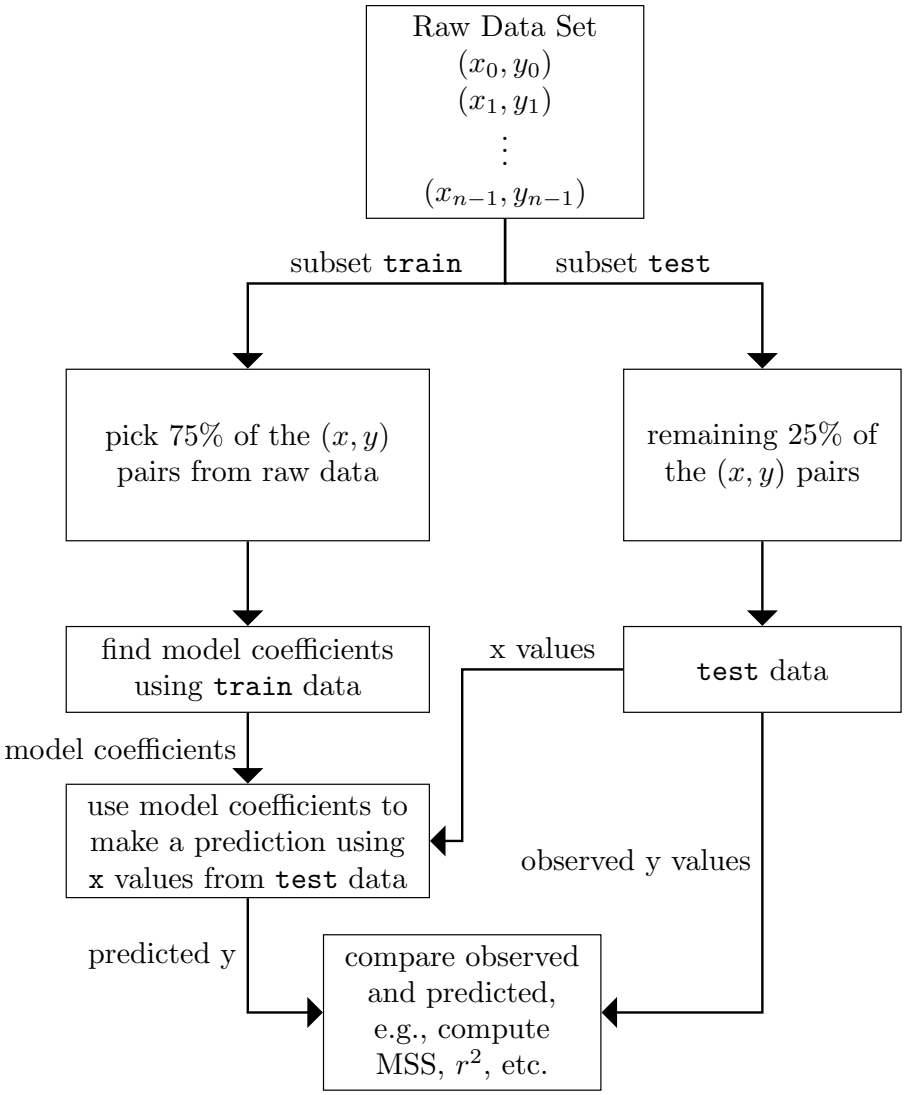
$$\text{MSS} = \frac{1}{n}\sum_{\text{Test Data}}(a + bx_i - y_i)^2 \tag{1.66}$$

As an example, we will consider the *machine data* set at the UCI Machine Learning data repository.[11] This is a **`csv`** file but the columns are unlabeled.[12]

[11] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository `http://archive.ics.uci.edu/ml`. Irvine, CA: University of California, School of Information and Computer Science.

[12] The file can be downloaded from `https://archive.ics.uci.edu/ml/machine-learning-databases/cpu-performance/machine.data`.

Figure 1.4: Concept of the train-test split.



We can read this file directly into our notebook by specifying the URL of the file instead of the file name in **pd.read_csv**. We must tell **pandas** at read time that the columns are unlabeled so that it does not accidentally use the first line of the data file as the column headers. Then we can manually label to columns, if we want to. Strictly speaking, we don't have to label the columns, as they will be numbered, and we could treat this like a spreadsheet or an array, and just refer to the column number.

```
data = pd.read_csv("https://archive.ics.uci.edu/ml/machine-
       learning-databases/cpu-performance/machine.data",
       header=None)
data.columns=["vendor","Model","MYCT","MMIN", "MMAX","CACH",
       "CHMIN","CHMAX","PRP","ERP"]
```

Each line refers to a different type of computer, and the columns give the computer vendor; cycle type (nSec); minimum and maximum main memory (kB); cache memory (kB); minimum and maximum channels; and published and estimated relative performance. At the present, we will be interested in the columns labeled **PRM** and **ERP** because these data are approximately linearly related. This is what the first part of the data set looks like:

	vendor	Model	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP	ERP
0	adviser	32/60	125	256	6000	256	16	128	198	199
1	amdahl	470v/7	29	8000	32000	32	8	32	269	253
2	amdahl	470v/7a	29	8000	32000	32	8	32	220	253
3	amdahl	470v/7b	29	8000	32000	32	8	32	172	253
4	amdahl	470v/7c	29	8000	16000	32	8	16	132	132

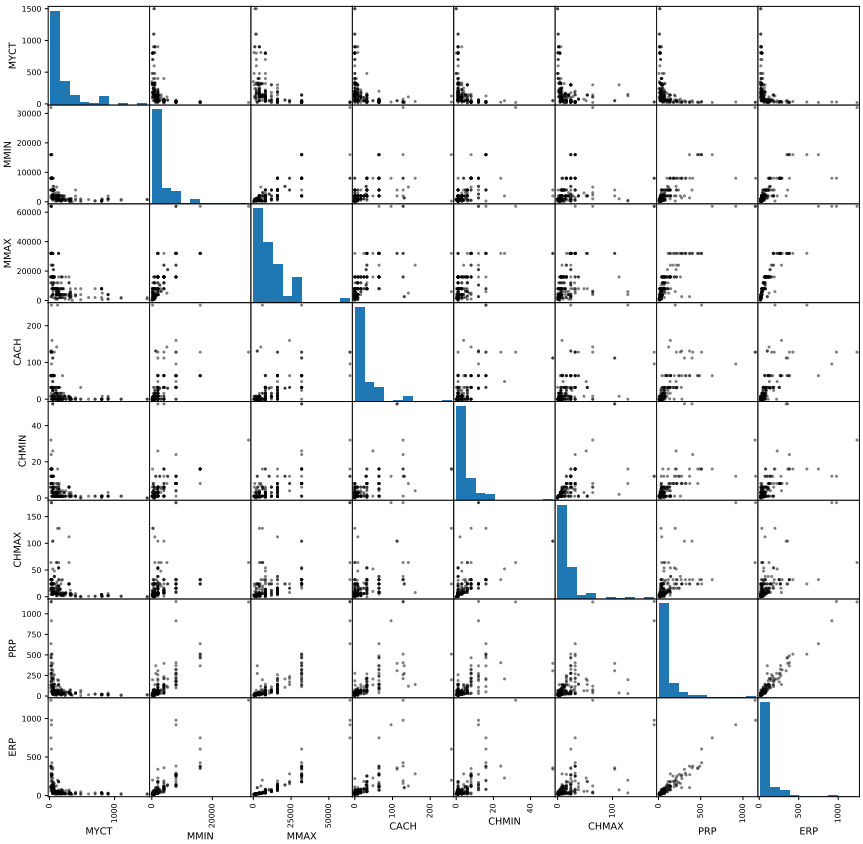
When we have a data set with multiple columns it is sometimes interesting to plot every column as a function of every other function. In other words, pick any column, and use it as the x-axis of a plot, and then pick any other column and use it as the y-axis of a plot. We can do this with the **pandas** function[13] **pd.plotting.scatter_matrix**. For example, to make a scatter matrix plot of the computer data,

```
rr = pd.plotting.scatter_matrix(data,
     figsize=(15,15),
     marker="o",s=10,c="black")
```

The only required argument is the name of the **DataFrame** (the first argument). The other arguments are optional. Here we have specified **figsize** (to specify plot dimensions and aspect ratio); **marker** (to indicate which scatter plot

[13]Details are given online at https://pandas.pydata.org/pandas-docs/stable/generated/pandas.plotting.scatter_matrix.html.

Figure 1.5: A **pandas** scatter matrix plot of the computer data set.



marker[14] to use); and **c** which is used to specify the color of the marker. The diagonals in the scatter plot show histograms of the individual variables. The resulting **scatter_matrix** for **data** is illustrated in figure 1.5. We can see from the frames adjacent to the bottom left that the columns we have labeled **PRP** and **ERP** look approximately linear. This suggests that it might be interesting to examine this relationship more closely.

We'll begin by separating the data set into a test set and a training set. We will reserve 75% of data for training, and 25% for testing. We'll do this by randomly selecting indices of the data. First, we pull out the x and y values, and point to them with Python variables **xvals** and **yvals**

```
xvals=data["PRP"]
yvals=data["ERP"]
```

[14] A complete list of scatter plot markers is given online at `https://matplotlib.org/api/markers_api.html`.

Then we will use the function **random.shuffle** to shuffle the list **[0, 1, ...,n-1]**, where **n** is the length of the data set. The **shuffle** function randomly rearranges the values of its argument in place, much like shuffling a deck of cards. We will call the list of indices **ind**, both before it is shuffled and after it is shuffled.

```
import random
n=len(data)
ind=list(range(n))
random.shuffle(ind)
```

At this point, the list **ind** still contains each of the integers **[0, 1, ..., n-1]**, but they have been rearranged in a random order. Since there are **n** total indices, then 75% of the indices (rounded to the nearest integer) is given by **ntrain=round(0.75*n)**. The idea is to extract the data pairs with indices given by the first **ntrain** numbers in the shuffled **ind** as the training set, and use the remaining data as the test set. Python lets us do this as follows:

```
ntrain=round(.75*n)
xtrain=xvals[ind[:ntrain]]
ytrain=yvals[ind[:ntrain]]
xtest=xvals[ind[ntrain:]]
ytest=yvals[ind[ntrain:]]
```

In this code, **ind[:ntrain]** gives the first 75% of the indices in the array **ntrain**, while **ind[ntrain:]** gives the remaining 25%. There is no-overlap, and all indices are included. By referring to **xvals[list_of_indices]** Python picks out those values in **xvals**, and only those values in **xvals**, that are expressed in **list_of_indices**. Here we have replaced **list_of_indices** with **ind[:ntrain]** to get a training set, and **ind[ntrain:]** to get a test set.

First, we'll use our old friend **scipy.stats.linregress** on the training set:

```
from scipy.stats import linregress as LR
b,a,r,p,stderr=LR(xtrain,ytrain)
print(a,b,r)
```

```
-0.2168130767260834 0.9246615087756963 0.9653789783330483
```

This tells us that the regression on the training set gives (rounded to 2 decimals)

$$ERP = 0.92 - 0.22 \times PRP \tag{1.67}$$

with a Pearson correlation of $r = 0.97$. The MSS Error (eq. 1.66) for the linear regression can be calculated as

```
def MSS(a,b,X,Y):
    n=len(X)
    RSS=sum([(a+b*x-y)**2 for x,y in zip(X,Y)])
    return(RSS/n)
MSS(a,b,xtest,ytest)
```

```
1233.7745814823706
```

The MSS value by itself is not really meaningful if we are only interested in linear models. However, if we are comparing multiple models, then it becomes a useful measure of model quality, e.g., to distinguish between a linear, multi-linear, or quadratic model. We will discuss this is subsequent chapters. For now, the more useful measure is the r-value. We mention the MSS here for completeness and as an introduction to linear regression in **sklearn**.

The standard machine learning package in Python is called **scikit-learn**.[15] The library name is actually **sklearn**; the name **scikit-learn** actually refers to the URL where the package is documented.[16] Many of the functions in **sklearn** have very similar calling sequences, and are very general. The way in which these functions are used will seem very different from the calling sequences in typical Python functions, e.g., in **numpy**, and can be quite daunting for beginners. It can be particularly confusing when you want to do a very simple task such as linear regression of y on x, because the implementation is much more general.

We are going to use two sub-libraries in **sklearn** at this time. These will be **sklinearn.linear_model** and **sklearn.metrics**. The **linear_model** sub-library contains a number of functions including the one we want, which is called **LinearRegression**. The **metrics** sub-library includes various functions that are used to evaluate the relative success of a learning algorithm. For now we are only interested in the calculation of the MSS (performed by **mean_squared_error** and r^2 (the square of the Pearson correlation coefficient, calculated by **r2_score**).

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

The **LinearRegression** class has several methods that will perform what the linear regression. We initialize the class with with a **fit** method. The general format of a **fit** method is

[15] Pedregosa et al (2011) Scikit-learn: Machine Learning in Python. *JMLR* **12**:2825-2830.
[16] See `http://scikit-learn.org/stable/` for full documentation of the latest versions of **sklearn**.

```
fit(training, test)
```

where **training** is a multi-variate training set, and test is a multi-variate test set. In **sklearn**, the data sets are expected to have multiple columns, to represent the different components of the independent variable. These components are called features; we will discuss this in the next chapter. When y is a function of a single variable x then we have to do some massaging of the original data to get it into a format that **sklearn** is happy with. We do that with the **numpy** array **reshape** command with arguments (-1,1). Assume that we already have **xtrain** and **ytrain** as defined previously. Then our **sklearn**-compatible training sets **XT** and **YT** are derived as:

```
XT=np.array(xtrain).reshape(-1,1)
YT=np.array(ytrain).reshape(-1,1)
```

The linear regression is calculated using the **fit** method. The Pearson r value, the intercept (a) and slope (y) are found using the **score**, **intercept_** and **coef_** methods. The latter two methods are returned as arrays because in general **fit** works with a multidimensional data set.

```
r=LinearRegression().fit(XT,YT)
r=LinearRegression().fit(XT,YT)
r.intercept_[0], r.coef_[0,0]
```

```
-0.21681307672599814, 0.9246615087756955)
```

The resulting linear fit is the same as that obtained with **linregress**, except for numerical errors in the higher precision digits.

Most classes in **sklearn** also have a **predict** method. After doing a fit, we can then make a prediction, using a training set, based on the fit. For a one-dimensional data set, we need to massage the test set as well before we use **predict**:

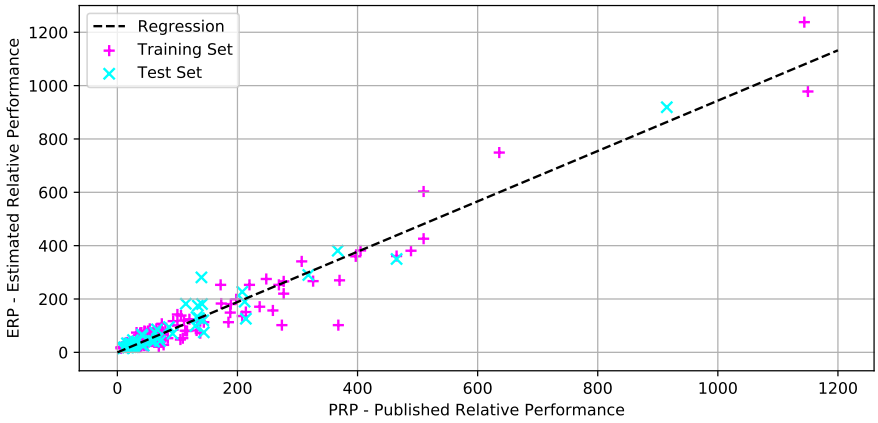
```
TX=np.array(xtest).reshape(-1,1)
TY=np.array(ytest).reshape(-1,1)
P=r.predict(TX)
```

The array **P** will contain values of y as determined by the fit $y = a + bx$ at each of the x values in **xtest**. We can calculate the MSS (eq. 1.66) using either the built-in function

```
mean_squared_error(TY,P)
```

```
1233.7745814823718
```

Figure 1.6: Least squares regression of ERP on PRP showing test and training points with different markers.



or by a manual calculation with **numpy** array manipulation:

```
sum([x*x for x in ((TY-P).T)[0]])/len(TY)
```

```
1233.7745814823716
```

We can also calculate r^2 based on the *test data set*. This is more meaningful that the r^2 value calculated previously because it does not used any data that was used to obtain the fit:

```
r2_score(TY,P)
```

```
0.9408303351597077
```

Finally, we may want to plot all of the data along with the fit on a single picture.

```
plt.scatter(xtrain,ytrain,c="magenta",marker="+",
   s=50,label="Training Set")
plt.scatter(xtest,ytest,c="cyan",marker="x",s=50,
   label="Test Set")
plt.plot([0,1200],[ERP(0),ERP(1200)],label="Regression",
   ls="--",c="k")
plt.grid()
plt.xlabel('PRP - Published Relative Performance')
plt.ylabel("ERP - Estimated Relative Performance")
plt.legend()
```

The resulting plot is illustrated in figure 1.6.

Here we put together the entire sequence of Python code we used to complete and evaluate the linear regression using **sklearn**.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
import random
import pandas as pd

# read the data into a Pandas DataFrame

data = pd.read_csv("https://archive.ics.uci.edu/ml/machine-
learning-databases/cpu-performance/machine.data",
header=None)
data.columns=["vendor","Model","MYCT","MMIN", "MMAX","CACH",
"CHMIN","CHMAX","PRP","ERP"]

# select the x and y values

xvals=data["PRP"]
yvals=data["ERP"]

# Shuffle indices for test/train split

n=len(data)
ind=list(range(n))
random.shuffle(ind)

ntrain=round(.75*n)
xtrain=xvals[ind[:ntrain]]
ytrain=yvals[ind[:ntrain]]
xtest=xvals[ind[ntrain:]]
ytest=yvals[ind[ntrain:]]

# format test and training sets for sklearn

XT=np.array(xtrain).reshape(-1,1)
YT=np.array(ytrain).reshape(-1,1)
TX=np.array(xtest).reshape(-1,1)
TY=np.array(ytest).reshape(-1,1)

# perform the regression and print some metrics
r=LinearRegression().fit(XT,YT)
r=LinearRegression().fit(XT,YT)
print("intercept=",r.intercept_[0], "slope=",r.coef_[0,0])

P=r.predict(TX)
MSE = mean_squared_error(TY,P)
R2=r2_score(TY,P)
print("MSE=",MSE, "R2=",R2)
```

Chapter 2

Multilinear Regression

In the examples of chapter 1, the analyses could be simplified to two variables. We wrote the data as ordered pairs, as in equation 1.1, and tried to find a linear relation in the form of $y = f(x) = a + bx$ that minimized the objective function 1.9. Sometimes a simple linear relationship is insufficient to express the relationship between the variables. We have several different options:

1. There may be other variables we can consider. If the original data is given as a large table with p columns, for example, we may want to consider a multivariate function of the form

$$y = f(x_0, x_1, \ldots, x_{p-1}) \tag{2.1}$$

   In this chapter, we will consider linear functions of this form:

$$y = c_0 x_0 + c_1 x_1 + \cdots c_{p-1} x_{p-1} + c_p \tag{2.2}$$

   where $c_0, \ldots, c_p$ are undetermined constants.

2. We may want to consider nonlinear functions $y = f(x)$, such as polynomials. Sometimes the choice of function is driven by the scientific discipline that the data arises from. In other cases, we have no idea what the form of $f(x)$ should be, and we want to try many different functions to try to find the best fit.

3. We may want to fit a piecewise-linear function $y = f(x)$ to approximate a non-linear function. This is often nearly as accurate as a nonlinear function. !

Mathematics of Multilinear Regression

As before, let n be the number of observations or data points. Suppose each data point has $p+1$ dimensions. Each data point is a tuple

$$(x_0, x_1, \ldots, x_{p-1}, y) \tag{2.3}$$

When $p = 1$ we have normal (x, y) data pairs. When we are usually presented with the data in a multi-columnar spreadsheet, we have to identify one particular column as the **response variable** (y) and the other columns as the **explanatory variables** (x_0, x_1, $\ldots$, x_{p-1}). Mathematically, it is more convenient to represent these vectors as follows:

$$\begin{aligned} y_0 &= f([x_0, x_1, \ldots, x_{p-1}]_0) = f(\mathbf{x}_0) \\ y_1 &= f([x_0, x_1, \ldots, x_{p-1}]_1) = f(\mathbf{x}_1) \\ &\ldots \\ y_{n-1} &= [f(x_0, x_1, \ldots, x_{p-1}]_{n-1}) = f(\mathbf{x}_{n-1}) \end{aligned} \tag{2.4}$$

The y values are the **observed response variables** and the $\mathbf{x}$ vectors are the **observed explanatory variables**. Each of the components or column variables are sometimes called **features**, and the vector $\mathbf{x}_i = [x_0, \ldots, x_{p-1}]_i$ is called a **feature vector**.[1] The matrix formed by the collection of all feature vectors is called a **feature matrix**. The ith row of the feature matrix consists of the feature (row) vector $\mathbf{x}_i$. Here we are using superscripts to indicate the observation or row index, and subscripts to indicate the feature. Our goal is to use least squares linear regression to find the unknown parameters a and $b_0, \ldots, b_{n-1}$ in

$$f(\mathbf{x}) = a + b_0 x_0 + b_1 x_1 + \cdots b_{p-1} x_{p-1} \tag{2.5}$$

that minimize the objective function

$$\mathcal{E} = \sum_{k=0}^{n-1} |f(\mathbf{x}_k) - y_k|^2 \tag{2.6}$$

To avoid double subscripts, let us denote the jth component of the vector $\mathbf{x}_i$ as x_j^i (instead of, say, $\mathbf{x}_{i,j}$) If each of the data points in equation 2.4 were to have some residual error $r_i = y_i + f(x_i)$ when trying to match the curve given by

[1] Note that we are describing feature vectors here as row vectors, whereas we typically, and elsewhere in this book, usually refer to vectors in their column form.

equation 2.5, then we would have the following system of equations.

$$\begin{aligned}
a + b_0x_0^0 + b_1x_1^0 + \cdots + b_{p-1}x_{p-1}^0 &= y_0 + r_0 \\
a + b_0x_0^1 + b_1x_1^1 + \cdots + b_{p-1}x_{p-1}^1 &= y_1 + r_1 \\
a + b_0x_0^2 + b_1x_1^2 + \cdots + b_{p-1}x_{p-1}^2 &= y_2 + r_2 \\
&\vdots \\
a + b_0x_0^{n-1} + b_1x_1^{n-1} + \cdots + b_{p-1}x_{p-1}^{n-1} &= y_{n-1} + r_{n-1}
\end{aligned} \tag{2.7}$$

Since each x_c^r refers to feature c of the observation vector $\mathbf{x}^r$, in terms of a data matrix, r is the row index and c is the column index. We can rewrite this as a matrix equation

$$\mathbf{Ab} = \mathbf{y} + \mathbf{r} \tag{2.8}$$

where the matrix $\mathbf{A}$ and column vectors $\mathbf{b}$, $\mathbf{y}$ and $\mathbf{r}$ are defined below:

$$\underbrace{\begin{bmatrix} 1 & x_0^0 & x_1^0 & \cdots & x_{p-1}^0 \\ 1 & x_0^1 & x_1^1 & \cdots & x_{p-1}^1 \\ 1 & x_0^2 & x_1^2 & \cdots & x_{p-1}^2 \\ & & & & \\ 1 & x_0^{n-1} & x_1^{n-1} & \cdots & x_{p-1}^{n-1} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} a \\ b_0 \\ b_1 \\ \vdots \\ b_{p-1} \end{bmatrix}}_{\mathbf{b}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}}_{\mathbf{y}} + \underbrace{\begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{bmatrix}}_{\mathbf{r}} \tag{2.9}$$

We can simplify the matrix notation if we define the jth column vector of $\mathbf{A}$ as $\mathbf{a}_j$:

$$\mathbf{a}_j = \begin{bmatrix} x_j^0 \\ x_j^1 \\ \vdots \\ x_j^{n-1} \end{bmatrix} \tag{2.10}$$

Then we can write the matrix $\mathbf{A}$ using matrix partition notation as

$$\mathbf{A} = \begin{bmatrix} \mathbf{1} & | & \mathbf{a}_0 & | & \mathbf{a}_1 & | & \cdots & | & \mathbf{a}_{p-1} \end{bmatrix} \tag{2.11}$$

The vertical line means that the vectors written between the vertical lines should be expanded in place to obtain the full expression for the matrix. Here $\mathbf{1}$ is a column vector consisting solely of 1's. Then if we define $b_{-1} = a$ and also define $\mathbf{a}_{-1} = \mathbf{1}$ this can be written as

$$\begin{aligned}
\mathbf{Ab} &= a\mathbf{1} + b_0\mathbf{a}_0 + b_1\mathbf{a}_1 + \cdots + b_{p-1}\mathbf{a}_{p-1} && (2.12) \\
&= b_{-1}\mathbf{a}_{-1} + b_0\mathbf{a}_0 + b_1\mathbf{a}_1 + \cdots + b_{p-1}\mathbf{a}_{p-1} && (2.13) \\
&= \sum_{j=-1}^{p-1} b_j\mathbf{a}_j && (2.14)
\end{aligned}$$

Our goal is to minimize the objective function, summed over the training set, of

$$\mathcal{E}(b_0, b_1, \ldots, b_{p-1}) = \sum_{i=0}^{n-1} \left|(\mathbf{Ab})_i - y_i\right|^2 \tag{2.15}$$

$$= (\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}(\mathbf{Ab} - \mathbf{y}) \tag{2.16}$$

We do this by setting $\partial\mathcal{E}/\partial b_i = 0$ and solving for the unknown parameters b_i. By the product rule,

$$\frac{\partial\mathcal{E}}{\partial b_i} = \left[\frac{\partial}{\partial b_i}(\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}\right](\mathbf{Ab} - \mathbf{y}) + (\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}\left[\frac{\partial}{\partial b_i}(\mathbf{Ab} - \mathbf{y})\right] \tag{2.17}$$

From equation 2.14,

$$\frac{\partial(\mathbf{Ab} - \mathbf{y})}{\partial b_i} = \frac{\partial(\mathbf{Ab})}{\partial b_i} = \mathbf{a}_i \tag{2.18}$$

Hence

$$0 = \frac{\partial\mathcal{E}}{\partial b_i} = (\mathbf{a}_i)^{\mathrm{T}}(\mathbf{Ab} - \mathbf{y}) + (\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}\mathbf{a}_i \tag{2.19}$$

Distributing the $\mathbf{a}_i^{\mathrm{T}}$ into the first term, and expanding the transpose and multiplying out the second gives

$$0 = \mathbf{a}_i^{\mathrm{T}}\mathbf{Ab} - \mathbf{a}_i^{\mathrm{T}}\mathbf{y} + \mathbf{(Ab)}^{\mathrm{T}}\mathbf{a}_i - \mathbf{y}^{\mathrm{T}}\mathbf{a}_i \tag{2.20}$$

But $(\mathbf{Ab})^{\mathrm{T}}\mathbf{a}_i = \mathbf{b}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{a}_i$ is a scalar, and the transpose of a scalar is itself

$$((\mathbf{Ab})^{\mathrm{T}}\mathbf{a}_i)^{\mathrm{T}} = (\mathbf{b}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{a}_i)^{\mathrm{T}} = \mathbf{a}_i^{\mathrm{T}}\mathbf{Ab} \tag{2.21}$$

Similarly,

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{y} = \mathbf{y}^{\mathrm{T}}\mathbf{a}_i \tag{2.22}$$

because each side of the equation is a scalar. Hence

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{y} = \mathbf{a}_i^{\mathrm{T}}\mathbf{Ab}, \qquad i = 0, 1, \ldots, p-1 \tag{2.23}$$

Transforming back to the full matrix form gives

$$\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{Ab} \tag{2.24}$$

The solution can be written analytically as

$$\mathbf{b} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y} \tag{2.25}$$

Rather than using this form to solve the system direction, however, it generally more efficient to use a linear solver on equation 2.24 and solve the system using a standard linear method, such as Gaussian reduction. Equation 2.24 are called the **normal equations** for the system. They give a linear system that can be

solved exactly for the coefficients b_i that minimize the objective function. The matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is sometimes called the **normal matrix N** of the system.

The normal matrix can be a bit difficult to understand in higher dimensions, so we consider an example with three features. Suppose that the feature vectors have three components denoted by $\mathbf{x} = (u, v, w)$. We will still denote the response variable by y. Then

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ \vdots & & & \\ 1 & x_{n-1} & y_{n-1} & z_{n-1} \end{bmatrix} \tag{2.26}$$

and

$$\mathbf{N} = \mathbf{A}^{\mathrm{T}}\mathbf{A} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{n-1} \\ y_0 & y_1 & \cdots & y_{n-1} \\ z_0 & z_1 & \cdots & z_{n-1} \end{bmatrix} \begin{bmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ \vdots & & & \\ 1 & x_{n-1} & y_{n-1} & z_{n-1} \end{bmatrix} \tag{2.27}$$

$$= \begin{bmatrix} n & \sum x_i & \sum y_i & \sum z_i \\ \sum x_i & \sum x_i^2 & \sum x_i y_i & \sum x_i z_i \\ \sum y_i & \sum x_i y_i & \sum y_i^2 & \sum y_i z_i \\ \sum z_i & \sum x_i z_i & \sum y_i z_i & \sum z_i^2 \end{bmatrix} \tag{2.28}$$

Thus the normal matrix is symmetric. The first row contains the sums of individual features summed over all input vectors. The remaining rows contains sums $\sum_k x_i^k x_j^k$ on the ij^{th} position, where there the sum is over the observation vectors k.

Multilinear Regression with Numpy

As with linear regression, it is fairly easy to write our own solver for this in Python using the standard linear system solver **`numpy.linalg.solve`**. This function takes a matrix **`A`** and a vector **`b`** and returns the solution of the equation $\mathbf{Ay} = \mathbf{b}$ as the output of **`solve(A,b)`**.

The following function takes a multi-factorial **`array`** of x values and an one-dimensional **`array`** of y values and returns the normal matrix $\mathbf{N}$=$\mathbf{A}^{\mathrm{T}}\mathbf{A}$ and the target vector $\mathbf{c} = \mathbf{A}^{\mathrm{T}}\mathbf{Y}$. From equation 2.24 we need to solve $\mathbf{c} = \mathbf{Nb}$ for the unknown coefficients **b**.

```
def Normal_Matrix(xdata, ydata):
    # make sure the data is a 2-dimensional array

    if xdata.ndim != 2:
        print("Not a 2x2 array")
        return(float("NAN"))

    # add a column of 1's to the front of the data

    nrows,ncols=xdata.shape
    column_ones=np.ones((nrows,1))
    A=np.hstack((column_ones, xdata))

    # calculate the transpose
    AT=A.T

    # form the normal matrix
    N=AT.dot(A)

    # form the target vector
    ATy = AT.dot(ydata)
    return(N,ATy)
```

Let's apply this to the computer hardware data set. As in chapter 1, we read the data set directly from the internet. We add the column names to represent the hardware properties (see page 19)

```
import pandas as pd
data = pd.read_csv("https://archive.ics.uci.edu/ml/
   machine-learning-databases/cpu-performance/machine.data",
   header=None)
data.columns=["vendor","Model","MYCT","MMIN","MMAX",
   "CACH","CHMIN","CHMAX","PRP","ERP"]
```

Next, we'll extract the desired x and y data as **numpy** arrays and used **random** shuffling of indices to create the test and training sets. The y variable will be given by the column labeled **PRP**; the x variable features will be given by the size columns **MYCT**, **MMIN**, **MMAX**, **CACH**, **CHMIN**, and **CHMAX**. To extract the necessary data:

```
import numpy as np
import random
X=np.array(data[["MYCT","MMIN","MMAX","CACH","CHMIN","CHMAX"]])
Y=np.array(data["PRP"])
ndata=len(X)
```

In the next block of code, we will shuffle indices to create test and training sets. By randomly rearranging the numbers 0 through n, we can get a random

selection of 75% of the indices by picking the first 75% of the shuffled indices. We will pick our training set by selecting using the first 75% of the shuffled indices. Here we only define the indices of the test and training set.

```
indices=list(range(ndata))
random.shuffle(indices)
ntrain=round(.75*ndata)
training_indices=indices[:ntrain]
test_indices=indices[ntrain:]
```

Next we actually select the test and training set

The **reshape** is discussed in chapter 1 (page 23).

```
XTRAIN=X[training_indices]
YTRAIN=Y[training_indices].reshape(-1,1)
XTEST=X[test_indices]
YTEST=Y[test_indices].reshape(-1,1)
```

We can apply the function **Normal_Matrix** that we have written to this training set to find a linear fit , by using **XTRAIN** and **YTRAIN** as input.

```
N, ATY=Normal_Matrix(XTRAIN,YTRAIN)
import numpy.linalg
numpy.linalg.solve(N,ATY)
```

```
array([[-5.57949071e+01],
       [ 4.86061221e-02],
       [ 1.25405020e-02],
       [ 6.31709086e-03],
       [ 7.04575470e-01],
       [-1.61978327e+00],
       [ 1.69010283e+00]])
```

Thus the multilinear fit is approximately (with some rounding to save space):

$$y = -56 + 0.049x_0 + 0.013x_1 + 0.0063x_2 + 0.70x_3 - 1.62x_4 + 1.69x_5$$

In the context of the original problem this can be interpreted as follows:

$$\begin{aligned}\text{PRP} = &- 56 + 0.049 \times \text{MYCT} + 0.013 \times \text{MMIN} + 0.0063 \times \text{MMAX} \\ &+ 0.70 \times \text{CACH} - 1.62 \times \text{CHMIN} + 1.69 \times \text{CHMAX}\end{aligned}$$

In order to determine a goodness of fit, we might want to find the MSS error on the test data. To do this, we need to make a prediction. A predicted value corresponding to the ith test vector $\mathbf{x}^k$ is given by

$$\hat{y}_i = b_{-1} + \sum_{k=0}^{p-1} \mathbf{x}_k^i \tag{2.29}$$

where b_{-1} is the constant coefficient and $\mathbf{x}_k^i$ is the kth feature of test vector $\mathbf{x}^i$. The notation can become a little confusing here because there are p features but $p+1$ coefficients. In Python, the coefficients end up being stored with indices $0, 1, \ldots, p$. The constant coefficient is the one with index zero. Among the remaining coefficients, we need to multiply **b[i]** times feature **i-1**. To avoid this confusion, we add a column vector of ones to the front of the matrix of test data. Not only does this makes it analogous to the **A** matrix (except that it contains test data rather than training data), the feature indices will now line up with the coefficient indices. Thus now we have

$$\hat{y}_i = \sum_{k=0}^{p} b_k \mathbf{x}_k^i \tag{2.30}$$

Here $\mathbf{x}_0^i = 1$, and $\mathbf{x}_k^i$, for $k = 1, 2, \ldots, p$ are the features $0, 1, \ldots, p-1$ of vector $\mathbf{x}^i$. Treating $\mathbf{x}_k^i$ as a matrix with column index k and row index i, and **b** as a row vector, this is simple matrix multiplication of the ith row of **x** with the coefficient array.

```
def predicted_values(coefficients, xvalues):

    # check for bad data shape
    n=len(coefficients)
    nvectors, nfeatures=xvalues.shape
    if n!= nfeatures+1:
        print("The number of coefficients is",n,
        "the vectors have length",nfeatures)
        return(float("NAN"))

    # find dot product of each row with coefficient vector
    output=[]
    for vector in xvalues:
        U=np.append([1],vector)
        y=float(U.dot(coefficients))
        output.append(y)
    output=np.array(output).reshape(-1,1)
    return (output)
```

We can obtain an array of predicted values on the test set by a single function call:

```
PRE=predicted_values(sol,XTEST)
```

It is useful to calculate several sum-squared-errors to evaluate the result. Let $\hat{y}_i$ denote $f(x)_i$ where $f(x)$ is the model equation. The **residual sum squared error** is

$$\text{RSS} = \sum_{x \in \text{test}} (\hat{y}_i - y_i)^2 \tag{2.31}$$

The **Mean Squared Error** (MSS) is the mean of the RSS,

$$\text{MSS} = \frac{\text{RSS}}{n_{\text{test}}} = \frac{1}{n_{\text{test}}} \sum_{x \in \text{test}} (\hat{y}_i - y_i)^2 \tag{2.32}$$

```
def MSS(YO, YP):
    n=len(YP)
    return(float(sum((YO-YP)**2)/n))
MSS(YTEST, PRE)
```

```
5975.775754556709
```

The **sklearn** package calls this the **mean_squared_error**. Finally, there is a third sum square error that is sometimes called the **Total Sum of Squares** (TSS Error). It is defined as

$$\text{TSS} = \sum_{y \in test} (\overline{y} - y_i)^2 \tag{2.33}$$

where $\overline{y} = \frac{1}{n} \sum_{test} y_i$. Note that this TSS does not depend on the values of either x_i or on the fit, only on the values of y_i. It is purely a property of observed values of y in the test data. In Python, the TSS can be calculated directly as

```
def TSS(YOBS):
    ybar = np.mean(YOBS)
    return(float(sum((ybar-YOBS)**2)))
TSS(YTEST,PRE)
```

```
1534606.5192307695
```

A more absolute measure than either the RSS or the MSS is the R^2 value, which is defined as follows:

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}} = 1 - \frac{n \times \text{MSS}}{\text{TSS}} \tag{2.34}$$

Putting these together,

```
def MYR2(YOBS,YPRE):
    n=len(YOBS)
    RSS=n*MSS(YOBS,YPRE)
    tss=TSS(YOBS)
    return(1-RSS/tss)
MYR2(YTEST,PRE)
```

```
0.7975113911332077
```

Thus the prediction has an $R^2 \approx 0.80$ (rounded to two significant figures). Values of R^2 that are closer to 1 are generally thought to indicate a better fit, while values of R^2 that are closer to 0 are generally thought to indicate a poorer fit. However, the value of R^2 does not tell the whole story. The subject of evaluation of multi-linear fits is discussed in detail in statistics books.

Multilinear Regression with Sklearn

Linear regression is built into the **sklearn** package. Assuming we have already read our data and split it into test and training sets as described above, we can use **LinearRegression()** almost immediately. First, load the libraries.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Then perform the regression. Note that the data sets must be properly formatted, as described in chapter 1. The code given above and used to format the arrays **XTRAIN**, **YTRAIN**, etc., will work.

```
r=LinearRegression().fit(XTRAIN,YTRAIN)
print(r.intercept_)
print(r.coef_)
```

```
[-55.79490706]
[[ 0.04860612  0.0125405   0.00631709  0.70457547 -1.61978327
1.69010283]]
```

We obtain the same coefficients that we did with **linregress**, except that they are formatted differently. The constant coefficient is given in the output **intercept_**, while the slopes are given in the output **coef_**. The mean RSS error and the R^2 are found using functions in the **metrics** package.

```
R2=r2_score(YTEST,P)
MSE = mean_squared_error(YTEST,P)
print(R2,MSE)
```

```
0.7975113911332077 5975.775754556711
```

These values are the same as the ones we obtained with our direct calculation. In general it is easier to use the implementation in **sklearn**, once students understand how it works, instead of the code of **linregress**, because less additional code is required. They both produce the same result.

Chapter 3

Polynomial Regression

In this chapter we turn to the idea of applying the least square concept to fitting a polynomial to data. Some authors call this **polynomial regression**, because in the end we are finding a polynomial that fits the data in the least-square sense. On the other hand, this is still a form of linear regression because the equations that need to be solved to minimize the objective function are all linear in the unknown coefficients, and hence they can be solved analytically.

Mathematical Background

Suppose we are faced with n data pairs as in equation 1.1 and want to fit a polynomial of degree p

$$P(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_p x^p \tag{3.1}$$

As before, we will want to minimize an objective function of the form

$$\mathcal{E} = \sum_{i=0}^{n-1} (y_i - P(x_i))^2 \tag{3.2}$$

Suppose that there is some residual error r_i at each prediction, so that

$$P(x_i) = y_i + r_i \tag{3.3}$$

Substituting equation 3.3 into equation 3.3 for each data point $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ gives

$$\begin{aligned} c_0 + c_1 x_0 + c_2 x_0^2 + \cdots + c_p x_0^p &= y_0 + r_0 \\ c_0 + c_1 x_1 + c_2 x_1^2 + \cdots + c_p x_1^p &= y_1 + r_1 \\ c_0 + c_1 x_2 + c_2 x_2^2 + \cdots + c_p x_2^p &= y_2 + r_2 \\ &\vdots \\ c_0 + c_1 x_{n-1} + c_2 x_{n-1}^2 + \cdots + c_p x_{n-1}^p &= y_{n-1} + r_{n-1} \end{aligned} \tag{3.4}$$

Reformulating equation 3.4 as a matrix equation

$$\overbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^p \\ 1 & x_1 & x_1^2 & \cdots & x_1^p \\ 1 & x_2 & x_2^2 & \cdots & x_2^p \\ & & & \vdots & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^p \end{bmatrix}}^{\text{Define this as } \mathbf{A}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_p \end{bmatrix}}_{\mathbf{c}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}}_{\mathbf{y}} + \underbrace{\begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{bmatrix}}_{\mathbf{r}} \tag{3.5}$$

and define the matrix $\mathbf{A}$ and column vectors $\mathbf{c}$, $\mathbf{y}$, and $\mathbf{r}$ as shown, then

$$\mathbf{Ac} = \mathbf{y} + \mathbf{r} \tag{3.6}$$

Observe that $\mathbf{Ac}$ is a column vector containing $P(x_i)$ in the ith row. Thus from equation 3.2

$$\begin{aligned} \mathcal{E} &= \sum_{i=0}^{n} (y_i - (\mathbf{Ac})_i)^2 \\ &= \sum_{i=0}^{n} (\mathbf{y}\text{-}\mathbf{Ac})_i (\mathbf{y}\text{-}\mathbf{Ac})_i \\ &= \sum_{i=0}^{n} ((\mathbf{y}\text{-}\mathbf{Ac})^{\mathrm{T}})_i (\mathbf{y}\text{-}\mathbf{Ac})_i \\ &= (\mathbf{y}\text{-}\mathbf{Ac})^{\mathrm{T}} (\mathbf{y}\text{-}\mathbf{Ac}) \end{aligned}$$

Differentiating and using the product rule,

$$0 = \frac{\partial \mathcal{E}}{\partial c_i} = (\mathbf{y}\text{-}\mathbf{Ac})^{\mathrm{T}} \frac{\partial}{\partial c_i} (\mathbf{y}\text{-}\mathbf{Ac}) + \left[\frac{\partial}{\partial c_i} (\mathbf{y}\text{-}\mathbf{Ac})^{\mathrm{T}} \right] (\mathbf{y}\text{-}\mathbf{Ac}) \tag{3.7}$$

If we define

$$\mathbf{a}_j = \begin{bmatrix} x_0^j \\ x_1^j \\ x_2^j \\ \vdots \\ x_{n-1}^j \end{bmatrix} \tag{3.8}$$

then we can write $\mathbf{A}$ in partitioned form as

$$\mathbf{A} = \left[\begin{array}{c|c|c|c} \mathbf{a}_0 & \mathbf{a}_1 & \cdots & \mathbf{a}_p \end{array} \right] \tag{3.9}$$

and

$$\mathbf{Ac} = c_0\mathbf{a}_0 + c_1\mathbf{a}_1 + \cdots c_p\mathbf{a}_p \tag{3.10}$$

Hence

$$\frac{\partial \mathbf{Ac}}{\partial c_i} = \mathbf{a}_i \tag{3.11}$$

This simplifies equation 3.7 to

$$0 = (\mathbf{y}\text{-}\mathbf{Ac})^{\mathrm{T}}(-\mathbf{a}_i) + \left[(-\mathbf{a}_i)^{\mathrm{T}}\right](\mathbf{y}\text{-}\mathbf{Ac}) \tag{3.12}$$

Distributing the transpose in the first term, and the multiplication in both terms,

$$-\mathbf{y}^{\mathrm{T}}\mathbf{a}_i + \mathbf{c}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{a}_i - \mathbf{a}_i^{\mathrm{T}}\mathbf{y} + \mathbf{a}_i^{\mathrm{T}}\mathbf{Ac} = 0 \tag{3.13}$$

Each term in this equation is a scalar. Furthermore, since a scalar is its own transpose,

$$\mathbf{a}_i^T\mathbf{y} = (\mathbf{y}^{\mathrm{T}}\mathbf{a}_i)^{\mathrm{T}} = \mathbf{y}^{\mathrm{T}}\mathbf{a}_i \tag{3.14}$$

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{Ac} = (\mathbf{c}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{a}_i)^{\mathrm{T}} = \mathbf{c}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{a}_i \tag{3.15}$$

Substitution and dividing by 2,

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{y} = \mathbf{a}_i^{\mathrm{T}}\mathbf{Ac} \tag{3.16}$$

Since this must hold for every column of the matrix,

$$\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{Ac} \tag{3.17}$$

Equations 3.17 are called the **normal equations** for the polynomial least squares regression. They resemble the normal equations for the normal equations we found for a multilinear fit 2.24 but the form of the matrix $\mathbf{A}$ is somewhat difference.

Polynomial Regression in Python

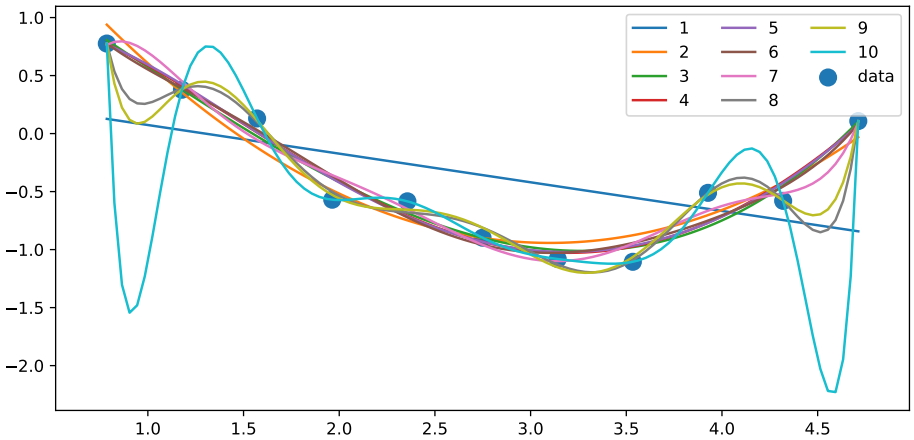
The main error that students make with polynomial regression is over-fitting. We will demonstrate this with a toy example. Using Python we will generate a sample of 11 data points evenly spaced on the interval $[\pi/4, 3\pi/2]$ that are randomly distributed about the function $y = \cos(x)$, with a uniform distribution on the interval ± 0.2 in the y direction. To ensure reproducibility of the data we'll set a random number seed using **`np.random.seed`**.

```
import numpy as np
npts=11
np.random.seed(99)
x=np.linspace(math.pi/4, 1.5*math.pi, npts)
y=np.cos(x) + .4*(np.random.random(npts)-.5)
```

A least squares polynomial fit is performed with **np.polyfit(x,y,n)**, which returns a **numpy array** of coefficients for a polynomial of degree **n** that fits the data in the least squares sense. If we save that list of coefficients to any **array**, such as **c**, then we can find the values of the polynomial at any number or list or array of numbers **x** with the function **np.polyval(c, x)**. The following block of code will find the fits for each degree 1, 2, . . . , 10 and plot the results on a single figure.

```
xplot=np.linspace(math.pi/4, 1.5*math.pi, 100)
for degree in range(1,npts):
    c=np.polyfit(x,y,degree)
    yplot=np.polyval(c,xplot)
    plt.plot(xplot,yplot,label=str(degree))
plt.scatter(x,y, label="data", s=100)
plt.legend(ncol=3)
```

Figure 3.1: Least squared fits of degrees 1 through 10 using **polyfit** to a toy data set. The data points in the tod set are at: (0.79, 0.78), (1.18, 0.38), (1.57, 0.13), (1.96, -0.57), (2.36, -0.58), (2.75, -0.9), (3.14, -1.08), (3.53, -1.11), (3.93, -0.51), (4.32, -0.58), (4.71, 0.11).



The results in figure 3.1 illustrate the danger of over-fitting. The higher degree fits are modeling the noise (the random variation) and not the original data. The highest degree fit actually passes through each point exactly. While this returns zero error it does not actually model the original function.

To see how we can find the *optimal degree of fit*, we will begin by first defining some auxiliary functions in Python. The first function will simplify our training set/test set split. We will define **train_test_split(x,y,f)** to return a tuple containing four arrays: the training set for x, the training set for y, the test set for x, and the test set for y. A typical way to call our new function would be

```
XTRAIN, YTRAIN, XTEST, YTEST=train_test_split(x,y,.75)
```

This would select 75 percent for the training set and 25 percent for the test set. The implementation of **train_test_split** follows. If the fraction **f** is not specified in the call to invoke **train_test_split**, it will default to 75% for training. (A similar function is defined in **sklearn**; see page 132).

```
def train_test_split(x,y,f=.75):

    # Force between 10 and 90 percent training data

    f=max(.1, min(f,.9))

    # shuffle the indices of the original data

    n=len(x)
    ntrain=round(f*n)
    indices=list(range(n))
    random.shuffle(indices)

    # select the first f (fraction) for training
    # and the remaining for test

    ind_train=indices[:ntrain]
    ind_test=indices[ntrain:]

    # select the the points in the training and test data

    XTRAIN=x[ind_train]; YTRAIN=y[ind_train]
    XTEST=x[ind_test]; YTEST=y[ind_test]
    return(XTRAIN,YTRAIN,XTEST,YTEST)
```

Next, we define a function **fitsets** which uses **polyval** to fit the training set with the best polynomial of each degree 1, 2, ..., **maxdeg**. After determining each set of coefficients, it calculates the MSS value using the test data set. The function then returns a list of MSS values, one value for each degree fit. For example, to perform linear, quadratic, and cubic fits, and return the three MSS values, we would run the following:

```
XTR,YTR,XT,YT=train_test_split(x,y,.75)
fitsets(XTR,YTR,XT,YT,3)
```

```
array([0.76316081, 0.05956944, 0.02600236])
```

The code to implement **fitsets** follows.

```
def fitsets(XTRAIN,YTRAIN,XTEST,YTEST,maxdeg):
    ntest=len(XTEST)
    results=[]
    for p in range(1,maxdeg):
        fit=np.polyfit(XTRAIN, YTRAIN,p)
        yfit=np.polyval(fit,XTEST)
        MSS=sum((yfit-YTEST)**2)/ntest
        results.append([p,MSS])
    return(results)
```

Now the problem is that the MSS values we have obtained are strongly dependent are the randomization: if we had made a different train/test split, we would get a different result. The best way to avoid a bias as a result of this problem is to run **fitsets** many many times with many many splits.

So here is the basic plan.

1. For trial = 1, 2, . . . , **nruns** (the desired number of times we want to repeat the randomization):
   (a) Shuffle the data and do a train/test split
   (b) For each of degree 1, 2, . . . , **maxdeg** (the highest degree polynomial we are interested in fitting, i.e., the largest value of p):
      i. Find the best fit coefficients $c_0, c_1, \ldots, c_p$ using the *training* set.
      ii. Evaluate the fit (e.g., calculate the MSS) using the *test* set.
2. At this point you should have **nruns** MSS values for *each* degree polynomial.
   (a) Calculate the mean and standard deviation of the MSS values for each degree.
   (b) Plot the means as a function of degree (figure 3.2)
   (c) Plot the means +/- 1.96 standard deviations to get an idea of the 95% spread of the data, cutting off the plot at the MSS=0 axis. Fill in the area between the upper curve and the lower curve, as illustrated in figure 3.2.

The usefulness of plotting the spread in this manner will become obvious shortly. The distance between the central and upper and lower curves in the final step

is totally arbitrary. The value of 1.96 times the standard deviation could easily be replaced by one, two, or three times the standard deviation. The idea is to represent the spread, not its precise value. The number 1.96 is commonly used by statisticians because when data is randomly distributed according to a normal distribution, 95% of it will fall within 1.96 standard deviations of the mean.

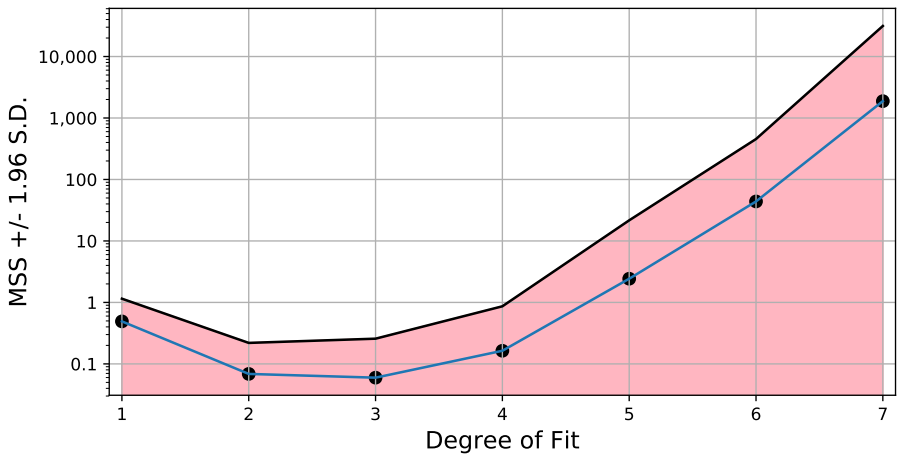
Here is a block of code that will repeat the process **nruns** times.

```
nruns=100
maxdeg=7
MSS_DAT=[]
for rz in range(nruns):
    XTR,YTR,XT,YT=train_test_split(x,y,.75)
    MSS=fitsets(XTR,YTR,XT,YT,maxdeg)
    MSS_DAT.append(list(MSS))
MSS_DAT=np.array(MSS_DAT)
means=np.mean(MSS_DAT,axis=0)
stdevs=np.std(MSS_DAT,axis=0)
```

The last two lines calculate the mean and standard deviation of the MSS for each degree fit. The first two lines are data set dependent and need to be set by the user. The number of runs needed will depend on the file size. For a small file, we will cover all possible randomizations with a fairly small **nruns**; there is no point is in setting **nrun** in the millions if there are on eleven points in the file. For example, there are only 11!/(7!4!)= 330 different possible ways that we can choose 7 points out of 11. This is maximum number of different possible test/training splits. Second, the degree of the fit is limited by the number of points. For a small data set, for example, there is no point in attempting to fit a polynomial of degree greater than $n-1$ to n points (2 points for line, 3 points for quadratic, 4 points for a cubic, etc.). For larger data sets we can try higher degree fits but even then it is likely that some of the points will be very nearly co-linear (or co-something, where something is some sort of polynomial); the effect is to make the data set ill conditioned to higher degree fits. This is a subject that is covered in more advanced numerical analysis courses. Suffice it to say that if you start getting messages that say your data is ill conditioned, it means you are trying to fit a polynomial that is much higher degree than your data warrants.

Finally, we produce some code that will display our results. We'll start by defining a **range DOTS** that represents the values on the x-axis; the correspond to the degrees of the polynomials. The next three lines use **pyplot.plot** the mean and upper and lower spread defined by +/- 1.96 standard deviations about the mean. We use **pyplot.fill_between** to fill the space between the upper and lower curves with a light pink background. We then draw the dots with

Figure 3.2: MSS mean and envelope of 1.96 standard deviations as a function of degree fit for the data shown in figure 3.1, using 100 test runs. The lower edge to the envelope is not visible because when 1.96 standard deviations is subtracted from the mean the result is negative.



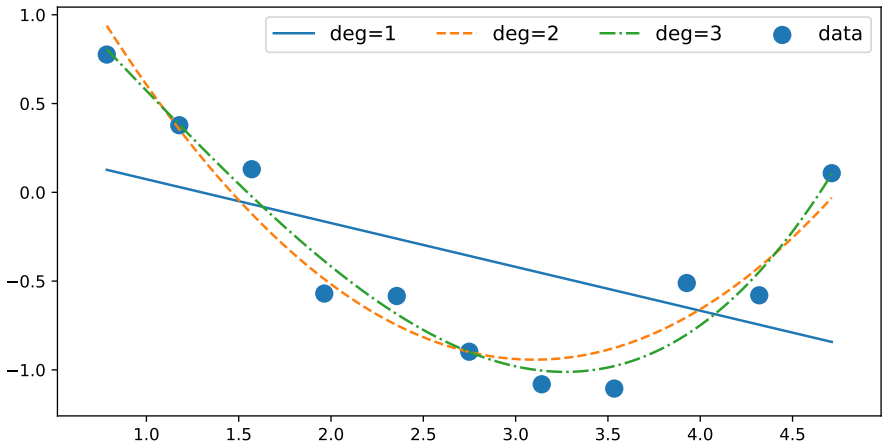
`pyplot.scatter` after we back-fill because if we do it in the opposite order the filling will over-draw the dots. The **`pyplot.yscale("log")`** forces the y-axis to be drawn with a logarithmic scale. This will only plot positive values, and is very useful when plotting the MSS because there is often a very large range of values.

```
import matplotlib.pyplot as plt
DOTS=range(1,maxdeg+1)
plt.plot(DOTS,means+1.96*stdevs,c="k")
plt.plot(DOTS,means-1.96*stdevs,c="k")
plt.plot(DOTS,means)
plt.fill_between(DOTS, means+1.96*stdevs,
  means-1.96*stdevs,color="lightpink")
plt.scatter(DOTS,means,s=50,c="k")
plt.yscale("log")
```

The results for the toy data set from figure 3.1 are shown in figure 3.2. We see that the MSS is significantly better with a quadratic than a linear fit, but nothing is added by adding higher degrees. In fact, both the average of the MSS values and the spread of the MSS rapidly increase by approximately a factor of ten for each additional degree that we add to the fit. The take-away message from figure 3.2 is that a quadratic (degree 2) has both the lowest mean spread and lowest variance error of all possible degrees for this data. We we found in figure 3.1 that a degree 10 polynomial could fit all the data points exactly (with

zero error), and that degree 8 or 9 nearly fit them exactly, we were fitting the noise, and not the data. This illustrates the danger of over-fitting.

Figure 3.3: Degree 1, 2, and 3 fits for toy data set from figure 3.1, where the fit is based on the entire data set, with the y axis expanded.



The data is re-plotted in figure 3.3 with the y-axis expanded and only the fits for degrees 1, 2, and 3 shown. Visually we can see little advantage to 3 over 2, and but there is a significant advantage to 2 over 1. This corroborates the analysis that led to our conclusion of an optimal degree of 2 for this data. Of course the original data was not polynomial at all (as we know)!

The Variance-Bias Tradeoff

To understand why the error first decreases and then increases, we need to delve into probability theory. If you haven't had a course in probability the next couple of paragraphs might not make much sense. You can skip to the conclusion at the end of the section instead. The derivation depends on something called the **expectation** in the world of probability theory. The expectation of a quantity u is denoted by a linear operator $E[u]$ (see equation A.66).

Suppose that are data points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ represent sample of some deterministic function $y = f(x)$ that is measured with some noise that has variance σ^2. Then

$$E[y_i] = y_i \tag{3.18}$$

$$\text{var}[y] = \sigma^2 \tag{3.19}$$

We will make some estimate $\hat{y}$ of y that differs from y due to noisy sampling. Because of the linearity of the expectation operator, and because y_i is a constant,

$$E[(y_i - \hat{y})_i^2] = E[y_i^2 + \hat{y}_i^2 - 2y_i\hat{y}_i] \tag{3.20}$$

$$= E[y_i^2] + E[\hat{y}_i^2] - 2y_i E[\hat{y}_i] \tag{3.21}$$

Using equation A.83,

$$E[y_i^2] = \text{var}[y_i] + E[y_i]^2 \tag{3.22}$$

$$E[\hat{y}^2] = \text{var}[\hat{y}_i] + E[\hat{y}_i]^2 \tag{3.23}$$

Hence

$$E[(y_i - \hat{y})_i^2] = \text{var}[y_i] + E[y_i]^2 + \text{var}[\hat{y}_i] + E[\hat{y}_i]^2 - 2y_i E[\hat{y}_i] \tag{3.24}$$

Since $E[y_i] = y_i$ and $\text{var}[y_i] = \sigma^2$,

$$E[(y_i - \hat{y})_i^2] = \sigma^2 + y_i^2 + \text{var}[\hat{y}_i] + E[\hat{y}_i]^2 - 2y_i E[\hat{y}_i] \tag{3.25}$$

But

$$y_i^2 + E[\hat{y}_i]^2 - 2y_i E[\hat{y}_i] = (y_i - E[\hat{y}_i])^2 = E[y_i - \hat{y}_i]^2 \tag{3.26}$$

Therefore

$$E[(y_i - \hat{y})_i^2] = \underbrace{\sigma^2}_{\text{noise}} + \underbrace{\text{var}[\hat{y}_i]}_{\text{variance}} + \underbrace{E[y_i - \hat{y}_i]^2}_{\text{bias}} \tag{3.27}$$

This is our conclusion:

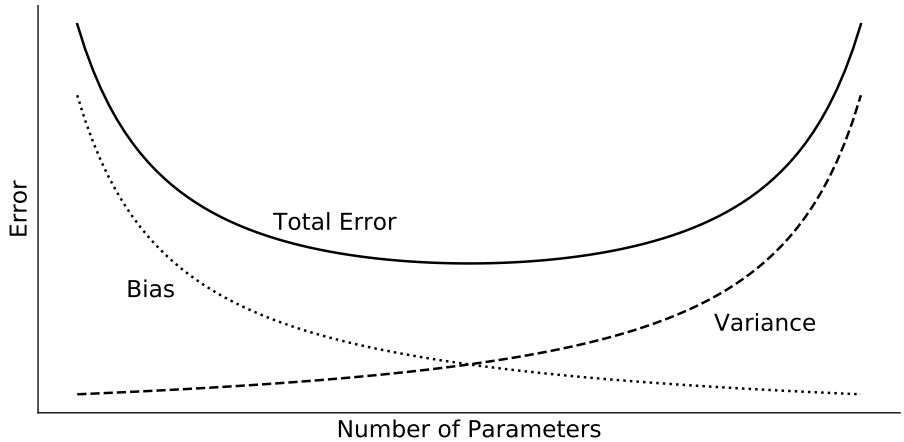
$$\text{Total Error} = \text{ Noise} + \text{Variance} + \text{Bias}$$

The total error is the sum of three terms:

1. An unavoidable and irreducible measurement noise. This is inherent to the observation and measurement process and cannot be avoided. It is due to normal random variation and cannot be controlled.

2. Variance in the model, which describe fluctuations in the description of the data. These fluctuations increase as you add more parameters to the model. When we are fitting a polynomial, for example, by adding higher degree terms, we add more parameters. Visually, this adds more "bumps" to the plot. Statistically, this adds more degrees of freedom to the fit, and there are more places where the model can go wrong.

Figure 3.4: Illustration of the bias-variance tradeoff. As the number of parameters increases, the variance increases while the bias decreases. The optimal fit lies at some point in-between.



3. Bias in the model. The bias decrease as you add more parameters, because you are fitting the data more closely (figure 3.4).

This result is usually called the **variance/bias trade-off** because as you add more parameters to the model, the bias decreases (which improves the fit) but the variance increases (which worsens the fit). The optimal fit is somewhere in between. Hence, a trade-off between the two must be made.

Polynomial Fit: A Real Data Set

We'll apply the methods discussed in the previous sections to the price of Brazilian coffee beans.[1] We will compare the price of *arabica* coffee beans to the price of *robusta* coffee beans in Reals per 60 kilogram bag over a 58 month period starting in January 2014.[2] The data have been placed into the files **`robusta.txt`** and **`arabaca.txt`** in the **`data`** folder on the github site that accompanies this

[1]Data taken from the USA Foreign Agricultural Global Agricultural Information Network (GAIN) Report, BR18027, 15 Nov 2018, downloaded from `https://gain.fas.usda.gov/RecentGAINPublications/CoffeeSemi-annual_SaoPauloATO_Brazil_11-5-2018.pdf` on 22 Nov 2018.

[2]The basic unit of Brazilian currency is the Real. The exchange rate on 22 Nov 2018 was approximately 1 Real to 0.26 US Dollars (Google Finance).

book. You should copy them into a local folder if you want to follow along and reproduce this code.

The files are tab-separated columns, formatted with one column per year 2014 through 2018 and one line per month. The months are labeled with the month name. The last few entries are zero-filled in the data set to indicate no data (there are other ways to handle this, but in this data set we have pre-filled in the zero). The first two lines of the data contain text that can be ignored. We can read this type of file using **pandas.read_csv** and using the **skiprows** and **sep** options. We indicate tabs instead of commas by setting **sep="\t"**.

```
robusta = pd.read_csv("robusta.txt",sep="\t",skiprows=2)
arabica = pd.read_csv("arabica.txt",sep="\t",skiprows=2)
```

Here is a glance at what the data files look like if we use the **print** command to print them out:

```
        Month    2014    2015    2016    2017    2018
0     January  226.82  283.28  389.27  495.19  331.57
1    February  243.48  299.58  393.61  449.93  319.12
...
10   November  277.02  375.25  521.31  365.36    0.00
11   December  277.14  378.98  487.60  362.30    0.00
```

Each data file has the same number of data points. We can stack each one into a single list, drop the last two elements, and divide by 60 to convert to price per kilogram.

```
years=list(robusta.columns)[1:]
robusta_prices=
  np.hstack([list(robusta[year]) for year in years])[:-2]/60

years=list(arabica.columns)[1:]
arabica_prices=
  np.hstack([list(arabica[year]) for year in years])[:-2]/60
```

We'll use these for our x and y data arrays, using the *robusta* price as our explanatory variable and the *arabica* price as a response variable.

```
x=np.array(robusta_prices)
y=np.array(arabica_prices)
```

We'll then use the functions **train_test_split** (page 41) and **fitsets** (page 42) to perform 200 simulations.

```
nruns=200
maxdeg=9
MSSes=[]
for rz in range(nruns):
    XTR,YTR,XT,YT=train_test_split(x,y,.75)
    results=np.array(fitsets(XTR,YTR,XT,YT,maxdeg)).T
    MSSes.append(list(results))
MSSes=np.array(MSSes)
means=np.mean(MSSes,axis=0)
stdevs=np.std(MSSes,axis=0)
```

Here is the basic code for making a plot of the results:

```
DOTS=range(1,maxdeg+1)
plt.plot(DOTS,means)
# 95 percent envelope
plt.plot(DOTS,means+1.96*stdevs,c="k")
plt.plot(DOTS,means-1.96*stdevs,c="k")
plt.fill_between(DOTS,means+1.96*stdevs,
  means-1.96*stdevs,color="lightpink")
```

After adding some additional code for scaling and labels, the result is plotted in figure 3.5. We can see that there is very little different in the first five degrees of fit, so some other measure needs to be made than MSS error.

Figure 3.5: MSS Error as a function of degree for the polynomial fits of the cost of *arabica* as a function of *robusta* coffee.

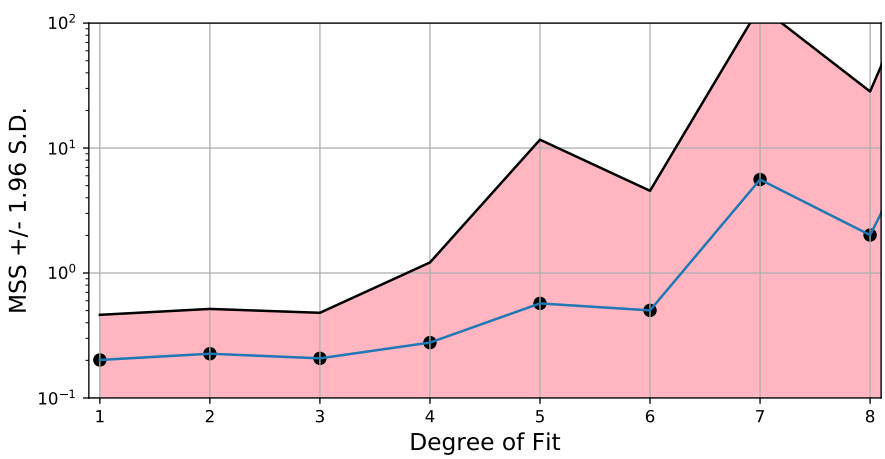
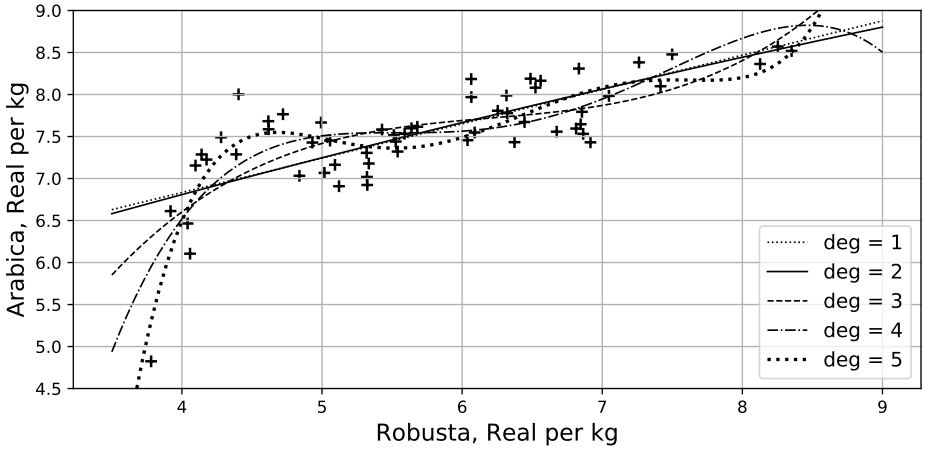


Figure 3.6: Plot of *arabica* coffee price in Real per kilogram as a function of *robusta* price along with polynomial fits for degrees 1 through 5.



The actual fits using the entire data set are illustrated in figure 3.6. This shows that the linear and quadratic fits are, indeed, indistinguishable, in the range of data provided (approximately $3.8 < x < 8.5$). The higher degrees are visually much more satisfying, However, care must be taken in using any of these, especially if any extrapolation is intended, because the polynomials make arbitrary predictions outside of the range of the original data. Observe how the 4th degree turns around and decreases beyond the maximum value, while the 3rd and 5th both continue to increase. This illustrates the extreme danger of extrapolation based on polynomial interpolation: in most cases it does not work.

Chapter 4

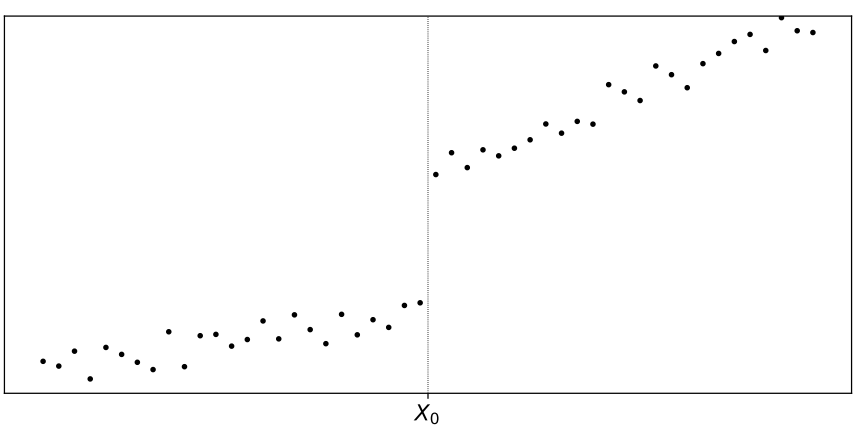
Stepwise Regression

In some situations there will be a natural location $x = X_0$ to partition our data set where it appears that the data is linear on either side of X_0, but it has different slopes on the left and on the right (see figure 4.1). In this case the data set $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ might be represented by a stepwise linear model

$$f(x) = \begin{cases} a_0 + b_0 x, & x < X_0 \\ a_1 + b_1 x, & x \geqslant X_0 \end{cases} \tag{4.1}$$

where a_0, a_1 are y-intercepts and b_0, b_1 are slopes. In a more general case, we

Figure 4.1: Illustration of a data set that appears to be linear but with two different slopes in the intervals $x < X_0$ and $x \geqslant X_0$.



might want to divide the domain into n sub-intervals separated at jump points

$X_0, X_1, \ldots, X_n$. The function then looks like:

$$f(x) = \begin{cases} a_0 + b_0 x, & x < X_0 \\ a_1 + b_1 x, & x_0 \leqslant x < X_1 \\ a_2 + b_2 x, & x_1 \leqslant x < X_2 \\ \vdots & \\ a_n + b_n x, & x_{n-1} \leqslant x < X_n \\ a_{n+1} + b_{n+1} x, & x \geqslant X_n \end{cases} \tag{4.2}$$

In the most general case, there is no constraint that the function be continuous, so there may be discontinuous steps, or jumps, at each of the points $x_0, x_1, \ldots, x_n$.

If we know the location of the break points $X_0, X_1, \ldots$ it is often sufficient to simply partition the data into multiple subsets and do independent linear regressions on (following chapter 1) on each subset. If we do not know the exact location of X_j, or we want to constrain the fits at one or more of the steps to be continuous, we no longer have a collection of simple linear problems, and the overall problem is no longer linear. Instead some sort of iterative process is necessary to determine where to place the steps.

Constrained (Continuous) Stepwise Linear Fit

If we want to constrain the fit to go through the point (X_0, y_0) it is more convenient to write these equations in the point-slope form instead of the slope intercept form. We'll begin with the two-segment case first. Then we have

$$f(x) = \begin{cases} y_0 + b_0(x - X_0), & x < X_0 \\ y_0 + b_1(x - X_0), & x \geqslant X_0 \end{cases} \tag{4.3}$$

The idea is to find a least squares fit of equation 4.3 to the data that minimizes that an objective function such as

$$\mathcal{E} = \sum_{i=0}^{n-1} (f(x_i) - y_i)^2 \tag{4.4}$$

$$= \sum_{x_i < X_0} (y_0 + b_0(x_i - X_0) - y_i)^2 + \sum_{x_i \geqslant X_0} (y_0 + b_1(x_i - X_0) - y_i)^2 \tag{4.5}$$

There is both good news and bad news in Python when it comes to stepwise regression. The bad news is that it is not implemented, even in **sklearn**. The good news is that we can still do it using built in **optimize.curve_fit** in **scipy** with only a little bit of preparation.[1]

Instead of going all willy-nilly and poking around in various places trying to find the best place for the step point, we want to be smart and let Python find the optimal step point. In general it can't do that on its own unless we supply it a good first guess because the objective function is so lumpy: it will just settle on the wrong local minimum.

First, we have to define a step-wise linear function. There are a number of ways to do this, but the easiest is to use the **numpy.piecewise** function. The general form of **piecewise** is

```
piecewise(x, conditions, functions)
```

where **conditions** is a list of logical conditions and **functions** is a list of functions. The value of **x** is tested against each of the conditions in succession. If the jth condition is true, then **x** is passed to the jth function. If all conditions are false, and one more function than condition is provided, the extra function is used as the "else" function.[2] In a case like this, where the functions reference in the second list in **piecewise** are only used once, it is most convenient to define them "on the fly" as **lambda** functions. A **lambda** expression of the form

```
lambda x: expression
```

where **expression** is some formula that depends on **x** can be used to define a temporary nameless function. We could instead use a block of code such as

```
def temp(x):
        return expression
```

and then replace the entire string "**lambda x: expression**" with "**temp(x)**." Our **piecewise** definition for equation 4.3 is

```
def f(x, X0, y0, b0, b1):
    return np.piecewise(x,
            [x < X0], [lambda x:y0+b0*(x-X0),
                       lambda x:y0+b1*(x-X0)])
```

An equivalent definition without using **lamda** functions is this:

[1]The method described here is a generalization of one proposed on a message board at https://stackoverflow.com/questions/29382903/how-to-apply-piecewise-linear-fit-in-python.

[2]See https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.piecewise.html for detailed documentation on **piecewise**.

```
def f(x, X0, y0, b0, b1):
        def left(x):
                return(y0+b0*(x-X0))
        def right(x):
                return(y0+b1*(x-X0))
    return np.piecewise(x, [x < X0], [left(x),right(x)])
```

Using **lambda** allows us avoid defining a lot of temporary functions like **left** and **right** if the case where we have multiple steps. We observe that it is not strictly necessary to use **piecewise** to define **f(x,X0,y0,b0,b1)**; we could have implemented with a simple **if**/**else** construct.

Next, we'll generate some toy data. We start by generating 50 points that are uniformly distributed over the interval [.5, 19.5]:

```
nran=50
xmin=.5; xmax=19.5
xran=np.random.rand(nran)*(xmax-xmin) + xmin
```

We'll then define our target function to satisfy

$$y = \begin{cases} 35 + 3(x - 10), & x < 10 \\ 35 - 2(x - 10), & x \geqslant 10 \end{cases} \tag{4.6}$$

We can do this using the **piecewise** implementation of **f** we defined above, as

```
def ytarget(x):
    return f(x, 10, 35, 3, -2)
```

Next, we'll randomly generate our points with y-values at each of the x-coordinates in **xran** but displaced from the line with a uniform distribution by ± 2.5:

```
yran=[ytarget(x)+5*(np.random.random()-.5) for x in xran]
```

Our goal is to find the stepwise linear fit to data in **xran**, **yran**. We'll use **scipy.optimize.curve_fit** for this.[3] The basic form if

```
curve_fit(f, xarray, yarray, parameters)
```

where **xarray** and **yarray** are arrays or lists of the x and y data (like **xran** and **yran**) and **f(x, *parameters)** is the function that is used to estimate **y** from **x** (like our **f**). The ***** in front the **parameters** means that there is a sequence of parameters that follows the argument **x** in **f**. These are the parameters that

[3]See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html for documentation.

curve_fit attempts to optimize. In the call to **curve_fit** we have the option of giving a first guess for these parameters. If we do not give a first guess there is a very good chance that **curve_fit** with either converge to the wrong answer or that it will fail to converge at all. There are two return values to **curve_fit**: the parameters, and the covariance matrix.

```
parms, covmat = optimize.curve_fit(f, xran, yran,[1,1,1,-1])
print(parms)
```

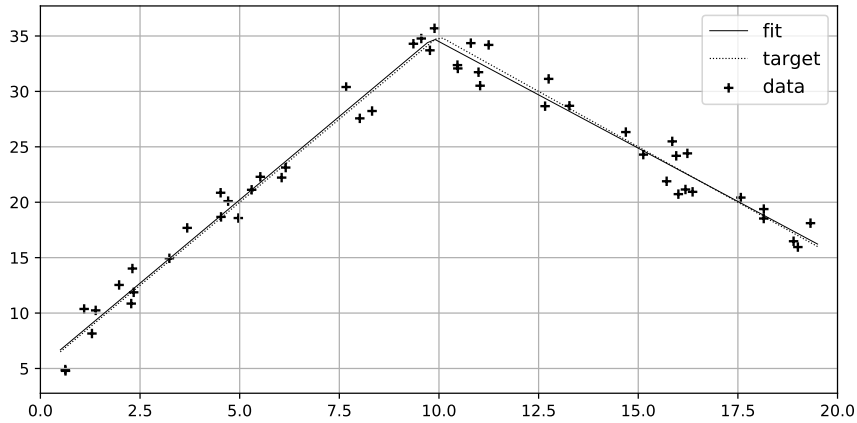
```
[ 9.8439359  34.79551181  3.00921039 -1.92330673]
```

The default initial guess for all parameters used by **optimize.curve_fit** is 1. We only changed one parameter in our first guess, to force the second slope to be negative. The return values for the parameters are in the same order as the parameters to **f**; in this example it converged to

$$y = \begin{cases} 34.79551181 + 3.00921039(x - 9.8439359), & x < 9.8439359 \\ 34.79551181 - 1.92330673(x - 9.8439359), & x \geqslant 9.8439359 \end{cases} \tag{4.7}$$

The resulting constrained linear fit and the target data are shown in figure 4.2.

Figure 4.2: Constrained fit found in equation 4.7 to data randomly distributed about the target curve in equation 4.6. The lines should not be identical because of the data randomization.



Partially Constrained Linear Fit

Suppose we only want to force continuity at one point, but we have more that one segment. We can do this, e.g., by defining a piecewise function such as

$$y = \begin{cases} y_0 + b_0(x - X_0), & x < X_0 \\ y_1 + b_1(x - X_1), & X_0 \leqslant x < X_1 \\ y_1 + b_2(x - X_1), & X_1 \leqslant x \end{cases} \tag{4.8}$$

The second segment (with right endpoint at $x = X_1$) and the third segment (with left endpoint at $x = X_1$) are constrained to pass through the point (X_1, y_1). The right end point of the first segment has coordinates (X_0, y_0) while the left end of the second segment has coordinates $(X_0, y_1 + b_1(X_0 - X_1))$. In general

$$y_0 \neq y_1 + b_1(X_0 - X_1) \tag{4.9}$$

so the function is not necessarily continuous at $x = X_0$. We could force it to be continuous by defining

$$b_1 = \frac{y_0 - y_1}{X_0 - X_1} \tag{4.10}$$

so that the fully continuous piecewise function would be

$$y = \begin{cases} y_0 + b_0(x - X_0), & x < X_0 \\ y_1 + \dfrac{y_0 - y_1}{X_0 - X_1}(x - X_1), & X_0 \leqslant x < X_1 \\ y_1 + b_2(x - X_1), & X_1 \leqslant x \end{cases} \tag{4.11}$$

We will leave equation 4.11 as an exercise for the student and implement equation 4.8 instead. First we'll generate a toy data set consisting of 100 points with three segments and target skip points at $(10, 35)$ and $(20, 15)$, with slopes 3, -2, and 1.5 :

```
np.random.seed(99)
nran=100; xmin = 0.5; xmax=29.5;
xran = xmin +(xmax-xmin)*np.random.rand(nran)
```

The three-step piecewise function that implements equation 4.8 is

```
def f(x, x0, x1, y0, y1, b0, b1,b2):
    y=np.piecewise(x,[x < x0, (x>=x0)&(x< x1)],
            [lambda x:y0 + b0*(x-x0),
             lambda x:y1 + b1*(x-x1),
             lambda x:y1 + b2*(x-x1)])
    return(y)
```

We'll let our target have turning points at (10,35) and (20, 15); we'll give it slopes of 3, -2, and 1.5. The reader should verify that this is continuous at $x = 20$ and discontinuous at $x = 10$.

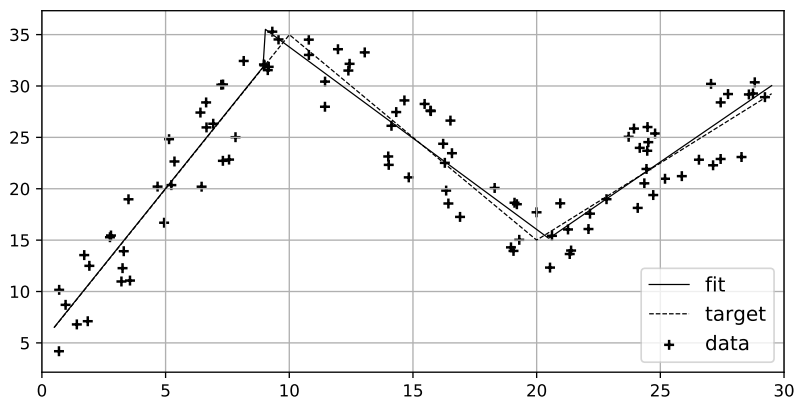
```
def ytarget(x):
    y=f(x, 10, 20, 35, 15, 3, -2, 1.5)
    return y
yran=[ytarget(x)+10*(np.random.random()-.5) for x in xran]
```

The optimization code, with starting guess for parameters $X_0 = 8$, $y_0 = 15$, $X_1 = 25$ $y_1 = 1$, $b_0 = 1$, $b_1 = -1$ and $b_2 = 1$, is

```
parms, covmat = optimize.curve_fit(f, xran, yran,
  [8, 15, 25, 1, 1, -1, 1])
print(parms)
```

```
[ 8.99888865 20.49222946 32.02324234 15.16658127  3.00014888
 -1.77597516 1.65071387]
```

Figure 4.3: Piecewise linear fit with three segments. The segments at the second jump point are constrained to be continuous but not at the first.



There is no general heuristic that always works to give a good first guess. Here are some things you can try:

1. Make a scatter plot of the results

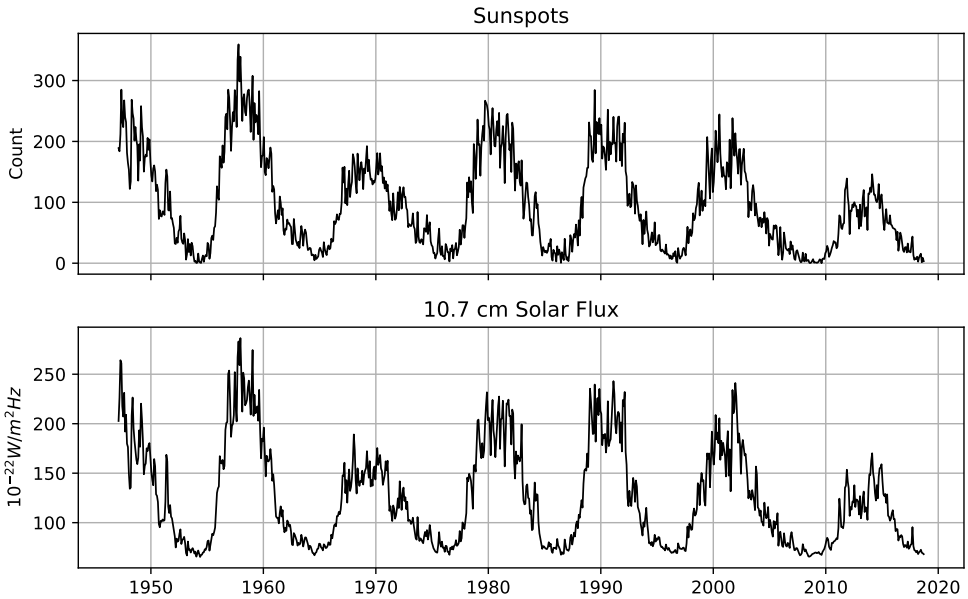
2. Fit the data without any first guess; if it works, add a line plot to your scatter plot. If the turning points look like they fall in the right place, you are done.

3. If there is a weird turning point, make a first guess for your parameters:

   (a) From the scatter plot, estimate the turning points as best you can. At the very minimum, things tend to work better if your guesses are ordered properly, e.g., use 1, 2, 3 and not 1, 3, 2 if it looks like the values are 1.1., 2,2, and 3.5.

   (b) Assign slopes of either 1 or -1 to each segment. You can try to read a slope off the graph, but that probably isn't necessary unless you have a lot of segments.

Figure 4.4: Comparison of sunspot count and observed 10.7 cm solar flux for approximately 70 years. Each value plotted is an average over a single month, so the 28 day variation is suppressed in this plot. Data from `ftp://ftp.geolab.nrcan.gc.ca/data/solar_flux/monthly_averages/solflux_monthly_average.txt`

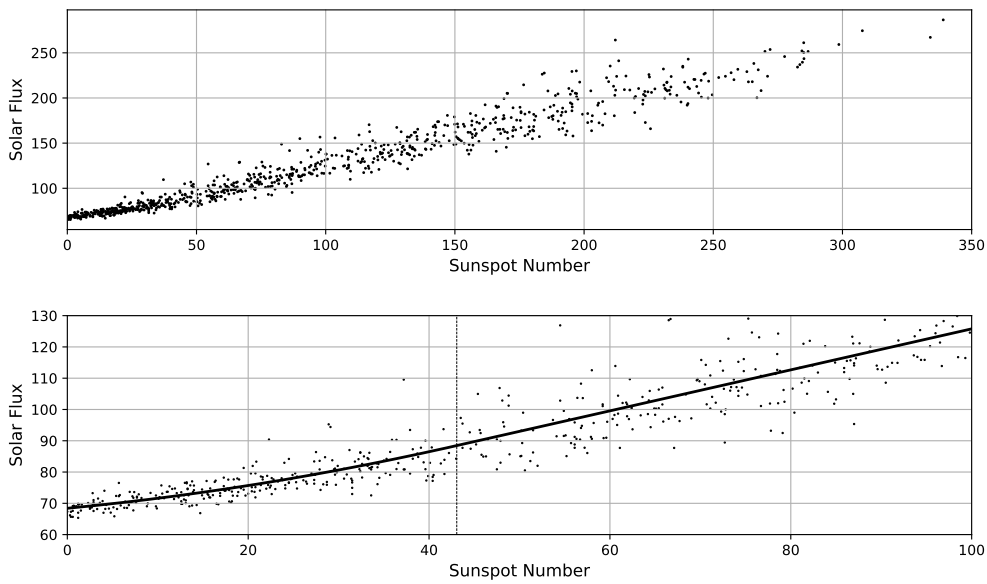


Non-linear Stepwise Regression

The observed number of sunspots and 10.7 cm solar flux both vary with 28 day, 11 year, and 22 year cycles (figure 4.4. The 28 day cycle is associated with the rate of solar rotation. The causes of the 11 and 22 year cycles are unknown. While the number of sunspots can be observed directly with telescopes

(and have been counted for several centuries), it is the solar flux that has a direct impact on upper atmospheric density, the weather, and communication. Since the sunspot record is much longer than the flux record, if one could find a relationship between the two observables, one might be able to extrapolate the flux backwards for several centuries and include it into historical weather models.[4]. We plot the solar flux as a function of sunspot number in the top

Figure 4.5: Top: solar flux as a function of sunspot number, from Feb 1947 through Sept 2018. Bottom: constrained piecewise nonlinear fit using equation 4.17. The scale is magnified to show the area where the parabolic fit is made (to the left of vertical line) and the linear fit (to the right of the vertical line).



of figure 4.5. On a close inspection, the data appears to be somewhat curved for smaller values of sunspot number (under around 40), and linear for larger values. This suggests we try a piecewise fit of

$$y = \begin{cases} a_0x^2 + b_0x + c_0, & x < x_0 \\ a_1 + b_1x, & x > x_0 \end{cases} \tag{4.12}$$

for some parameters a_0, a_1, b_0, b_1, c_0 and x_0. Because the data looks continuous, we want to ensure continuity at x_0, that is, we want

$$a_0x_0^2 + b_0x_0 + c_0 = a_1 + b_1x_0 \tag{4.13}$$

[4]The data used in this example is in the file **`flux-and-sunspot.txt`** We omit the code for loading and extracting the columns; it is illustrated in the accompanying notebook.

Because the data looks smooth, we want to ensure continuity of the derivatives at x_0, so we want

$$2a_0x_0 + b_0 = b_1 \tag{4.14}$$

Substituting equation 4.14 into 4.13 gives

$$a_0x_0^2 + b_0x_0 + c_0 = a_1 + (2a_0x_0 + b_0)x_0 \tag{4.15}$$

Canceling the common b_0x_0 term and solving for a_1,

$$a_1 = c_0 - a_0x_0^2 \tag{4.16}$$

Thus equation 4.12 becomes

$$y = \begin{cases} ax^2 + bx + c, & x < x_0 \\ c - ax^2 + (2ax_0 + b)x, & x > x_0 \end{cases} \tag{4.17}$$

where we have dropped the indices on a, b and c because they all now refer to the original 0 index. The parameters with the index 1 in equation 4.12 have been constrained by our two conditions to depend on the parameters with the index 0.

A **piecewise** implementation of this is

```
def f(x, X0, a, b, c):
    return np.piecewise(x,
            [x < X0], [lambda x:a*x*x+b*x+c,
                       lambda x:(c-a*X0*X0)+(2*a*X0+b)*x])
```

The optimization can then be performed as before.

```
parameters, covmatrix=optimize.curve_fit(f,X, Y, [10, 1, 1, 1])
X0,a,b,c=parameters
print("The solution is: ")
print(round(a,5),"x**2 + ",round(b,5),
    "x + ", round(c,5), ", x <", round(X0,2))
print(round(c-a*X0**2,5),"+",round(2*a*X0+b,5),"x, x>",
    round(X0,2))
```

```
The solution is:
0.00441 x**2 +  0.27555 x +  68.41115 , x < 43.06
60.23217 + 0.65541 x, x> 43.06
```

A plot of the resulting fit is shown in the bottom of figure 4.5.

Chapter 5

Ridge Regression

Multicollinearity

One of the common problems with models of the form 2.5

$$\hat{y} = a + b_0 x_0 + b_1 x_1 + \cdots b_{p-1} x_{p-1} = a + \mathbf{b}^{\mathrm{T}} \mathbf{x} \tag{5.1}$$

is that of **multicollinearity**. Here a and $b_0, \ldots, b_{p-1}$ are $p+1$ undetermined coefficients, and we have defined the constant vector

$$\mathbf{b} = \begin{bmatrix} b_0 \\ \vdots \\ b_{p-1} \end{bmatrix} \tag{5.2}$$

and the p-dimensional feature vector

$$\mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_{p-1} \end{bmatrix} \tag{5.3}$$

As before, each of the coordinates x_i of the feature vector $\mathbf{x}$ is called a **feature** of $\mathbf{x}$. When multicollinearity occurs (or nearly occurs) it is possible to predict one (or more) explanatory variable from the others. When a system is multicollinear, it means that one of the columns of the matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ in equation 2.24 can be expressed as a linear combination of some of the other columns of the matrix, or equivalently, that one of the variables can be expressed as a linear combination of the other variables. If this happens, the normal matrix becomes singular, is not invertible, and equation 2.24 does not have any solution. Even if the system

is only approximately multicollinear, the system will be **ill conditioned**, and the matrix will (for all practical purposes) be almost singular. This means that numerical methods such as inverses and linear solutions will usually fail to work properly, even though such things may sometimes exist.

Mathematical Background of Ridge Regression

The solution proposed by least squares is to find a predictor that minimizes the objective function

$$\mathcal{E} = \sum_{\mathbf{x}_i, y_i \in \text{Training Set}} \left| y_i - a - \mathbf{b}^{\mathrm{T}} \mathbf{x}_i \right|^2 \tag{5.4}$$

where the sum is over all the training points. In **ridge regression**, a method originally formulated to for the solution of integral equations[1] this problem is approached by constraining the least square solution to a space

$$|\mathbf{b}|^2 = \mathbf{b}^{\mathrm{T}} \mathbf{b} = \sum_{i=0}^{p-1} b_i^2 \leqslant K \tag{5.5}$$

where the constant K is a parameter of the problem. Equivalently, the problem can be solved by minimizing the revised objective function

$$\mathcal{E}_{\text{ridge}} = \sum_{i=0}^{n_{\text{train}}-1} \left| y_i - a - \mathbf{b}^{\mathrm{T}} \mathbf{x}_i \right|^2 + \lambda \sum_{i=0}^{p-1} b_i^2 \tag{5.6}$$

Here n_{train} is the size of the training set and i is an index that starts at 0 and runs over the items in the training set. The parameter λ is a constant that is fixed by the user in advance.

Equation 5.6 is the **dual objective function** of the constrained system given by equations 5.4 and 5.5, and the parameter λ is the **dual** parameter to K. In other works, for every parameter K in the constrained optimized system (5.4,5.5) there is an equivalent parameter λ in the optimized dual objective function 5.6. Dual problems are discussed in textbooks in advanced linear algebra; they are equivalent formulations of the same problem from two different perspectives. To understand the difference between the two problems consider the following:

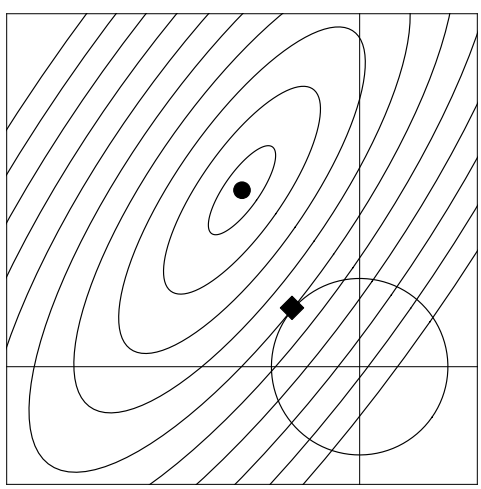
[1] Tikhonov, A. N. (1943). [On the stability of inverse problems (in Russion)]. *Doklady Akademii Nauk SSSR*. **39**:195-198. The application to the least squares problem was proposed by Heorl, A.E. and Kennard, R.W. (1970) "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, **12**(1):55-67.

1. When $\lambda \to 0$, then $\mathcal{E}_{\text{ridge}} \to \mathcal{E}$. The optimized solution to 5.6 will be the same as the solution to 5.4. This is an unconstrained solution, equivalent to $K \to \infty$ in 5.5.

2. When $\lambda \to \infty$ the ridge potential $\mathcal{E}_{\text{ridge}}$ is also driven to ∞ unless we drive the coefficients $b_i \to 0$. Thus the only way to minimize $\mathcal{E}_{\text{ridge}}$ as λ becomes large is to constrain all the b_i to a small neighborhood of the origin. This is equivalent to driving $K \to 0$.

3. The typical situation will be somewhere in-between cases 1 and 2. The solution to the unconstrained problem is typically a point $\hat{\mathbf{b}}$ in $\mathbf{b}$ space that is distinct from the origin, and K is some number that is smaller than $|\hat{\mathbf{b}}|$. The constraint is that all parameters fall in a unit ball of radius K about about the origin, while the usual least squares solution is at some distance outside this ball. In two dimensions (easiest to visualize) the level contours of $(\mathbf{Ab\text{-}y})^{\mathrm{T}}(\mathbf{Ab\text{-}y})$ will be ellipses, and the constraint will correspond to a circle of radius K centered at the origin. The optimal solution will occur at precisely the point where the circle of radius K is tangent to a level curve of $\mathcal{E}$.

Figure 5.1: Two dimensional representation of $\mathbf{b}$ space showing the level curves of the objective function for normal least squares regression. In ridge regression, the optimal solution for a normal least squares fit (the black dot) is moved to the first point of tangency of the level curves to the constrained circle of radius K about about the origin (diamond). So long as the ball of radius K does not encompass the black dot, the components of $\mathbf{b}$ tend to move toward the origin.



To see why this helps in the case of an ill-conditioned system, we will examine

what happens to the least squares solution. The equations are identical to those in chapter 1 except for the last term. Equation 2.16 becomes

$$\mathcal{E}(b_0, b_1, \ldots, b_{p-1}) = (\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}(\mathbf{Ab} - \mathbf{y}) + \lambda \mathbf{b}^{\mathrm{T}}\mathbf{b} \tag{5.7}$$

Differentiating with respect to b_i and setting $\partial\mathcal{E}/\partial b_i = 0$ (as we did in equation 2.19; refer to chapter 2 for calculation of the first part of the derivative)

$$0 = \frac{\partial \mathcal{E}}{\partial b_i} = (\mathbf{a}_i)^{\mathrm{T}}(\mathbf{Ab} - \mathbf{y}) + (\mathbf{Ab} - \mathbf{y})^{\mathrm{T}}\mathbf{a}_i + \frac{\partial}{\partial b_i}(\lambda \mathbf{b}^{\mathrm{T}}\mathbf{b}) \tag{5.8}$$

$$= -2\mathbf{a}_i^{\mathrm{T}}\mathbf{y} + 2\mathbf{a}_i^{\mathrm{T}}\mathbf{Ab} + 2\lambda b_i \tag{5.9}$$

The last term follows because $\mathbf{b}^{\mathrm{T}}\mathbf{b} = \sum b_i^2$. Since this equation holds for all i, we can convert the column vectors into the matrix form, giving

$$\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{Ab} + \lambda\mathbf{b} = (\mathbf{A}^{\mathrm{T}}\mathbf{A} + \lambda\mathbf{I})\mathbf{b} \tag{5.10}$$

where $\mathbf{I}$ is an identity matrix. Solving for $\mathbf{b}$ gives

$$\mathbf{b} = (\mathbf{A}^{\mathrm{T}}\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y} \tag{5.11}$$

Thus instead of finding the inverse of an ill-conditioned matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, we add a small parameter to the diagonal elements of the matrix to make it better suited.

Example of a Adding Diagonal Elements

To illustrate how this conditions the matrix, suppose that we have a simple system with two raw data points: (3,5) and (6.01, 10). The second data point is almost exactly twice the second. then

$$\mathbf{A} = \begin{bmatrix} 3 & 5 \\ 6.01 & 10 \end{bmatrix} \tag{5.12}$$

and

$$\mathbf{A}^{\mathrm{T}}\mathbf{A} = \begin{bmatrix} 34 & 68.03 \\ 68.03 & 136.1201 \end{bmatrix} \tag{5.13}$$

Using **`np.linalg.inv`**, we can find the inverse:

$$(\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1} = \begin{bmatrix} 54448.03999517 & -27211.99999759 \\ -27211.99999759 & 13599.99999879 \end{bmatrix} \tag{5.14}$$

If the original y values were, say, 1 and 1.1, then the solution for the least squares coefficients would be

$$\begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\,\mathbf{y} \tag{5.15}$$

$$= \begin{bmatrix} 54448.03999517 & -27211.99999759 \\ -27211.99999759 & 13599.99999879 \end{bmatrix} \begin{bmatrix} 3 & 6.01 \\ 5 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ 1.01 \end{bmatrix}$$

$$= \begin{bmatrix} 87908.11243221 \\ -43934.53199611 \end{bmatrix} \tag{5.16}$$

When a small parameter is added to the diagonal,

$$\mathbf{A}^{\mathrm{T}}\mathbf{A} = \begin{bmatrix} 34+\lambda & 68.03 \\ 68.03 & 136.1201+\lambda \end{bmatrix} \tag{5.17}$$

these numbers rapidly decrease. We can simulate this in Python:

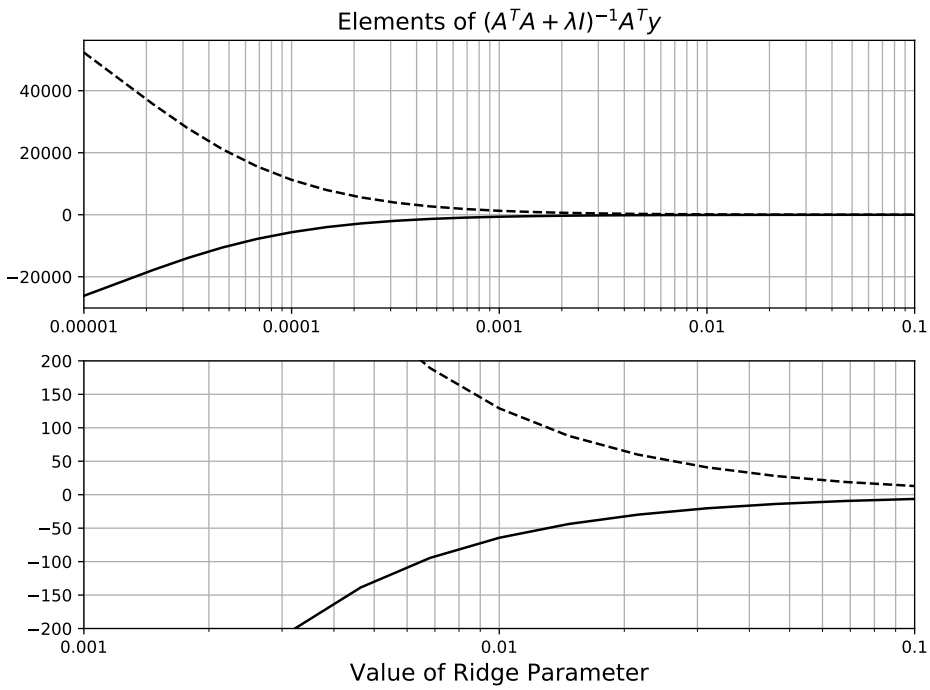
```
import numpy as np
import numpy.linalg as LA
def toy_ridge(lam,A,y):
    AT=A.T
    U=A.dot(AT)+lam*np.identity(2)
    UINV=LA.inv(U)
    return((UINV.dot(AT)).dot(y))
```

The function **toy_ridge** returns the two values of $(\mathbf{A}^{\mathrm{T}}\mathbf{A}+\lambda\mathbf{I})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y}$ as a function of the parameter **lam**, the matrix **A**, and the vector **y**. We can use the example above as input and loop through several orders of magnitude parameter using the **numpy geomspace** function. The following loops through 25 values of λ from 0.00001 to 0.1 that are geometrically spaced (each successive value is the same multiple of the previous value) and collects the results in the arrays **first** and **second**.

```
A=np.array([[3,5],[6.01,10]])
y=np.array([1,1.1])
first=[]
second=[]
xvals=np.geomspace(.00001,.1,25)
for lam in xvals:
    y1,y2=toy_ridge( lam, A, y)
    first.append(y1)
    second.append(y2)
```

The two resulting parameters are plotted on two different y-scales in figure 5.2. There is some controversy as to where it is appropriate to turn off the procedure, i.e., which value of λ to pick as optimal, although the choice with the minimal MSE is sometimes preferred.

Figure 5.2: Elements of the vector $(\mathbf{A}^{\mathrm{T}}\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y}$ as a function of the parameter λ are plotted for $.00001 < \lambda < .01$ (top) and $.001 < \lambda < .01$ (bottom).



Ridge Regression Example in Python

We begin by considering the automobile miles per gallon data set at the UCI machine learning repository.[2] Each of the 398 vehicles listed in the are described by 8 features: miles per gallon; number of cylinders; engine displacement; horsepower; weight; acceleration; model year; origin; and car name. The number of cylinders is integer; the model year is a year; the origin is an integer that describes the country of origin; and the car name is a string. The remaining values are real (floating point) values. Some values are missing and are represented by question marks in the data file, which has fixed width columns. '

We can read the data into a **`pandas DataFrame`** using **`read_fwf`**. We need to

[2]The file is available at `https://archive.ics.uci.edu/ml/datasets/auto+mpg`. The original is documented at Quinlan, R (1993) "Combining Instance-Based and Model-Based Learning," in *Proceedings on the Tenth International Conference of Machine Learning*, 236-243, University of Massachusetts, Amherst.

tell **pandas** that the data file uses the **"?"** for missing data using the **na_values** option, as this is not a standard format. This will replace the **"?"** with Python **nan** symbols in the appropriate locations. Since the columns are not labeled in the data file, we also need to tell that to **pandas** using the option **header=None**, since normally the first line of the data file contains the labels. Finally, we will remove all rows with missing data using the method **dropna** (with **axis=0**, meaning drop all rows with **nan** in them; the alternative is to set **axis=1** to drop the columns).

```
import pandas as pd
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
   machine-learning-databases/auto-mpg/auto-mpg.data",
   header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
  "model","origin","carname")
data = data.dropna(axis=0)
```

For the present analysis, we will deal only with the numerical data and not the flags. We will extract those columns by name and copy them into a new **DataFrame** called **cardata**.

```
cardata=data[["mpg","cyl","displ","hp","weight","accel"]].copy()
```

We can make a scatter matrix matrix of the data to see how the variables appear to relate to one another using **pd.plotting.scatter_matrix**.

```
rr = pd.plotting.scatter_matrix(cardata, figsize=(10,10),
                                marker="o",s=10,c="black")
```

The result is shown in figure 5.3. From the figure, some of the features look like they could be derived from some of the other features.

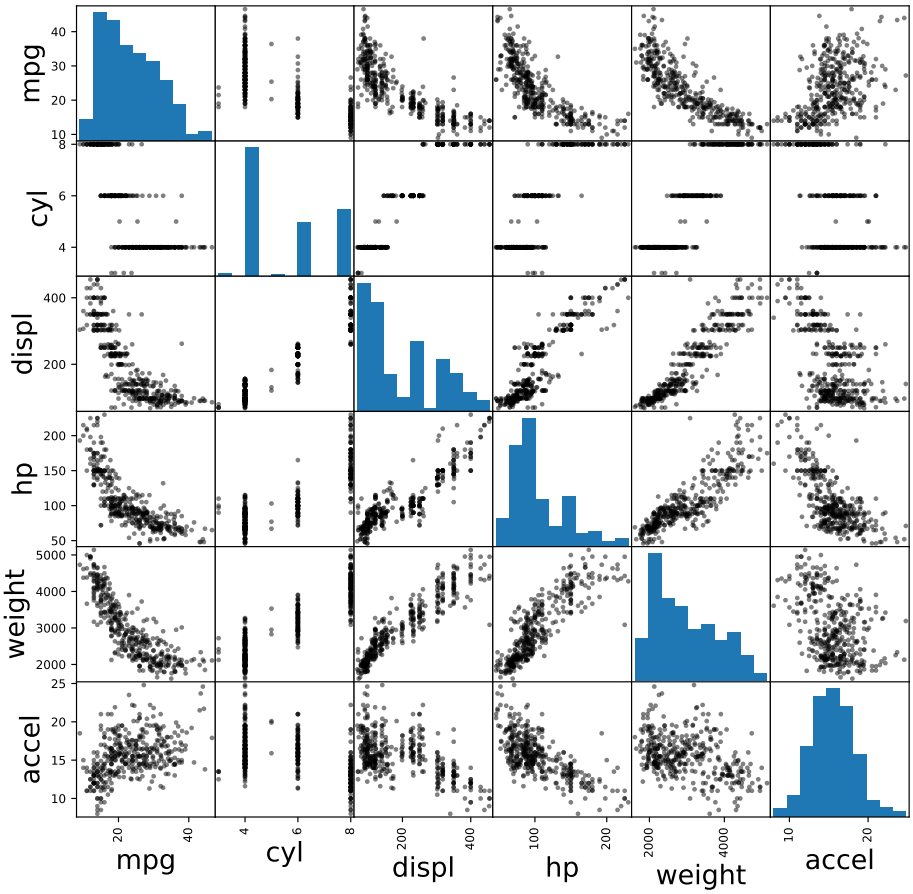
Our goal is to predict the miles per gallon **mpg** column from the other numerical parameters. First we will define our **X** and **Y** data arrays:

```
X=np.array(cardata[["cyl","displ","hp","weight", "accel"]])
Y=np.array(cardata["mpg"])
n=len(Y)
```

The we extract test and training sets. As usual, we will reserve 25% of the data for the test set, and use the rest for training.

```
import random
indices=list(range(n))
random.shuffle(indices)
ntrain=round(.75*n)
```

Figure 5.3: A **pandas** scatter matrix plot of the MPG data set.




```
training_indices=indices[:ntrain]
test_indices=indices[ntrain:]
XTRAIN=X[training_indices]
YTRAIN=Y[training_indices].reshape(-1,1)
XTEST=X[test_indices]
YTEST=Y[test_indices].reshape(-1,1)
```

For comparison, we will start with a straightforward multilinear regression as we did in chapter 2 using **sklearn**.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
r=LinearRegression().fit(XTRAIN,YTRAIN)
print("intercept:", r.intercept_)
print("Coefficients:", r.coef_)
```

```
intercept: [44.30044754]
Coefficients: [[-0.65190687  0.00126933 -0.0270908  -0.00549576
0.11840116]]
```

```
P=r.predict(XTEST)
R2=r2_score(YTEST,P)
MSE = mean_squared_error(YTEST,P)
print("R2:",R2)
print("MSE:", MSE)
```

```
R2: 0.6149567770383708
MSE: 19.78151940492196
```

The model is (approximately)

$$\text{mpg} = 44.3 - 0.652 \times \text{cyl} + 0.00127 \times \text{displ} - 0.0271 \times \text{hp} - \\ 0.00550 \times \text{weight} + 0.118 \times \text{accel}$$

and has an MSE of approximately 19.78.

Next, we will try to do better with ridge regression. Ridge regression can be performed using the function **ridge** in **sklearn.linear_model**.[3]

If we pick the value that we want to use for the ridge coefficient (the parameter **alpha** in **ridge**) then we can do ridge regression very easily:

```
from sklearn.linear_model import Ridge
r=Ridge(alpha=1)
print("intercept:", r.intercept_)
print("Coefficients:", r.coef_)
```

```
intercept: [44.28441584]
Coefficients: [[-0.64395933  0.00115032 -0.02706653 -0.00549747
0.11827546]]
MSE: 19.776001061911092
```

The default value for **alpha** is 1, and using the default value does not give any significant improvement in the MSE.

In order to do better, we can write a little code block to loop through values using **np.logspace**. We want to use **np.logspace** so that we can loop over several orders of magnitude.[4] The general format is similar to **np.linspace**, except that we specify the logarithms of the endpoints:

```
np.logspace(start,stop,num)
```

[3]Full documentation is given online at `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html`

[4]See `https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.logspace.html` for details.

means to return an array of **num** points starting at 10^{start} and ending at 10^{stop}. (It is possible to switch to other bases besides base 10.) The steps are equally space in log (logarithm) space.

Since we don't have any idea what value to to use we'll generate 100 values evenly separated (in log space) over seven orders of magnitude. Then we will aggregate the coefficients and the mean square errors as we go along. Our goal will be to plot the MSE as a function of the parameter **alpha**, and then to pick out the set of coefficients that give the smallest MSE. For each value of **alpha**, we will fit the data with the training set, then figure out the MSE with the testing data. We'll append the error to the array **errs** and append the list of coefficients to the array **coefs**.

In order to clearly see what is happening, we need to plot each coefficient as a function of **alpha**. Since each set of coefficients was aggregated as a row of **coefs**, we can take the transpose of the entire array so that the history of each coefficient is given in a successive row of **coefs**. We plot the absolute value of each coefficient so that we can display the y axis in a logarithmic scale. If we don't do this the plot will not be very informative. We also want to display the x axis in a logarithmic scale to show all the values of the parameter **alpha**. The result is shown in the top plot of figure 5.4.

```
alphas = np.logspace(-1,6,100)
coefs=[]
errs=[]
for a in alphas:
    r.set_params(alpha=a)
    r.fit(XTRAIN,YTRAIN)
    coefs.append(list(r.coef_[0]))
    PRED=r.predict(XTEST)
    MSE = mean_squared_error(YTEST,PRED)
    errs.append(MSE)
```

Here is the code for plotting the absolute value of the coefficients as function of the ridge parameter.

```
coefs=np.array(coefs).T
for row in coefs:
    plt.plot(alphas, np.abs(row))
plt.xscale("log")
plt.yscale("log")
plt.grid()
plt.ylabel("Absolute Value of Fit Coefficient", fontsize=12)
plt.xlabel("Value of Fit Coefficient", fontsize=12)
plt.plot([6000,6000],[.00001,1],ls=":",c="k")
plt.ylim([.00001,1])
```

A plot of the MSE as a function of ridge coefficient is shown in bottom of figure 5.4 (the code for this is not shown here; see the notebook for this chapter on github for complete code). A closer examination of the minimum (see the notebook) shows that the minimum MSE occurs when **alpha=6000** (approximately, to one significant figure). A vertical dotted line is drawn on both plots in figure 5.4 at this point. We can plug this number back into **ridge** to get the revised fit:

```
r=Ridge(alpha=6000)
r.fit(XTRAIN,YTRAIN)
print("intercept:", r.intercept_)
print("Coefficients:", r.coef_)
r.fit(XTRAIN,YTRAIN)
PRED=r.predict(XTEST)
MSE = mean_squared_error(YTEST,PRED)
print("MSE:",MSE)
```

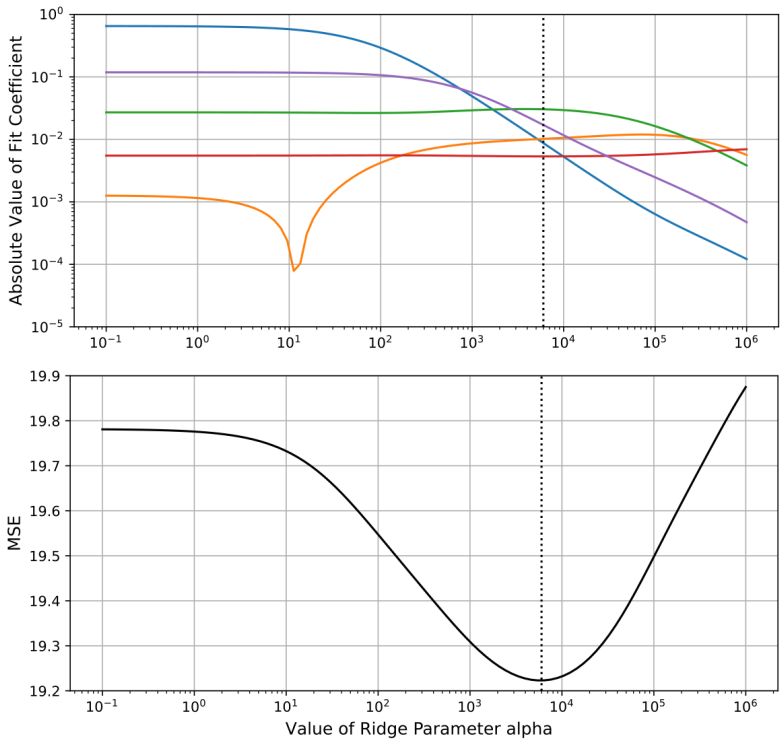
```
intercept: [44.45672731]
Coefficients: [[-0.00872798 -0.01020507 -0.03033813 -0.00534507
0.01700801]]
MSE: 19.222977416869142
```

The equation for the revised model is (approximately)

$$\text{mpg (revised)} = 44.5 - 0.00873 \times \text{cyl} - 0.0102 \times \text{displ} + 0.0303 \times \text{hp} - 0.00534 \times \text{weight} + 0.0170 \times \text{accel}$$

Important Note: The results we have shown here are strongly dependent on the randomness of test/train split. If you run the calculation over again, you will find that the minimum occurs at a different location. You should perform the calculation *many* times to get a statistical average.

Figure 5.4: Result of varying the ridge parameter **`alpha`** over several orders of magnitude. Top: absolute value of regression coefficients. It is not unusual for the values of individual components to flip signs as they move towards the origin. Since this is a logarithmic plot of the absolute value, sign flipping is illustrated by a dip and turn (such as the notch around alpha=10). Eventually (for larger value of alpha) all of the coefficients are driven towards zero. Bottom: Mean square error. The vertical dotted line in both graphs shows the location of the minimum MSE. Observe that near the optimal value all of the components of **b** are approximately the same order of magnitude.



Chapter 6

Lasso Regression

Ridge regression, which we discussed in chapter 5 is a special case of a class of methods known as **P-norm regularization**. In these methods our goal is to find the optimal model (i.e., choice of coefficients a and $\mathbf{b}$= $[b_0, b_1, \ldots, b_{p-1}]^{\mathrm{T}}$)

$$y = a + \mathbf{b}^{\mathrm{T}}\mathbf{x} \tag{6.1}$$

by constraining the least squares objective function

$$\mathcal{E} = \sum_{i=0}^{n-1} |y_i - a - \mathbf{b}^{\mathrm{T}}\mathbf{x}_i|^2 \tag{6.2}$$

such that

$$\big(||\mathbf{b}||_P\big)^P = \sum_{i=0}^{p-1} |b_i|^P < K \tag{6.3}$$

for some constant K. Here there are n data points $(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{n-1}, y_{n-1})$, where each $\mathbf{x}_j$ has p features. In ridge regression, we use the 2-norm, and the dual problem becomes

$$\mathcal{E}_{\text{ridge}} = \sum_{i=0}^{n-1} |y_i - a - \mathbf{b}^{\mathrm{T}}\mathbf{x}_i|^2 + \lambda|\mathbf{b}|^2 \tag{6.4}$$

In **lasso regression**[1] this becomes

$$\mathcal{E}_{\text{lasso}} = \sum_{i=0}^{n-1} |y_i - a - \mathbf{b}^{\mathrm{T}}\mathbf{x}_i|^2 + \lambda|\mathbf{b}| \tag{6.5}$$

[1]Tibshirani, R. (1996). "Regression Shrinkage and Selection via the lasso." *Journal of the Royal Statistical Society. Series B (methodological).* **58**: 267-88.

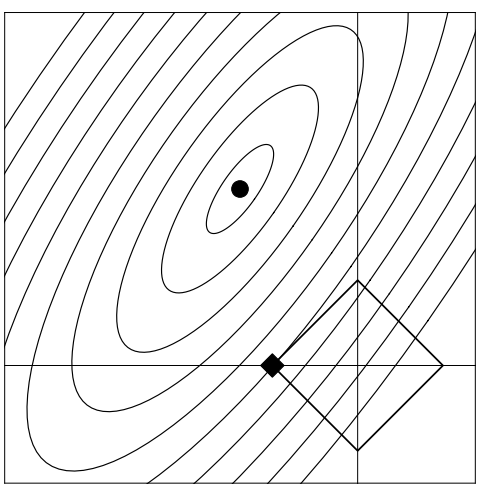
Sometimes the two methods are combined in an **elastic net**:[2]

$$\mathcal{E}_{\text{Elastic Net}} = \sum_{i=0}^{n-1} |y_i - a - \mathbf{b}^{\mathrm{T}}\mathbf{x}_i|^2 + \lambda|\mathbf{b}| + \mu|\mathbf{b}|^2 \tag{6.6}$$

The derivation of the coefficients **b** for ridge regression in chapter 5 needs to be modified by replacing the term $(\partial/\partial b_i)\lambda|\mathbf{b}|^2 = \lambda b_i$ with $(\partial/\partial b_i)\lambda|\mathbf{b}|$, which equals 1, 0, or -1, depending on whether b_i is positive, negative or zero. We will not present the calculations here.

Most of the same remarks that were made about ridge regression in chapter A also apply here. However, the constraint given by the 1-norm is now a diamond centered about the origin with corners on the coordinate axis, rather than a circle (it becomes a polyhedron in higher dimensions).

Figure 6.1: Two dimensional representation of **b** space showing the level curves of the objective function for normal least squares regression. In lasso regression, the optimal solution for a normal least squares fit (the black dot) is moved to the first point of tangency of the level curves to the constrained diamond that intersects the coordinate axes at points $b_i = K$. So long as this dimaond does not encompass the black dot, the components of **b** tend to move toward the origin. Compare with figure 5.1



The new situation with lasso regression is illustrated by comparing figure 6.1 with figure 5.1. In both figures, the ellipses give the level curves corresponding

[2]The elastic net has is reducible to a support vector machine. See Q. Zhou et al. (2014) *A Reduction of the Elastic Net to Support Vector Machines with an Application to GPU Computing* `https://arxiv.org/abs/1409.1976`

to unconstrained linear regression. In the absence of any constraint, the optimal solution is found at the black dot in the center of all the contours. By adding a constraint, the solution must be within the constraining p-ball. In ridge regression, this was a point of tangency that was closer to the origin than the unconstrained solution. In lasso regression, the first point where a contour touches the constraining ball is always going to be on a corner, and the corners always lie on a coordinate axis. So the lasso will force some parameters to zero. This effectively removes some features from the problem and reduces the dimension of the solution. This is sometimes call **dimensionality reduction** or **feature reduction**. The parameters that have been set to zero correspond to features or dimensions in feature space that the lasso solution deems are extraneous or unnecessary to the solution of the problem. If the original data is, in fact, truly multicollinear, then we would expect that there should be dimensions that need to be removed.

Lasso Regression in Python

We'll illustrate the lasso in Python with the same example we used to illustrate the ridge algorithm. Let's assume we've loaded the **mpg** data file and defined the training and test set as we did in chapter 5 (see page 67). We can run the lass algorithm using the function **lasso** from **sklearn.linear_model**.[3]

```
from sklearn.linear_model import Lasso
```

The code is similar to the code for **Ridge**. The default parameter value is **alpha=1**.

```
r=Lasso(alpha=1)
r.fit(XTRAIN,YTRAIN)
PRED=r.predict(XTEST)
MSE = mean_squared_error(YTEST,PRED)
print("intercept:", r.intercept_)
print("Coefficients:", r.coef_)
print("MSE:",MSE)
```

```
intercept: [45.5965519]
Coefficients: [-0.         -0.00460263 -0.05139156 -0.00531516
0.        ]
MSE: 18.19745098344398
```

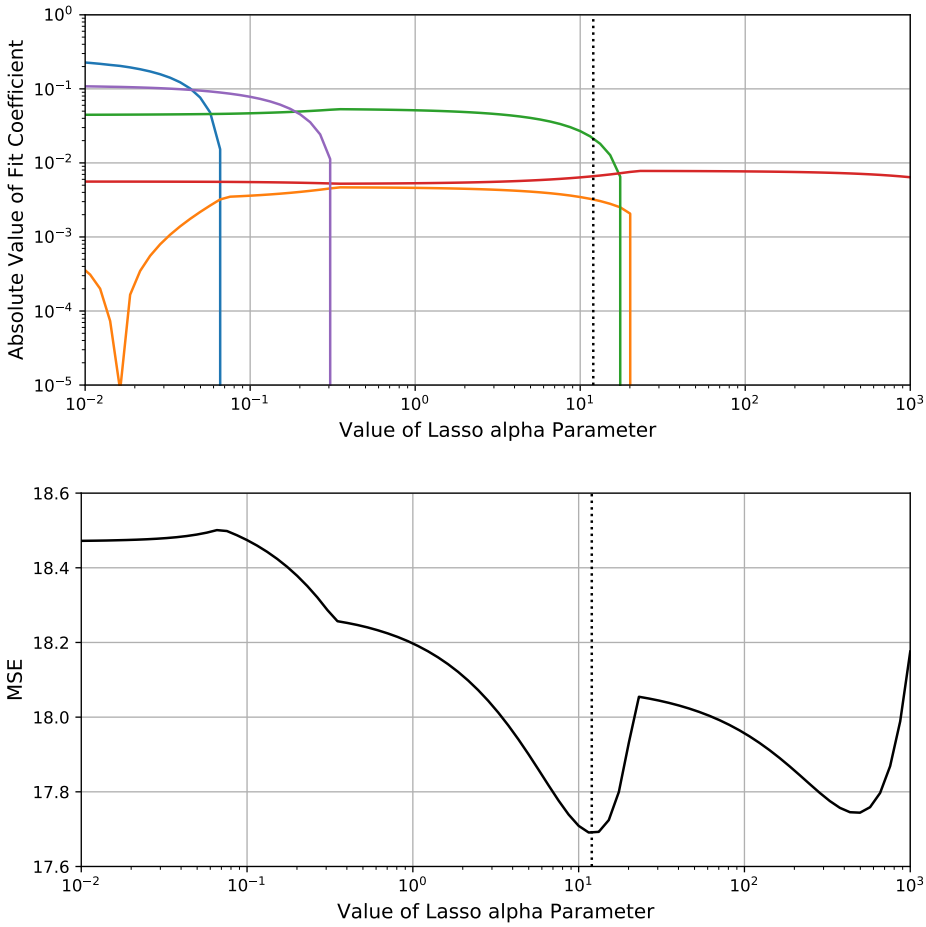
[3]Full documentation and examples of the **Lasso** function are given at https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

At this value of **alpha** two of the coefficients have already been set to zero. To compare the MSE with other values, we repeat our test of running through a logarithmic sequence of values. We'll illustrate from 0.001 through 1000 using **logspace(-3,3,100)** (six orders of magnitude):

```
alphas = np.logspace(-3,3,100)
coefs=[]
errs=[]
intcpts=[]
for a in alphas:
    r.set_params(alpha=a)
    r.fit(XTRAIN,YTRAIN)
    coefs.append(list(r.coef_))
    intcpts.append(float(r.intercept_))
    PRED=r.predict(XTEST)
    MSE = mean_squared_error(YTEST,PRED)
    errs.append(MSE)
coefs=np.array(coefs).T
```

This code is almost identical to the code we used to explore ridge regression on this data (see page 70). We can plot the resulting coefficients and mean square errors as a function of the Lasso **alpha** value as shown in figure 6.2 (code for generating the plot not shown). As expected, the coefficients are forced to zero one by one as we increase the Lasso **alpha** parameter.

Figure 6.2: Result of varying the lasso parameter **alpha** over several orders of magnitude. Top: absolute value of regression coefficients. It is not unusual for the values of individual components to flip signs as they move towards the origin. Since this is a logarithmic plot of the absolute value, sign flipping is illustrated by a dip and turn (such as the notch around alpha=10). When a coefficient is driven to zero, the curve drops vertically off the bottom of the plot. Bottom: Mean square error. The vertical dotted line in both graphs shows the location of the minimum MSE. Compare with figure 5.4 which solved the same problem with Ridge regression.



Chapter 7

Spline Regression

In cubic spline interpolation we fit a piecewise polynomial to the n data points (x_0, y_0), ..., (x_{n-1}, y_{n-1})

$$f(x) = \begin{cases} S_0(x) = b_{00} + b_{01}x + b_{02}x^2 + b_{03}x^3, & x_0 \leqslant x < x_1 \\ S_1(x) = b_{10} + b_{11}x + b_{12}x^2 + b_{13}x^3, & x_1 \leqslant x < x_2 \\ S_2(x) = b_{20} + b_{21}x + b_{22}x^2 + b_{23}x^3, & x_2 \leqslant x < x_3 \\ \vdots & \\ S_{n-2}(x) = b_{n-2,0} + b_{n-2,1}x + b_{n-2,2}x^2 + b_{n-2,3}x^3, & x_{n-2} \leqslant x \leqslant x_{n-1} \end{cases} \tag{7.1}$$

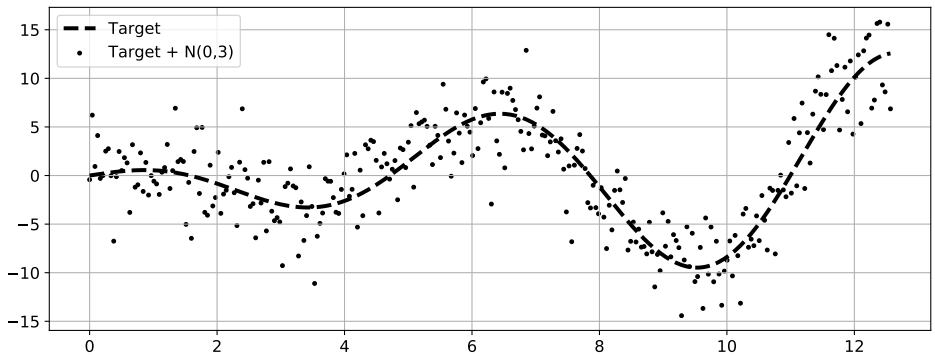
where $a = a_0 < a_1 < \cdots < a_n = b$, subject to the following conditions:

1. $f(x_i) = y_i$ for $i = 0, \ldots, n-1$;
2. The function is continuous at each of intermediate the $n-2$ **knot points** a_1, ..., x_{n-2};
3. The first and second derivative are continuous at each knot points (giving us an additional $2(n-2) = 2n-4$ conditions);
4. The second derivative is equal to zero at the end points a and b (2 additional conditions).

These $(n) + (n-2) + (2n-4) + 2 = 4(n-1)$ conditions are known as **free cubic spline interpolation** conditions. There are the same number of conditions as there are unknown parameters in equation 7.1. It is possible to solve for these coefficients exactly.[1]

[1]The derivation of the spline fit is given in most books on numerical analysis, such as Burden, R.L. and Faires, J.D. (2001) *Numerical Analysis*, Brooks-Cole.

Figure 7.1: Toy data set used to illustrate smoothing splines. The target function is $y = x\cos(x) + \epsilon$. Here ϵ is a random variable with a normal distribution that has a mean of 0 and standard deviation of $\sigma = 3$. For sufficiently large samples, we would expect 95% of the data to fall with 1.96σ of the original curve and nearly all of the data to fall within 3σ of the curve.



In a **smoothed cubic spline** the following objective function is minimized:

$$\mathcal{E}_{\text{smoothed}} = \lambda \sum_{i=0}^{n-2} \left[\frac{y_i - S_i(x)}{\sigma_i} \right]^2 + (1-\lambda) \int_a^b f''(x)^2 dx \tag{7.2}$$

Here the second term is a *roughness penalty*, λ is a relative weight factor, and σ^2 is the expected variance in the noise for the ith data point. When $\lambda \to 0$, the fit will approach a straight line; when $\lambda \to 1$, the fit will approach a full cubic spline that passes through every data point. [2]

In **free cubic spline regression**[3], we set the number of knot points to $K < n$. The knot points do not have to coincide with data points, and the first restriction in eliminated. Instead there are only $3K - 2$ restrictions, down from the much larger $4n - 2$ (typically $n \gg K$). Furthermore, the knot points do not have to coincide with any data point (or data point x values), and may either be pre-set by the user or set automatically by a minimization procedure. Since there are far more data points than conditions the parameters themselves must be fit by a minimization process.

[2]Reinsch, C (1967) "Smoothing by Spline Functions", *Numerische Mathematick*, **10**:177-183; Pollock, D. (1999). "Smoothing with Cubic Splines," in *Handbook of Time Series Analysis, Signal Processing, and Dynamics.*

[3]Green, PJ and Silverman, B (1993) *Nonparametric Regression and Generalized Linear Models*, CRC Press, available online at `https://books.google.com/books/about/Nonparametric_Regression_and_Generalized.html?id=-AIVXozvpLUC`.

Smoothing Splines in Python

We'll begin by building a toy data set that has a normal random distribution about a target function.

```
X=np.linspace(0,4*math.pi,300)
np.random.seed(99)
n=len(X)
Y=X*np.cos(X)+ np.random.normal(0,3,n)
```

A plot of the target function and toy randomized data set is illustrated in figure 7.1. To generate a *single* train/test split, we can use our usual code block:

```
n=len(X)
indices=list(range(n))
random.shuffle(indices)
ntrain=round(.75*n)
ntest=n-ntrain
training_indices=indices[:ntrain]
training_indices.sort()
test_indices=indices[ntrain:]
test_indices.sort()
XTRAIN=X[training_indices]
YTRAIN=Y[training_indices]
XTEST=X[test_indices]
YTEST=Y[test_indices]
```

In Python the user can specify the location of the knot points and do a least squares spline fit using the function **`LSQUnivariateSpline`** in the package **`scipy.interpolate`**. The basic format[4] is

```
LSQUnivariateSpline(X,Y,t)
```

where **`X`** and **`Y`** are the x and y data arrays, and t is an array of x values for the knot points. The knot points must be ordered such that $t_0 < t_1 < t_2 < \cdots$. We illustrated with two different *arbitrary* selections of knot points: 3 knots at $t = 2$, $t = 6$, and $t = 10$; and 12 knots equally spaced from $t = 1$ through $t = 12$:

```
t1=[2,6,10]
t2=np.linspace(1,12,12)
```

Now we will generate functions that correspond to each set of knots. We only use the training data to generate the fit functions.

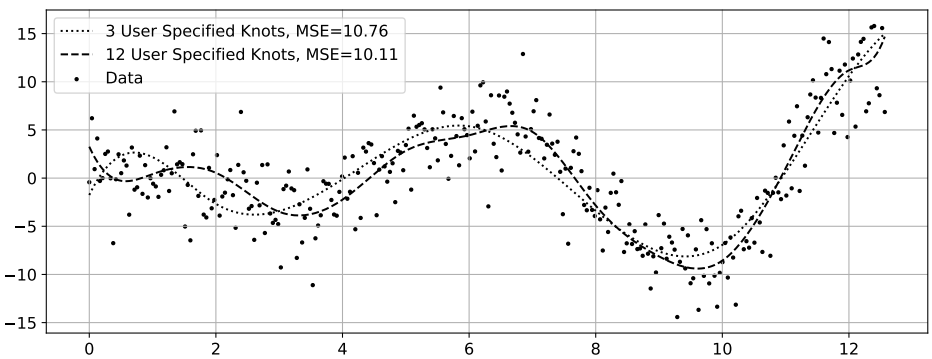
[4]Full documentation can be found at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.LSQUnivariateSpline.html`.

```
from scipy.interpolate import LSQUnivariateSpline as LS
fit1=LS(XTRAIN,YTRAIN,t1)
fit2=LS(XTRAIN,YTRAIN,t2)
```

Each of the two functions **fit1** and **fit2** can be used to generate predictions, either on single numbers or on arrays. Thus **fit1(12)** will return a single value, with **fit1(XTEST)** will return an entire array of values, one number corresponding to each value in **XTEST**. We will use the test data to produce predictions.

```
pred1=fit1(XTEST)
pred2=fit2(XTEST)
```

Figure 7.2: Spline fits plotted on entire data set, generated using 3 and 12 user supplied knot locations. The fits were produced on the test data, then applied to the entire data set for illustration. Consequently the actual shape of the fit will vary depending on the randomization training/test split, as will the overall mean squared error.



Finally, we can calculate the mean squared error as

$$\mathrm{MSE} = \frac{1}{n_{\mathrm{test}}} \sum_{i \in \mathrm{test}} \left(y_{i,\mathrm{predicted}} - y_{i,\mathrm{data}}\right)^2 \tag{7.3}$$

```
MSE1=sum((pred1-YTEST)**2)/ntest
MSE2=sum((pred2-YTEST)**2)/ntest
print("Mean Squared Error MSE1 =", MSE1)
print("Mean Squared Error MSE2 =", MSE2)
```

```
Mean Squared Error MSE1 = 9.938314251388844
Mean Squared Error MSE2 = 9.460801051538377
```

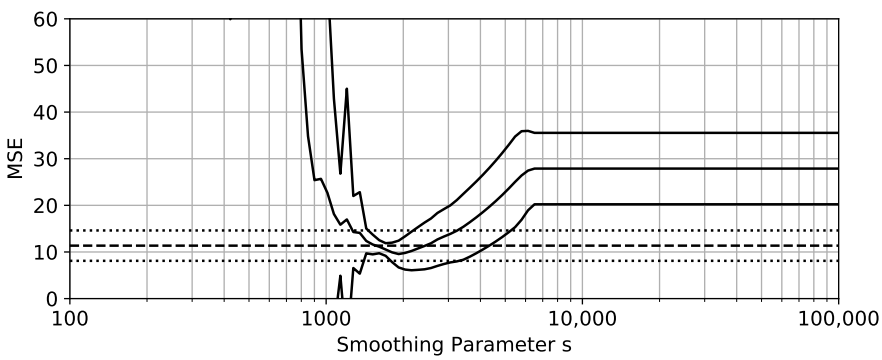
The actual value of the mean squared error will vary with the randomization split. Illustrations of the typical fit are shown in figure 7.2.

The Python function **UnivariateSpline** will arbitrarily determine the number of knots depending on a floating point parameter **s**.[5]. When **s** is small, the fit will approach an interpolating spline with knots at all the points; and when **S** is large the fit will *approach* a single straight line.[6] The function **UnivariateSpline** also returns a function that can be applied to a single data value or an array or list of values to obtain predictions. A single fit is performed with the call

```
UnivariateSpline(XTRAIN, YTRAIN, s=sval)
```

where **sval** is the desired parameter value. The effect of varying the spline parameter **sval** using the toy data set is illustrated in figures 7.3 and 7.4. Figure 7.3 shows the mean square error as a function of the spline parameter $\pm 1.96\sigma$ after doing 25 train/test splits. Each train/test split has a different fit, hence there is some variation in the prediction. Sample plots for a single split as a function of **s** are illustrated in figure 7.4. For low values of **s**, the spline fits the noise and not the underlying pattern. It gets pulled towards every single data points (top frame of figure 7.4). For extremely large values of **s** the fit will only return two knots so that the fit will always resemble a single cubic with two turning point (bottom frame of figure 7.4). The optimal fit is at an intermediate value, somewhere around **s**=3000 (for this data set). The

Figure 7.3: Dependence of Mean Square Error (MSE) on smoothing parameter.



The number of knots is shown in figure 7.5 as a function of smoothing parameter for the toy data set, using a single train/test split. For this data set, the number

[5]Full documentation can be found at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.UnivariateSpline.html`

[6]In reality, the function will always return a minimum of two knots, so that there is still a spline fit, so the resulting fit is never a straight line, but will, overall, resemble a single cubic with two turning points.

of knots falls off rapidly with **s** between around 1500 and 3000. This corresponds to the valley in the MSE shown in figure 7.3.

Figure 7.4: Effect of changing the spline parameter on the spline fit for various values of the spline parameter. For small values of the spline parameter, the curve is dominated by the noised and is pulled towards every point. For larger values, the curve is dominated by the me

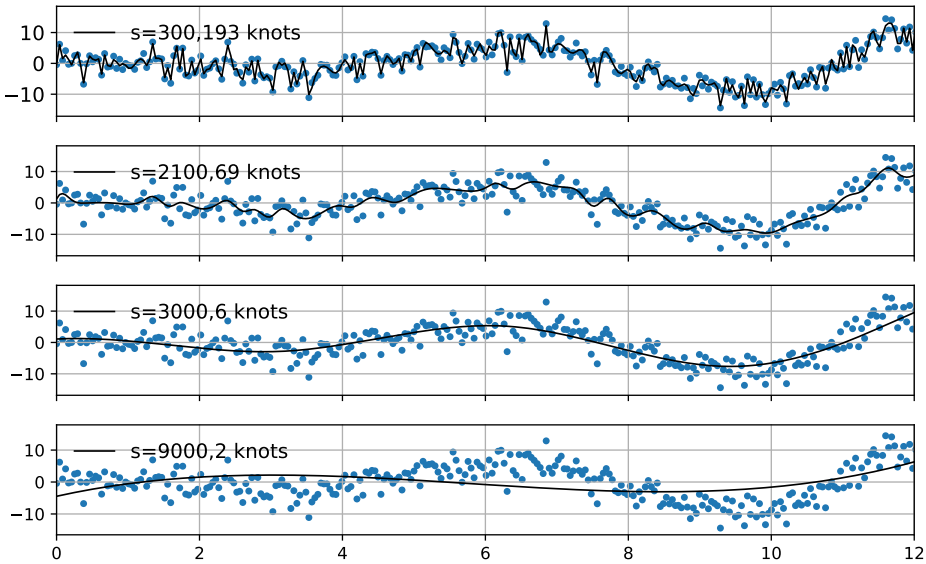
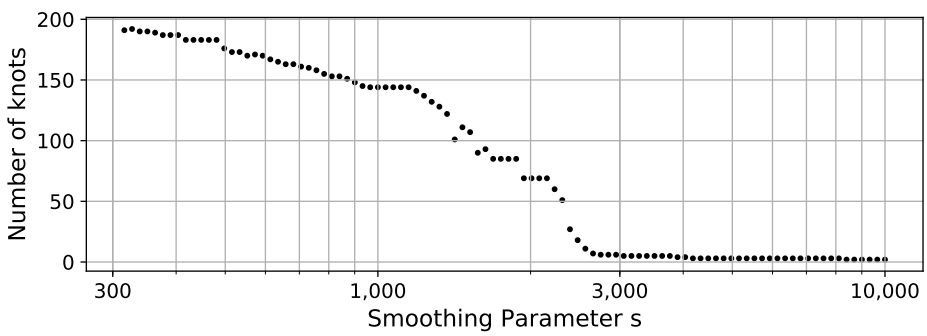


Figure 7.5: Number of knots as a function of smoothing parameter for a single train/test spit, us



Chapter 8

Nonlinear Regression

The goal of least squares nonlinear regression is the same as linear regression. Given a set of n data points (x_0, y_0), $\ldots$, (x_{n-1}, y_{n-1}), find the parameters $a_0, a_1, \ldots$ such that the objective function

$$\mathcal{E} = \sum_{i=0}^{n-1} |f(x_i; a_1, a_2, \ldots) - y_i|^2 \tag{8.1}$$

is minimized, where $f(x_i)$ depends nonlinearly on the parameters a_j.

The problem in nonlinear regression is that when the partial derivatives with respect to the unknown parameters $a_1, a_2, \ldots$ are calculated, they are not linear in $a_1, a_2, \ldots$. Hence it is rarely possible to find explicit analytic formulas for the solutions. An example of a nonlinear function that is commonly fit to data in chemistry and biology is the **Hill** function. A Hill function has the form

$$y = \frac{x^m}{a^m + x^m} \tag{8.2}$$

Here the unknown parameters are a and m, where $a > 0$ and m can either positive or negative. Typically a Hill function occurs in an n-step reaction and gives the relative rate at which a reaction occurs, but there are also situations where non-integral values of m will occur. The corresponding objective function is then

$$\mathcal{E} = \sum \left[\frac{x_i^m}{a^m + x_i^m} - y_i \right]^2 \tag{8.3}$$

The equations for the partial derivatives $\partial\mathcal{E}/\partial a = 0$ and $\partial\mathcal{E}/\partial m = 0$ are highly nonlinear and we cannot easily solve for an an explicit solution for a and m.

Nonlinear Optimization

Instead of the matrix formulation used in linear or polynomial regression, a numerical iteration is required when the objective is not linear in the coefficients. The most commonly used procedures are generally variations of **gradient descent**, sometimes also called the **method of steepest descent**. In these methods, a first guess (often supplied by the user) is made for the parameters. This could be as simple as setting all unknowns equal to 1. Then partial derivatives $\partial\mathcal{E}/\partial a_i$ are calculated with respect to all unknown parameters. An incremental update to all parameters is made that takes the value of $\mathcal{E}$ **downhill**, i.e., in such a way so as to decrease its value. If no such value can be found that $\mathcal{E}$ is already at a local minimum. The process is then repeated iteratively until $\mathcal{E}$ cannot be decreased any further or until a convergence criterion is met.

Gradient descent is analogous to climbing down a hill. Wherever you are standing, look for the greatest slope down the hill and follow it. Imaging pouring a pitcher of water on the ground, and following its path. It will follow the path of steepest descent to a local minimum. If you are climbing the hill, look for the steepest slope up (in this case the analogous method is called the **method of steepest ascent**). In one dimension, we pick a small number η (the **learning rate** and a first guess x_0, then take successive guesses as

$$x_{n+1} = x_n - \eta f'(x) \tag{8.4}$$

In multidimensional optimization equation 8.4 is replaced by

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}) \tag{8.5}$$

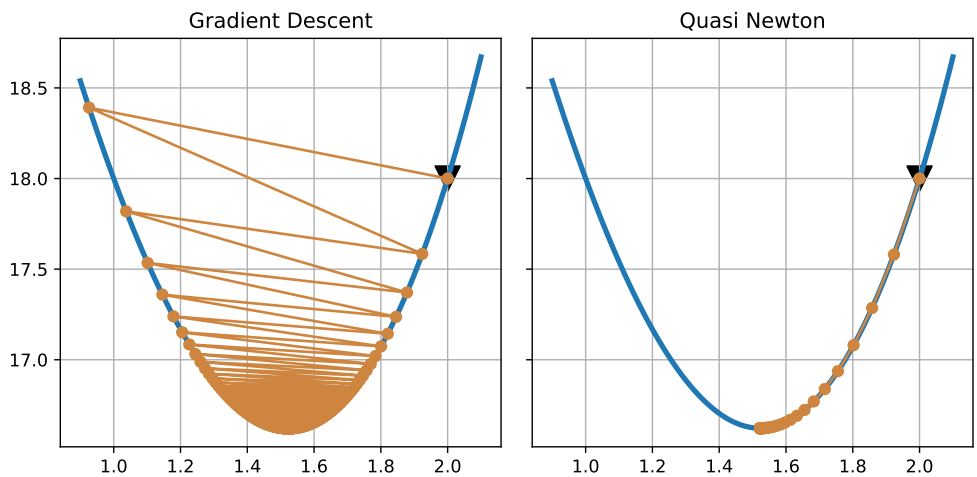
where ∇ is the vector gradient operator. The problem with gradient descent is that if the step size is too small it may take a very long time to converge, and if the step size is too large it will jump past the minimum and bounce back and forth across the minimum rather than converging directly towards the minimum (figure 8.1, left). In the worst case it could get stuck bouncing back and forth between two points.

One common improvement to gradient descent methods is to modify the value of η as the iteration proceeds to take this overstepping into account. Suppose we have some guess at x_n and we want to take our next guess at some point $x = x_n + \Delta x$. A Taylor series expansion about x_n gives

$$f(x) \approx f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)(\Delta x)^2 \tag{8.6}$$

$$\approx f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2 \tag{8.7}$$

Figure 8.1: Comparison of gradient descent (left) and quasi-Newton (right) methods for $f(x) = (x-5)^2 + x^3 + 1$. The gradient descent method method requires 3220 iterations while the quasi-Newton method only required 106 iterations to converge to step of 10^{-10} starting from $x_0 = 2$. Both methods used $\eta = .179$.



Since $f(x_n)$, $f'(x_n)$, and $f''(x_n)$ are constants,

$$f'(x) \approx f'(x_n) + f''(x_n)(x - x_n) \tag{8.8}$$

Ideally we would like to pick our next iteration x_{n+1} is at the minimum. But if this is true, then at the minimum $f'(x_{n+1}) = 0$. If $x = x_{n+1}$ then our minimization condition is $f'(x) = 0$. Hence, setting $x = x_{x+1}$

$$0 = f'(x_n) + f''(x_n)(x_{n+1} - x_n) \tag{8.9}$$

Solving for x_{n+1} gives

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{8.10}$$

In multiple dimensions the equivalent equation is

$$x_{n+1} = x_n - \mathbf{H}^{-1}\nabla f(x_n) \tag{8.11}$$

where ∇ is the gradient operator and $\mathbf{H}$ is the Hessian matrix of second partial derivatives. A number of methods have been developed to work around the fact that it is difficult, if not impossible, to compute the Hessian matrix efficiently in most cases. Many of these are variations on secant approximations to the derivatives. Examples include Broyden's[1] and the BFGS (Broyden-Fletcher-

[1]Broyden, CG. "A Class of Methods for Solving Nonlinear Simultaneous Equations." *Mathematics of Computation. American Mathematical Society.* **19**(92): 577-593 (1965).

Goldfarb-Shanno) method.[2] Some other methods use randomization; examples include stochastic gradient descent[4] and simulated annealing,[5] but they are beyond the scope of this text.

Here is an example of gradient descent code in Python:

```
def gradient_descent(f,df,x0,eta=0.001,nmax=25,conv=1E-10):
    xold=x0
    results=[]
    for i in range(nmax):
        F = f(xold)
        DF = df(xold)
        xnew = xold - eta * DF
        delta = abs(xnew-xold)
        if delta < conv:
            break
        results.append([xold, F])
        xold=xnew
    print(i,"iterations, delta=",delta)
    return(results)
```

We use the algorithm to find the minimum $f(x) = (x-5)^2 + x^3 + 1$.

```
def f(x):
    return((x-5)**2 + x**3+1)
def df(x):
    return(2*(x-5)+3*x**2)
def d2f(x):
    return(2+6*x)
```

The code to generate the figure on the left-frame of figure 8.1 using $\eta = 0.179$ and $x_0 = .2$ is then:

```
U=gradient_descent(f,df,2,nmax=100000,eta=.179)
XP,YP=zip(*U)
xplot=np.linspace(.9, 2.1,100)
yplot=f(xplot)
plt.plot(xplot,yplot)
plt.plot(XP,YP,marker="o")
```

[2]Fletcher, R "A New Approach to Variable Metric Algorithms," *Computer Journal*, **13** (3): 317-322 (1970); Goldfarb, D "A Family of Variable Metric Updates Derived by Variational Means," *Mathematics of Computation*, **24** (109): 23-26 (1970); Shanno, DF "Conditioning of quasi-Newton methods for function minimization," *Mathematics of Computation*, **24** (111): 647-656 (1970), and the Davidson-Fletcher-Powell method.[3]

[4]See Robbins, H and Monro, S. (1951) "A Stochastic Approximation Method" *Ann. Math. Statist.*, **22**(3):400-407.

[5]Metropolis, N. et. al. (1953) "Equation of State Calculations by Fast Computing Machines." *J. Chem. Phys.* **21**:1087-1092.

```
3220 iterations, delta= 9.990253069247501e-11
```

A quasi-Newton method requires the second derivative and can be implemented as follows.

```
def quasi_newton(f,df,d2f, x0,eta=0.2,nmax=25,conv=1E-10):
    xold=x0
    results=[]
    for i in range(nmax):
        F = f(xold)
        DF = df(xold)
        DF2 = d2f(xold)
        xnew = xold - eta * DF/abs(DF2)
        delta = abs(xnew-xold)
        if delta < conv:
            break
        results.append([xold, F])
        xold=xnew
    print(i,"iterations, delta=", delta)
    return(results)
```

Using the same control parameters, we can run this algorithm as follows.

```
U=quasi_newton(f,df,d2f, 2,nmax=1000,eta=.179)
XP2,YP2=zip(*U)
xplot=np.linspace(.9,2.1,100)
yplot=f(xplot)
plt.plot(xplot,yplot)
plt.plot(XP2,YP2,marker="o")
```

```
106 iterations, delta= 8.229839032480868e-11
```

The same level of convergence was achieved in approximately one-thirtieth as many iterations. There is no crisscrossing past the minimum in the quasi-Newton approach, as shown in the right-hand frame of figure 8.1.

Now suppose we want to find the energy minimum for an objective function such as equation 8.1. We can plot the corresponding energy landscape if we first implement the objective function

```
def Ehill(a,m):
    return sum((fhill(xi,a,m)-yi)**2 for (xi,yi) in zip(x,y))
```

Then we can use **`plt.contour`** to plot a parametric scan of parameter space:

```
EScape=[[Ehill(a,m) for a in np.linspace(0,5,50)]
  for m in np.linspace(0,5,50)]
cp=plt.contour(EScape, levels=np.logspace(-1,1,15),colors="k")
```

To perform a gradient descent on this objective function we need an equation for the gradient

$$\nabla\mathcal{E} = \left(\frac{\partial\mathcal{E}}{\partial a}, \frac{\partial\mathcal{E}}{\partial m}\right) \tag{8.12}$$

From equation 8.3,

$$\frac{\partial\mathcal{E}}{\partial a} = 2\sum_i \left[\frac{x_i^m}{a^m + x_i^m} - y_i\right]\frac{\partial y}{\partial a}(x_i; a, m) \tag{8.13}$$

$$\frac{\partial\mathcal{E}}{\partial m} = 2\sum_i \left[\frac{x_i^m}{a^m + x_i^m} - y_i\right]\frac{\partial y}{\partial m}(x_i; a, m) \tag{8.14}$$

where $y(x_i; a, m) = x_i^m/(x_i^m + a^m)$ is given by equation 8.2. Differentiating,

$$\frac{\partial}{\partial a}\left[\frac{x_i^m}{x_i^m + a^m}\right] = \frac{-m x_i^m a^{m-1}}{(a^m + x_i^m)^2} \tag{8.15}$$

$$\frac{\partial}{\partial m}\left[\frac{x_i^m}{x_i^m + a^m}\right] = \frac{(a^m + x_i^m)x_i^m \ln x_i - x_i^m(a^m \ln a + x_i^m \ln x_i)}{(a^m + x_i^m)^2} \tag{8.16}$$

If we implement a function for y and ∇y as follows,

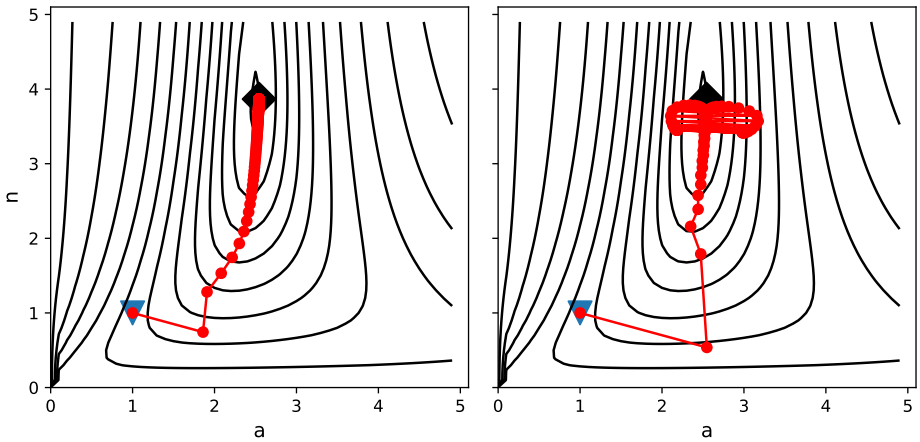
```
from math import log
def fhill(x,a=1,n=1):
    return x**n/(a**n+x**n)
def Gradfhill(x,a,n):
    AX = a**n + x**n
    DEN= AX**2
    da = -(n * x**n * a**(n-1) )/DEN
    dn = (AX * x**n * log(x) - \
      x**n *(a**n * log(a) + x**n * log(x)))/DEN
    return (da, dn)
```

then we can calculate the gradient of the objective function $\nabla\mathcal{E}$ with the following code.

```
def GradEhill(a,n):
    sa=0; sn=0
    for xi,yi in zip(x,y):
        da,dn = Gradfhill(xi, a,n)
        sa += 2*(fhill(xi,a,n)-yi)*da
        sn += 2*(fhill(xi,a,n)-yi)*dn
    return np.array((sa,sn))
```

Let **f** and **df** be Python functions that implement the objective function and its gradient, and let **x0** be a vector that gives a starting point for the gradient descent. Then a gradient descent algorithm in two-dimensions can be written as

Figure 8.2: Energy landscape generated by equation 8.1. On the left a learning rate of $\eta = 0.5$ is used; on the right, $\eta = 0.9$. The red curves show successive iterations from the initial guess (triangle) toward the energy minimum (the diamond).




```
def GD2D(f,df,x0,eta=0.001,nmax=25,conv=1E-10):
    xold=x0
    results=[]
    for i in range(nmax):
        F = f(*xold)
        DF = df(*xold)
        xnew = xold - eta * DF
        delta = np.linalg.norm(xnew-xold)
        if delta < conv:
            break
        results.append(xold.tolist())
        xold=xnew
    print(i,"iterations, delta=",delta)
    return(results)
```

This code is basically the same as our earlier code is except that the function call uses the splat operator (**`*xold`**) to ensure that the function applies to all arguments, rather than a vector; and the absolute value is replaced by a vector norm. A function call to gradient descent with the data set given above, using a starting guess of $m = 1$ and $a = 1$, and a learning rate of $\eta = .5$, results in convergence to one part in 10^{-10} in 553 iterations.

```
U=GD2D(Ehill,GradEhill,np.array((1,1)),nmax=5000,eta=.5)
```

```
553 iterations, delta= 9.675597431945335e-11
```

An attempt to use a faster learning rate ($\eta = 0.9$) results in the same problem

as it did in one dimension: jumping back and forth across the valley (figure 8.2). This is normally addressed either by quasi-Newton methods that utilize the Hessian matrix or make some sort of secant-type of approximation to the derivatives.

Curve Fitting in Python

The function **scipy.optimize.curve_fit** uses the Levenberg-Marquardt algorithm[6] This is a commonly used hybrid of Newton's method and gradient descent, and like most methods, has the danger that it can converge to the wrong local minimum. Hence it is important to feed the function a good starting point for the parameters. (We previously came across **curve_fit** in our study of stepwise regression, on page 54.)

We'll illustrate the method by generating a small number of data points near a hill function. We intentionally keep the data set small because similar problems in biology will also only have small data sets. Data set sizes of around ten points would be typical. First, we'll define a hill function to use as our target curve.

```
def fhill(x,a=1,m=1):
    return x**m/(a**m+x**m)
```

We will set a target value of $m = 4$ and $a = 2.5$, and randomly define *around* 25 points spread about the Hill function with a normal $N(0, .1)$ distribution. To ensure that all data points are positive, we'll drop any data pairs that come up with negative y values.[7]

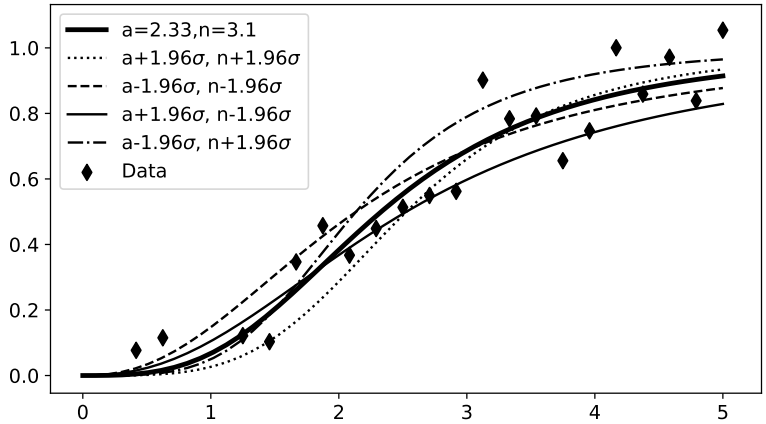
```
atarget=2.5; mtarget=4
n=25
xs=np.linspace(0,5,n)
ys=fhill(xs,atarget,mtarget)+np.random.normal(0,.1,n)
x,y=np.array([(x,y) for (x,y) in zip (xs,ys) if y>0]).T
n=len(x)
print("There are",n,"data points")
```

```
There are 21 data points
```

[6]Levenberg, K (1944) "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics.* **2**:164-168. Independently discovered by Marquardt, D. (1963) "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics.* **11**(2):431-441.

[7]This is because the Hill function usually represents either a concentration or flux gradient in a positive direction.

Figure 8.3: Optimized fit and 95% error predictions for Hill function example.



Now we can fit the curve with **curve_fit**.[8], which returns both a list of parameters and a covariance matrix. We can use the diagonal entries to estimate the one-standard-deviation errors of each of the parameters. We'll use values of 1 for our initial guess for all parameters (passed into **curve_fit** with **p0**:

```
parameters,covmatrix=curve_fit(fhill,x,y,p0=(1,1))
afit,mfit=parameters
asig,msig = np.sqrt(np.diag(covmatrix))
print("Predicted a = ",round(afit,2),"+/-",round(1.96*asig,2))
print("Predicted m = ",round(mfit,2),"+/-",round(1.96*msig,2))
```

```
Predicted a =  2.33 +/- 0.2
Predicted m =  3.1 +/- 0.79
```

Because of the nonlinearity of the Hill function, the ±95% estimates of the parameters do not make a neat envelope about the center, but form a criss-cross pattern as illustrated in figure 8.3.

[8]Full documentation on **curve_fit** is given at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html`

The Wrong Way to Fit Exponentials

Consider the population data in table 1.1. We can read this data into a **Pandas DataFrame** using **read_csv**.

```
import pandas as pd
import numpy as np
data=pd.read_csv("us-population.txt")
X= np.array(data["Year"])
Y=np.array(data["Population"])
n=len(X)
```

We can easily fit an exponential that minimizes the objective function

$$\mathcal{E} = \sum_i (e^{a+b(x_i-c)} - y_i)^2 \tag{8.17}$$

using **scipy.optimize.curve_fit** as we did above. Here **c=1790** is a constant that we will set before we begin the problem; it could be absorbed into the constant **a**, but we will not for numerical stability.

First, we define a function **f(x,a,b)** that returns the value of $e^{a+b(x-1790)}$:

```
def f(x,a=1,b=1):
    return np.exp(a+b*(x-1790))
```

Then we can do the fit, find the standard deviations, and mean squared error (which is equal to the objective function divided by the number of points fit).

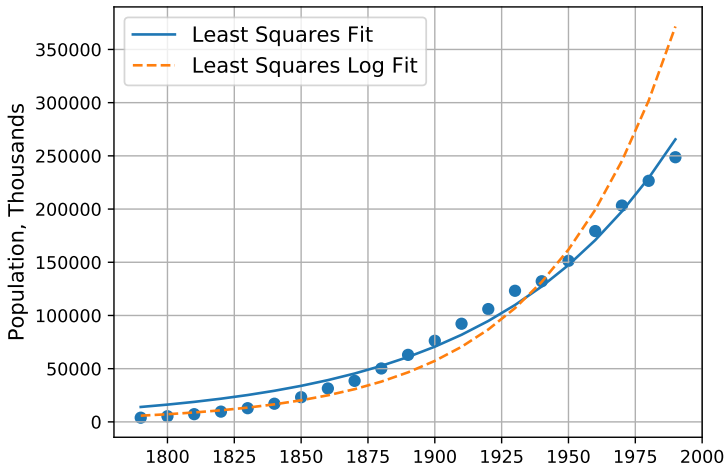
```
parameters,covmatrix=curve_fit(f, X,Y,p0=(1,1))
a,b=parameters

siga=covmatrix[0,0]**.5
sigb=covmatrix[1,1]**.5
print("a=",round(a,3),"+/-",round(1.96*siga,3))
print("b=",round(b,3),"+/-",round(1.96*sigb,3))

lsfit=f(X,*parameters)
print("MSE=",sum((Y-lsfit)**2)/n)
```

```
a= 9.548 +/- 0.201
b= 0.015 +/- 0.001
MSE= 90900652.15033811
```

Thus the optimal exponential fit in the *least squares sense* is

$$y = e^{9.548+0.0159(x-1790)} = e^{-18.913+0.0159x} \tag{8.18}$$

Figure 8.4: Comparision of fits to the US population data. The dashed line is a leasat squares fit the natural logarithm of the data; the solid line is a true least squares fit.



Now let us consider a second way of looking at the data. If our model is

$$y = e^{a+bx} \tag{8.19}$$

then it is also true that

$$u = \ln y = \ln \left\{ e^{a+bx} \right\} = a + bx \tag{8.20}$$

Thus if y is exponential in x, then u is linear in x. Suppose we do a least squares fit on $u = \ln y = a + bx$ as a function of x. Then we can transform back:

$$y = e^{u} = e^{a+bx} \tag{8.21}$$

So lets try doing that. We can use **polyfit** to get the coefficients.

```
M,B=np.polyfit(X,np.log(Y),1)
print("slope =    ",round(M,3))
print("intercept=",round(B,3))
logfit=np.exp(M*X+B)
print("MSE =      ",sum((Y-logfit)**2)/n)
```

```
slope =     0.021
intercept= -28.505
MSE =       1193297724.9124706
```

The linear regression of $\ln y$ on x is

$$\ln y = -28.505 + 0.021x \tag{8.22}$$

which suggests an exponential curve of

$$y = e^{-28.505+0.021x} \tag{8.23}$$

We can see that equations 8.18 and 8.23 are a bit different. Furthermore, the mean squared error for the second fit is about 13 times larger than the MSE for the first fit. This implies that the second fit is not a minimum of the objective function given in equation 8.17.

While either curve may actually "look" good to the unaided eye on a plot (e.g., figure 8.4), finding a true minimum of the objective function is always better. For this example, doing a linear fit of the logarithmically transformed function leads to large errors as x gets large (see accompanying Jupyter notebook).It was quite common to use the log-transform when calculations had to be performed by hand or slide rule, and easy to use optimization methods had not yet been implemented. Be wary of older physics and chemistry lab manuals, especially those written before 1990. There is no excuse for not at least attempting to try a good optimization first. If you can't get the parameters to converge, you can always try the log-transform as a first approximation.

Chapter 9

Regression Trees

Tree models can be used for both regression and classification. The algorithms for each are very similar. The acronym CART, for *Classification and Regression Trees*, is sometimes used.[1] When a tree is used to find a numerical result (e.g., a floating point value) it is called a **regression tree**. When one is used to place an item into a category such as "tastes like chicken" or "tastes like chocolate" it is called a **classification tree** or a **decision tree**. We will discuss regression trees in this chapter and return to classification trees in chapter 18.

In a **regression tree**, a problem is classified by repetitive subdivisions based on a decision process that minimizes the RSS error. Consider, for example, a situation in which there are n explanatory variables (features) $x_0, \ldots, x_{n-1}$ and one response variable y. Then a multi-linear regression would fit a model of the form

$$y = a + b_0 x_0 + b_1 x_1 + \cdots + b_{n-1} x_{n-1} \tag{9.1}$$

In the tree-based approach, we do not attempt to fit a continuous model. Instead, we will fit a step-wise continuous model where the function is constant in each of a number of sub domains of the of the original feature space. To do this, the domain is progressively divided into K distinct, non-overlapping regions. A single function value is assigned to each sub-domain, usually based on the mean value of the input data.

Let $\mathbf{x}$ be the vector of explanatory inputs. Then the model can be written as

$$y = c_0 \delta(R_0, \mathbf{x}) + c_1 \delta(R_1, \mathbf{x}) + \cdots c_{K-1} \delta(R_{K-1}, \mathbf{x}) = \sum_{k=0}^{K-1} c_k \delta(R_k, \mathbf{x}) \tag{9.2}$$

[1] The acronym is generally attributed to the title of the book, Breiman, L, et. al. (1984) *Classification and Regression Trees.* Brooks/Cole.

Here δ is a set membership indicator function:

$$\delta(R_k, \mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} \in R_k \\ 0, & \text{otherwise} \end{cases}$$

If there is only a single input then $\mathbf{x}$ collapses to the scalar x. Note that since $\mathbf{x}$ can only be a member of a single region, only a single term in the sum will be picked out by the sum in the final expression in equation 9.2. The first argument to each δ is an n-dimensional region. If there is only one feature in the explanatory variable, then the first argument is an interval. The second argument to δ may be either scalar x (for a single-dimensional feature space) or vector $\mathbf{x}$ (for the multidimensional feature space).

Here's how the regression tree algorithm works. Suppose there are N training vectors and K regions. Let $y_0, \ldots, y_{N-1}$ be the corresponding values of the response variable. The algorithm will continually seek to minimize the objective function

$$\mathcal{E}_{\text{Regression Tree}} = \sum_{y_i \in R_1} (y_i - \hat{y}_1)^2 + \cdots + \sum_{y_i \in R_K} (y_i - \hat{y}_K)^2 \tag{9.3}$$

$$= \sum_{j=0}^{N-1} \sum_{i=0}^{K-1} (y_j - \hat{y}_i)^2 \delta(R_i, x_j) \tag{9.4}$$

where $\hat{y}_i$ is the mean response in region i. The objective function calculates the residual sum of square (RSS) model error.

The problem is that we don't know where the K partitions are initially. Thus we need to locate them. We start by defining a single partition as the complete domain. Then we define **cut points** (figure 9.1) in each feature. If the explanatory variables are $x_0, x_1, \ldots, x_{n-1}$ then a cut point is a value z such that divides the dimension x_j into two regions, $x_j < z$ and $x_j > z$. We find cut points that minimize the RSS when the points are split over the two regions. Each variable is considered, so the split might be parallel to any axes. A region is split only along one explanatory variable at a time, so the regions remain rectangular. Once a cut point is found, the process is repeated again in each new sub-region.

At each division, we can predict the response (the response variable, y). This is the mean of the training observations in any given region. As we repeat this process then we get a hierarchy of values, hence a regression tree. The values of the response are plateaus over hyper-rectangular regions in the domain (figure 9.2. Iff the process is repeated too long there will be one leaf on the tree for each value in the training set, which will have been overfit! Thus the recursive subdivision is normally stopped at or before there are a minimum number of training points in each subdomain.

Figure 9.1: Example of effect of cut points on explanatory variable domain for four successive decisions. In the first split, the algorithm splits the domain along the line $x_0 = 150$. In the second split, the sub-domain $x_0 > 150$ is split along the line $x_1 = 180$. In each decision, a single sub-domain is split, producing a sequence of decisions on which domain is split next. At the end of the splitting process, a single value can be assigned to each domain. This is the model value for that y.

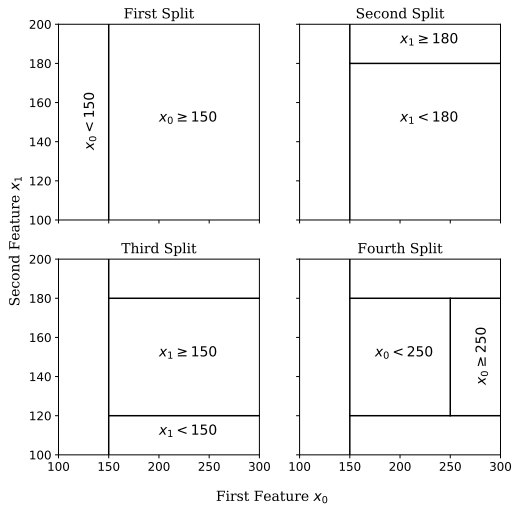
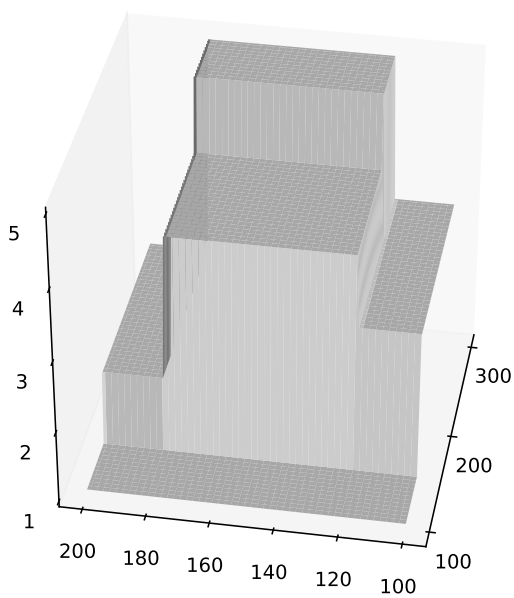


Figure 9.2: The response function is a multidimensional step function on the partitioned domain. The level of each plateau is the mean of the training values in that sub-domain.



Python Example: Single Feature

We will examine the 10.7 cm solar flux data for for July 2015 (see also page 58). This data is given in the accompanying data file **solar-data-july-2015.txt**.[2] This file contains a number of solar and geomagnetic indices, reported at one day intervals, with the most recent data first. The first 15 lines contain metadata describing the content of the file. We will read the data file with **read_csv**.

```
rawdata=pd.read_csv("solar-data-july-2015.txt",sep="\t",
   skiprows=15)
```

We'll select the **"Flux"** column (column index 1) as our response variable, and reverse its order to get it into the correct time order. Next we convert to **numpy arrays** and reshape for compatibility for **scikit** as we have done before.

```
Y=np.flip(np.array(rawdata["Flux"]))
n=len(Y)
X=np.array(list(range(n)))
x=X.reshape(-1,1)
y=Y.reshape(-1,1)
```

The decision tree is created with **DecisionTreeRegressor** in **sklearn.tree**.[3] . The standard format of **DecisionTreeRegressor** specifies the depth at which the partitioning is stopped.

```
DecisionTreeRegressor(max_depth=3)
```

where the number **3** should be replaced with the desired number of cuts. We will create three different decision trees, which terminate after 3, 4, and 5 cuts.

```
from sklearn.tree import DecisionTreeRegressor
regr2= DecisionTreeRegressor(max_depth=2)
regr3= DecisionTreeRegressor(max_depth=3)
regr4= DecisionTreeRegressor(max_depth=4)
```

The actual regression fits are performed with the **fit** method.

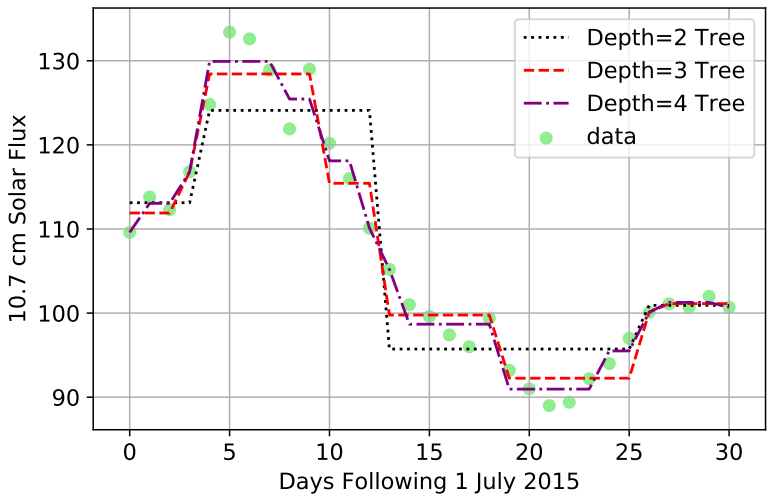
```
regr2.fit(x,y)
regr3.fit(x,y)
regr4.fit(x,y)
```

Predictions are made with the **predict** method of **DecisionTreeRegressor**:

[2] Original data source: `http://www.solen.info/solar/old_reports/2015/july/indices.html`, accessed 19 November 2018.

[3] Full documentation can be found at `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`.

Figure 9.3: Fits produced by regression tree on solar flux data for July 2015. Three different fits are shown, with the iteration terminated after either two, three, or four cuts are performed. As the number of cuts increases the fit approaches a linear step function with a different step at each data point.




```
ypred2=regr2.predict(x)
ypred3=regr3.predict(x)
ypred4=regr4.predict(x)
```

Each of these returns arrays of predicted y values of the same length as the input x values. We can compare the results, as illustrated in figure 9.3. The basic **`pyplot`** commands (pretty-printing commands are included in the accompanying notebook but are omitted here) are:

```
include{matplotlib.pyplot as plt}
plt.plot(x,ypred2,c="k",ls=":", label="Depth=2 Tree")
plt.plot(x,ypred3,c="k",label="Depth=3 Tree")
plt.plot(x,ypred4,c="k",ls="--",label="Depth=4 Tree")
plt.scatter(x,y,c="k")
```

We can numerically compare the fits by calculating a mean residual sum of squares for eac method.

$$\text{MSE} = \frac{1}{n}\sum(f(x_i) - y_i)^2 \tag{9.5}$$

```
for j in range(2,8):
    r=DecisionTreeRegressor(max_depth=j)
    r.fit(x,y)
    p=r.predict(x)
    MSE=sum((Y-p)**2)/n
    print("The MSE for a depth of ",j," is ", round(MSE,2))
```

```
The MSE for a depth of  2  is  25.71
The MSE for a depth of  3  is  8.32
The MSE for a depth of  4  is  3.73
The MSE for a depth of  5  is  0.81
The MSE for a depth of  6  is  0.11
The MSE for a depth of  7  is  0.0
```

This, of course, verifies what we expect, that the fit will approach a point-by-point interpolation as the depth of the tree increases.

Python Example: Multiple Features

We will illustrate regression trees for data sets with multiple features using the automobile miles per gallon data set from the UCI machine learning data repository that we studied in chapter 5 (see page 67). We'll load the data directly from the internet using **read_fwf**, indicating that missing data is specified in the file by a question mark (**"?"**) and that the file does not have headers.

```
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
   machine-learning-databases/auto-mpg/auto-mpg.data",
   header=None,na_values="?")
```

Since the file does not have any headers we will supply them ourselves. We don't have to supply headers; we could refer to the columns by number, but headers are easier. We will also simply remove any records that have missing data in one of the fields.[4]

```
data.columns=("mpg","cyl","displ","hp","weight","accel",
   "model","origin","carname")
data = data.dropna(axis=0)
```

Since humans have trouble visualizing higher dimensional data sets, we will consider a two-dimensional feature-subspace consisting of displacement and horsepower, for the purposes of illustrating the algorithm. We have to reshape the single-dimensional response data for compatibility with **scikit**.

[4]See footnote on page 66 for data file information.

```
X=np.array(data[["displ","hp"]])
Y=np.array(data["mpg"]).reshape(-1,1)
n=len(Y)
```

The regression tree, fit, and prediction are generated exactly as before:

```
r=DecisionTreeRegressor(max_depth=3)
r.fit(X,Y)
YP=r.predict(X)
```

The array **YP** will have predicted values for all of the original data points, but it will not tell you the shape of the decision space. The predictions will be clustered into a small number of distinct values:

```
values=list(set(YP))
nval=len(values)
print(nval,"distinct values among ", len(YP)," predictions")
```

```
8 distinct values among  392  predictions
```

To visualize the locations of the different plateaus in (HP, DISP) space, we can calculate the center of each cluster.

```
def cluster_center(value, X, Y, Z):
    x,y=np.array([[x,y] for (x,y,z)
       in zip (X,Y,Z) if z==value]).T
    x=np.mean(x)
    y=np.mean(y)
    return (x,y)
HP=X[:,1]; DISP=X[:,0]
cluster_centers = [cluster_center(u, DISP, HP, YP)
   for u in values]
print(" HP    DISP    MPG")
for (hp,disp),mpg in zip(cluster_centers, values):
    print("%6.2f %6.2f %5.2f" %(hp,disp,mpg))
```

```
 HP    DISP    MPG
 84.93  62.69 34.80
380.57 186.93 13.41
325.55 144.61 15.31
232.44  98.26 19.06
299.17 111.67 23.72
129.33  97.23 24.66
114.54  77.25 28.95
101.81  63.88 31.71
```

Of course these numbers give the MPG at the centers of the clusters of given *data* values. They do not give the decision boundaries. One way to plot the

decision boundaries is blanket the entire feature space with a tight grid and make a contour plot.[5] To generate the grid we use **np.meshgrid**:

```
xmin, xmax = DISP.min() - 1, DISP.max() + 1
ymin, ymax = HP.min() - 1, HP.max() + 1
xx, yy = np.meshgrid(np.arange(xmin, xmax, 1),
                     np.arange(ymin, ymax, 1))
```

We need to unwind the **xx** and **yy** grids to make them compatible with **predict**, and bind them together into a single two array. Then we need to reshape **Z** to match the feature dimensions.

```
ZP=r.predict(np.c_[xx.ravel(), yy.ravel()])
Z=ZP.reshape(xx.shape)
```

The contour lines themselves can be plotted with

```
cs = plt.contour(xx, yy, Z, colors="k")
```

Now we have enough information to calculate the plateau centers.

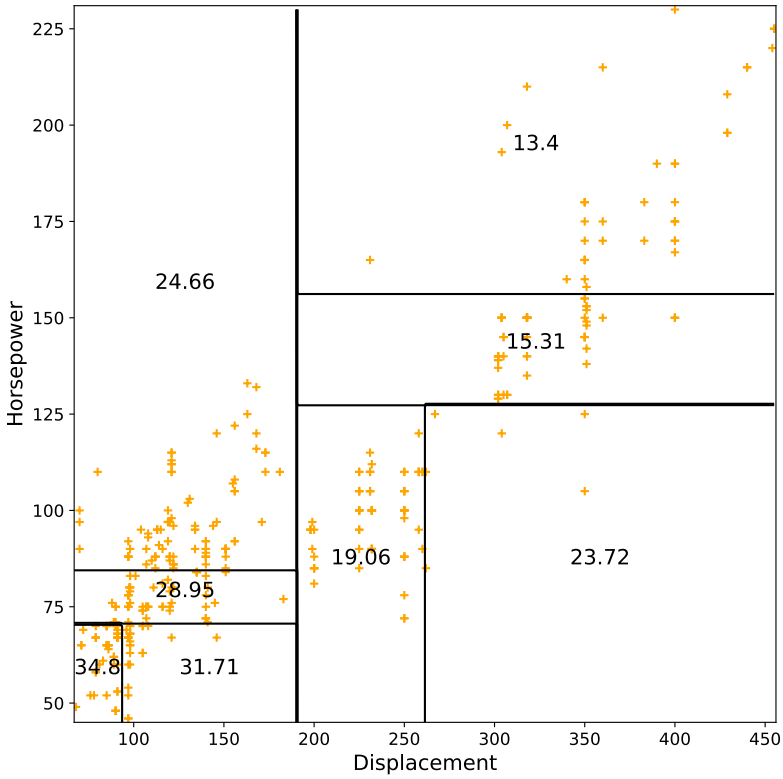
```
def find_plateau_centers(Z,xmin,xmax,ymin,ymax):
    U=np.unique(Z)
    centers=[]
    for x in U:
        locs=[]
        for ix in np.arange(xmin,xmax,1):
            for iy in np.arange(ymin,ymax,1):
                zval=Z[int(iy-ymin)][int(ix-xmin)]
                if zval==x:
                    locs.append((ix,iy))
        center=list(np.mean(locs,axis=0))
        centers.append((x,center))
    return(centers)
```

We'll add labels, and put the predicted MPG markers and data points on the plot as well.

```
plt.xlabel("Displacement",fontsize=15)
plt.ylabel("Horsepower",fontsize=15)
plt.scatter(DISP,HP,marker="+",c="lightgray")
centers = find_plateau_centers(Z,xmin,xmax,ymin,ymax)
for value, location in centers:
    plt.text(*location, str(round(value,2)), fontsize=15,
             horizontalalignment="center")
```

[5]This method is based on the example at `https://scikit-learn.org/stable/auto_examples/tree/plot_iris.html`.

Figure 9.4: Decision tree plateaus for the two-feature mpg data set.



The resulting contour plot is shown in figure 9.4.

Chapter 10

Logistic Regression

All of the topics we have discussed so far involve **regression**, one of the two major sub-disciplines of machine learning. In a regression task we study how one variable (the response variable, also called the dependent variable, or its mean) behaves as a function of another variable (the explanatory or independent variable, or its features). Now we turn to **logistic regression**, which is our first example of **classification**.

In a classification task (also called categorization) items are identified as being in one of two or more categories. Such tasks fall (broadly) into two types of problems: **supervised learning** and **unsupervised learning**. The goal of both of the types of tasks is to find a function $g(\mathbf{x})$ that returns the class number (or name) that each feature vector belongs to.

For example, suppose we are building a model to identify whether or not a vehicle is "full size sedan", "small sedan," a "large sedan," or a "sub-compact." Feature vectors would contain properties such as

$$\mathbf{x} = [\text{length}, \text{weight}, \text{cargo volume}, \ldots] \tag{10.1}$$

The goal would be to return a either a class name (such as "sub-compact") or a class number (such as 3) that distinguishes car $\mathbf{x}_i$ from car $\mathbf{x}_j$.

In **supervised learning**, training sets contain matched pairs of feature vectors and class labels. Most of the methods we will talk about, including logistic regression, are supervised. In **unsupervised learning** the training sets do not contain any class labels. Clustering (chapter 21) is an example of unsupervised learning.

Logistic regression[1] is an example of a supervised classifier. It is easiest to visualize and describe when there are only two categories, but the method can be extended to multi-class categories.

In the simplest case there are two categories C_1 and C_0. Let x be some explanatory variable that determines which category we are in. Our goal is to find the probability function $P(C_i|x)$ that will make this assignment. Let

$$y = P(C_1|x) \tag{10.2}$$

Then since there are only two categories, and the total probability must be one,

$$1 - y = P(C_0|x) \tag{10.3}$$

The **odds ratio** for C_1 is

$$\text{odds ratio} = \frac{P(C_1|x)}{1 - P(C_1|x)} = \frac{y}{1-y} \tag{10.4}$$

The **logit** function is defined as

$$\text{logit(u)} = \ln \frac{u}{1-u} \tag{10.5}$$

Thus the log of the odds ratio z

$$z = \ln \frac{y}{1-y} = \text{logit}(y)$$

In logistic regression, we do a linear regression of z on x. Thus our model is

$$\ln \frac{y}{1-y} = a + bx \tag{10.6}$$

for some unknown parameters a and b. Solving for y gives

$$y = \frac{1}{1 + e^{-(a+bx)}} \tag{10.7}$$

Once we determine the probability funtion y, we will want to know the value of y that we can use to distinguish between categories. If we conclude that $x \in C_1$ when $y > 1/2$ and $x \in C_0$ otherwise, then the two classes are separated when $x = x_0$, where x_0 is the solution of

$$\frac{1}{1 + e^{-(a+bx_0)}} = \frac{1}{2} \tag{10.8}$$

[1] Cox, D. (1958) *The regression analysis of binary sequences.* Journal of the Royal Statistical Society B. **20**: 215-242.

Cross multiplying gives

$$2 = 1 + e^{-(a+bx_0)} \tag{10.9}$$

Subtracting 1 from both sides and then taking natural logarithms,

$$0 = -a - bx_0 \tag{10.10}$$

Hence $x_0 = -a/b$ or the negative of the predicted intercept divided by the slope.

If there are multiple features in the explanatory variable, but still only a pair of categories in the response, the logistic regression probability becomes

$$y = \frac{1}{1 + e^{-(a+\mathbf{x}^{\mathrm{T}}\mathbf{b})}} \tag{10.11}$$

where $\mathbf{x}$ is the feature vector and $\mathbf{b}$ is a vector of coefficients. If if there are multiple categories as well as multiple features, the probability function becomes

$$y = \frac{e^{\mathbf{x}^{\mathrm{T}}\mathbf{b}_k}}{\sum_k e^{\mathbf{x}^{\mathrm{T}}\mathbf{b}_k}} \tag{10.12}$$

where the sum is over all categories (classes), $\mathbf{x}$ is a feature vector, and $\mathbf{b}_k$ is a vector of weights (coefficients) for class k. This is sometimes called the **softmax** function.

The function y gives the probability that x falls into category C_1. We normally turn this into a binary decision by making the following assignment:

$$\text{Assign } x \text{ to } \begin{cases} C_1, & \text{if } y \geqslant 1/2 \\ C_0, & \text{if } y < 1/2 \end{cases} \tag{10.13}$$

During the linear regression process we minimize the objective function

$$\mathcal{E} = \sum (\hat{y}(x_i) - y_i)^2 \tag{10.14}$$

where $\hat{y}(x_i)$ is given by equation 10.7.

Unfortunately the probability function 10.7 is non-linear in the parameters a and b, and it is not possible to analytically determine explicit solutions of the equations that minimize the objective function. Instead an iterative nonlinear optimizer is required to solve these equations implicitly. Logistic Regression is typically implemented using a maximum likelihood estimator. A maximum likelihood estimator (MLE) finds the maximum value of the likelihood or log-likelihood function (natural logarithm of the likelihood), or the most probable outcome. Under certain conditions a MLE will produce the same result as least squares minimization. Typical iteration methods include variations on gradient

descent algorithms, which attempt to follow the shortest path to an energy minimum (e.g., the minimum RSS error) by iterating on a formula similar to Newton's method.[2]

Logistic Regression in Python

We will begin by loading the auto mpg data file (see pages 66 and 102) with **read_fwf**. As before, we label the columns (because they are unlabeled) and need to tell **pandas** that missing fields are labeled with a **"?"**. Then we remove all rows that missing data.

```
data=pd.read_fwf("https://archive.ics.uci.edu/ml
     /machine-learning-databases/auto-mpg/auto-mpg.data",
     header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
     "model","origin","carname")
data = data.dropna(axis=0)
print(data[:5])
```

```
    mpg  cyl  displ     hp  weight  accel  model  origin                     carname
0  18.0    8  307.0  130.0  3504.0   12.0     70       1  "chevrolet chevelle malibu"
1  15.0    8  350.0  165.0  3693.0   11.5     70       1          "buick skylark 320"
2  18.0    8  318.0  150.0  3436.0   11.0     70       1         "plymouth satellite"
3  16.0    8  304.0  150.0  3433.0   12.0     70       1              "amc rebel sst"
4  17.0    8  302.0  140.0  3449.0   10.5     70       1                "ford torino"
```

The line **data = data.dropna(axis=0)** removes the records which contain missing data. In this data file, missing data was identified by a **'?'** and replaced by the Python symbol **nan** on input. Since there are only around half a dozen records that have missing data, it doesn't significantly affect the analysis if we just remove them and get on with it. In many real-world analyses, however, this is not so. As data sets get larger – into the millions of records and thousands of features – it becomes nearly impossible to avoid records with missing features. For example,a medical data set may be missing x-ray interpretations from a tenth of the subjects, and missing cholesterol measurements from another quarter of the subjects. Instead of throwing the records out we can sometimes get away with replacing the missing value with an **imputed** or inferred value. The function **SimpleImputer**[3] from **sklearn.impute**, for example, allows us to replace the missing value with the mean, median, most frequent value, or any other constant we wish.

[2]See chapter 3.3.2 of Y. Abu-Mostafa et. al., *Learning from Data* for a detailed discussion of the gradient descent algorithm for logistic regression.

[3]For full documentation see `https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html`

We are going to identify each car as either a "gas guzzler" or "not a gas guzzler" based on average fuel economy of American cars in 2016. In 2016, this was 24.7 miles per gallon.[4] We will indicate a "gas guzzler" with a flag value of 1.

$$y = \begin{cases} 1, & \text{if mpg} \geqslant 24.7 \\ 0, & \text{if mpg} < 24.7 \end{cases} \tag{10.15}$$

Our goal will be to see if we can identify a "gas guzzler" based on the weight of the car. We'll define a training set using equation 10.15.

```
mpg2016=24.7
weight=data["weight"]
mpg=data["mpg"]
```

We'll use the function **test_train_split** that we defined on page 41 to separate the data into test and training sets. We'll do a 50/50 split. As usual with single feature explanatory variables, we need to reshape the x-data.

```
import random
random.seed(99)
XTRAIN,YTRAIN, XTEST, YTEST=train_test_split(X,Y,f=.5)
XTRAIN=XTRAIN.reshape(-1,1)
XTEST=XTEST.reshape(-1,1)
```

Logistic regression is performed with the class **LogisticRegression**[5] from **sklearn.linear_model**. As usual, we need to load the class from the library, then use the **fit** method to actually perform the logistic regression.

```
from sklearn.linear_model import LogisticRegression as LR
r=LR().fit(XTRAIN, YTRAIN)
```

As with other linear models in **scikit**, the intercept and coefficients are accessed by the class fields **intercept_** and **coef_**.

```
b=r.coef_[0,0]
a=r.intercept_[0]
x0=-a/b
print("a (intercept)=",a)
print("b (slope)=",b)
print("x0 (50% point)=",x0)
```

[4] Reported at https://www.reuters.com/article/us-autos-emissions/u-s-vehicle-fuel-economy-rises-to-record-24-7-mpg-epa-idUSKBN1F02BX, downloaded 12 Dec 2018.

[5] The logistic regression class is fully documented at https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

```
a (intercept)= -4.847731365621908
b (slope)= 0.001869482929460507
x0 (50% point)= 2593.086724263837
```

The predicted probability function is approximately (to three non-zero significant figures):

$$y = \frac{1}{1 + e^{4.848 - .000187x}} \tag{10.16}$$

A car is classified as a gas guzzler if it weighs more than approximately 2593 pounds. This is summarized in figure 10.1. A quantitative discussion of how to interpret the results of a binary decision is deferred until chapter 11.

There are several different ways of evaluating binary classification methods such as logistic regression; these are used by so many methods that we will defer the general discussion to chapter 11. For now we will only consider one measure: the **accuracy**. The accuracy measures the proportion (or percentage, if you prefer) of correctly classified vectors. There is a simple way to do this in Python once we have a test vector **YTEST** and a prediction vector **YP** that is built from the model using as input **XTEST**. As usual with **sklearn** methods there is a **predict** function.

```
YP=r.predict(XTEST)
```

Each of **YTEST** and **YP** will be an **array** of 1's and 0's of the same length, and also the same length as **XTEST**. The expression **YTEST==YP** will generate an **array** of values that either **True** and **False**. All we have to do to count the number of cases where the prediction is the correct is to count the **True**'s. Since Python implements **True** internally as a 1 and **False** as a 0, this can be done by summation:

```
print("accuracy=",sum(YP==YTEST)/len(YTEST))
```

```
accuracy= 0.8214285714285714
```

Thus about 82% of the cars were correctly predicted to be in the correct category, either as gas guzzler or not a gas guzzler. This measure does not tell us which category is better predicted, nor does it tell us if either type of failure (incorrectly predicted to be a gas guzzler or incorrectly predicted to not be a gas guzzler) is more likely. This information comes from other measures.

Example in Python with Multiple Features

We'll repeat the example with the same data set used in the previous section, but this time will attempt to distinguish between gas-guzzlers and non-guzzlers using four features: displacement, horsepower, weight, and acceleration.

We've already created the training values, so we can use the same data for **y**. To get the x data, we not to extract the appropriate columns from the original data set after we have pre-processed it by removing rows with missing data as before. First we create the multi-feature x data.

```
X2=np.array(data[["displ","hp","weight","accel"]])
XTRAIN,YTRAIN, XTEST, YTEST=train_test_split(X2,Y,f=.5)
```

We do the fit on the multi-feature data.

```
r2=LR().fit(X2, Y)
```

We can pull out the coefficients as intercept as before.

```
b=r2.coef_
a=r2.intercept_
print("a (intercept)=",a)
print("b (slope)=",b)
YP=r2.predict(XTEST)
print("accuracy=",sum(YP==YTEST)/len(YTEST))
```

```
a (intercept)= [-0.20819552]
b (slope)= [[ 0.01457418 -0.01558174  0.00213798 -0.38627897]]
accuracy= 0.8520408163265306
```

The accuracy has improved slightly, from 82% to about 85%. This could change with a different randomization and train/test split. The classes are divided by the hyperplane

$$a + \mathbf{x}^{\mathrm{T}}\mathbf{b} = 0 \tag{10.17}$$

which is approximately given by

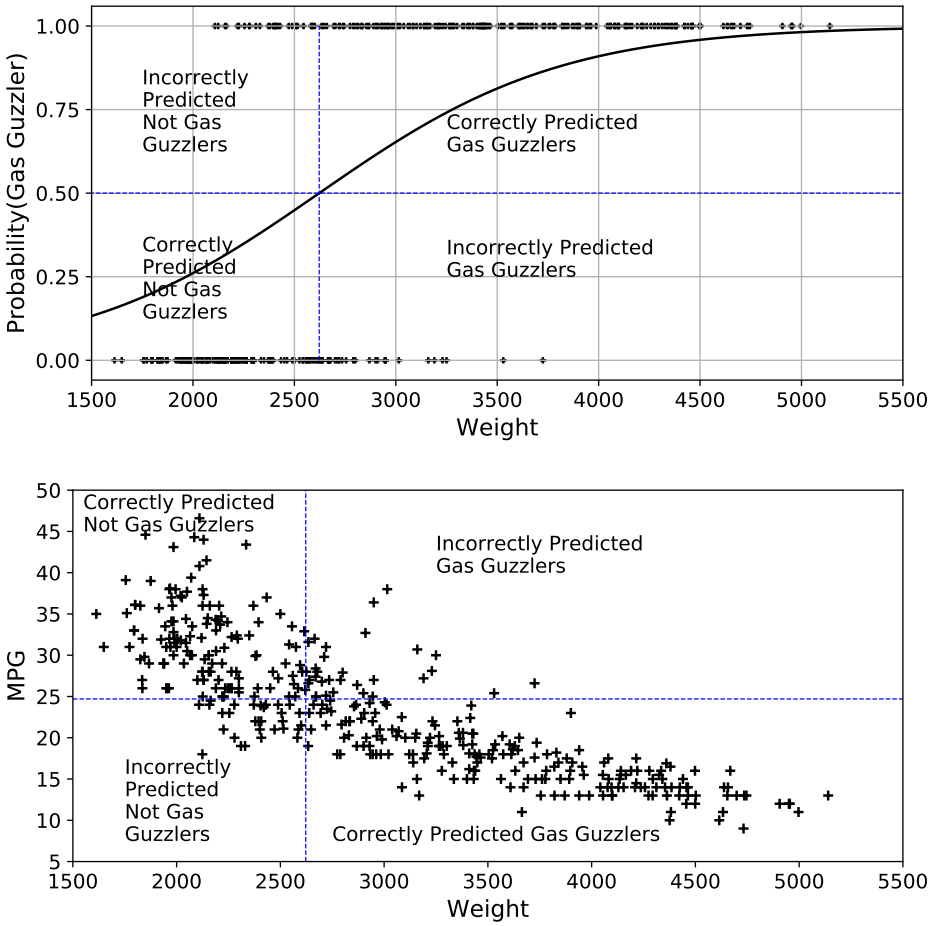
$$\begin{aligned} &-0.208 + 0.0146 \times \text{disp} - 0.0156 \times \text{hp} + .00214 \times \text{weight} \\ &\qquad - 0.0386 \times \text{accel} = 0 \end{aligned} \tag{10.18}$$

for the gas guzzler classification data. Similarly, the probability function is given by

$$y = \frac{1}{1 + e^{-(a+\mathbf{x}^{\mathrm{T}}\mathbf{b})}} \tag{10.19}$$

where the expression $(a + \mathbf{x}^{\mathrm{T}}\mathbf{b})$ in the exponential should be replaced by the expression on the left hand side of the equal sign in equation 10.18.

Figure 10.1: Results of logistic regression on gas guzzler classification. Top: logistic probability function determined, showing correctly and incorrectly data points as they fall into four possible categories. The actual data points are plotted as either a one (identified as a gas guzzler) or a zero (not a gas guzzler). Bottom: plot of miles per gallon as a function of weight, showing the MPG cutoff that was used to define a gas-guzzler and the 50% probability cutoff in terms of the corresponding weight classification. The four quadrants in the bottom plot show the same information as the four quadrants in the top plot.



Chapter 11

Evaluating Binary Classification

Numerical Measures of Success

One way to determine if a binary (two-way) decision (or classification) process is any good is to count the number of correct decisions and the number of incorrect classifications. Each input vector (which may be a scalar x) is associated with a decision, which we will indicate with a 0 (a **negative** decision) or a 1 (a **positive** decision). We develop our model using a training set consisting of **exemplars**: pairs of vectors and the correct decisions, either 0's and 1's. We will test our model with a similarly formatted data set (the test set), which also consists of vector/decision pairs, but these pairs were not used in the original model building process. Both the training and test data contain exemplar pairs, but only the training set is used to build the model. After the model is built with training exemplars, we feed the test exemplars to it and examine the decision process. A model like logistic regression can report two types of result for any given input x.

- A probability $p(x)$ that x is falls into the positive category, i.e., that $y = 1$
- A decision $y_p = 1$ or $y_p = 0$, based on a threshold, that the predicted value of the category is either $y = 1$ (a positive result) or $y = 0$ (a negative result).

We then may measure the correctness of the decision process by comparing the exemplar y to the the predicted value y_p. We then conclude that result is a

- A **true positive** (TP) if $y = 1$ and $y_p = 1$
- A **false positive** (FP) if $y = 0$ and $y_p = 1$
- A **true negative** (TN) if $y = 0$ and $y_p = 0$
- A **false negative** (FN) if $y = 1$ and $y_p = 0$

If we count the number of true positives, false positives, true negatives, and false negatives in our prediction we can display them as a **confusion matrix** (see also figure 11.1)

$$\text{confusion matrix} = \begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix} \tag{11.1}$$

Let N and P be the total number of negative and positive exemplars,

Figure 11.1: The confusion matrix counts the numbers of true positives, false positives, true negatives, and false negatives in a decision process.

		Predicted	
		0	1
Actual	0	True Negative	False Positive
	1	False Negative	True Positive

$$N = TN + FP = \text{total number of negative exemplars (0's)} \tag{11.2}$$

$$P = TP + FN = \text{total number of postive examplars (1's)} \tag{11.3}$$

Then the **true positive rate** TPR (the fraction of positive exemplars that are actually predicted to be positive) is given by

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{11.4}$$

The true positive rate is also called the **recall** or **sensitivity** by some authors. The **false positive rate** FPR is the proportion of negatives that that incorrectly identified as positives:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{11.5}$$

The FPR is also called the **fallout** or **false alarm rate**. The **specificity** gives the fraction of negatives exemplars that are correctly classified,

$$\text{specificity} = \frac{TN}{N} = \frac{TN}{FP + TN} = 1 - FPR \tag{11.6}$$

The fraction of all predicted positive values (1's) that have positive exemplars (true positives) is called the **precision** and may be considered a measure of the positive predictive power:

$$\text{precision} = \frac{TP}{TP + FP} \tag{11.7}$$

The **accuracy** of model gives the proportion of all predictions that match the exemplars (see, for example, page 112).

$$\text{accuracy} = \frac{TP + TN}{P + N} \tag{11.8}$$

The **error rate** is closely related to the accuracy:

$$\text{error rate} = 1 - \text{accuracy} = \frac{FP + FN}{P + N} \tag{11.9}$$

An ideal model will have a high positive prediction rate (precision near 1); a high true positive rate (recall near 1); and high accuracy (near 1). Often an increase in one measure will lead to a decrease in another, so it is common to look for a more balanced, geometric measure. One such measure is the **F-measure**:

$$F = \frac{2}{\dfrac{1}{\text{precision}} + \dfrac{1}{\text{recall}}} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{11.10}$$

If the recall is high and the precision is low then the F-measure is close to $2 \times$ precision.

Evaluation Metrics in Python

All of the evaluation metrics we have discussed above are easy to calculate manually in Python. There are also functions pre-implemented in **`sklearn`** that implement them. We will demonstrate with the logistic regression model of gas guzzlers from chapter 10 (pages 110ff).

Suppose **`YTEST`** is the test array of exemplars (1's and 0's from page 111) and **`YP`** is the predicted classification (also an array of 1's and 0's, from page 112).

We can determine the confusion matrix by counting true positives, false positives, true negatives, and false negatives.

```
def ConfMat(OBS,PRED):
    OPS=list(zip(OBS,PRED))
    TP = sum([(O==1) and (P==1) for O,P in OPS])
    TN = sum([(O==0) and (P==0) for O,P in OPS])
    FP = sum([(P==1) and (O==0) for O,P in OPS])
    FN = sum([(P==0) and (O==1) for O,P in OPS])
    M = np.array([[TN,FP],[FN,TP]])
    return M
ConfMat(YTEST,YP)
```

```
array([[66, 16],
       [17, 97]])
```

We can also use the **confusion_matrix** function in **sklearn**.[1]

```
from sklearn.metrics import confusion_matrix
confusion_matrix(YTEST,YP)
```

```
array([[66, 16],
       [17, 97]])
```

We can compute the other measures described above easily enough as well.

```
def Metrics(OBS,PRED):
    OPS=list(zip(OBS,PRED))
    TP = sum([(O==1) and (P==1) for O,P in OPS])
    TN = sum([(O==0) and (P==0) for O,P in OPS])
    FP = sum([(P==1) and (O==0) for O,P in OPS])
    FN = sum([(P==0) and (O==1) for O,P in OPS])
    P = sum(OBS)
    N = len(OBS)-P
    TPR = TP/P
    TNR = TN/N
    Specificity = TN/(FP+TN)
    Accuracy = (TP+TN)/(P+N)
    Precision = TP/(TP+FP)
    F=2/(1/Precision+1/TPR)
    return(TPR, TNR, Specificity, Accuracy, Precision,F)
TPR,TNR,Spec,Acc,Prec,F=Metrics(YTEST,YP)
print("TPR =          ",round(TPR,3))
print("TNR =          ",round(TNR,3))
print("Specificity = ",round(Spec,3))
print("Accuracy =     ",round(Acc,3))
print("Precision =    ",round(Prec,3))
print("F =            ",round(F,3))
```

[1]The **confusion_matrix** function is fully documented at `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html`.

```
TPR =          0.851
TNR =          0.805
Specificity =  0.805
Accuracy =     0.832
Precision =    0.858
F =            0.855
```

We can also get the TPR with the function **recall_score**[2] ; and can use the functions **accuracy_score**[3], **precision_score**[4], **f1_score**[5]. These are all also in **sklearn.metrics**.

```
from sklearn.metrics import recall_score
recall_score(YTEST,YP)
```

```
0.8508771929824561
```

```
from sklearn.metrics import accuracy_score
accuracy_score(YTEST,YP)
```

```
0.8316326530612245
```

```
from sklearn.metrics import precision_score
precision_score(YTEST,YP)
```

```
0.8584070796460177
```

```
from sklearn.metrics import f1_score
f1_score(YTEST,YP)
```

```
0.8546255506607929
```

ROC Curves

The ROC curve[6] measures the ability of a binary classifier to distinguish between two categories as a threshold is varied.[7] For logistic regression, we typically use a threshold of 0.5 in the probability function, but we could consider varying this threshold to any level in the range (0,1). The ROC curve is produced as follows.

[2]See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html for full documentation.

[3]See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html for full documentation.

[4]See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html for full documentation.

[5]See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html for full documentation.

[6]ROC is an acronym for Receiver Operating Characteristic. The name comes from the analysis of radar signals, where its uses is similar to its use hear – discrimination ability as a threshold is varied – but the name itself has little meaning to us today so we just use the term ROC.

[7]See, for example, Fawcett,T (2006) An introduction to ROC analysis. *Pattern Recognition Letters.* **27**:861-874; Swets,JA (2000) Better Decisions Through Science. *Scientific American.* **283**(4):82-87.

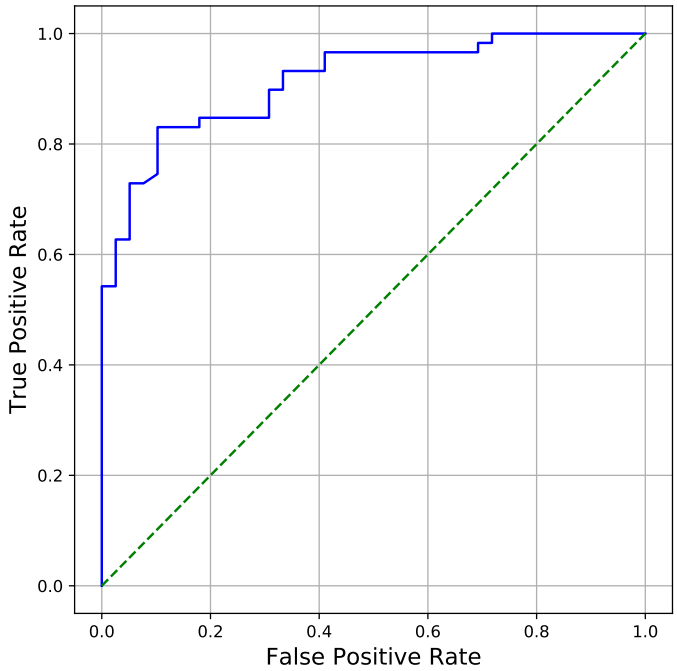
1. For each feature vector/exemplar pair $(\mathbf{x}, y(\mathbf{x}))$ in the test data, find the probability $p_j = \Pr(\mathbf{x}_j \in C_1) = \Pr(y_j = 1|\mathbf{x}_j)$.
2. Sort the pairs (y_j, p_j) produced in step 1 in decreasing order of p_j.
3. Let P be the total count of $y = 1$'s in the test data.
4. Let N be the total count of $y = 0$'s in the test data.
5. Set $TP = FP = 0$
6. For each (y_j, p_j) in in the list of k pairs:
   (a) If $y_j = 1$ then at the threshold of p_j the prediction will be a true positive (TP); increment TP.
   (b) Otherwise, if $y_j = 0$, a prediction of 1 at a threshold of p_j will be a false positive (FP); increment FP.
   (c) Plot the point $(FP/N, TP/P)$. This is a point on the ROC curve for a threshold corresponding to the probability p_j.
7. The **roc curve** is the curve plotted by joining all points from the origin $(0, 0)$, through all the $(FP_i/N, TP_i/P)$, to $(1, 1)$.

The function **ROC** given here will generate the points on the ROC curve.

```
def ROC(Y,Prob):
    if (len(Y)!=len(Prob)):
        print("Length mismatch")
        return([])
    P=sum(Y); N=len(Y)-P
    if (P<1) or (N<1):
        print("There must be both positive and
          negative examples")
        return([])
    py_pairs = sorted(zip(Prob, Y), reverse=True)
    FP=0; TP=0
    ROC_CURVE=[]
    pprev=float("-inf")
    for p,y in py_pairs:
        if p != pprev:
            ROC_CURVE.append([FP/N, TP/P])
            pprev=p
        if y>0:
            TP+=1
        else:
            FP+=1
    ROC_CURVE.append([FP/N, TP/P])
    return(ROC_CURVE)
```

The input to **ROC** consists of two arrays: the array of exemplars, and the corresponding array of probabilities. The return values are a pair of lists consisting of the false positive rate (which should be plotted on the x axis) and the true positive rate (which should be plotted on the y axis). Let **r** be the model developed using logistic regression on page 111ff, and let **XTEST** and **YTEST** be the corresponding test exemplar data. Then we can generate probabilities for

Figure 11.2: ROC curve for the logistic regression model for classification of gas guzzlers from chapter 10. This figure was generated using the code shown in this chapter. ROC curves should be bowed up and to left. The further a curve bows into this area, it will tend to have fewer false positives and more true positives, indicating a better model.



logistic regression using the method **predict_prob**.

```
probs=r.predict_proba(XTEST)[:,1]
FPR,TPR=zip(*ROC(YTEST,probs))
```

We then plot this ROC curve (as shown in figure 11.2.

```
plt.plot(FPR,TPR,c="k")
plt.grid()
plt.plot([0,1],[0,1],c="k",ls="--")
plt.xlabel("False Positive Rate",fontsize=14)
plt.ylabel("True Positive Rate",fontsize=14)
```

An identical ROC curve can be produced using **roc_curve** from **sklearn.metrics**.[8] The necessary code for producing the ROC using **roc_curve** follows.

```
probs=r.predict_proba(XTEST)[:,1]
fpr, tpr, threshold = roc_curve(YTEST,probs)
plt.plot(fpr,tpr, c="k")
plt.plot([0,1],[0,1],c="k",ls="--")
```

[8]Full documentation can be found at `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html`.

We can also add labels and grid lines.

```
plt.xlabel("False Positive Rate", fontsize=14)
plt.ylabel("True Positive Rate", fontsize=14)
plt.grid()
```

The ROC curve plots the TPR as a function of the FPR as a function of the threshold. Annotating the ROC curve with the thresholds helps us to understand how this curve is useful. For the gas guzzler case, the actual thresholds in the data occur at somewhat irregular locations:

```
print(threshold)
```

```
[0.9878745  0.87493554 0.84872465 0.80448668 0.77099723
 0.67454239 0.6607842  0.65873713 0.60012308 0.53894802
 0.50710895 0.4888726  0.480673   0.44892449 0.43946876
 0.41891812 0.39471658 0.37742706 0.37315109 0.32753671
 0.31363434 0.30236211 0.30197753 0.30159322 0.29624224
 0.29548232 0.18053469]
```

Since (as humans) it is easier to understand round numbers like 0.3, 0.4, 0.5, etc, none of which are exactly in the data set, we will need to interpolate. We can use the **`interp1d`** function from **`scipy.interpolate`** to interpolate for the values of (FPR, TPR) at the values of the threshold we are interested in and then mark them on the ROC curve. This works be creating a function **`f(t)`**, where **`t`** is a threshold, that returns a point **`[fpr,tpr]`**.

```
from scipy import interpolate
x=threshold
y=np.vstack([fpr,tpr])
f=interpolate.interp1d(x,y)
```

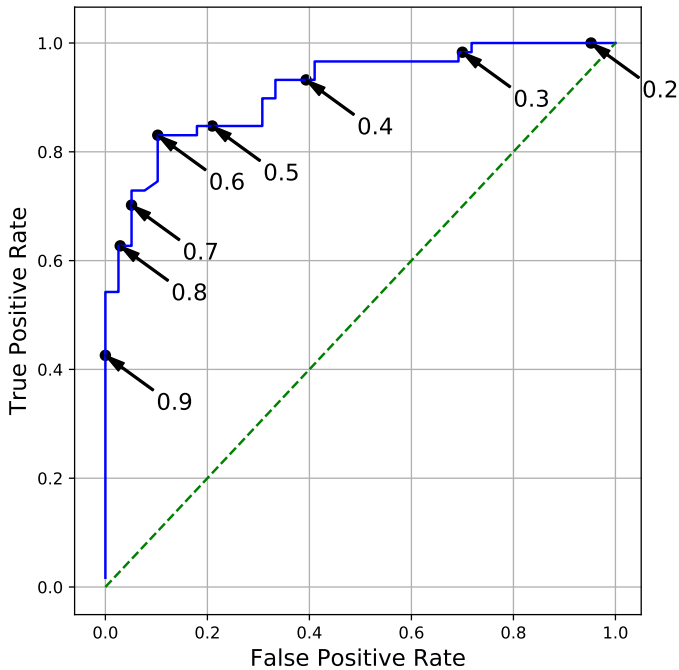
We can annotate at intervals from, say, .2 to .9 in units of 0.1.

```
tvals=np.arange(.2,1,.1)
xyvals=f(tvals)
xvals,yvals=xyvals
```

Then we should add the following code to the same Jupyter cell as the code for the ROC plot.

```
plt.scatter(xvals,yvals, c="k")
for t,x,y in zip(tvals,xvals,yvals):
    plt.annotate(str(round(t,1)),(x,y),
        xytext=(x+.1,y-.1), fontsize=14,
        arrowprops={"width":1.0, "facecolor":"black",
        "headwidth":6})
```

Figure 11.3: Same ROC curve plotted in figure 11.2 annotated by threshold.



As we can see from figure 11.3, high threshold points start at the bottom left of the figure with both low FPR and low TPR. In general, as we reduce the threshold, we increase the TPR without significantly decreasing the FPR until the curve bows over to the right. As the threshold continues to decrease, the curve bows further to the right, with the TPR increase less quickly and the FPR increasing more quickly. Eventually the TPR more or less levels off and further lowering the threshold only serves to increase the FPR. Ideally we would like a high TPR and a low FPR. This means we want to be in the upper left hand corner of the picture. The way to do this is to find a model (and a threshold) that has its bow as close to the the upper left corner as possible. For example, with this particular train/test split, increasing the threshold to 0.6 from 0.5 would decrease the FPR without significantly change the TPR. Decreasing the threshold to 0.4, on the other hand, would improve the TPR at a cost to the FPR. A look at figure 12.6 shows that some models with some data sets can push the classifier almost to the limit, while other models of the same data are barely different from random guessing (the 45 degree dashed line).

The area under an ROC curve (AUC, for Area Under Curve) is another commonly used metric. The AUC gives the probability that a method will decide that a particular vector with $y = 1$ with be assigned to a class with $y_p = 1$ is

higher than the probability that a particular vector with $y = 0$ will be classified with $y_p = 1$.[9] This is sometimes called a Wilcoxin test of ranks.[10] We can calculate this area under the curve using a trapezoidal rule integration formula. If we subdivide the interval $[a, b]$ into sub-intervals at points x_k, and let Δx_k be the interval width, then:

$$\int_a^b f(x)dx \approx \sum \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k \tag{11.11}$$

```
def TRAP(xvals, yvals):
    XLEFT = xvals[:-1]; XRIGHT= xvals[1:]
    YLEFT = yvals[:-1]; YRIGHT= yvals[1:]
    DX = XRIGHT-XLEFT
    return(sum(0.5*(YLEFT+YRIGHT)*DX))
```

Given the arrays **FPR** and **TPR** that we found on page 121, the AUC is given by

```
TRAP(np.array(FPR), np.array(TPR))
```

```
0.9309477963200686
```

We can also find the AUC using **auc**[11] or **roc_auc_score**[12] in **sklearn.metrics**.

```
from sklearn.metrics import auc,auc_score
print(roc_auc_score(YTEST,probs))
print(auc(FPR,TPR))
```

```
0.9309477963200685
0.9309477963200685
```

The AUC is thus another measure of the accuracy or goodness of prediction of the model. A model that has perfect prediction will have a square ROC curve with a bow in the upper left hand corner. The area under such a curve will be one. A model that does no better than guessing will have an ROC curve that falls on the 45 degree line and have an area under the curve one half.

[9] Bradley, AP (1997) The Use of the Area Under the Curve in the Evaluation of Machine Learning Algorithms. *Pattern Recognition.* **30**(7):1145-1159.

[10] Hanley,JA and McNeil,BJ (1982) The meaning and use of The Area under a Receiver Operating Characteristic (ROC) Curve. *Radiology* **143**:29-36.

[11] For details see `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html`.

[12] For details see `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html`.

Chapter 12

Artificial Neural Networks

Artificial neural networks (ANN's) are used for both regression and classification. Many different types have been implemented, ranging from the simplest, single neuron (perceptron) model, to large mult-layer network (deep learning networks). Other types include recurrent, associative, convolutional, and genetic network. In this chapter we will the use of feed-forward networks in both classification and regression.

The Perceptron

Artificial neural networks (ANN's) are built by combining single **perceptrons**[1] (sometimes called McCulloch-Pitts neurons[2]). At its simplest, the perceptron linearly combines inputs $x_1, x_2, \ldots, x_n$ with weighting factors $w_1, w_2, \ldots, w_n$ (figure 12.1). The output is written in terms of the input $\mathbf{x}$ and weights $\mathbf{w}$ as

$$y = \sigma\big(w_0 + \mathbf{w} \cdot \mathbf{x}\big) = \sigma\bigg(\sum_{k=0}^{n} w_i x_i\bigg) \tag{12.1}$$

where $\sigma(u)$ is a **threshold function** (figure 12.2). In the last step we have extended the feature vector $\mathbf{x}$ by one dimension. This incorporates an extra input with strength $x_0 = 1$. The threshold function is inspired by biology: neurons only fire if the sum of the input weights exceed a sufficient threshold. The threshold function $\sigma(u)$ is required to be non-decreasing and saturable, in

[1] Rosenblatt, F (1957) *The Perceptron–a perceiving and recognizing automaton.* Report 85-460-1, Cornell Aeronautical Laboratory

[2] McCulloch, W., Pitts, W. (1943) *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, **5**:115-133.

the sense that $\sigma(u) \to 1$ as $u \to \infty$ and $\sigma(u) \to 0$ as $u \to -\infty$. Typical functions are the unit step function

$$\sigma(u) = \begin{cases} 1, & u > 0 \\ 0, & u \leqslant 0 \end{cases} \tag{12.2}$$

and the logistic sigmoid

$$\sigma(u) = \frac{1}{1 + e^{-x}} \tag{12.3}$$

Any such definition allows the threshold function to be rebranded as a probability distribution: it will fire only if $\sigma(\sum(\text{input})) > 0$. With a suitable choice of weights the perceptron can be used to find a hyperplane that linearly separates the data in two sets. This plane can either represent a best fit (linear regression) or separation (for classification).

Figure 12.1: A single neuron perceptron with a threshold function.

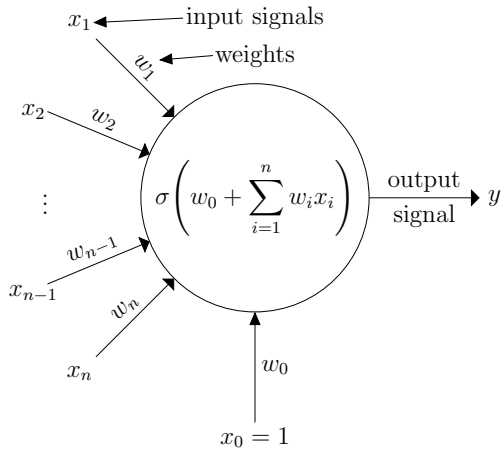
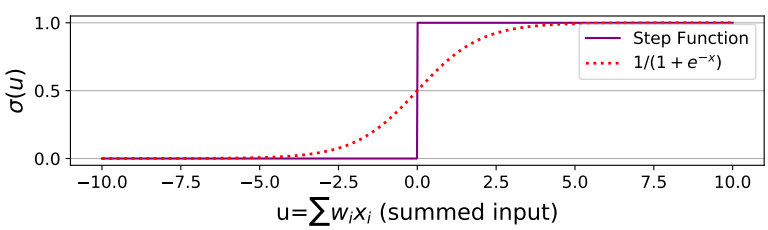


Figure 12.2: Typical neural network threshold functions must saturate at $\pm\infty$ at 1 and 0, should give a probability closer to 1 when $x > 0$, and a probability closer to 0 when $x < 0$.



To understand why this will lead to a linear separation in the data, consider the nature of the argument of σ:

$$w_0 + \mathbf{w} \cdot \mathbf{x} \begin{cases} > 0, & \text{Neuron will fire} \\ \leqslant 0, & \text{Neuron will not fire} \end{cases} \tag{12.4}$$

At the boundary, $w_0 + \mathbf{w} \cdot \mathbf{x} = 0$, the equation of a hyperplane.

Suppose, for example, that we have a single perceptron whose input is a two dimensional feature vector denoted by (x, y), and whose output is denoted by z. By equation 12.1, the output is

$$z = \sigma(w_0 + w_1 x + w_2 y) \tag{12.5}$$

The perceptron will fire only when $w_0 + w_1 x + w_2 y > 0$ Inputs (x, y) on one side of the line

$$y = -\frac{w_0}{w_2} - \frac{w_1}{w_2} x \tag{12.6}$$

will cause the perceptron to fire, while those on the other side will not.

We use the following **learning rule** to update the perceptron: weights are updated in proportion to (a) the strength of the input and (b) the magnitude of the error. This is applied as follows:

1. Randomly initialize all weights
2. Let $\eta < 1$ be a small positive **learning rate**.
3. For each data point:
   (a) Read in the next data point $(x_1, x_2, \ldots, x_n)$ and observed y value
   (b) Compute the predicted $y' = \sigma(\mathbf{w} \cdot \mathbf{x})$
   (c) Compute the prediction error $\epsilon = y' - y$
   (d) Update the weights by $\Delta \mathbf{w} = \epsilon \eta \mathbf{x}$
4. If the weights have not converged, repeat at step 3

We can demonstrate the linear separation by using the following data set. Three points are in category 1 and the other point is in category zero.

```
data= [((0, 0), 1), ((0, 1), 0), ((1, 0), 1), ((1, 1), 1)]
```

We define a unit step function:

```
def sigma(x): # step function
    return(0 if x < 0 else 1)
```

A perceptron algorithm that will run **steps** iterations

```
def perceptron(data, steps,eta=.5):
    w=np.random.random(3)
    xvecs = [np.array([1,*x]) for x,y in data]
    yvals = [y for x,y in data]
    for j in range(steps):
        wold=w
        for xvec, y in zip(xvecs,yvals):
            output = sigma(xvec.dot(w))
            err=y-output
            w+=err*eta*xvec
    return(w)
```

Figure 12.3: Lines of separation for 25 runs of the perceptron on a two-feature data set. The data point with $(x, y) = (0, 1)$ (red dot) is in category zero; the other three points (blue dots) are in category one. All of the lines correctly separate the two categories.

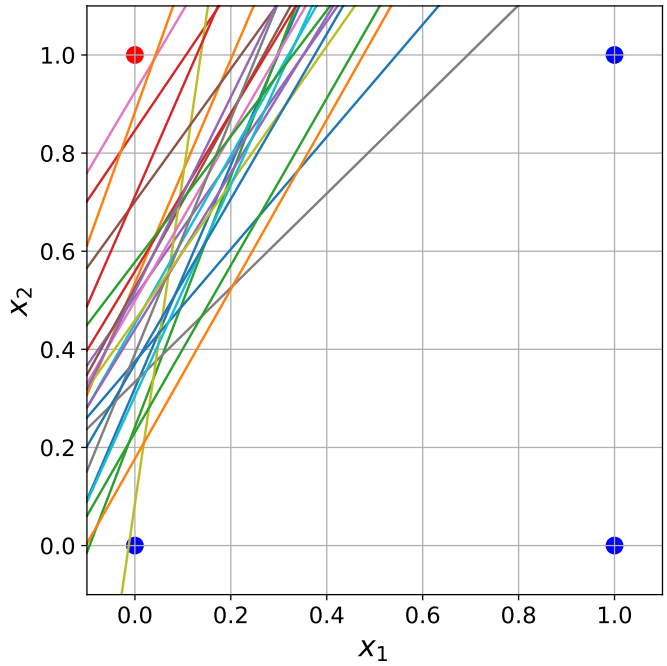
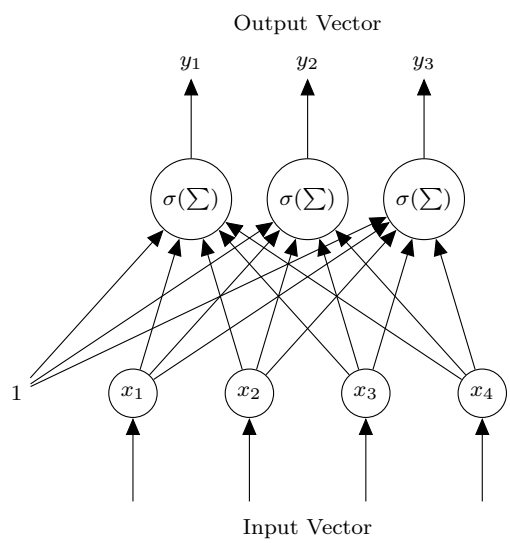


Figure 12.4: A single layer perceptron with $p = 4$ inputs and $n = 3$ outputs. There is only a single layer. The node labels $\sigma(\sum)$ are shorthand for $y_i = \sigma(\sum_{k=1}^{p} w_{ik} x_k)$. Each arrow has a different weight w_{ik} associated with it, where the first index refers to the output node and the second index refers to the input node.



This algorithm will typically converge in about half a dozen iterations (the reader should verify this). If we run this algorithm 25 times and plot the line of separation, we will get a different line of separation each time (figure 12.3). Each line linearly separates the data, but at a different location.

Individual neurons can be combined together into a network called a single layer perceptron (figure 12.4). Suppose that there are n output neurons $\mathbf{y} = (y_1, \ldots, y_n)$ and p input neurons $\mathbf{x} = (1, x_1, \ldots, x_p)$. (In figure 12.4 we have $n = 3$ and $p = 4$.) We have no idea what weights to use, so we use some very small random numbers to initialize all the weights w_{ij}. Using the given input and initial weight values, we will calculate an observed output $(y_1, \ldots, y_n) = \mathbf{y} = \sigma(\mathbf{w} \cdot \mathbf{x})$. During training we will have some training values $\mathbf{y}' = (y'_1, \ldots, y'_n)$, which will be different from the actual values $\mathbf{y} = (y_1, \ldots, y_n)$. We will use this error to update the the vector of weights by (replacing step 3) above

$$\Delta \mathbf{w} = \eta(\mathbf{y}' - \mathbf{y}) \cdot \mathbf{x} \tag{12.7}$$

where η is a very small number (the learning rate). The process is then repeated with the new weights and the iteration is repeated until the algorithm converges.

Multilayer Perceptrons and Back-Propagation

In a **multilayer perceptron** the neurons are arranged in layers as a feed-forward network (figure 12.5). If we only had linear combinations, adding more neurons would not do anything for us, since the linear combination of a linear combination is still linear. But by adding nonlinearity it turns out that we can fit (and separate) practically any type of reasonably shaped function. Each neuron processes its input by calculating the weighted sum of the extended vector (with weight w_0) and then applies the threshold function σ.

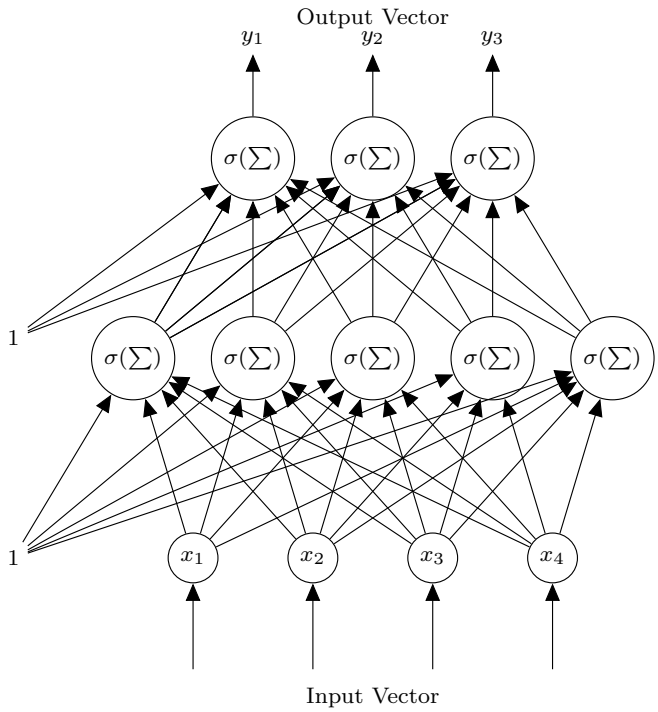
Weights in the multi-layer perceptron are updated using the **back-propagation** rule.[3] This rule minimizes the objective function

$$\mathcal{E} = \frac{1}{2} \sum (y'_i - y_i)^2 \tag{12.8}$$

where the sum is taken over the training data. Here y'_i is a value in the training data and y_i is a value in the output vector. We note that this is the same objective function used for regression problems. The leading factor of $1/2$ is not necessary, but is added for convenience, to cancel out the factor of 2 that arises

[3] Rumelhart, D.E. (1986) "Hinton, G. E. et al "Learning representations by back-propagating errors". *Nature.* **323** (6088): 533-536.

Figure 12.5: A multilayer perceptron may have zero or more hidden layers. This figure illustrates a multilayer perceptron with four inputs, 3 outputs and a five cell hidden layer.



during differentiation. If u_i is the output of any neuron in any given layer. The if that neuron has p inputs $u_1, \ldots, u_p$ from the underlying layer,

$$u_j = \sigma\left(\sum_{k=1}^{p} w_{ki} u_k\right) \tag{12.9}$$

To find the optimal w_{ij} we need to solve for the partial derivatives

$$\frac{\partial \mathcal{E}}{\partial w_{ij}} = \frac{\partial \mathcal{E}}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} \tag{12.10}$$

If u_j is an output layer, then $u_j = y_j$, and therefore

$$\frac{\partial \mathcal{E}}{\partial u_j} = \frac{\partial}{\partial y_j} \frac{1}{2} \sum (y_i' - y_i)^2 = y_j' - y_j \tag{12.11}$$

Equation 12.11 can be solved for any threshold function σ. If use a logistic function as $\sigma(x) = 1/(1 + e^{-x})$ then

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x} \frac{1}{1 + e^{-x}} = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{e^{-x}\sigma(x)}{1 + e^{-x}} \tag{12.12}$$

But for the logistic function

$$1 - \sigma(x) = 1 - \frac{1}{1 + e^{-x}} = \frac{e^{-x}}{1 - e^{-x}} \tag{12.13}$$

and thus

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) \tag{12.14}$$

Using the fact that $u_j = \sigma\big(\sum_{k=1}^{p} w_{ki} u_k\big)$, for the output layer

$$\frac{\partial u_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sigma\left(\sum_{k=1}^{p} w_{ki} u_k\right) \tag{12.15}$$

$$= u_j(1 - u_j) \frac{\partial}{\partial w_{ij}} \sum_{k=1}^{p} w_{ki} u_k \tag{12.16}$$

$$= u_j(1 - u_j) u_i \tag{12.17}$$

Thus (for the output layer)

$$\frac{\partial \mathcal{E}}{\partial w_{ij}} = y_j(1 - y_j)(y'_j - y_j) u_i \tag{12.18}$$

The reader is referred to more advanced books for an analysis of the hidden layers.[4] The resulting learning rule is

$$\Delta w_{ij} = -\eta \delta_j y_i \tag{12.19}$$

where

$$\delta_j = \begin{cases} y_j(1 - y_j)(y'_j - y_j) & \text{for the output layer} \\ y_j(1 - y_j)(\sum \delta_k w_{jk}) & \text{for hidden layers} \end{cases} \tag{12.20}$$

Neural Networks for Classification in Python

We will build a neural network to classify a wine as either a red wine or a white wine based on its chemical properties. We will use data from the wine quality data set at the UCI machine learning data repository.[5] This data set

[4] See, for example, Hassoun, MH (1995) *Fundamentals of Artificial Neural Networks.* MIT Press.

[5] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In *Decision Support Systems*, Elsevier, **47**(4):547-553. See `https://archive.ics.uci.edu/ml/datasets/wine+quality` for details on the files.

contains two files, one for red wine, and one for white wine. Each record lists 11 physio-chemical attributes of a particular wine, such as fixed acidity or free sulfur dioxide present. There is also a flag for each wine, ranking the wine quality on a scale of 1 through 9, but we will not use that rank for this analysis. Overall, the files rank a total of 6497 wines, of which 1599 are red wines and the rest are white wines. Our goal is to see if we can build a neural network to identify if a wine is either red or white based on the physio-chemical properties.

We'll start by reading the files from the repository using **read_csv**. The files use semi-colons as delimiters so we use the **sep** option in **read_csv**.

```
red=pd.read_csv("https://archive.ics.uci.edu/ml/
     machine-learning-databases/
     wine-quality/winequality-red.csv", sep=";")
white=pd.read_csv("https://archive.ics.uci.edu/ml/
     machine-learning-databases/
     wine-quality/winequality-white.csv", sep=";")
```

We'll extract all of the numerical features (except for wine quality) and join them together in a single file as the **X** data set. For the **Y** data we'll assign a **1** to every red wine and a zero to every white wine.

```
XRED=np.array(red.drop(columns=["quality"]))
XWHITE=np.array(white.drop(columns=["quality"]))
X=np.vstack([XRED,XWHITE])
Y=[*(len(XRED)*[1]),*(len(XWHITE)*[0])]
```

Now we will generate our test and training sets. We'll reserve 25 percent of the data for testing and use the rest for training. Since we a pretty familiar with separating data into test and training sets now we won't do it manually; instead, we'll use the function **train_test_split** from **sklearn.model_selection**. This does essentially the same thing as the function we implemented on page 41, but returns the files in a different order.

```
from sklearn.model_selection import train_test_split
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y,test_size=.25)
print(len(XTRAIN),len(YTRAIN),len(XTEST),len(YTEST))
```

```
4872 4872 1625 1625
```

Before we run the classifier we should scale the training data so that all the features are in the range [0,1]. We'll do this with the function **MinMaxScaler** in **sklearn.preprocessing**. Scaling is a two step process: fitting the training data, then transforming the test data.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
XTRAINT=scaler.fit_transform(XTRAIN)
XTESTT=scaler.transform(XTEST)
```

Neural network classifiers are implemented by the class **MLPClassifier** in **skearn.neural_network**.[6] Like other classifiers in **sklearn**, we must first load the class, it has **fit**, **predict**, and **predict_proba** methods. We'll try a network with three hidden layers, with 3, 4, and 4 nodes respectively. We run the classifier with the scaled training data, and test it with the scaled test data.

```
from sklearn.neural_network import MLPClassifier as ANN
from sklearn.metrics import recall_score, precision_score, \
     accuracy_score,confusion_matrix,roc_curve
r = ANN(solver='lbfgs', alpha=1e-5,
    hidden_layer_sizes=(3,4,4), random_state=1)
r.fit(XTRAINT,YTRAIN)
YP=r.predict(XTESTT)
print(confusion_matrix(YTEST,YP))
print("Accuracy= ",accuracy_score(YTEST,YP))
print("Recall=   ",recall_score(YTEST,YP))
print("Precision=",precision_score(YTEST,YP) )
```

```
[[1227    1]
 [   9  388]]
Accuracy=  0.9938461538461538
Recall=    0.9773299748110831
Precision= 0.9974293059125964
```

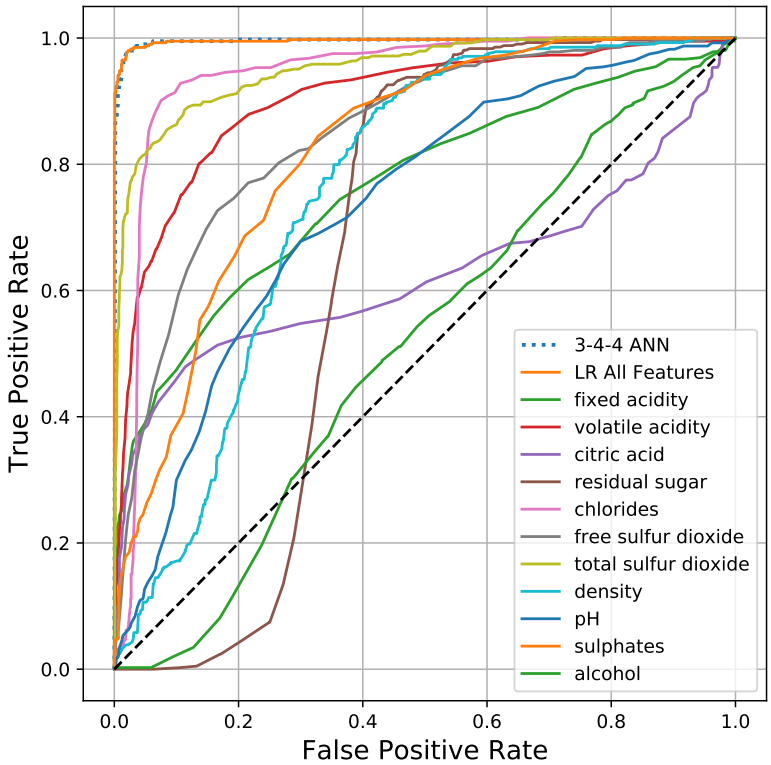
This neural network gives an almost perfect classification with nearly zero errors. The weights between the layers are given by the class attribute **coefs_**:

```
r.coefs_
```

```
[array([[ -0.26659625,  -0.96204366,  -2.10951461],
        [ -0.2925765 ,   1.9031335 ,  -1.59558436],
        [ -0.56391113,  -1.55868509,  -1.32513858],
        [ -0.06126505, -19.78825583,   0.0249536 ],
        [ -0.41401844,   3.25117443,  -1.13642887],
        [  0.09467749,   5.97113368,  -0.36960465],
        [ -0.75797344,  -6.99567027,  -0.52472058],
        [  0.47021644,  22.93805106,  -0.64796793],
        [ -0.08831896,  -0.24643248,  -3.2363302 ],
        [ -0.66192791,   3.45332855,  -1.16089679],
        [ -0.75606913,   1.3960695 ,  -2.13782063]]),
 array([[ 0.33143394,  0.61949583, -0.89191002,  0.37497069],
        [-3.47854508, -0.29337686, -0.40651741,  1.7427559 ],
        [-0.53714922, -0.0960453 ,  0.75653219,  2.12817985]]),
 array([[ 0.65702384,  2.39633517, -3.59304495, -0.7735582 ],
        [ 1.73829533, -1.07994953,  0.50659443,  0.34597349],
        [-0.68874086, -0.14885159,  0.3366933 , -0.14863807],
        [ 5.05292914,  8.10009074, -6.1980924 ,  0.02578735]]),
```

[6]See https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html for full documentation of this class.

Figure 12.6: What makes red wine red? The ROC curves for classification of wine as either red or white are shown here both for an artificial neural network (ANN) with three hidden layers and for logistic regression. The ANN and full-feature logistic curves show nearly 100% accuracy but do not shed any light on which feature is most important in the classification. By classifying separately on each feature we can compare the relative importance of each column in the feature matrix. The code is included in the Jupyter notebook for chapter 12.




```
array([[-6.83652208],
       [ 4.74698817],
       [ 2.03901842],
       [-0.73313378]])]
```

Similarly, the constant weights added to each layer are given by class attribute **intercepts_**

```
r.intercepts_
```

```
[array([-1.08806994,  0.3550354 , -5.63349954]),
 array([-5.76922512, -0.57229715, -0.88995946, -4.78294479]),
 array([10.31318447,  6.96088531, -1.01002004, -0.62791223]),
 array([31.59734969])]
```

We can also evaluate this classifier with an ROC curve. We'll need to calculate the probabilities first with the class method **predict_proba** and then send them **roc_curve** as we discussed in chapter 11 (see figure 12.6). The ROC curve is nearly ideal because, almost reaching the upper left hand corner, because the classification works so well on this data set.

```
YPROB=r.predict_proba(XTEST)[:,1]
fpr, tpr, threshold = roc_curve(YTEST,YPROB)
plt.plot(fpr,tpr)
plt.xlabel("False Positive Rate", fontsize=14)
plt.ylabel("True Positive Rate", fontsize=14)
plt.grid()
plt.plot([0,1],[0,1],ls="--")
fig=plt.gcf()
fig.set_size_inches(6,6)
```

Network Visualization

Sometimes it is helpful to visualize the neural network itself. If the network is deep (e.g., with many thousands of layers, or if the layers have many thousands of nodes) this is difficult to do. In our case, we have 11 inputs, and three hidden layers. The hidden layers have sizes that are most four nodes, so it should be possible to actually draw a picture of it. While **sklearn** does not actually provide any methods to do that, the general graph drawing package **graphviz** provides a relatively straight-forward solution. The following function will draw such a figure using **graphviz**.[7]

```
from graphviz import Digraph
def VisualizeNN(hidden, inputs, outputs,\
 file="NNVisualization"):
    layers=[len(inputs),*hidden,len(outputs)]
    dot=Digraph()
    i=1
    def nodename(i,j):
        return "L["+str(i)+"]["+str(j)+"]"
    for layer in layers:
        for j in range(1,layer+1):
            if i==1:
                thelabel=inputs[j-1]
                theshape="ellipse"
            elif i==len(layers):
                thelabel=outputs[j-1]
                theshape="ellipse"
```

[7]Graphviz is documented at `https://graphviz.readthedocs.io/en/stable/manual.html`.

```
            else:
                thelabel=" "
                theshape="circle"
            dot.node(nodename(i,j),label=thelabel,
               shape=theshape)
            if i>1:
                thisnode=nodename(i,j)
                ilowerlayer=i-1
                ilowerlayers=layers[ilowerlayer-1]
                for k in range(1,ilowerlayers+1):
                    dot.edge(nodename(ilowerlayer,k),
                      thisnode)
        i+=1
    outputfile=uniqueFileName(file,type="gv")
    dot.render(outputfile, view=True)
```

Python's implementation of **Graphviz** will automatically generate an output file as both graphviz language and as a **pdf** file. By default, this codes writes a plot to **NNVisualization.pdf**. A copy of the graphviz source language code is written to a second file **NNVisualization.gv**. The **view=True** line at the very end of the code will open up a window with the visualization on the screen. Additional visualizations can be generated by running the **DOT** application from the command line on the **gv** file.[8]

The function **VisualizeNN** as written above also requires a function that prevents overwriting existing files. The function **uniqeFileName** checks to see if a file already exists, this function will create a new file name by adding a number to the end of the desired file name.

```
from os.path import isfile
def uniqueFileName(filename, type=""):
    # make sure the file name is unique (new)
    nameparts = filename.rsplit(".",1)
    # determine the base file name
    if len(nameparts)<2:
        nameparts.append(type)
    left, right = nameparts
    # generate a unique output file name
    i = 1
    fname = left+"."+right
    while isfile(fname):
        fname = left+str(i)+"."+right
        i += 1
    return(fname)
```

We can then produce a visualization of our wine network as follows. We'll label the input nodes with the names of the columns of the data. Since some of the

[8]See https://graphviz.gitlab.io/download/ and http://www.graphviz.org/documentation/.

names are very long, we'll insert newline characters where they have spaces.

```
input_labels=list(set([x.replace(" ","\n") \
   for x in list(red.columns)])-set(["quality"]))
output_labels=["Output = 1 for Red Wine or 0 for White Wine"]
VisualizeNN([3,4,4], input_labels,output_labels)
```

The visualization is shown in figure 12.7.

Neural Networks for Regression in Python

Neural networks can also be used for regression. The output nodes then produce y values rather than categories. This is implemented by **sklearn** in **MPLRegressor**.[9] We will demonstrate by trying to predict automobile miles per gallon as we did using ridge and lasso regression (see page 66). We start by loading the data file, adding headers, and then eliminating any records that have missing features.

```
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
     machine-learning-databases/auto-mpg/auto-mpg.data",
     header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
     "model","origin","carname")
data = data.dropna(axis=0)
```

We'll extract the **"mpg"** column for our **Y** values. We want to predict this from the number of cylinders (**"cyl"**), the engine displacement (**"displ"**), the engine horsepower (**"hp"**), the car weight (**"weight"**), and the car acceleration (**"accel"**). We'll extract these columns into our **X** data.

```
X=data[["cyl", "displ","hp","weight","accel"]]
Y=data["mpg"]
```

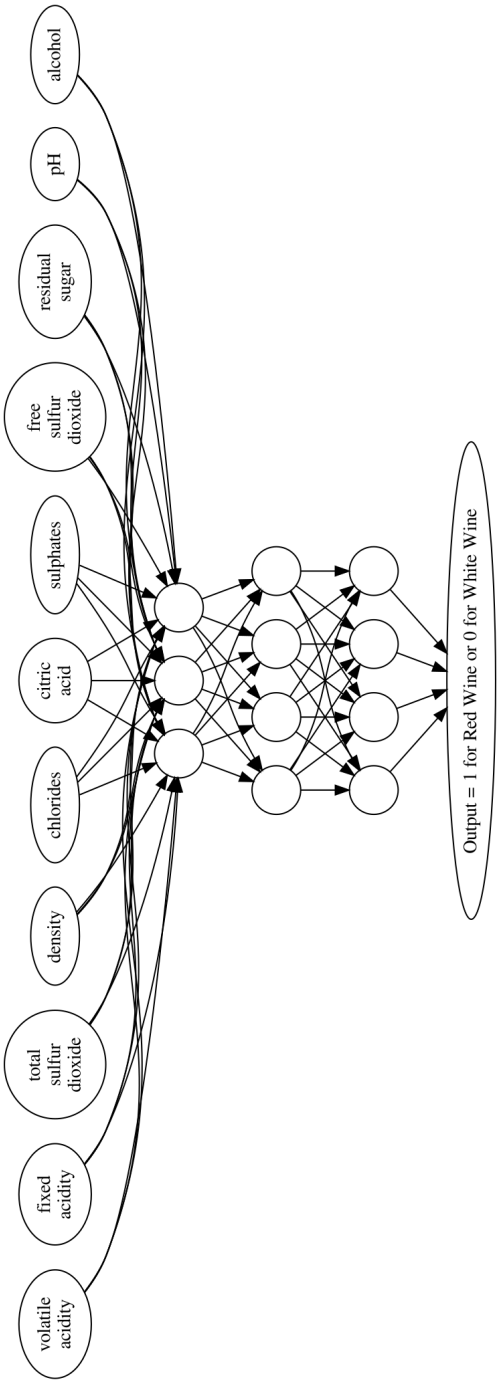
This type of data works better if we scale the input to the unit interval. We'll use **MinMaxScaler** for this. Here are our includes.

```
from sklearn.neural_network import MLPRegressor as REG
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_squared_error
```

Next we'll load the regressor and do 25 train-test splits to get an idea of the variation in the residual mean squared error, scaling the features to a range of

[9]See https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html.

Figure 12.7: Neural network with three hidden layers used to identify red wine.



[0,1]. We'll arbitrarily try a network with three hidden layers that each have nine nodes. We need the following includes:

```
scaler = MinMaxScaler()
for j in range(nruns):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y))
    XTRAINT=scaler.fit_transform(XTRAIN)
    XTESTT=scaler.transform(XTEST)
    r = REG(solver='lbfgs', alpha=1e-5,random_state=1,
          hidden_layer_sizes=(9,9,9))
    r.fit(XTRAINT,YTRAIN)
    YP=r.predict(XTESTT)
    errors.append(mean_squared_error(YP,YTEST))
print(np.mean(errors),np.std(errors))
```

```
15.926269318692174 3.3713083042333074
```

This is somewhat better than we did with ridge regression (see page 71). The disadvantage here is that we don't really have a meaningful model. We can certainly write out a complex system of equations involving the final weights and the various features, but these weights don't really tell us much. At least in linear or ridge regression, the parameters gave us a sort of slope, which tells us how much the dependent variable is expected to change for a unit change in a given feature. To visualize the results we can project the predicted and observed values onto the two dimensional weight/mpg plane. The results for are shown in 12.8. There is no clear curve or shape to the fit, just a cluster of predicted points within the larger cluster of observed points. This demonstrates how artificial neural networks can be very good at making predictions or simulating real-world data, but very poor at actually explaining the data or giving a physically meaningful model.

Multi-class Network Example: Digit Recognition

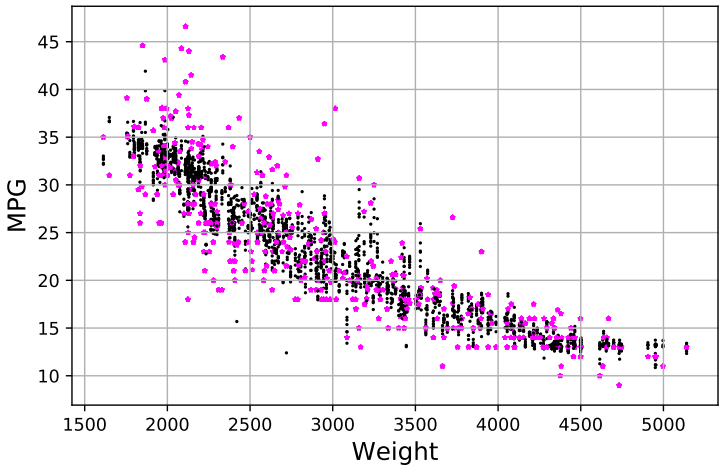
The MNIST (Modified National Institute of Standards and Technology, USA) data set of handwritten digis contains some 70,000 examples of handwritten digits (60,000 marked training images and additional 10,000 test images).[10] Examples of some of the handwritten digits in the data file are shown in figure 12.9.

The data files are large enough that it is easier to deal with them in their original zipped format using Python's **`gzip`** package.[11] The images are stored as matrices of unsigned bytes, with each byte containing a pixel value from 0 to

[10] LeCun, Y et. al. (1998) "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* **86**(11):2278-2324. The data files are available for download from `http://yann.lecun.com/exdb/mnist/`. This website describes the file formats in detail.

[11] The code presented here was inspired by a post at `codeinspiredbyhttps:`

Figure 12.8: Observed and predicted values for a single instance of the neural network model with three hidden layers of five nodes each, projected into the weight/mpg plane. The dots give predictions and the markers give observations.



255 representing a gray scale level. Labels are also unsigned bytes, and consist of the integer values 0 through 9. The authors' give the following warning: "please note that your browser may uncompress these files without telling you you." There are four files:

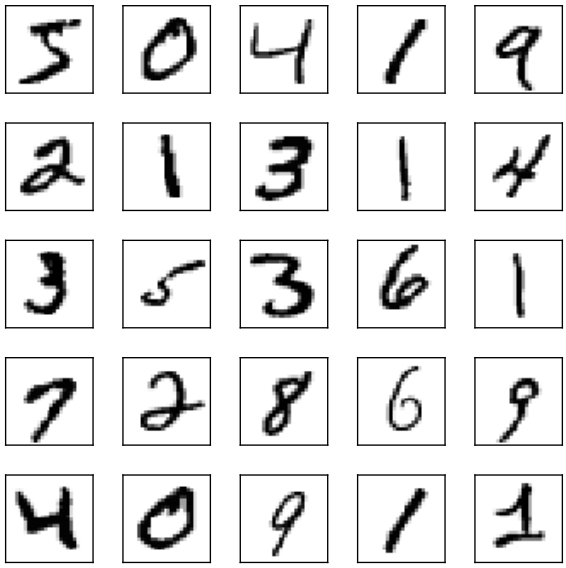
File Name	File Description
`train-images-idx3-ubyte.gz`	Train set 28x28 images
`train-labels-idx1-ubyte.gz`	Training set 0-9 labels
`t10k-images-idx3-ubyte.gz`	Test set 28x28 images
`t10k-labels-idx1-ubyte.gz`	Test set 28x28 labels

We will not perform multiple train/test splits on this data; in fact, we will not do any train/test splits. Instead, we will use the single split provided by the authors.

The function **`read_MNIST_labels(zipfile)`** will read either of the MNIST zip files and return a single array. The kth element of the array will give the exemplar value (a digit from 0 through 9) that corresponds to the kth row of the corresponding image file. We we use this function to read the label files into our **`Y`** arrays.

`//stackoverflow.com/questions/40427435/extract-images-from-idx3-ubyte-file-or-gzip-via-python` and is based on the information in the LeCun web site (op cit).

Figure 12.9: Sample digits from the MNIST data set



```
def read_MNIST_labels(zipfile):
    f = gzip.open(zipfile,'r')
    Y=[]
    buf=f.read(8) # skip header
    while (True): # read till end of file
        buf = f.read(1 * 32) # grab 32 bytes
        # convert to floating point
        labels = np.frombuffer(buf,
            dtype=np.uint8).astype(np.int64)
        if len(labels)>0:
            Y=Y+list(labels)
        else:
            break
    return (Y)
```

The function, **read_MNIST_images (zipfile)**, will read either of the image files and return a **numpy array** where each row represents a single image, with the 28×28 image expanded into a single 784 element array. Each row can then be used as a single feature vector, with the pixel at the (i, j) coordinate corresponding to a single feature (this pixel is in the $28i + j$-th position of the row). We will use this function to read the image files into our **X** arrays.

```
import gzip
def read_MNIST_images(zipfile):
    image_size = 28
    num_images=1
    maximages=100000
    f = gzip.open(zipfile,'r')
    X=[]
    i=0
    f.read(16) # skip header information
    while(True): # continue until end of file
        # read one image at a time
        buf = f.read(image_size * image_size * num_images)
        # convert unsigned integer data to floating point
        data = np.frombuffer(buf,
            dtype=np.uint8).astype(np.float32)
        if (len(data)<1) or (i>maximages):
            break
        # reshape array to a single vector
        data = data.reshape(num_images,
            image_size, image_size, 1)
        data=np.ravel(data)
        X.append(data)
        i+=1
    return(X)
```

First, read the data files.

```
XTRAIN=read_MNIST_images('train-images-idx3-ubyte.gz')
print(len(XTRAIN),"images read")
XTEST=read_MNIST_images('t10k-images-idx3-ubyte.gz')
print(len(XTEST),"images read")
```

```
60000 images read
10000 images read
```

```
YTRAIN=read_MNIST_labels("train-labels-idx1-ubyte.gz")
print(len(YTRAIN), YTRAIN[:5])
YTEST=read_MNIST_labels("t10k-labels-idx1-ubyte.gz")
print(len(YTEST), YTEST[:5])
```

```
60000 [5, 0, 4, 1, 9]
10000 [7, 2, 1, 0, 4]
```

If the files are not in your working directory you need to specify the full (or relative) path to their location.

Next, we'll look at the images to see if they really look like letters. This part is not necessary but it makes the problem more interesting. The following function will display 25 digits starting at the kth location in the array.

```
def display_MNIST(kmin,outfile="MNIST-digit-examples.pdf"):
    maxrows=5
    fig,ax=plt.subplots(nrows=maxrows,ncols=5)
    for irow in range(maxrows):
        for icol in range(5):
            k=5*irow+icol+kmin
            img=np.array(XTRAIN[k]).reshape(28,28)
            # subtract from 1 to reverse image
            ax[irow][icol].imshow(1-img,
               aspect="auto",cmap="gray")
            ax[irow][icol].set_xticks([])
            ax[irow][icol].set_yticks([])
    fig.set_size_inches(5,maxrows)
    fig.tight_layout()
    fig.savefig(outfile)
```

The image displayed in figure 12.9 was generated with the code **display_MNIST(0)**.

As usual, we will transform the feature vectors into a standard interval using **MinMaxScaler**.

```
scaler = MinMaxScaler()
XTRAINT=scaler.fit_transform(XTRAIN)
XTESTT=scaler.transform(XTEST)
```

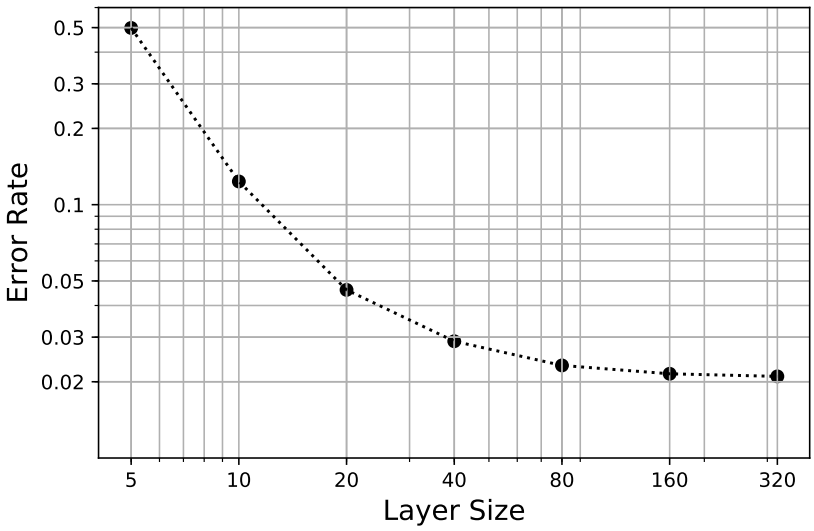
We can then train and test a network with, say, three hidden layers, each containing 80 neurons. Here are the results for a single instance of training such a network.

```
size=80
r = ANN(solver='lbfgs', alpha=1e-5,
    hidden_layer_sizes=(size, size, size), random_state=1)
r.fit(XTRAINT,YTRAIN)
YP=r.predict(XTESTT)
print(confusion_matrix(YTEST,YP))
acc=accuracy_score(YTEST,YP)
print("Accuracy= ",acc)
```

```
[[ 968    0    3    0    1    2    2    0    2    2]
 [    0 1122    3    0    0    1    2    1    6    0]
 [    4    2 1004    5    2    0    2    8    5    0]
 [    1    0    7  991    0    3    0    2    4    2]
 [    0    0    5    0  960    0    5    2    1    9]
 [    2    0    1   11    0  864    6    0    5    3]
 [    3    3    1    1    4    5  937    1    3    0]
 [    0    3    8    4    2    0    1 1001    4    5]
 [    4    0    2    5    1    5    1    4  950    2]
 [    1    2    0    4   10    4    3    8    6  971]]
Accuracy=  0.9768
```

If we put this code in a loop for layer size (the variable **`size`**) we can explore how the accuracy varies with network size. Ideally we would want to vary the test/train split and the randomness of the initial weight assignment to the network, but for illustration purposes we will only calculate a single instance for each size. Since it is a very large problem to exhaustively consider a large number of different configurations, we only explore 3-level networks where each layer has the same number of nodes. In figure 12.10 we plot the error rate as a function of layer size. The error rate is one minus the model accuracy.

Figure 12.10: Error rate as a function of layer size s for a neural network consisting of three hidden layers, each of the same size. The error rate is equal to 1 minus the accuracy. One instance is plotted for each size network for the MNIST digits data.



Chapter 13

Naive Bayes

The **Naive Bayes'** method, which we study in this chapter, the **K-Nearest Neighbors** (KNN) algorithm (chapter 14), the **linear discriminant analysis** (LDA) algorithm and **quadratic discriminant analysis** (QDA) method (chapter 15) are all examples of **probabilistic classifiers**. These classifiers calculate the probability of class membership based on observations of the existing data distribution and assumptions about probability distribution of the population from which the data was drawn.

An important tool for probabilistic methods is Bayes' Rule.[1] Bayes' Rules states that the probability of an observation $\mathbf{x}$ falling into a cluster c_k is

$$P(c_k|\mathbf{x}) = \frac{P(c_k)P(\mathbf{x}|c_k)}{P(\mathbf{x})} \tag{13.1}$$

class conditional where

$$P(c_k) = \text{Probability of cluster } k \tag{13.2}$$

$$P(\mathbf{x}) = \text{Probability of observation } \mathbf{x} \tag{13.3}$$

$$P(\mathbf{x}|k) = \text{Probability of observation } \mathbf{x} \text{ assuming cluster } c_k \tag{13.4}$$

Equation 13.1 can be written in an algorithm-ready form as follows:

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{normalization}} \tag{13.5}$$

[1] Bayes' Rule (sometimes called Bayes' Theorem) is named for Thomas Bayes (1701-1761), one of the early founds of of probability theory. Details can be found in any textbook on probability theory. We have re-written it here in terms of class membership; it is more commonly stated in terms of events A and B, e.g., as $P(A|B) = P(A)P(B|A)/P(B)$. See theorem A.3.

where "prior" is a first guess at the probability; "liklihood" is the class-conditional probability based on a theoretical description of the population and the evidence given by the data; an dclass conditional "posterior" is an updated estimate of the class probability, given the observed data. The denominator $P(\mathbf{x})$ is called a "normalization" because it is fixed for any given data point.

A probabilistic method works by finding the classification scheme that maximizes the probability. In other words, the optimal classification is given by

$$y_p = \arg\ \max_{c_k} \left[P(c_k|\mathbf{x})\right] = \arg\ \max_{c_k} \left[\frac{P(c_k)P(\mathbf{x}|c_k)}{P(\mathbf{x})}\right] \tag{13.6}$$

$$= \arg\ \max_{c_k} \left[P(c_k)P(\mathbf{x}|c_k)\right] \tag{13.7}$$

where the last line follows because we can effectively ignore the denominator in the maximization as it is fixed.

Equation 13.7 gives us a **Bayesian Classifier**. We initialize the classifier using the observed class distribution as the prior probability. We then must make some assumption about the likelihood function; the most common assumption is a multivariate Gaussian distribution. Then we use an iterative maximization algorithm to find the posterior probability.

In a **Naive Bayes Classifier**, we assume all of the features are independent

$$P(\mathbf{x}|c_k) = \prod_{j=1}^{d} P(x_j|c_k) \tag{13.8}$$

where d is the number of features (the dimension of $\mathbf{x}$). From equation 13.7, the predicted class is

$$y_p = \arg\ \max_{c_k} \left[P(c_k)P(\mathbf{x}|c_k)\right] = \arg\ \max_{c_k} \left[P(c_k)\prod_{j=1}^{d} P(x_j|c_k)\right] \tag{13.9}$$

This considerably simplifies the mathematics – for example, a multivariate Gaussian becomes the product of one variable Gaussians.[2]

This is not always a very accurate assumption. Consider what it means for the auto mpg data set (page 66): in predicting **`mpg`**, we would assume that the factors **`cyl`**, **`displ`**, **`hp`**, **`weight`**, and **`accel`** are assumed to be independent. In fact, these factors are hardly independent. We can get a hint as this by

[2]Pseudocode for the algorithms of Bayesian and Naive Bayes' classifiers are given in Zaki, MJ and Meira, W (2014) *Data Mining and Analysis*, Cambridge Univ. Press, chapter 18.

calculating the correlation between the columns (see page 158). Cars with more cylinders and larger displacement tend to have higher acceleration, for example. So while naive Bayes' method is often a first method of choice, care should be taken to ensure that the factors are independent first. Independent factors can be extracted with an independent components analysis (ICA, page 155). Nevertheless, we can often still plow blindly ahead and often will get quite good results using the naive Bayes method.

Naive Bayes in Python

We will create a 3-class data set from the auto mpg data set (see page 66) that describes the number of cylinders as a function of two variables, miles per gallon (**mpg**) and acceleration (**accel**). First we load the standard packages and read the data file with **read_fwf**. Then we label the columns and remove records with missing data.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
data=pd.read_fwf("https://archive.ics.uci.edu/ml
    /machine-learning-databases/auto-mpg/auto-mpg.data",
    header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
    "model","origin","carname")
data = data.dropna(axis=0)
```

Since we only want three classes, we'll extract the records for cars that have either 4, 6 or eight cylinders. We'll just take the three columns we need.

```
cardata=np.array(data[["cyl","mpg","accel"]])
cars=np.array([line for line in cardata if line[0] in [4,6,8]])
```

We'll define the **X** data as the 2nd and third column (Python indices 1 and 2, skipping the first column). We'll take the **Y** data from the first column (the cylinder data, Python index 0). At the same time we'll rescale **Y** so that 4, 6, and 8 are mapped to 0, 1, and 2 respectively.

```
X=cars[:,1:]
Y=cars[:,0]/2-2
```

Dividing by 2 converts 4, 6, and 8 into 2, 3, 4; subtraction by 2 gives us the desired class labels of 0, 1, and 2.

Several different naive Bayes' methods are implemented in the Python package **sklearn.naive_bayes**.[3] The difference between these depends on the probability distributions that the features are assumed to follow such as Gaussian, multinomial, or Bernoulli. We will demonstrate the Gaussian model, implemented in **GaussianNB**.[4] We'll load the required packages first.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,confusion_matrix
```

Here we do the train/test split, load the classifier, do a fit, a prediction, and an evaluation. We'll do 100 train/test splits and calculate the mean error (error = 1 - accuracy) +/- 1.96 standard deviations. Then we'll print a single confusion matrix to demonstrate the results (this will be the confusion matrix for the final calculation in the loop).

```
errs=[]
nsplits = 100
for split in range(nsplits):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    gnb=GaussianNB()
    gnb.fit(XTRAIN,YTRAIN)
    YP=gnb.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("%d Splits: Mean Error=%7.6f +/- %7.6f (95%%)"\
      %(nsplits, np.mean(errs),1.96*np.std(errs)))
```

```
100 Splits: Mean Error=0.120928 +/- 0.064840 (95%)
[[42  2  0]
 [ 4 15  0]
 [ 1  5 28]]
```

The 1.96 SD bounds is an estimate of the 95% variation in the error. Thus we expect sample of this size to have an error rate of somewhere between 6% and 19% approximately 95% of the time. The average error rate is around 12%, and we would like to do better.

There are several possible reasons reasons for a high error rate:

1. We might be missing some information by limiting ourselves to just two of the features in the data. We will return to this question in the section below on **Adding More Features** and **Encoding Categorical Features**.

[3] The differences between these methods are described at `https://scikit-learn.org/stable/modules/naive_bayes.html`.

[4] See `https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB` for full documentation of **GaussianNB**.

2. **GaussianNB** assumes that the data is drawn from a normal distribution.

```
from scipy.stats import kurtosis
kurtosis(X, fisher=False)
```

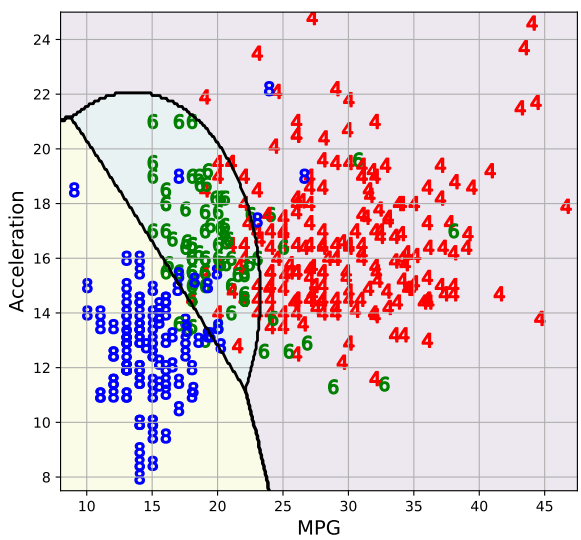
```
array([2.45939109, 3.46390269])
```

A normal distribution has a kurtosis of 3.0; thus **mpg** has a flatter distribution that normal ($k < 3$) and **accel** has a pointier distribution than normal ($k > 3$) (see page 256).

3. There is an extensive class overlap illustrated by the data markers in figure 13.1. A number of 6 and 8 cylinder cars are buried deep within the region corresponding to the 4-cylinder cars and it would be unlikely that the features we have chosen would be sufficient to identify them.

4. Naive Bayes, by its nature requires that the features be independent. In fact, acceleration and miles per gallon are not independent variables. To some extent we can deal with this by doing an Independent Component Analysis (ICA). We will return to this below.

Figure 13.1: Decision boundary for Gaussian naive bayes classifier for the number of cylinders as either 4, 6, or 8 from the auto mpg data set.



The function **visualize_classifier_contours** can be used to display the decision boundaries. They are drawn by making a contour plot and applying a

classifier on an exhaustive grid.[5] The algorithm generates an array of (x, y) values using **numpy.meshgrid** , makes predictions for every pair, draws the decision contours, and then displays and saves the figure. It uses **contour** to draw the class boundaries, and **contourf** to color in the classes.

Because we know the classes are 0, 1, and 2, we can pick any numbers between the classes – such as 0.5 and 1.5 – to use to draw the class boundaries. We also draw the original data points as small numbers.

```
def visualize_clasifier_contours(clf,X,Y,DX,DY,thelevels,colors,
  markers,xlab="X",ylab="Y", filename='classifier-figure.pdf'):
    xinterval = X[:, 0].min() - 1, X[:, 0].max() + 1
    yinterval = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(*xinterval, DX),
                         np.arange(*yinterval, DY))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    c=plt.contour(xx, yy, Z, alpha=1,levels=thelevels,
      linewidths=2,colors="k")
    c=plt.contourf(xx, yy, Z, alpha=.1)
    nclasses=len(markers)
    for i in range(nclasses):
        xyvals=X[Y==i]
        x,y=zip(*xyvals)
        plt.scatter(x,y,marker=markers[i],color=colors[i],s=100)
    plt.xlabel(xlab,fontsize=14)
    plt.ylabel(ylab,fontsize=14)
    plt.grid()
    fig=plt.gcf()
    fig.tight_layout()
    fig.set_size_inches(6,6)
    fig.savefig(filename)
```

Here we give the code to invoke the visualization.

```
colors=["red", "green","blue"]
markers=["$4$","$6$","$8$"]
visualize_clasifier_contours(gnb,X,Y,.1,.1, [0.5,1.5],colors,
    markers,"MPG","Acceleration")
```

The result is shown in figure 13.1.

[5]This method is suggested at the sklearn webpage https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#sphx-glr-auto-examples-neighbors-plot-classification-py. The code shown here is a modification of the example code.

Adding More Features

Next, we'll see what happens if we add all the numerical features to the data.

```
X=np.array(data[["mpg","displ","hp","accel"]])
```

We'll also consider all the the possible **Y** values, not just 4, 6, or 8 cylinders:

```
Y=np.array(data["cyl"])
unis=np.unique(Y).tolist()
print(unis)
Y=np.array([unis.index(j) for j in Y])
```

```
[3, 4, 5, 6, 8]
```

Now we have five classes to distinguish instead of two, and we have four features in the **X** array. The five classes are labeled as class 0 (3 cylinders); class 1 (4 cylinders); class 2 (5 cylinders); class 3 (6 cylinders); and class 4 (8 cylinders).

As before, we'll do 100 runs and compute the average error.

```
errs=[]
for j in range(100):
    XTRAIN,XTEST,YTRAIN,YTEST=train_test_split(X,Y)
    gnb=GaussianNB()
    gnb.fit(XTRAIN,YTRAIN)
    YP=gnb.predict(XTEST)
    errs.append(1-accuracy_score(YP,YTEST))
print("%d Splits: Mean Error=%7.6f +/- %7.6f (95%%)"\
      %(nsplits, np.mean(errs),1.96*np.std(errs)))
print("Last confusion matrix:")
print(confusion_matrix(YP, YTEST))
```

```
100 Splits: Mean Error=0.043673 +/- 0.037956 (95%)
Last confusion matrix:
[[ 1  0  0  0  0]
 [ 0 46  1  1  0]
 [ 0  0  0  0  0]
 [ 0  0  0 20  0]
 [ 0  0  0  1 28]]
```

This is a significant improvement; we've decreased the error from about 12% to under 5%.

Next we turn to the following question: can we make use of any of the remaining data in the file?

Encoding Categorical Features

Lets look a little closer at the features in the data set we just examined. To get a list of the column headers of a **pandas DataFrame** we use the class attribute **dataframe.columns**:

```
data.columns
```

```
Index(['mpg', 'cyl', 'displ', 'hp', 'weight', 'accel', 'model',
 'origin','carname'],
 dtype='object')
```

This returns a **pandas Index** object.[6] For our purposes we can treat this object as a list of strings giving the Relevant Papers: Relevant Papers:column headers, in order of the columns.

The columns **'mpg'**, **'displ'**, **'hp'**, **'weight'** and **'accel'** contain numerical data that represent real valued quantities. These are called **quantitative features** or **quantitative variables** because (a) they are described by numbers; and (b) the numbers have physical units (e.g., miles per gallon, cubic inches, horsepower, pounds, feet per second squared).

The columns **'cyl'**, **'model'**, **'origin'** and **'carname'** represent features that can be used to categorize the data into different subsets. Features such as **'model'** and **'carname'** are strings that, taken together, give a unique identifier for each type of car. Feature **'origin'** is a numerical representation of the place where the car was manufactured: 1 (North America); 2 (Europe); or 3 (Asia). Finally, **'cyl'** gives the number of cylinders, in the set $\{3, 4, 5, 6, 8\}$.

```
Y=np.array(data["cyl"])
unis=np.unique(Y).tolist(); unis
```

```
[3, 4, 5, 6, 8]
```

While both **'origin'** and **'cyl'** are numbers, neither of them have physical units. We call features such as **'origin'**, **'cyl'**, **'carname'**, or **'model'** **categorical features** or **categorical variables**.

Suppose we want to predict the number of cylinders of vehicle based on the following: **'mpg'**, **'displ'**, **'hp'**, **'weight'**, **'accel'** and **'origin'**. Except for **'origin'** all of these features are numerical. To represent **'origin'** we need to somehow represent the information numerically. Unfortunately the values

[6]For details see https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Index.html.

that are already in the data set: 1 for North America, 2 for Europe, and 3 for Asia are not a good a representation, because this imposes an ordering on the data. There is no particular reason why values should be ordered in this manner and results could end up being biased. We fix this be introducing a new encoding where each value that a category can take on is represented by a different feature. Since **'origin'** can take on three values, we add three new features, i.e., three more columns to the feature matrix. These columns could be called **'Amer'**, **'Euro'** and **'Asia'**. The values in the columns will be set to either **1** or **0**, according to the following rule:

- If **'origin'=1** set **'Amer'=1**, **'Euro'=0**, and **'Asia'=0**
- If **'origin'=2** set **'Amer'=0**, **'Euro'=1**, and **'Asia'=0**
- If **'origin'=3** set **'Amer'=0**, **'Euro'=0**, and **'Asia'=1**

This creates an encoding where each column only has a **1** in the individual cars corresponding to a particular origin, and otherwise has a **0** for that feature. This is called a **1-Hot** or **One-Hot** encoding. Each car has a one, or a hot value, only in the column corresponding to the correct category. The word **hot** is used in the sense of an electrical circuit: when it is hot, it is turned on, and when it is not hot, it is turned off.[7]

The **pandas** function **get_dummies** implements **One-Hot** encodings.[8] To demonstrate it we'll create a temporary new **DataFrame** with **origin** as a column and print some of the data.

```
DF=data[["mpg","displ","hp","accel","origin"]]
DF[16:21]  # print lines 16 through 21
```

	mpg	displ	hp	accel	origin
16	18.0	199.0	97.0	15.5	1
17	21.0	200.0	85.0	16.0	1
18	27.0	97.0	88.0	14.5	3
19	26.0	97.0	46.0	20.5	2
20	25.0	110.0	87.0	17.5	2

We then tell **get_dummies** which columns represent categorical variables by giving it a list of the column names. Since we no longer need **DF** we'll overwrite it with the modified **DataFrame**.

```
DF=pd.get_dummies(DF,columns=["origin"])
DF[16:21]
```

[7]Some modelers prefer to use one fewer new variable then there are categories and set the value for the last category to all zero's. In this case, an Asian car would be identified by setting *both* **'Amer'=0** and **'Euro'=0**, without the need for the third variable **'Asia'**.

[8]See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

```
        mpg     displ   hp      accel   origin_1 origin_2 origin_3
16      18.0    199.0   97.0    15.5    1        0        0
17      21.0    200.0   85.0    16.0    1        0        0
18      27.0    97.0    88.0    14.5    0        0        1
19      26.0    97.0    46.0    20.5    0        1        0
20      25.0    110.0   87.0    17.5    0        1        0
```

Finally, convert the temporary data frame to a feature matrix.

```
X=np.array(DF)
print(X[16:21]) # print lines 16 through 20
```

```
[[ 18.  199.   97.   15.5   1.    0.    0. ]
 [ 21.  200.   85.   16.    1.    0.    0. ]
 [ 27.   97.   88.   14.5   0.    0.    1. ]
 [ 26.   97.   46.   20.5   0.    1.    0. ]
 [ 25.  110.   87.   17.5   0.    1.    0. ]]
```

Since we want to predict the number of cylinders as a category, starting with category zero, we'll use the **unis** list defined on the previous page:

```
Y=[unis.index(j) for j in Y]
```

This converts each 3 to a 0, 4 to a 1, 5 to a 2, 6 to 3, and 8 to a 4.

Here we run the Naive Bayes' method through 100 train/test splits.

```
errs=[]
for j in range(100):
    gnb=GaussianNB()
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    gnb.fit(XTRAIN,YTRAIN)
    YP=gnb.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("%d Splits: Mean Error=%7.6f +/- %7.6f (95%%)"\
      %(nsplits, np.mean(errs),1.96*np.std(errs)))
print("Last confusion matrix:")
print(confusion_matrix(YP,YTEST))
```

```
100 Splits: Mean Error=0.359490 +/- 0.147373 (95%)
Last confusion matrix:
[[ 0  0  0  0  0]
 [ 1 42  2  3  0]
 [ 0  0  0  0  0]
 [ 0  0  0  1  1]
 [ 0  0  0 22 26]]
```

Rather than improving our prediction accuracy, we appear to have gotten worse. The error has increased to over 35%.

Independent Component Analysis (ICA)

One possible reason for the increase in error is that the data features do not represent independent variables. Independent Component Analysis (ICA) is a method that extracts the independent components of a data set. We will demonstrate the use of independent component analysis using the FastICA algorithm[9]. It is implemented in the package **sklearn.decomposition.FastICA**.[10]

To use ICA we must transform the **X** matrix into its independent components. The implementation requires us to specify the number of components we want. After we perform the transformation, we than use the transformed matrix in place of our original feature matrix. For example, to find a transformation with two components:

```
from sklearn.decomposition import FastICA
ICA=FastICA(n_components=2)
ICA.fit(X)
F=ICA.transform(X)
```

We would then extract test and training data from the transformed matrix **F**. There is no change to the exemplar values **Y**. For a single run, the code is the same as before, with **X** replaced by **F**.

```
gnb=GaussianNB()
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(F,Y)
gnb.fit(XTRAIN,YTRAIN)
YP=gnb.predict(XTEST)
print(1-accuracy_score(YP,YTEST))
print(confusion_matrix(YP,YTEST))
```

```
0.05102040816326525
[[ 2  0  0  0  0]
 [ 0 51  1  2  0]
 [ 0  0  0  0  0]
 [ 0  2  0 14  0]
 [ 0  0  0  0 26]]
```

The 5% error rate using only two components and a single split is encouraging. Next, we'll look at what happens if we average over 100 splits. We'll also look at varying the number of components

[9] Hyvärinen, A and Oja, E (2000) "Independent Component Analysis: Algorithms and Applications," *Neural Networks*, **13**:411-430.

[10] For full documentation on **FastICA** see `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.FastICA.html`.

```
nsplits=100
for ncomp in range(2,8):
    ICA=FastICA(n_components=ncomp)
    ICA.fit(X)
    F=ICA.transform(X)
    errs=[]
    for j in range(nsplits):
        gnb=GaussianNB()
        XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(F,Y)
        gnb.fit(XTRAIN,YTRAIN)
        YP=gnb.predict(XTEST)
        errs.append(1-accuracy_score(YTEST,YP))
    print("%d Splits, %d Components: Mean Error=%7.6f
    +/- %7.6f (95%%)"\
    %(nsplits,ncomp, np.mean(errs),1.96*np.std(errs)))
```

```
100 Splits, 2 Components: Mean Error=0.086224 +/- 0.045815 (95%)
100 Splits, 3 Components: Mean Error=0.111020 +/- 0.057708 (95%)
100 Splits, 4 Components: Mean Error=0.112143 +/- 0.052573 (95%)
100 Splits, 5 Components: Mean Error=0.136020 +/- 0.064687 (95%)
100 Splits, 6 Components: Mean Error=0.104694 +/- 0.054083 (95%)
100 Splits, 7 Components: Mean Error=0.308469 +/- 0.156993 (95%)
```

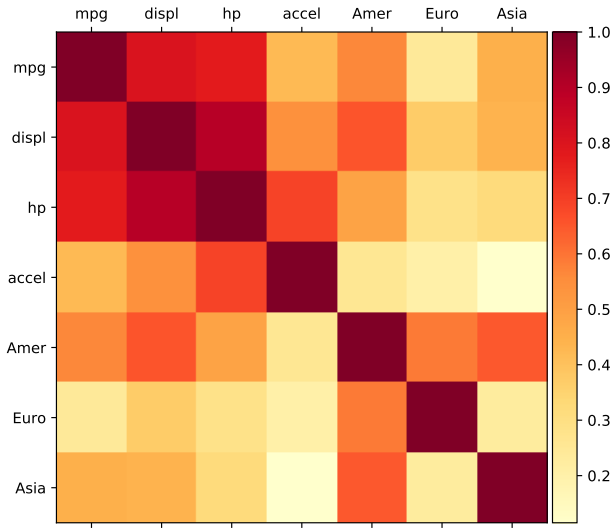
It seems that adding more independent components is making matters worse, and not better. This may be because we increasing the dimension of the problem from 2 to 7 without adding any additional data points. There are just under four hundred data points (total) in the data set. In any given train/test split, we use three quarters of them – about 300 data points – for training, and the rest – about one hundred – for testing. When we increase the number of dimensions of the data, those 300 data points in any given training set are representing up to a 7-dimensional vector space instead of a 2 or 3-dimensional vector space. Because we a not increasing the number of data points, large regions of the higher-dimensional space are not represented by the training set. Consequently we end up fitting the noise in in the training set and are not modeling the actual data. This is called the **curse of dimensionality** (see pages 183ff).

One way to get an idea of the true dimension of the data set is through a Principal Component Analysis (PCA) (see chapter 16). A PCA is a coordinate rotation of the data that is determined by the variance of each coordinate of the transformed variables.

```
from sklearn.decomposition import PCA
pca=PCA(n_components=7)
pca.fit(X)
print(pca.explained_variance_ratio_)
```

```
[9.77103869e-01 2.09376473e-02 1.64085614e-03 2.88975479e-04
```

Figure 13.2: Visualization of correlation matrix for auto mpg data file. The code to produce the plot is given in the notebook file for this chapter.




```
1.44868553e-05 1.41656401e-05 6.19652520e-37]
```

The first two component account for 97.7% and 2.09% of the variance, respectively. The third component accounts for only 0.16% and the last four components, taken together, account for less than 0.04% of the total variance. This is essentially a two-dimensional system.

In addition to the overlap between the classes there is also a considerable about of correlation between the features. The correlation tells us about the relationship or overlap between variables. Features that are strongly related have a correlations close to 1 or -1, while features that are not related to one another tend to have correlations closer to zero. The correlation matrix of **X** is found using **np.corrcoef**. By default, **np.corrcoef** assumes that different features are in different rows. Since we are treating the columns as features, we set the option **rowvar=False** to tell it compute the correlation between the columns rather than the correlation between the rows. The result will be a symmetric diagonal matrix with 1's in the diagonal.

```
np.round(np.corrcoef(X,rowvar=False),3)
```

```
array([[ 1.   , -0.805, -0.778,  0.423, -0.565,  0.244,  0.451],
       [-0.805,  1.   ,  0.897, -0.544,  0.656, -0.372, -0.441],
       [-0.778,  0.897,  1.   , -0.689,  0.49 , -0.285, -0.322],
       [ 0.423, -0.544, -0.689,  1.   , -0.258,  0.208,  0.115],
```

```
[-0.565,  0.656,  0.49 , -0.258,  1.   , -0.591, -0.649],
[ 0.244, -0.372, -0.285,  0.208, -0.591,  1.   , -0.23 ],
[ 0.451, -0.441, -0.322,  0.115, -0.649, -0.23 ,  1.   ]])
```

We can visualize the correlation matrix using **matshow**. This will produce a heat map of the matrix. Since we only care about high-vs-low correlation, we'll take absolute values first.

```
import matplotlib.pyplot as plt
plt.matshow(abs(cmat),cmap="YlOrRd")
```

After adding labels and a color bar (see notebook for chapter 13) we arrive at figure 13.2.

Chapter 14

K-Nearest Neighbors

K-nearest neighbors (KNN) is a supervised classification method in which cells are categorized according to their nearness. Like the Bayesian methods discussed in chapter 13 it is a probabilistic method. There is one input parameter, K, the number of near-neighbors to use. Given any feature vector $\mathbf{x}$, KNN will first determine the K feature vectors that are closest to $\mathbf{x}$ (its K nearest neighbors). KNN determines which cluster to which each of these neighbors is currently assigned. Bayes' rule is then used to determine the most likely class for $\mathbf{x}$. Suppose that there are $N_0, N_1, \ldots$ exemplars in class $c_0, c_1, \ldots$ in the data set. Then we define the prior probability for class c_j as the proportion of exemplars in class j, i.e.,

$$p(c_j) = \frac{N_j}{N} \tag{14.1}$$

where total number of vectors in the data set is

$$N = N_0 + N_1 + N_2 + \cdots \tag{14.2}$$

Further suppose that there are $n_0, n_1, \ldots$ exemplars from classes $c_0, c_1, \ldots$ in a ball of volume V centered about $\mathbf{x}$. The conditional probability for class c_i is the fraction of points from class c_i that lie within the ball. This is given by

$$p(\mathbf{x}|c_i) = \frac{n_i/N_i}{V} = \frac{n_i}{N_i V} \tag{14.3}$$

The unconditional probability is then

$$p(\mathbf{x}) = \frac{K/N}{V} = \frac{K}{NV} \tag{14.4}$$

where

$$K = n_1 + n_2 + \cdots \tag{14.5}$$

is the total number of points in the unit ball. Then by Baye's rule, the posterior probability for class c_i is

$$p(c_i|\mathbf{x}) = \frac{p(\mathbf{x}|c_i)p(c_i)}{p(\mathbf{x})} = \frac{\frac{n_i}{N_i V}\frac{N_i}{N}}{\frac{K}{NV}} = \frac{n_i}{K} \tag{14.6}$$

The predicted class is then (compare with equation 13.7)

$$y_p = \arg\max_{c_i} \; P(c_i|\mathbf{x}) = \arg\max_{c_i} \; n_i \tag{14.7}$$

In other words, the predicted class becomes the class with the most elements in the ball centered at **x**.

K-Nearest Neighbor Classification in Python

We will create a 3-class data set from the auto mpg data set (see page 66) that describes the number of cylinders as a function of two variables, miles per gallon (**mpg**) and acceleration (**accel**). First we load the standard packages and read the data file with **read_fwf**. Then we label the columns and remove records with missing data.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
data=pd.read_fwf("https://archive.ics.uci.edu/ml
    /machine-learning-databases/auto-mpg/auto-mpg.data",
    header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
    "model","origin","carname")
data = data.dropna(axis=0)
```

Since we only want three classes, we'll extract the records for cars that have either 4, 6 or eight cylinders. We'll just take the three columns we need.

```
cardata=np.array(data[["cyl","mpg","accel"]])
cars=np.array([line for line in cardata if line[0] in [4,6,8]])
```

We'll define the **X** data as the 2nd and third column (Python indices 1 and 2, skipping the first column). We'll take the **Y** data from the first column (the cylinder data, Python index 0). At the same time we'll rescale **Y** so that 4, 6, and 8 are mapped to 0, 1, and 2 respectively.

```
X=cars[:,1:]
Y=cars[:,0]/2-2
```

Dividing by 2 converts 4, 6, and 8 into 2, 3, 4; subtraction by 2 gives us the desired class labels of 0, 1, and 2.

The KNN classifier is called **KNeighborsClassifier** in **sklearn.neighbors**.[1] We'll arbitrarily try a classifier that uses 4 near neighbors.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

KNN=KNeighborsClassifier(n_neighbors=4)
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y,test_size=.25)
KNN.fit(XTRAIN, YTRAIN)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
   metric='minkowski', metric_params=None, n_jobs=1,
   n_neighbors=4, p=2, weights='uniform')
```

```
YP=KNN.predict(XTEST)
print(confusion_matrix(YTEST,YP))
print("accuracy = ",accuracy_score(YTEST,YP))
```

```
[[52  1  0]
 [ 5 14  3]
 [ 0  2 20]]
accuracy =  0.8865979381443299
```

We can visualize the decision boundaries as we did for the naive Bayes' method (page 150) using the code for **visualize_classifier_contours**. We can use the identical function code. The corresponding function call is:

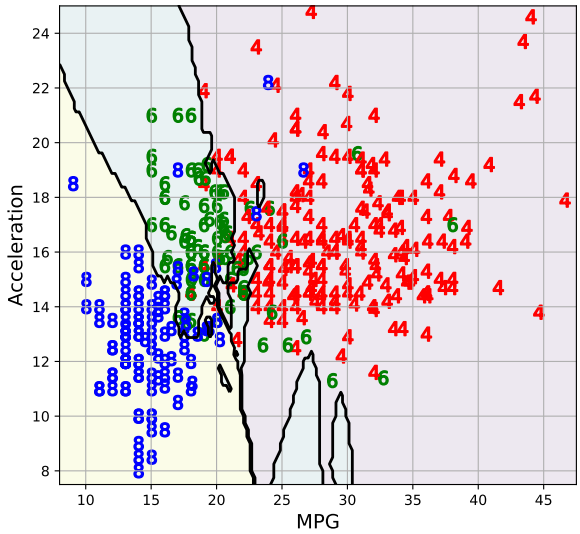
```
colors=["red", "green","blue"]
markers=["$4$","$6$","$8$"]
visualize_clasifier_contours(KNN,X,Y,.1,.1, [0.5,1.5],colors,
    markers,"MPG","Acceleration")
```

The result is shown in figure 14.1.

To determine whether a different number of near-near neighbors might have worked better, we can redo the train/test split. Here for each K value from 2 through 15 we'll do 25 train/test splits and generate a KNN classifier. Then

[1] See the sklearn web page at `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html` for documentation.

Figure 14.1: Decision boundary for KNN classifier using 4 near neighbors. Compare to figure 13.1 for a naive Bayes clasifier.

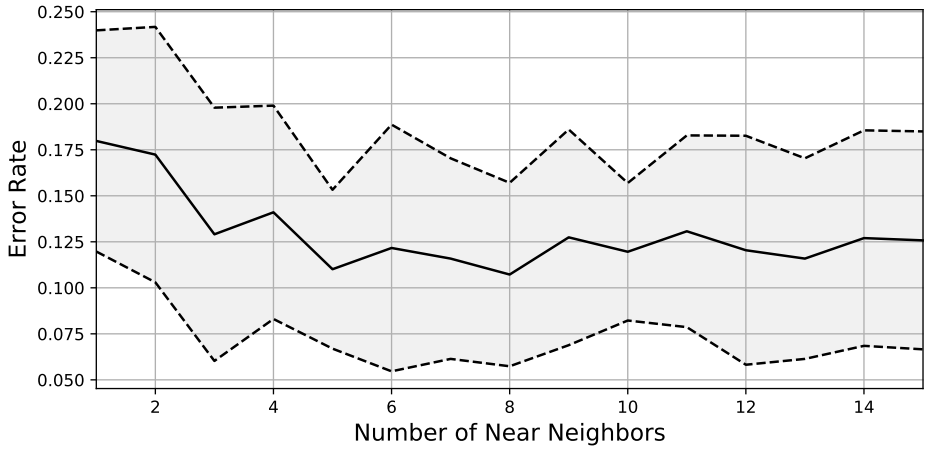


we'll figure out the error rate for each model. At the end we'll calculate the mean and standard deviation as a function of K (figure 14.2).

```
errs=[]
splits=25
for neighbors in range(1blue,16):
    KNN = KNeighborsClassifier(n_neighbors=neighbors)
    errors=[]
    for split in range(splits):
        XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
        KNN.fit(XTRAIN, YTRAIN)
        YP=KNN.predict(XTEST)
        err=1-accuracy_score(YTEST,YP)
        errors.append(err)
    errs.append(errors)
means=np.mean(errs,axis=1)
sigmas=np.std(errs,axis=1,ddof=1)
```

For this particular data set, it appears the KNN model shows significant improvement from $K = 1$ to $K = 3$, but beyond that there is very little additional improvement. There seems to be around a minimum level of about ten percent loss for all sizes.

Figure 14.2: Error rate $\pm 1.96\sigma$ as a function of the number of near neighbors used in the KNN models for the number of cylinders classification analysis.



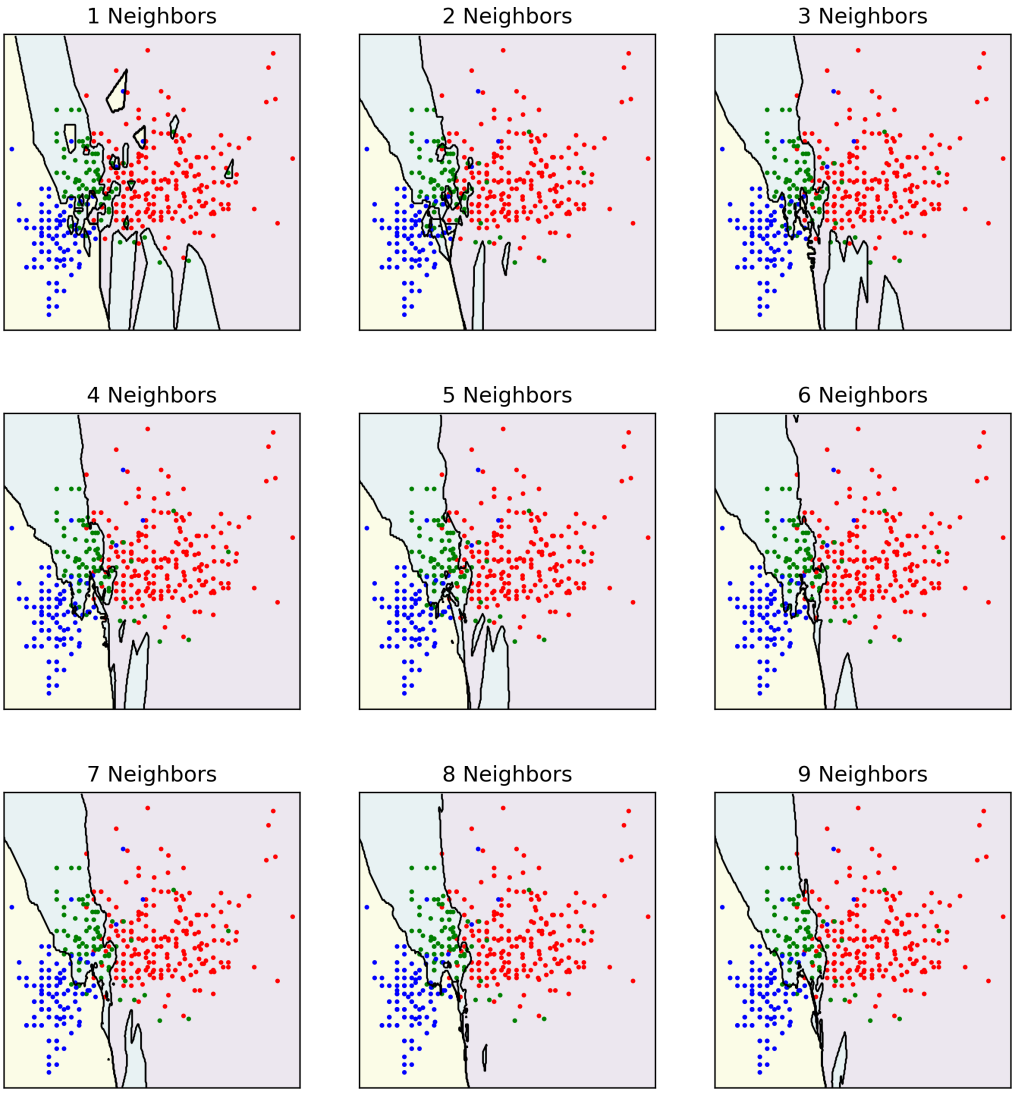
If we increase the number of quantitative features to four and the number of categories to five (3, 4, 5, 6 or 8 cylinders), as we did in chapter 13, the mean error be reduced to under about 4%. If we also include the origin using a 1-hot encoding, unlike Naive Bayes', the error is neither increased nor decreased. These results are demonstrated in the notebook for chapter 14.

For very small values of K (especially as $K \to 1$), the KNN algorithm will produce a boundary that is based on merged Voronoi cells (see page 220 and figure 21.1). The Voronoi cell $V(P_j)$ of a point P_j in a set $S = \{P_0, P_1, \dots\}$ is the region in the plane that is closer to P_j than any other point in S)

$$V(P_j) = \{P \in \mathbb{R}^2 | (d(P, P_j) < d(P, P_k)) \text{ for all } P_k \in S, P_k \neq P_j\} \tag{14.8}$$

This can be especially seen from isolated cells in the $K = 1$ frame in figure 14.3.

Figure 14.3: KNN classification boundaries for number of cylinders using different numbers of neighbors on the auto-mpg data, based on MPG (x-axis) and acceleration (y-axis). In the limit of a single near neighbors, some of isolated data points are classified by their Voronoi cells.



Chapter 15

Discriminant Analysis

Linear and quadratic discriminant analyses are refinements on Bayesian classification (page 146).[1] To understand them we should define some standard notations. Suppose we have a response variable y that can fall into any of K classes . Then

- The **prior probabilities** $p(c_0), p(c_0), \ldots, p(c_{K-1})$ are some initial first guesses of the probability that an observation will fall into each of the K classes.
- The **probability density function** $P(\mathbf{x}|c_j)$ denotes the probability density of a variable x that is known to be drawn from class j.

Then by Bayes' theorem, the probability that an observation with explanatory variable x falls into class j is given by

$$P(c_j|\mathbf{x}) = \frac{p(c_j)P(\mathbf{x}|c_j)}{\sum_{i=1}^{K} p(c_i)P(\mathbf{x}|c_i)} \tag{15.1}$$

The goal is then to find appropriate estimates for the $p(c_j)$ and $P(\mathbf{x}|c_j)$ so that we can calculate $P(c_j|\mathbf{x})$. Given $P(c_j|\mathbf{x})$ and an observation x, the class assignment is the value of j that maximizes $P(c_j|\mathbf{x})$.

In **Linear Discriminant Analysis** (LDA), we make the following assumptions:

[1]For more details see James, G. et al (2013) *Introduction to Statistical Learning.* Springer. Pseudocode for LDA is given in Zaki, MJ and Meira, W (2014) *Data Mining and Analysis*, Cambridge. A general review is Tharwat, A. (2016) *Linear vs. quadratic discriminant analysis classifier: a tutorial.* International Journal of Applied Pattern Recognition. **3**:145-10.

1. The data are distribution normally;
2. The explanatory variable x has the same standard deviation σ for all the classes;
3. If there is more than predictor (x, feature), then the explanatory variable has the same covariance matrix $\mathbf{\Sigma}$ for all the classes.

In **Quadratic Discriminant Analysis** (QDA) we relax assumption 3. In the following paragraphs, we discuss the mathematics of LDA. For the extension to QDA the reader should consult a more advanced textbook.[2]

When there is a single predictor (explanatory variable), then under these assumptions we can write

$$P(x|c_j) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu_j)^2/(2\sigma^2)} \tag{15.2}$$

If there are $d > 1$ features, we combine them into a vector $\mathbf{x}$ and write

$$P(\mathbf{x}|c_j) = \frac{1}{\sqrt{(2\pi)^d \det \mathbf{\Sigma}}} e^{-(1/2)(\mathbf{x}-\boldsymbol{\mu}_j)^{\mathrm{T}}\mathbf{\Sigma}(\mathbf{x}-\boldsymbol{\mu}_j)} \tag{15.3}$$

where $\boldsymbol{\mu}_j$ is a vector of the feature means for class j.

The math is simpler if there is only a single feature. In this case, $\mathbf{x}$ is a scalar; we substitute the Gaussian expression for $P(x|c_j)$ into the class probability $P(c_j|x)$ to obtain

$$P(c_j|x) = \frac{\dfrac{p(c_j)}{\sqrt{2\pi}\sigma} e^{-(x-\mu_j)^2/(2\sigma^2)}}{\sum_{i=1}^{K} \dfrac{p(c_i)}{\sqrt{2\pi}\sigma} e^{-(x-\mu_i)^2/(2\sigma^2)}} \tag{15.4}$$

We want to find the class j that maximizes $P(c_j|x)$. Since the natural logarithm is a monotonically increasing function of its argument, $P(c_j|x)$ is maximized when $\ln P(c_j|x)$ is maximized. But

$$\ln P(c_j|x) = \ln p(c_j) - \frac{(x-\mu_j)^2}{2\sigma^2} + \text{terms that do not depend on } j \tag{15.5}$$

$$= \ln p(c_j) + \frac{\mu_j}{\sigma^2} - \frac{\mu_j^2}{2\sigma^2} + \text{terms that do not depend on } j \tag{15.6}$$

The first terms in this equation are called the **discriminant** $\delta_j(x)$,

$$\delta_j(x) = \ln p_j - \frac{(x-\mu_j)^2}{2\sigma^2} \tag{15.7}$$

[2]Such as the ones by Hastie,T *et al* (2009) *The Elements of Statistical Learning*, 2nd Ed. Springer; or James, G, *et al*, *op cit.*

The corresponding discriminant for multifactorial problems is

$$\delta_j(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_j - \frac{1}{2}\boldsymbol{\mu}_j^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_j + \ln p(c_j) \tag{15.8}$$

In either case (scalar or vector $\mathbf{x}$), when the discriminant is maximized, the probability $P_j(x)$ is maximized. The general procedure is to make a first guess for μ and When the discriminant is maximized, the probability $P_j(x)$ is maximized.σ. The means are calculated as the class average from the training data, and σ is a weighted average of the standard deviations. If there are n training points and K classes then (for a single factor),

$$\sigma^2 = \frac{1}{n-K}\sum_{k=1}^{K}\sum_{x_j \in C(k)}(x_j - \mu_k)^2$$

where $C(k)$ is the set of all x training values for class k. These values are substituted into the formula for δ_k, and the class with the largest discriminant is chosen.[3]

LDA in Python

We use the three-class automobile data set that classifies cars based on the number of cylinders, as we did in chapters 13 and 14. We'll assume that we've already read the file and created the **X** and **Y** arrays and the test and training sets as described on page 147. Linear discriminant analysis can be performed by **Linear_Discriminant_Analysis** in **sklearn.discriminant_analysis**.[4]

```
from sklearn.discriminant_analysis import
    LinearDiscriminantAnalysis
errs=[]
nsplits=100
for j in range(nsplits):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    LDA = LinearDiscriminantAnalysis()
    LDA.fit(XTRAIN,YTRAIN)
    YP=LDA.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("LDA mean error=%7.5f std=%7.5f"
    %(np.mean(errs),np.std(errs)))
```

[3]Multifactorial formulas are given in Hastie,T, *et al*, *op cit.*

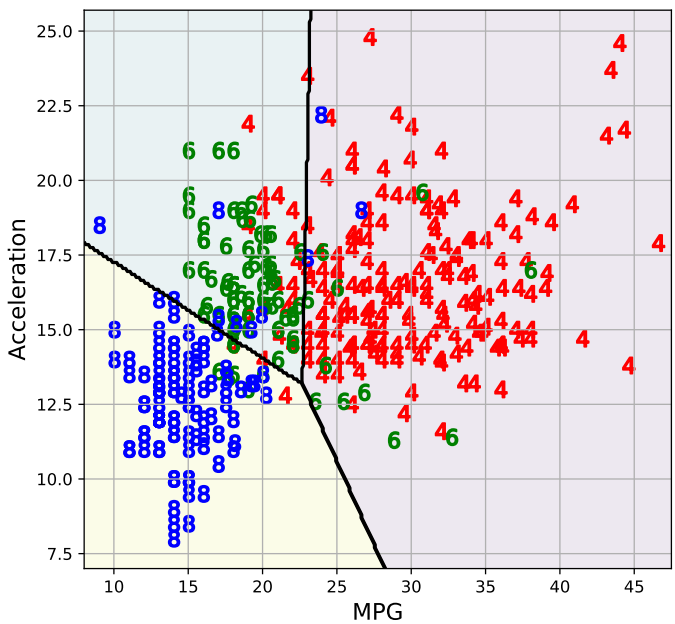
[4]For more details see the complete documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html`.

```
LDA mean error=0.05199 std=0.02215
```

We can plot the decision boundaries with **visualize_classifier_contours** (page 150). The result is shown in figure 15.1.

```
colors=["red", "green","blue"]
markers=["$4$","$6$","$8$"]
visualize_classifier_contours(LDA,X,Y,.1,.1, [0.5,1.5],colors,
    markers,"MPG","Acceleration",
    filename="LDA-MPG-decision-boundaries.pdf")
```

Figure 15.1: Decision boundaries for classes determined by number of cylinders using Linear Discriminant Analysis (LDA). Note that the decision boundaries are linear, compared to QDA, where they are quadratic arcs (figure 15.2). Compare also with figures 13.1 and 14.1



.

QDA in Python

We use the three-class automobile data set that classifies cars based on the number of cylinders, as we did above for LDA and in chapters 13 and 14. We'll assume that we've already read the file and created the **X** and **Y** arrays and the test and training sets as described on page 147. Quadratic discriminant analysis can be performed by **Quadratic_Discriminant_Analysis** in **sklearn.discriminant_analysis**.[5]

```
from sklearn.discriminant_analysis import
    QuadraticDiscriminantAnalysis
errs=[]
for j in range(nsplits):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    model = QDA()
    model.fit(XTRAIN,YTRAIN)
    YP=model.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("QDA mean error=%7.5f std=%7.5f"
    %(np.mean(errs),np.std(errs)))
```

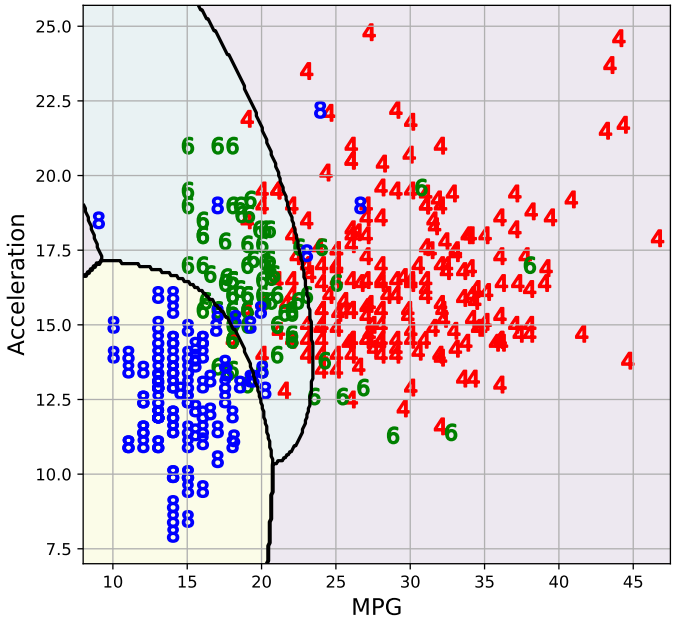
```
QDA mean error=0.13835 std=0.02989
```

We visualize the decision boundaries using **visualize_classifier_contours** as we did above. The resulting decision boundaries are plotted in figure 15.2.

```
visualize_classifier_contours(QDA,X,Y,.1,.1,
    [0.5,1.5],colors,markers,"MPG","Acceleration",
    filename="QDA-MPG-decision-boundaries.pdf")
```

[5] See https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis.html for the full documentation.

Figure 15.2: Decision boundaries for classes determined by number of cylinders using Quadratic Discriminant Analysis (LDA). Note that the decision boundaries are quadratic arcs, compared to LDA (figure 15.1), where they are linear. Compare also with figures 13.1 and 14.1



.

Chapter 16

Principal Component Analysis

PCA is not, in and of itself, a method of data classification. It is a tool that can be used for visualization or dimension reduction. To visualize the concept, think of the flight of an (American) football. The dynamics of the spinning, oblong shaped ball are quite complicated as it twists and turns in space. Now, attach a set of coordinates to the ball. The x axis is fixed along the main tip-to-tip axis of the football. Cut an imaginary plane perpendicular to the x axis that is halfway between the endpoints. Place the y and z axes in this plane. Where the yz planes intersects the football we see a circle around the origin. If we affix these axes to the football for all time, the dynamics is simple because nothing is moving. These are the principal axes or principal component axes (PCA) of the football.

PCA is similar to but different from ICA (page 155). In independent component analysis, we found the components that were independent in the probabilistic sense. That meant that their joint probability could be found by multiplying the probabilities of the individual components. PCA is a coordinate transformation. The coordinates are obtained in a ranked order to account for the variance in the distribution. The first component accounts for the most variance, and so forth. The principal components are, in general, not probabilistically independent.

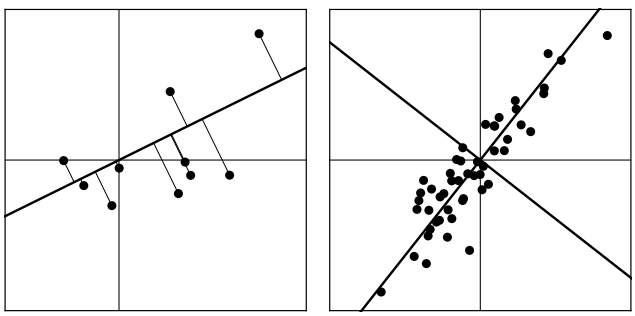
The idea is to find the principal components of a large cloud of data in n dimensional space. If we can somehow order these axes so that most of the data, like the football, is sort of symmetric or spread out, along the first (or first several) axis (axes), then we may be able to ignore the projections of the data into the other dimensions. This is helpful for two reasons. First, it will allow us to more easily visualize the data – we can project into planes consisting of pairs of principal component axes. Secondly, if it is really true that only a very small

percentage of the data falls into some of the dimensions, we can actually ignore those dimensions, and only study the remaining dimensions.

PCA is thus often used in combination with other techniques:

- To understand the primary directions of the spread of the data;
- To remove possibly non-useful information and simplify the data set before applying another method such as random forests;
- To visualize the data spread after clustering or other categorization techniques have been performed by projecting onto planes consisting of pairs of principal components.

Figure 16.1: The Principal Components of a data set. Left: Minimizing the distance to a line. Right: the Principal axes.



Principal Components

Suppose we have a collection of m data points $(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})$. In higher dimensions we represent each data point by a vector $\mathbf{x}$ that we call the **feature vector**. The individual components of each feature vector are called its features. When we are dealing with two dimensional data, the features are the x and y coordinates of each data point. Note that these feature vectors correspond the rows in the $\mathbf{X}$ data arrays that we have used in our examples so far in this text.

In linear regression (chapter 1) we found a line $y = f(x) = a + bx$ that minimized the sum of the vertical distances between the points and the line. This is sometimes called the regression of y on x. This is only meaningful if all of the noise (error) is in the y-direction. If the noise is only in the x direction, we

can switch coordinates and find what statisticians call the regression of x on y rather than the regression of y on x. But if there is noise along both axes what we really want is to find the line that minimizes the *perpendicular* distances to the points.

It is convenient to refer to the collection of all feature vectors as a **feature matrix**. The feature matrix corresponds to the arrays $\mathbf{x}$ that we have used in the examples so far in this text. Each row of the feature matrix contains a single feature vector. Thus if our data consists of a collection of data pairs (which we can label by their (x, y) coordinates), the feature matrix might look like

$$\mathbf{X}_0 = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_{m-1} & y_{m-1} \end{bmatrix} \tag{16.1}$$

The subscript 0 on $\mathbf{X}_0$ indicates that this is the raw data. More formally suppose that we have a k-dimensional data set with m vectors $\mathbf{x}_0, \ldots, \mathbf{x}_{m-1}$, that we write as row vectors

$$\mathbf{x}_j = \left[x_{j,0}, x_{j,1}, \ldots, x_{j,k-1}\right], \; j = 0, \ldots, m-1 \tag{16.2}$$

Then we define the **raw feature matrix** $\mathbf{X}_0$ as a matrix whose rows are composed of the data points:

$$\mathbf{X}_0 = \begin{bmatrix} \mathbf{x}_0 \\ \hline \vdots \\ \hline \mathbf{x}_{m-1} \end{bmatrix} \tag{16.3}$$

We also want **zero-center** the data, that is, transform it to a new coordinate system so that the average values in each axis are at the origin. If

$$\mu_x = \frac{1}{m}\sum_{j=0}^{m-1} x_j \text{ and } \mu_y = \frac{1}{m}\sum_{j=0}^{m-1} y_m$$

are the column means, then the zero centered matrix $\mathbf{X}$ (for planar data) is

$$\mathbf{X} = \begin{bmatrix} x_0 - \mu_x & y_0 - \mu_y \\ x_1 - \mu_x & y_1 - \mu_y \\ \vdots & \vdots \\ x_{m-1} - \mu_x & y_{m-1} - \mu_y \end{bmatrix} \tag{16.4}$$

The **scatter matrix** $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ occurs frequently in statistical analysis. For two dimensional data, it is computed as

$$\mathbf{X}^{\mathrm{T}}\mathbf{X} = \begin{bmatrix} x_0 - \mu_x & y_0 - \mu_y \\ x_1 - \mu_x & y_1 - \mu_y \\ \vdots & \vdots \\ x_{m-1} - \mu_x & y_{m-1} - \mu_y \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} x_0 - \mu_x & y_0 - \mu_y \\ x_1 - \mu_x & y_1 - \mu_y \\ \vdots & \vdots \\ x_{m-1} - \mu_x & y_{m-1} - \mu_y \end{bmatrix} \tag{16.5}$$

$$= \begin{bmatrix} x_0 - \mu_x & \cdots & x_{m-1} - \mu_x \\ y_0 - \mu_y & \cdots & y_{m-1} - \mu_y \end{bmatrix} \begin{bmatrix} x_0 - \mu_x & y_0 - \mu_y \\ x_1 - \mu_x & y_1 - \mu_y \\ \vdots & \vdots \\ x_{m-1} - \mu_x & y_{m-1} - \mu_y \end{bmatrix} \tag{16.6}$$

$$= \begin{bmatrix} \sum_{j=0}^{m-1} (x_j - \mu_x)^2 & \sum_{j=0}^{m-1} (x_j - \mu_x)(y_j - \mu_y) \\ \sum_{j=0}^{m-1} (x_j - \mu_x)(y_j - \mu_y) & \sum_{j=0}^{m-1} (y_j - \mu_y)^2 \end{bmatrix} \tag{16.7}$$

$$= (m-1)\mathbf{cov}(\mathbf{X}) \tag{16.8}$$

where $\mathbf{cov}(\mathbf{X})$ is the covariance matrix of the data set that is represented by $\mathbf{X}$. Note that this proportionality only works if the data set is centered. Formally, suppose $\mathbf{X}$ is a centered feature matrix. The eigenvectors of the scatter matrix tell us something about the directions of extent of the data, that is, the directions in space along which the data is spread. The directions along which these vectors point are called the **principal directions** of the centered data set. The **principal components** of a data set described by its feature matrix $\mathbf{X}$ are the projections $\mathbf{X}$ onto its principal directions.

In particular, once you know the principal directions $\mathbf{v}_i$ of a centered data set $\mathbf{X}$, then the principal components of that data set are the projections $\mathbf{X}\mathbf{v}_i$. If the $\mathbf{v}_i$ are the column vectors of a matrix $\mathbf{V}$, then we compute $\mathbf{X}\mathbf{V}$.

If we list the eigenvalues in order from largest to smallest, then sometimes there will be a few eigenvalues that are significantly larger than the others. These are considered **dominant eigenvalues**. Their corresponding eigenvectors are then the **dominant eigenvectors**. Suppose that there are p dominant eigenvectors and k features in $\mathbf{x}$. If $k > p$, we can project $\mathbf{X}$ along the p dominant principal components and ignore remaining $k - p$ features, thereby effectively reducing the dimensionality of the problem.

Theorem 16.1. Principal Component Calculation Theorem. *Let* $\mathbf{X}$ *represent a zero centered data set with singular value decomposition* $\mathbf{X}= \mathbf{U}\Sigma\mathbf{V}^T$. *Then the principal directions of* $\mathbf{X}$ *are the column vectors of* $\mathbf{V}$.

We can expand the scatter matrix in terms of its principal components, as described by Theorem 1. To do this, we expand the scatter matrix $\mathbf{S}=\mathbf{X}^{\mathrm{T}}\mathbf{X}$ using its singular value decomposition,

$$\mathbf{S} = \mathbf{X}^{\mathrm{T}}\mathbf{X} = (\mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}})^{\mathrm{T}}(\mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}) \tag{16.9}$$

Since the transpose of a product is the product of the transposes in the reverse order,

$$\mathbf{S} = \mathbf{V}\Sigma\mathbf{U}^{\mathrm{T}}\mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}} \tag{16.10}$$

Since $\mathbf{U}$ is is orthogonal, then $\mathbf{U}^{\mathrm{T}}\mathbf{U}=\mathbf{I}$ (the identity matrix), and thus

$$\mathbf{S} = \mathbf{V}\Sigma^2\mathbf{V}^{\mathrm{T}} \tag{16.11}$$

Right multiplying by $\mathbf{V}$,

$$\mathbf{S}\mathbf{V} = \mathbf{V}\Sigma^2\mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{V}\Sigma^2\mathbf{I} = \mathbf{V}\Sigma^2 \tag{16.12}$$

because $\mathbf{V}$ is orthogonal. Since Σ is diagonal, so is Σ^2. If s_i, $i = 0, 1, \ldots, k-1$ are the numbers on the diagonal of Σ, then $s_0^2, s_1^2, \ldots, s_{k-1}^2$ are on the diagonal of Σ^2.

Let $\mathbf{v}_0$, $\mathbf{v}_1$,..., $\mathbf{v}_{k-1}$ be the column vectors of $\mathbf{V}$. Then $\mathbf{S}\mathbf{V} = \mathbf{V}\Sigma^2$ becomes

$$\mathbf{S}\left[\begin{array}{c|c|c|c} \mathbf{v}_0 & \mathbf{v}_1 & \cdots & \mathbf{v}_{k-1} \end{array}\right] = \left[\begin{array}{c|c|c|c} \mathbf{v}_0 & \mathbf{v}_1 & \cdots & \mathbf{v}_{k-1} \end{array}\right] \begin{bmatrix} s_0^2 & 0 & \cdots & 0 \\ 0 & s_1^2 & & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & s_{k-1}^2 \end{bmatrix} \tag{16.13}$$

$$= \left[\begin{array}{c|c|c|c} s_0^2\mathbf{v}_0 & s_1^2\mathbf{v}_1 & \cdots & s_{k-1}^2\mathbf{v}_{k-1} \end{array}\right] \tag{16.14}$$

Equating the columns,

$$\mathbf{S}\mathbf{v}_j = s_j^2\mathbf{v}_j,\ j = 0, 1, \ldots, k-1 \tag{16.15}$$

Thus the s_j^2 are the eigenvalues of $\mathbf{S}$ and the $\mathbf{v}_j$ are the corresponding eigenvectors. It is customary to order these so that $s_0^2 \geqslant s_1^2 \geqslant s_2^2 \geqslant \cdots \geqslant s_{k-1}^2$.

The algorithm for finding the principal components of $\mathbf{X}$ is as follows.

1. Center $\mathbf{X}$.
2. Compute the matrices $\mathbf{U}$, $\mathbf{V}$ and Σ where the singular value decomposition of $\mathbf{X}$ is $\mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}$.
3. The principal components are the column vectors of $\mathbf{V}$.

PCA in Python: Gaussian Cloud

We'll demonstrate PCA first for a simple two dimensional Gaussian cloud. We use **numpy.random.multivariate_normal** to produce an oval-shaped cloud of normally distributed points oriented along a diagonal axis and centered off-origin. The function **multivariate_normal** has to be given a mean value and a covariance matrix. The mean value is an array that gives the coordinates of the center; we'll use the point (5,10). The covariance matrix must be symmetric, but to ensure that the cloud is not circular, the two diagonal elements must be non-zero. We'll set the diagonal elements to 3 and 1.7, and the off-diagonal elements to 2. The third argument to **multivariate_normal** is the number of random points that we want to generate. We will request 500 points.

```
from numpy.random import multivariate_normal as MVN
mu=np.array([5,10])
sigma=np.array([[3,2],[2, 1.7]])
X=MVN(mu, sigma, 500)
```

The principal components can be found using the class **PCA** in the scikit-learn library **sklearn.decomposition**.[1]

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)
pca.fit(X)
for component, fraction in zip(pca.components_,
   pca.explained_variance_ratio_):
    print(round(100*fraction,5),"percent:", component)
```

```
94.59073 percent: [0.8068687  0.59073082]
5.40927 percent: [-0.59073082  0.8068687 ]
```

When we fit the method we need to specify how many principal components we want to find (with the parameter **n_components**). We can find as many components as there are dimensions (features) in the data, although usually we are interested in reducing the dimensionality and will often only look at the first few components. The returned attribute **components_** gives an array of principal components, and the attribute **explained_variance_ratio** returns an array of the fraction of the variance explained by each principal component. The principal components are returned in order of largest fraction (corresponding to largest principal value) to smallest, so that if you only ask for, say, two, you will get the two most significant principal components. The variances tell you how much these components contribute to the total variance. For example, in

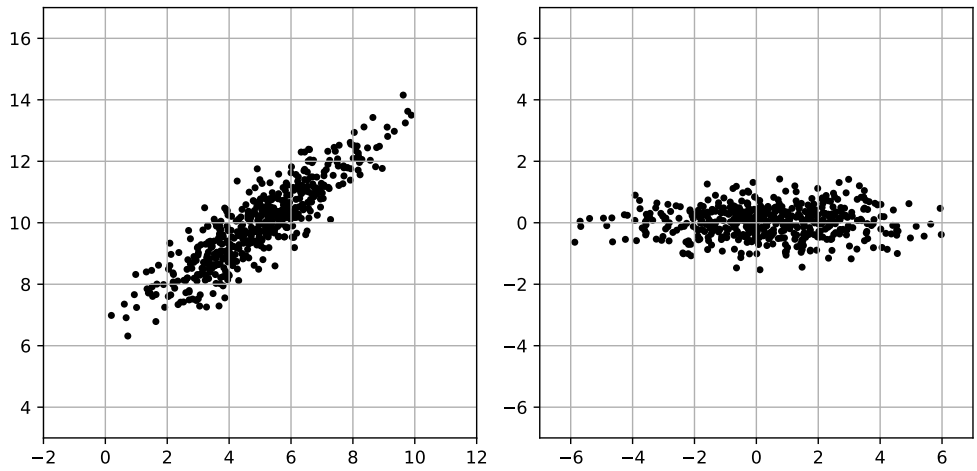
[1] See `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html` for full documentation.

our random cloud, approximately 94.6% of the variance is explained by the first principal component.

After we have fit the data, we can transform the data into the coordinate system given by the principal coordinates using the method **`transform`**. The original and transformed cloud are plotted in figure 16.2

```
P=pca.transform(X)
```

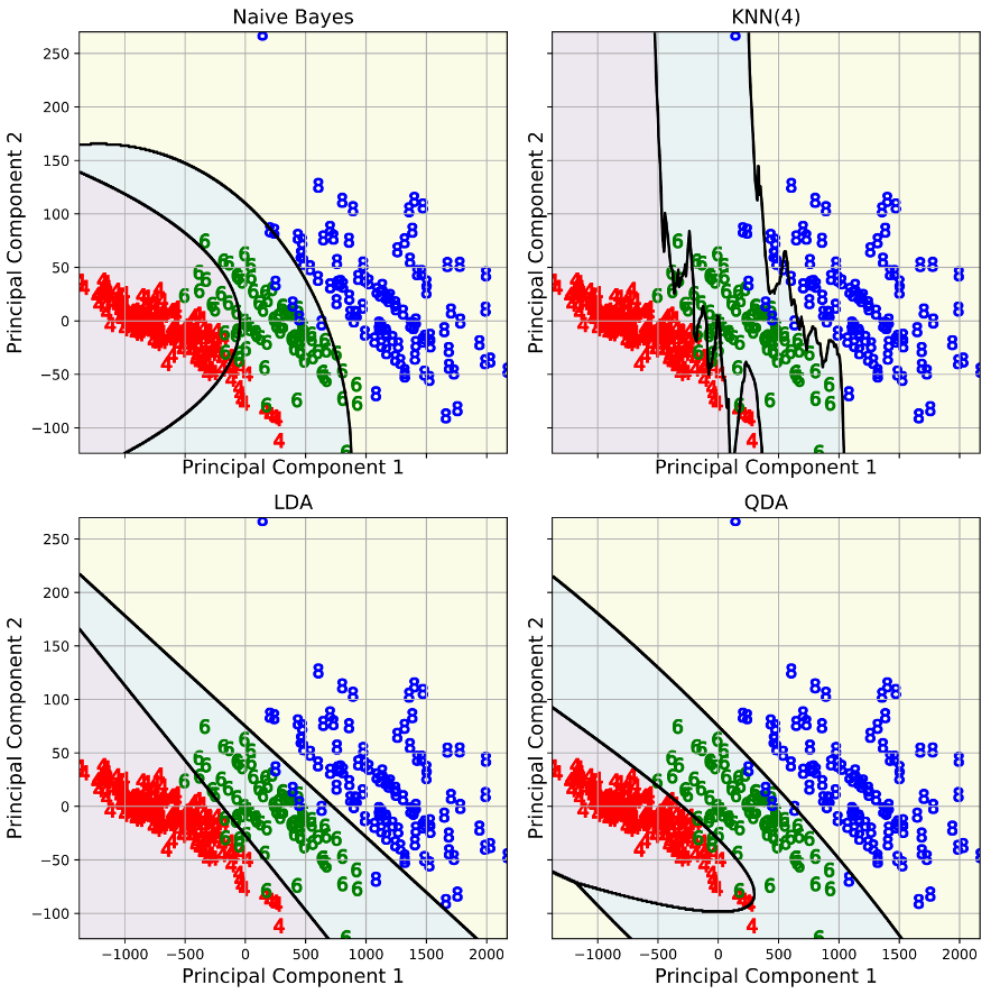
Figure 16.2: Left: A Gaussian cloud of 500 points centered at the point (5,10) with covariance matrix $\begin{bmatrix} 3 & 2 \\ 2 & 1.7 \end{bmatrix}$. Right: The same data is plotted along its two principal components. The first principal component is oriented horizontally, and the second component is oriented vertically. In the transformed coordinate system, the cloud is now centered at the origin.



PCA in Python: Automobile Classification

In chapters 13,14, and 15 we classified automobiles as having either 4, 6, or 8 cylinder engines based on an arbitrary projection of the five quantitative features **`mpg`**, **`displ`**, **`hp`**, **`weight`**, and **`accel`** onto a single two dimensional plane defined by **`accel`** and **`mpg`**. As the scatter matrix plot (figure 5.3) shows, some of these variables appear to be highly correlated.

Figure 16.3: Comparison of decision boundaries for the automobile engine cylinder classification problem in principal component space for Naive Bayes', KNN, LDA, and QDA methods. Compare to figures 13.1, 14.1, 15.1, and 15.2.



Here we read the full data file as before and remove records that contain missing data. Then we extract the five columns mentioned in the previous paragraph as our **X** data. For the class labels **Y** we use the number of cylinders (identified as a 0, 1, or 2 for 4, 6 or 8 cylinders) as we did in chapters 13,14, and 15.

```
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
   machine-learning-databases/auto-mpg/auto-mpg.data",
   header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
   "model","origin","carname")
data = data.dropna(axis=0)
cars=np.array(data[["cyl","mpg","displ","hp","weight","accel"]])
cars=np.array([line for line in cars if line[0] in [4,6,8]])
Y=cars[:,0]/2-2
X=cars[:,1:]
```

The correlation between the columns is given by **np.corrcoef** with the parameter **rowvar=False** (by default, the correlation between the rows is found). The **round** function can be used to make the output more readable. It works the same way as Python's usual **round** function, however, it can be applied to a full **array** with a single call. Here we round to 3 digits to the right of decimal place.

```
import numpy as np
np.corrcoef(X,rowvar=False)
np.round(np.corrcoef(X,rowvar=False),3)
```

```
array([[ 1.   , -0.818, -0.78 , -0.843,  0.419],
       [-0.818,  1.   ,  0.903,  0.935, -0.561],
       [-0.78 ,  0.903,  1.   ,  0.869, -0.692],
       [-0.843,  0.935,  0.869,  1.   , -0.431],
       [ 0.419, -0.561, -0.692, -0.431,  1.   ]])
```

As expected, two pairs of the columns have correlations above 0.9 in magnitude ((**weight**,**displ**), .935; and (**hp**,**displ**), .903, and three other pairs have correlations above 0.8 in magnitude ((**weight**,**hp**), .869; (**weight**, **mpg**), -.843; (**displ**,**mpg**), -.818). Over 99 percent of the variance is explained by the first principal component.

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)
pca.fit(X)
comps=pca.components_
explain=pca.explained_variance_ratio_
for comp, frac in zip(comps,explain):
    print(round(100*frac,5),"percent:", comp)
```

```
99.76511 percent: [-0.0076786   0.11404841  0.03920751
   0.99267056 -0.00138875]
```

```
0.19845 percent: [-0.01797915  0.94308579  0.30724002
  -0.1206763  -0.03614883]
```

We can transform the **X** feature matrix into its principal component frame, then repeat our classification from chapters 13,14, and 15 on the transformed data.

```
P=pca.transform(X)
gnb=GaussianNB()
KNN=KNeighborsClassifier(n_neighbors=4)
LDA = LinearDiscriminantAnalysis()
QDA=QuadraticDiscriminantAnalysis()
XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(P,Y)
gnb.fit(XTRAIN,YTRAIN)
KNN.fit(XTRAIN,YTRAIN)
LDA.fit(XTRAIN,YTRAIN)
QDA.fit(XTRAIN,YTRAIN)
YPGNB=gnb.predict(XTEST)
YPKNN=KNN.predict(XTEST)
YPLDA=LDA.predict(XTEST)
YPQDA=QDA.predict(XTEST)
print("accuracy (GNB) = ",accuracy_score(YTEST,YPGNB))
print("accuracy (KNN) = ",accuracy_score(YTEST,YPKNN))
print("accuracy (LDA) = ",accuracy_score(YTEST,YPLDA))
print("accuracy (QDA) = ",accuracy_score(YTEST,YPQDA))
```

```
accuracy (GNB) =  0.9175257731958762
accuracy (KNN) =  0.9381443298969072
accuracy (LDA) =  0.979381443298969
accuracy (QDA) =  0.9690721649484536
```

The accuracy levels are much higher than we obtained for the arbitrary projection onto a single coordinate plane. The decision boundaries for these four classifiers are illustrated in figure 16.3.

Principal Components and Image Compression

The SVD of an $n \times m$ matrix $\mathbf{M}$ can be written as (theorem B.13)

$$\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}} = s_0\mathbf{u}_0\mathbf{v}_0^{\mathrm{T}} + s_1\mathbf{u}_1\mathbf{v}_1^{\mathrm{T}} + \cdots + s_{n-1}\mathbf{u}_0\mathbf{v}_{n-1}^{\mathrm{T}} \tag{16.16}$$

where the $\mathbf{u}_j$ and $\mathbf{v}_j$ are the column vectors of the orthonormal matrices $\mathbf{U}$ and $\mathbf{V}$, and Σ is a diagonal matrix of singular values $s_0, s_1, \ldots$. We order the indices so that that the singular values are arranged largest to smallest in this expansion. The vectors $\mathbf{v}_j$ are the principal components as defined previously. If the leading singular values are significantly larger than the trailing singular

values it is sometimes possible to truncate the expansion 16.16 without any significant loss of information.

Consider, in particular, the case where **M** represents an pixel by pixel representation of an image. We can read the color **jpg** image in figure 16.4 and convert it into a grayscale array representation using the **PIL.Image** library.

Figure 16.4: A 600 × 400 color image of the front of the Ching Young Village shopping center in Hanalei, Kauai, Hawaii. (Photo by the author 5/27/17.)

```
import numpy as np
from PIL import Image
g=Image.open("BUILDING.JPG")
G=g.convert("L") # convert to grayscale
A=np.asarray(G)  # convert to numpy array
```

The truncated sum is found by adding up the first **j** outer products in **numpy**. The singular values, which fall off rapidly during the first 50 indices, are plotted in figure 16.5. **np.linalg.svd**,

```
U,Sigma,V=np.linalg.svd(A)
UT=U.T
def PC(n):
    M=np.zeros_like(A).astype(float)
    for j in range(n):
        B=np.outer(UT[j], V[j])*Sigma[j]
        M+=B
    return(M)
```

Thus, to display an image with the truncation after 50 components,

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(PC(50), cmap="gray")
```

Results for various truncations are shown in figure 16.6.

Figure 16.5: The relative variance ratio of each term in the expansion is related to the value of the singular value. This shows the singlar value as a function of position on diagonal for the image in figure 16.4. The singular values are arranged from largest to smallest. The variance is the square of the singular value and the relative variance ratio is s_i^2/s_0^2.

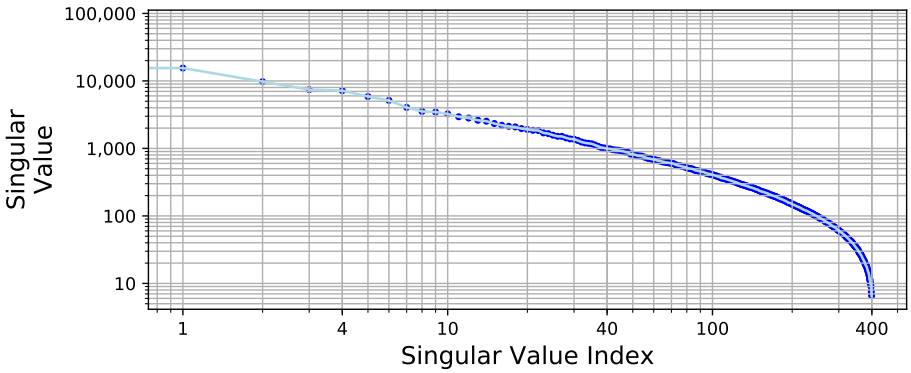
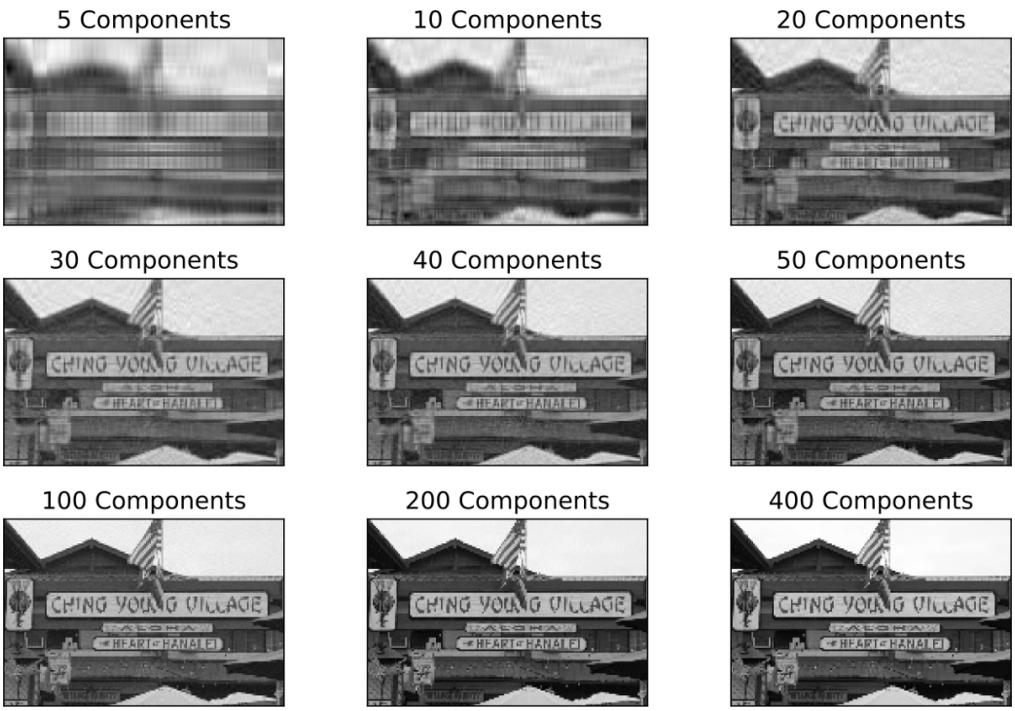


Figure 16.6: Truncations of the SVD after 5, 10, 20, 30, 40, 50, 100, 200, or all 400 terms in eq. 16.16.



The Curse of Dimensionality: Why Do We Care?

Intuitively one might think that the more information we feed into a problem the easier it will be to make a decision. As we have seen, however, sometimes adding more data – in the form of added features – increases the error rate. One reason of this this known as the **curse of dimensionality**.

Here's a simple way to look at the problem. Suppose you have N points with two features. Say you scale each dimension to a unit-cube and you divide each axis into 10 subdivisions. Then in two dimensions, your N points represent the data in 100 sub-squares of the unit square. If you add a third feature, you have 10^3 or 1000 blocks, but still N points. As your dimensions increase, you still have N points spread out over 10^d dimensions. The number of points per block, or the data density, decreases, as you increase the dimension. This is only one aspect of the problem. There are several counter-intuitive aspects of high-dimensional data sets.

It is easier to do calculations using a unit sphere than a unit cube. The volume of a sphere S_d of radius r in d dimensions[2] is

$$V(S_d) = \frac{\pi^{d/2} r^d}{\Gamma(1 + d/2)} \tag{16.17}$$

where the **gamma function** is defined as

$$\Gamma(x) = \int_0^\infty z^{x-1} e^{-z} dz \tag{16.18}$$

The gamma function has the following useful properties: $\Gamma(n) = (n-1)!$ when n is a positive integer; $\Gamma(1) = 1$; $\Gamma(1/2) = \sqrt{\pi}$; $n\Gamma(n) = \Gamma(n+1)$.

Consider the volume of a small spherical shell of thickness ϵ. Then

$$\frac{V_{shell}}{V(S_d)} = \frac{Kr^d - K(r-\epsilon)^d}{Kr^d} \tag{16.19}$$

where $K = \pi^{d/2}/\Gamma(1 + d/2)$ is a constant for any given d.

$$\frac{V_{shell}}{V(S_d)} = \frac{r^d - (r-\epsilon)^d}{r^d} = 1 - \left(1 - \frac{\epsilon}{r}\right)^d \tag{16.20}$$

For large dimensions,

$$\lim_{d\to\infty} \frac{V_{shell}}{V(S_d)} = \lim_{d\to\infty} \left[1 - \left(1 - \frac{\epsilon}{r}\right)^d\right] = 1 \tag{16.21}$$

[2]For a derivation see Zaki, MJ and Meira, W (2014) *Data Mining and Analysis*, pages 175-179.

Nearly all the volume is concentrated on the surface. For hypercubes or hyperrectangles (like data sets) **nearly all the data will be concentrated on the edges and corners of the data set**.

Now consider a d-dimensional unit hypercube with corners at

$$\overbrace{(\pm 1, \pm 1, \cdots, \pm 1)}^{\text{length d}} \tag{16.22}$$

Define the unit vector $\mathbf{e}_i$ along the ith axis as

$$\mathbf{e}_i = \underbrace{(\overbrace{0, 0, \ldots, 0}^{\text{all 0's}}, 1, \overbrace{0, \ldots, 0, 0}^{\text{all 0's}})}_{\text{length d}} \tag{16.23}$$

where the only 1 is in the ith position. Define the d-dimension vector $\mathbf{1}$ as

$$\mathbf{1} = \overbrace{(1, 1, \ldots, 1)}^{\text{all 1's}} \tag{16.24}$$

The vector $\mathbf{1}$ is the main diagonal of the hypercube. The angle θ_i between any coordinate axis $\mathbf{e}_i$ and the main diagonal is found by the dot product:

$$\mathbf{e}_i \cdot \mathbf{1} = \|\mathbf{e}_i\| \|\mathbf{1}\| \cos \theta_i \tag{16.25}$$

Since $\mathbf{e}_i \cdot \mathbf{1} = 1$, $\|\mathbf{e}_i\|$ and $\|\mathbf{1}\| = \sqrt{d}$, this gives $\cos \theta_i = \dfrac{1}{\sqrt{d}}$ As the number of dimensions becomes very large,

$$\lim_{d \to \infty} \cos \theta_i = \lim_{d \to \infty} \frac{1}{\sqrt{d}} = 0 \implies \lim_{d \to \infty} \theta_i = \frac{\pi}{2} \tag{16.26}$$

Thus all of the diagonals become perpendicular to all of the coordinate vectors. **The number of dimensions appears to double!** An unfortunate consequence of this is that PCA projections onto lower dimensions can obfuscate clusters along the diagonals.

A third counter-intuitive result is that **the probability density of a Gaussian becomes concentrated at distances $1/\sqrt{d}$ from the center** as the dimension becomes large.[3] This is in contrast to what we normally expect from the Gaussian where everything is concentrated in the center.

[3]See Blum A, Hopcroft J, Kannan R (2018) *Foundations of Data Science* (draft) `https://www.cs.cornell.edu/jeh/book.pdf`, accessed 23 March 2019.

Chapter 17

Support Vector Machines

In the general classification problem we have a collection of n points $\mathbf{x}_i$, $i = 0, \ldots, n-1$ in d-dimensional space, and we want to group them into K different classes. In a *linear separation* problem, we would find planes that separate the classes. Consider the two dimensional case, where we usually write each point in a plane as a pair (x, y). Since we are not treating y as anything special, we can write the coordinates of a single point sa $\mathbf{x} = (x_0, x_1)$, with the obvious generalization to d dimensional space of $\mathbf{x} = (x_0, \ldots, x_{d-1})$. A line normally written as

$$y = a + bx \tag{17.1}$$

then becomes by a change of variables $x \to x_0$ and $y \to x_1$

$$x_1 = a + w_0 x_0, \tag{17.2}$$

where we have written the slope as $w_0 = b$. The d-dimensional generalization is then

$$x_{d-1} = a + w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_{d-2} x_{d-2}. \tag{17.3}$$

Upon rearrangement,

$$0 = a + w_0 x_0 + w_1 x_1 \tag{17.4}$$

in the plane, where $w_1 = -1$ is a fixed constant; and

$$0 = a + w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_{d-1} x_{d-1} \tag{17.5}$$

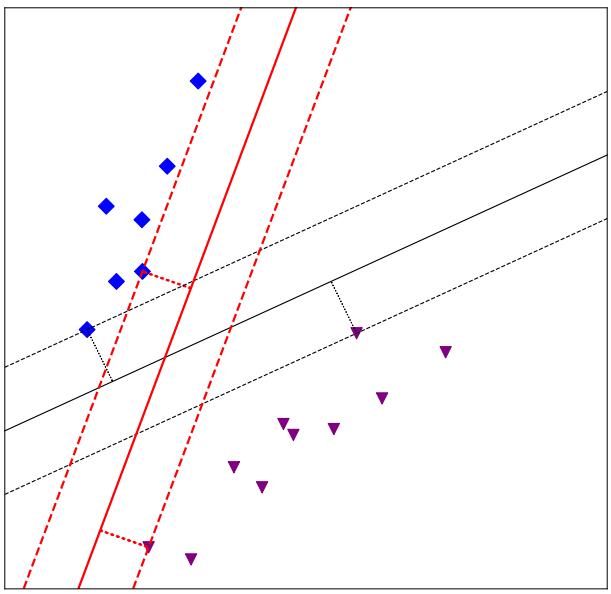
in d dimensions, where $w_{d-1} = -1$ is a fixed constant. This d-dimensional generalization of the line is called a **hyperplane in d dimensions**. If we define the **weight vector**

$$\mathbf{w} = (w_0, w_1, w_2, \ldots, w_{d-1})^{\mathrm{T}} \tag{17.6}$$

where $d = 2$ for the usual 2-dimensional line, and $d > 2$ for a hyperplane in more than two dimensions then **in either case**,

$$0 = a + \mathbf{w}^{\mathrm{T}}\mathbf{x} \tag{17.7}$$

Figure 17.1: Support vectors for two different lines that linearly separate the same pair of classes. The short line segments are parallel to the weight vector $\mathbf{w}$ for each separation. The ribbon separating the classes is drawn in such a way that the passes down its center and the length $\eta\mathbf{w}$ is equal to one half of its width, where η is chosen to minimize the distance among all points in each class.



If the line $y = a + bx$, which we have rewritten as $a + \mathbf{w}^{\mathrm{T}}\mathbf{x} = 0$ separates the two classes, then the points in either class can be identified because

$$a + \mathbf{w}^{\mathrm{T}}\mathbf{x} \begin{cases} > 0 & \text{points in class 1} \\ = 0 & \text{points on the line} \\ < 0 & \text{points in class 2} \end{cases} \tag{17.8}$$

The vector $\mathbf{w}$ is perpendicular to the line $a + \mathbf{w}^{\mathrm{T}}\mathbf{x} = 0$ (figure 17.1). To see this, consider the two dimensional case (line $y = a + bx$). Then by equation 17.4

$$\mathbf{w} = (w_0, w_1) = (b, -1) \tag{17.9}$$

Similarly, a vector $\mathbf{v}$ parallel $y = a + bx$ would have a change in y equal to the slope b for every change in x of 1, i.e.,

$$\mathbf{v} = (1, b) \tag{17.10}$$

Since

$$\mathbf{v} \cdot \mathbf{w} = 0 \tag{17.11}$$

the vectors are perpendicular.

Suppose the line $y = a + bx$ is drawn so that it separates two classes, i.e., one class is completely on one side of the line and the other class is on the other side. Next, consider sweeping the vector $\eta\mathbf{w}$ (where η is a real number) along the line $y = a + bx$. At some place along the line, $\eta\mathbf{w}$ will point directly towards a data point. The smallest distance

$$M = \min_j |\eta\mathbf{c}| \tag{17.12}$$

where the minimum is taken over all the points in either class is called the **support**. The corresponding vector $\mathbf{x}$ in feature space is called the **support vector**. It turns out[1] that weights for the optimal line are a linear combination of the support vectors $\mathbf{x}_j$ so that

$$\mathbf{w} = \sum_j c_j \mathbf{x}_j \tag{17.13}$$

The linear decision function 17.8 for a given vector $\mathbf{x}'$ then

$$a + \mathbf{w}^{\mathrm{T}}\mathbf{x}' = a + \sum_j c_j \mathbf{x}_j^{\mathrm{T}}\mathbf{x}' \tag{17.14}$$

where the sum is over the support vectors. An classifier that makes a decision based on finding the optimal support in this way is called a **support vector machine**. The decision function 17.14 still leads to a linear separation, but it can be extended to non-linear class boundaries by making a kernel separation. In a kernel substitution, the dot product $\mathbf{x}_j^{\mathrm{T}}\mathbf{x}'$ is replaced by a nonlinear function $K(\mathbf{x}_j, \mathbf{x}')$.[2]

SVM in Python

Here we apply SVM to the car classification problem studied in chapters 13, 14 and 15. Automobiles from the UC, Irvine **`auto-mpg`** data set are categorized as having either 4, 6, or 8 cylinders (other vehicles are removed from the analysis). We can read the data file directly with **`read_fwf`**. Then we will remove any rows that have missing data. Since the original data file does not have any column labels, we will need to add the column labels in manually.

[1] See, for example, Cortes, C and Vapnik, V (1995) "Support Vector Networks." *Machine Learning.* **20**:273-297.

[2] See, for example, chapter 21 of Zaki, MJ and Meira, W (2014) *Data Mining and Analysis.* Cambridge for details.

```
import pandas as pd
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
   machine-learning-databases/auto-mpg/auto-mpg.data",
   header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
   "model","origin","carname")
data = data.dropna(axis=0)
```

We select the quantitative features (except for cylinders) and place them into our feature matrix **X**, after removing any cars that do not fit into one of the desired categories of 4, 6 or 8 cylinders. Then we scale the number of cylinders to a value of 0, 1, or 2 to represent the class values of 4, 6 or 8 cylinders, respectively, by dividing by 2 and then subtracting 2. These become our **Y** values.

```
import numpy as np
cars=np.array(data[["cyl","mpg","displ","hp","weight","accel"]])
cars=np.array([line for line in cars if line[0] in [4,6,8]])
Y=cars[:,0]/2-2
X=cars[:,1:]
```

Next we convert the feature matrix into its principal components frame as we did on page 179 for this same data set. Our transformed feature matrix in the new coordinate system is **P**.

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)
pca.fit(X)
P=pca.transform(X)
```

The SVM implementation is very sensitive to scaling an should be rescaled to a unit interval, so we will use **MinMaxScaler**. Then we will do a train/test split on the rescaled principal components.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
scaler = MinMaxScaler()
Q=scaler.fit_transform(P)
```

Our final version of the feature matrix is now in **Q**. SVM support is provided by class **SVC** in **sklearn.svm**.[3]

```
from sklearn.svm import SVC
```

[3]See `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html` for full documentation.

The **sklearn.svm** implementation includes several options for kernels, such as **kernel="linear"**, **kernel="rbf"** (default), **kernel="poly"** (polynomial), and **kernel="sigmoid"**. The linear kernel is the usual dot product,

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}' \tag{17.15}$$

The polynomial kernel is

$$K(\mathbf{x}, \mathbf{x}') = (\gamma\mathbf{x}^{\mathrm{T}}\mathbf{x}' + C)^n \tag{17.16}$$

where C is specified by the option **coef_=0**, γ by the option **gamma=1**, and n by the option **degree=3**. The radial basis function kernel is

$$K(\mathbf{x}, \mathbf{x}') = e^{-\gamma|\mathbf{x}-\mathbf{x}'|^2} \tag{17.17}$$

and the sigmoid kernel is given by

$$K(\mathbf{x}, \mathbf{x}') = \tanh(\gamma\mathbf{x}^{\mathrm{T}}\mathbf{x}' + C) \tag{17.18}$$

To compare the kernels, we'll write a simple function in Python that will do 100 train-test splits on a model

```
def svm_evaluate(classifier, X, Y, N):
    errs=[]
    for j in range(N):
        XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
        classifier.fit(XTRAIN,YTRAIN)
        YP=classifier.predict(XTEST)
        errs.append(1-accuracy_score(YTEST,YP))
    return(np.mean(errs),np.std(errs))
```

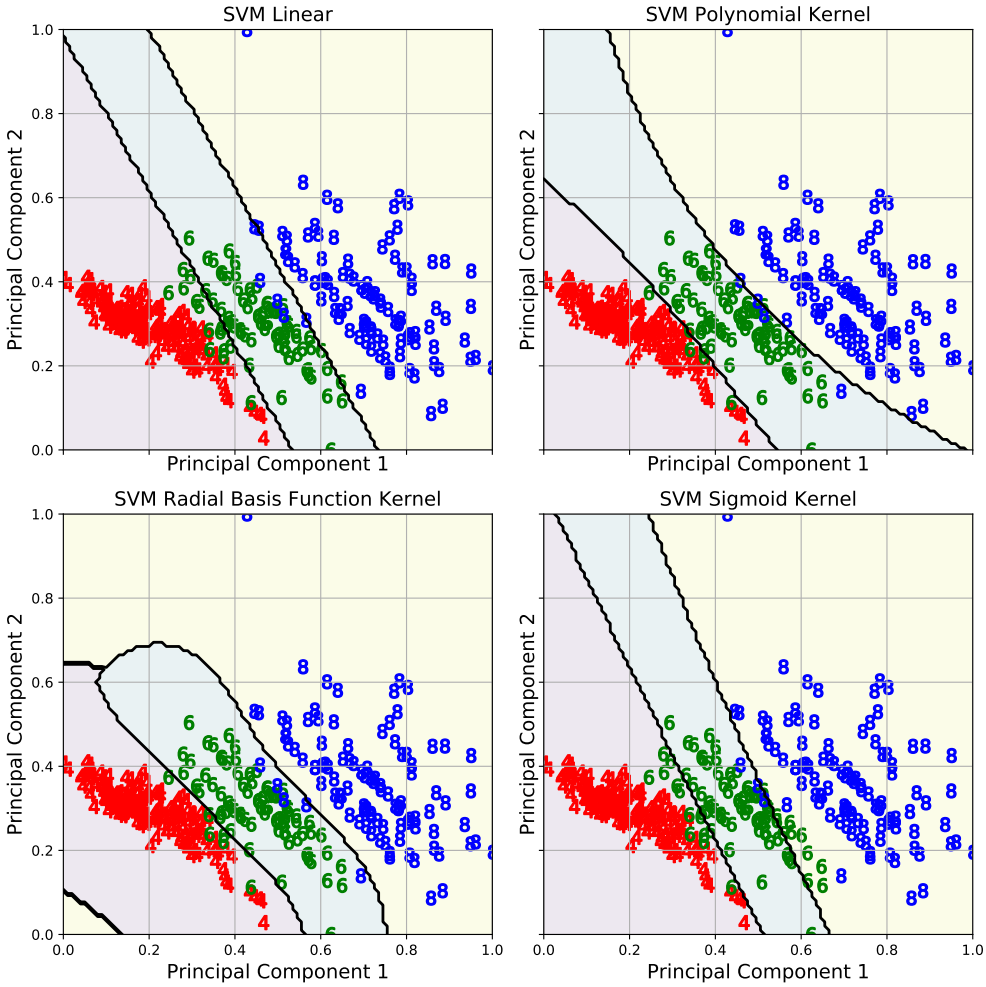
Then we'll run through each of the kernel's as follows, using the scaled and transformed data as input.

```
print("Linear Kernel     mean error=%7.5f sd=%7.5f" \
      %svm_evaluate(SVC(kernel="linear",gamma=10), Q, Y,100))
print("Polynomial Kernel mean error=%7.5f sd=%7.5f" \
      %svm_evaluate(SVC(kernel="poly", degree=3, gamma=10,
      coef0=3), Q, Y,100))
print("RBF Kernel        mean error=%7.5f sd=%7.5f" \
      %svm_evaluate(SVC(kernel="rbf",gamma=20), Q, Y,100))
print("Sigmoid Kernel    mean error=%7.5f sd=%7.5f" \
      %svm_evaluate(SVC(kernel="sigmoid", gamma=1, coef0=0),
      Q, Y,100))
```

```
Linear Kernel     mean error=0.05598 sd=0.02441
Polynomial Kernel mean error=0.03196 sd=0.01505
RBF Kernel        mean error=0.02948 sd=0.01501
Sigmoid Kernel    mean error=0.06969 sd=0.02470
```

The decision boundary for SVM's with various kernels is plotted in figure 17.2.

Figure 17.2: Decision boundary using SVM for the automobile classification problem as 4, 6 or 8 cylinders. The linear and polynomial (degree=3) classifiers use **`gamma=10`**; the radial basis function uses **`gamma=20`**; and the sigmoid uses **`gamma=1`**. All classifiers have **`coef0=0`** except for the polynomial, which sets it to 3. Compare also with figures 13.1, 14.1, 15.1, 15.2 and 16.3.



Chapter 18

Decision Trees

As with regression trees (see chapter 9), decision trees are formed by recursively partitioning the domain of the predictor variable.[1] In most implementations the tree is binary, i.e., a hierarchy of partitions is formed in which regions are recursively divided into two **sub-nodes**. The resulting tree is rarely balanced, hence the **sub-tree** from the right child may be much deeper than than the sub-tree from the left child. As the regions associated with successive partitions become smaller, they will contain fewer predictors. When an appropriate stopping condition is reached, that particular branch will cease to split and the predictors in the final node (called a **leaf**) are assigned to a class based on the target classes (the y values assigned to the x values in the data set that fall into the corresponding region). This differs from a regression tree, in which the predicted y values are assigned a valued based on the average of the target values (the actual y-values).

The decision on where to partition a region is called a **splitting criterion**. For multivariate data, each variable must be considered at each split, and a decision is made to determine the optimal split. The optimal split tells us (a) which variable to use; and (b) where to place the cut point. Many different methods can be used to make this decision. There is no best method; like everything else in machine learning, it generally depends on the data set. Here are some of the more common methods:

[1] For a more detailed survey of the mathematics see Rokach, L, and Maimon, O. (2005) "Top-Down Induction of Decision Trees. Classifiers – A Survey." *IEEE Trans. on Systems, Man, and Cybernetics – Part C.* **35**(4): 476-486.

- **Information Gain**. The **entropy** is

$$S = -\sum_i p_i \log p_i$$

where p_i is the proportion assigned to each class for a given split, the logarithm is calculated in base 2, and we define $p \log p = 0$ when $p = 0$. The corresponding information gain is obtained from the individual entropies.

$$\text{Information Gain} = -\sum_i \frac{|S_i|}{|S|} S_i$$

where the sum is over all the features.

- **Gini Impurity**. For each feature k (variable) let n_i be the number of data points that belong to class i. If there are n total data points then if we define the proportion $p_i = n_i/n$,

$$G = 1 - \sum_{i=1}^{K} p_i^2$$

where K is the number of classes.

- Total RSS - see chapter 9.

- **Chi squared**, where

$$\chi^2 = \sum_i \frac{(y_i - \hat{y}_i)^2}{\hat{y}_i}$$

A higher chi-squared value gives more information.

Different implementations use different stopping criteria, and most use some combination. The most common ones set a minimum number of points in each leaf node and a maximum tree depth.

There is a trade-off between stopping condition and tree size. If the stopping condition is two strict, the tree will be very small(i.e., not too deep) and therefore not terribly useful. If the stopping condition is not very weak, there is a high likelihood of overfitting the data. One way to address this is to use pruning methods. The idea is to overfit the data and then selectively cut back parts of the tree in a way that the fit will become "more optimal.' A variety of algorithms employ a number of pruning methods to cut back trees in this way.[2]

[2]See, for example, James, G et al (2013) *Introduction to Statistical Learning.* Springer. Pages 307-311.

Decision Trees in Python

Here we apply a decision tree to the car classification problem studied in chapters 13 through 17. Automobiles from the UC, Irvine **auto-mpg** data set are categorized as having either 4, 6, or 8 cylinders (other vehicles are removed from the analysis). We will use all of the quantitative features, as we did when we applied support vector machines to the same data set in chapter 17. However, unlike what we did with SVM's, we will classify with a decision tree without first finding the principal components. As usual, we read the data file directly with **read_fwf** and remove any rows that have missing data. The original data file does not have the columns labeled so we add them column labels in manually.

```
import pandas as pd
data=pd.read_fwf("https://archive.ics.uci.edu/ml/
   machine-learning-databases/auto-mpg/auto-mpg.data",
   header=None,na_values="?")
data.columns=("mpg","cyl","displ","hp","weight","accel",
   "model","origin","carname")
data = data.dropna(axis=0)
```

We select the quantitative features (except for cylinders) and place them into our feature matrix **X**, after removing any cars that do not fit into one of the desired categories of 4, 6 or 8 cylinders. Then we scale the number of cylinders to a value of 0, 1, or 2 to represent the class values of 4, 6 or 8 cylinders, respectively, by dividing by 2 and then subtracting 2. These become our **Y** values.

```
import numpy as np
cars=np.array(data[["cyl","mpg","displ","hp","weight","accel"]])
cars=np.array([line for line in cars if line[0] in [4,6,8]])
Y=cars[:,0]/2-2
X=cars[:,1:]
```

We create a train/test split using the default splitting (75%/25%):

```
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.metrics import accuracy_score
```

Here we compute the average error of 100 train/test splits.

```
nsplits=100; depth=3; errs=[]
for j in range(nsplits):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
    DT=tree.DecisionTreeClassifier(max_depth=depth)
    DT.fit(XTRAIN, YTRAIN)
    YP=DT.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("Decision Tree Depth = %d mean error = %7.6f SD = %7.6f"\
     %(depth,np.mean(errs),np.std(errs)))
```

```
Decision Tree Depth = 3 mean error = 0.020412 SD = 0.014651
```

We have used the **DecisionTreeClassifier**[3] from **sklearn.tree** to perform this classification task. We told it to stop splitting at a maximum depth of 3.

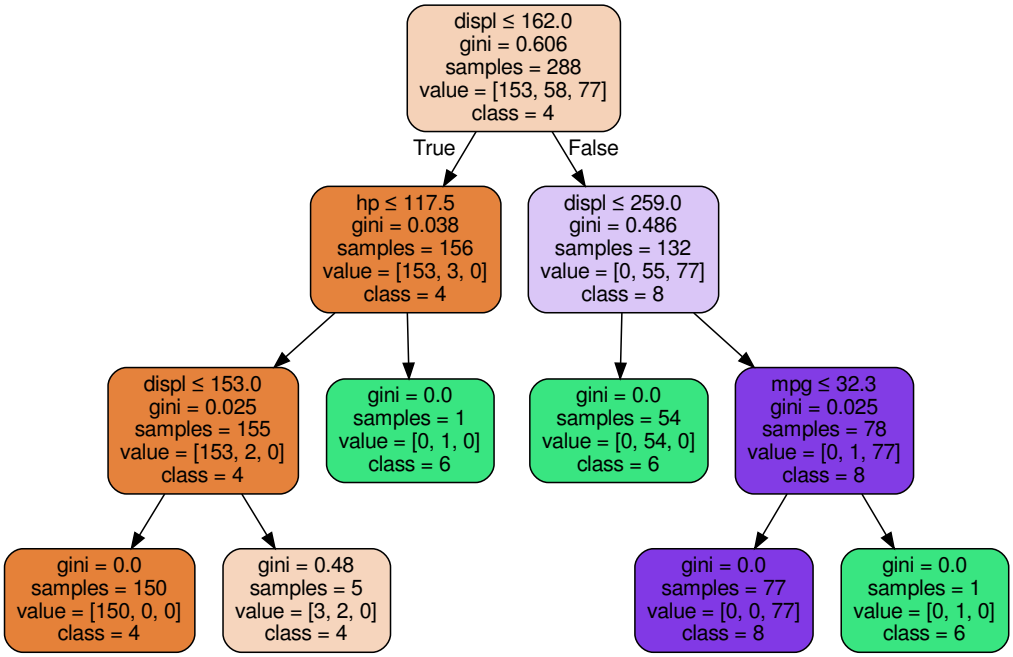
The **export_graphviz** method will export the tree data into a **graphviz** data structure, which can be either saved or displayed on the screen. We can use **pydotplus**, for example to display the files inline. The resulting tree is illustrated in figure 18.1.

```
import graphviz
import pydotplus
from IPython.display import Image
#
# generate the Graphviz code
#
dot= tree.export_graphviz(DT, out_file=None, rotate=False,
     feature_names=["mpg","displ","hp","weight","accel"],
     class_names=list(map(str,[4,6,8])),
     filled=True, rounded=True, special_characters=True)
#
# convert the graphviz to Python image data
#
g = pydotplus.graph_from_dot_data(dot)
#
# Save the file
#
g.write_pdf("My-Decision-Tree.pdf")
#
# Display the image in Jupyter Notebook
#
Image(g.create_png())
```

We could have instead chosen to convert our data to the first two principal components, transform to that frame, and scale the data. As we saw on page 179, these two components account for over 99.9% of the variance in the data. In the following code we use the default tree depth (we are not setting a value to **max_depth**). This means the tree will continue splitting nodes until the each leaf contains the minimum number of nodes required to split. Since we are not setting the minimum value required to split (**min_samples_split**), the nodes will continue to split until each leaf has two nodes. The result is somewhat less accurate. The decision boundary for a typical train/test split in the space for the first two principal components is illustrated in figure 18.2.

[3]See https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html for full documentation of the classifier.

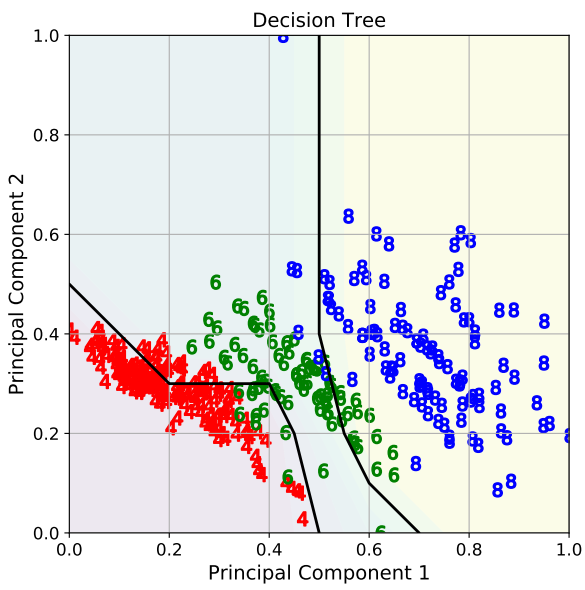
Figure 18.1: Decision tree visualization produced using **`export_graphviz`** for the automobile classification problem. This classification uses all features, not the principal components. The **`values`** line in each node lists the number of samples that remained in each class after each division. The class labels refer to the number of cylinders, 4, 6, or 8.



```
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

pca=PCA(n_components=2)
pca.fit(X)
P=pca.transform(X)
scaler = MinMaxScaler()
Q=scaler.fit_transform(P)

nsplits=100
errs=[]
for j in range(nsplits):
    XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(Q,Y)
    DT=tree.DecisionTreeClassifier()

    DT.fit(XTRAIN, YTRAIN)
    YP=DT.predict(XTEST)
    errs.append(1-accuracy_score(YTEST,YP))
print("Decision Tree (2 PC) mean err = %7.6f SD = %7.6f"\
     %(np.mean(errs),np.std(errs)))
```

```
Decision Tree (2 PC) mean err = 0.072268 SD = 0.027139
```

Figure 18.2: Decision boundary using a decision tree in principal component space (first two components) for the automobile cylinder classification problem. Compare with figures 16.3 and 17.2 for other PCA decision boundaries on the same classification problem.



Chapter 19

Tree Ensemble Methods

Ensemble methods involve creating a mixture of multiple methods to improve classification. Here we will look at three types of ensembles that are used to improve classification by decision trees.

- **Boosting**. A boosting method takes a weak classifier, such as a decision tree, and uses some technique to iteratively improve it. New classifications are determined as weighted averages of the earlier classifications.
- **Bagging**. Bagging is also known as **bootstrap aggregation**. The basic idea is to generate multiple different training sets, e.g., as subsets of the original data, and use them to build a better classifier by finding a weighted average of the individual classifiers.
- **Random Forests**. This is similar to bagging, except that each sub-classifier is based on a subset of the feature space.

While these methods *tend* to do better as the number of trees is increased this is not guaranteed. There is always the danger of overfitting the data. And they all work better with weak learning algorithms, so if a single, simple decision tree already works well, then none of these methods are likely to help very much.

We will illustrate each method with the wine classification data set, using the quality values for white wine as our class labels (see, for example, page 131). The different wines are described by their physio-chemical properties: **`'fixed acidity'`**, **`'volatile acidity'`**, **`'citric acid'`**, **`'residual sugar'`**, **`'chlorides'`**, **`'free sulfur dioxide'`**, **`'total sulfur dioxide'`**, **`'density'`**, **`'pH'`**, **`'sulphates'`** and **`'alcohol'`**. The final column in the file is **`'quality'`**, which we will use as our class labels **`Y`**. The remaining columns

will be used as the feature matrix **X**.[1] The content looks something like this (the first three rows are shown; the first column is the row number).

	fixed acid-ity	volatile acidity	citric acid	residual sugar	chlor-ides	free sulfur dioxide	total sulfur dioxide	density	pH	sufph-	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6

When we will read the white wine quality data file from the archive we need to tell **read_csv** to use semi-colons as separators, to reflect the actual data file format. (Otherwise **read_csv** will expect the fields to be separated by commas.

```
import pandas as pd
import numpy as np
white=pd.read_csv("https://archive.ics.uci.edu/ml/
    machine-learning-databases/wine-quality/
    winequality-white.csv", sep=";")
Y=np.array(white["quality"])
X=np.array(white)[:,:-1]
```

We will be comparing the following three classifiers: a **gradient boosting** classifier, a **bagging** classifier, and a **random forest** classifier. Each of these methods perform computations on ensembles of individuals classifiers. The base classifier is a decision tree. We will call each ensemble of trees a **forest**. For example, a boosting classifier may make a prediction based on a forest composed of, say, 50 or 100 decision trees. The size of the forest will sometimes affect the prediction. To evaluate this dependence, we will explore different size forests (e.g., 2 trees, 3 trees, 10 trees, 50 trees, etc). Furthermore, for each forest size, we will perform a large number n (say $n = 100$) of train/test splits, just as we have with other classifiers, and compute the error for each split. The following pseudo-code summarizes our approach:

- for forest in sizes $N = 2, 3, 5, \ldots$ trees
  - do the following n times (say $n = 100$)
    - perform a train/test split
    - calculate the classification error
  - calculate
    - μ = mean classification error of the n train/test splits
    - σ = standard deviation of error of the n train/test splits
- plot $\mu \pm 1.96\sigma$ vs N (number of trees)

[1] Cortez, P, et. al. (2009) "Modeling wine preferences by data mining from physicochemical properties." *Decision Support Systems.* **47**(4):547-553. Data available at `https://archive.ics.uci.edu/ml/datasets/wine+quality`. The discrimination from these classifiers is much lower than that of classification methods typically published for this data set because other methods such as neural networks or SVM are typically used.

The following function, **evaluate_classifier(X,Y,nruns)**, will run any classifier **nruns** times using a feature matrix **X** and class labels **Y**. For each of the **nruns** it will perform a train/test split, fit the classifier to the training data, make predictions on the test data, and calculate the fit error. The return value is a tuple pair giving the mean and standard deviation of the classification error.

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
def evaluate_classifier(clf, X, Y, nruns):
    errs=[]
    for j in range(nruns):
        XTRAIN, XTEST, YTRAIN, YTEST=train_test_split(X,Y)
        clf.fit(XTRAIN, YTRAIN)
        YP=clf.predict(XTEST)
        errs.append(1-accuracy_score(YTEST,YP))
    return(np.mean(errs), np.std(errs))
```

A tuple containing (μ, σ), the mean and standard deviation of the prediction error (calculated as one minus the accuracy score), is returned by this function.

To get a baseline comparison for the ensemble methods, we will classify the wine data using a single decision tree. First we will use look at truncated decision trees, where tree splitting is stopped after a three and five divisions, respectively.

```
from sklearn import DecisionTreeClassifier
DT=DecisionTreeClassifier(max_depth=3)
DT5=DecisionTreeClassifier(max_depth=5)
DTTmu,DTTsig=evaluate_classifier(DT,X,Y,25)
DTT5mu,DTT5sig=evaluate_classifier(DT5,X,Y,25)
print("Max Depth = 3: ", round(DTTmu,3), round(DTTsig,3))
print("Max Depth = 5: ", round(DTT5mu,3), round(DTT5sig,3))
```

```
Max Depth = 3:  0.479 0.012
Max Depth = 5:  0.471 0.013
```

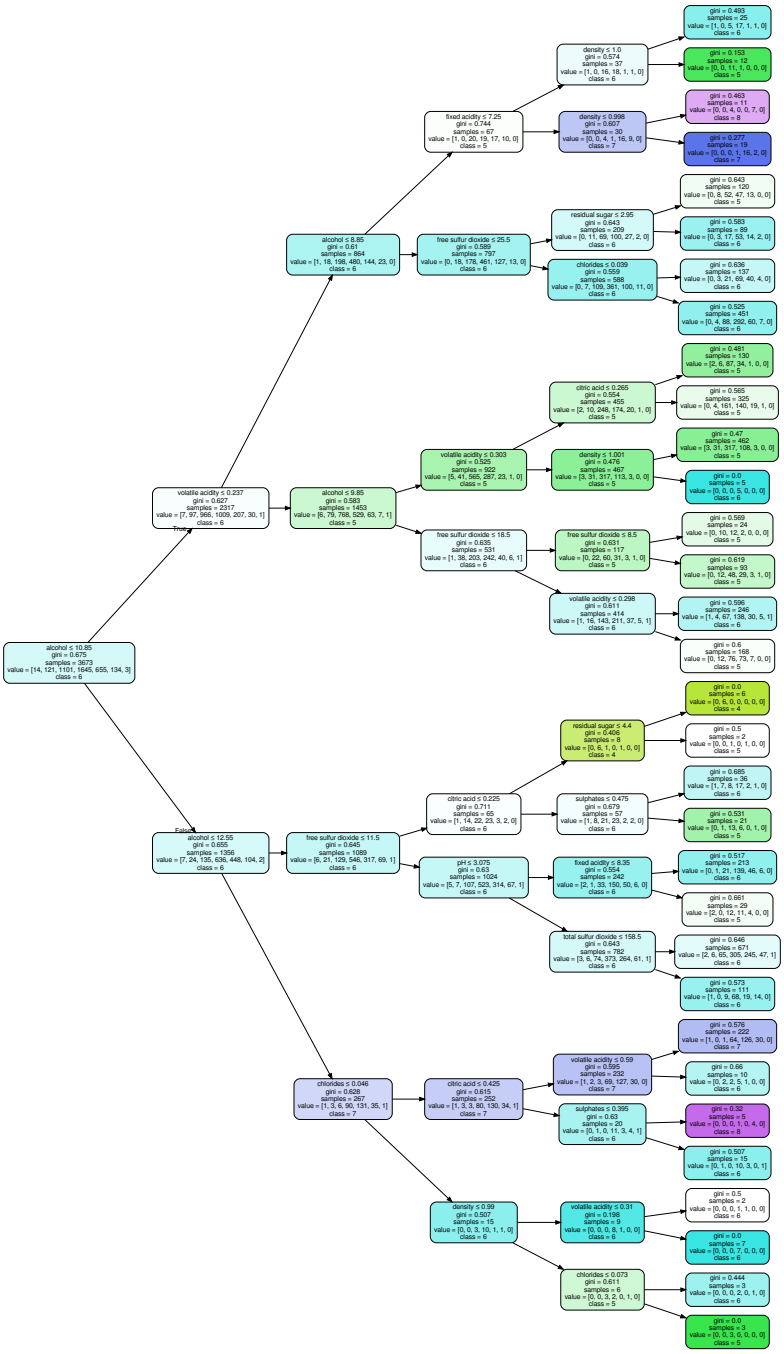
As we can see, there is not much error reduction by adding the additional two levels The second baseline will be a full decision tree, where the depth of tree splitting is not limited.

```
DT=DecisionTreeClassifier(max_depth=None)
DTFullmu, DTFullsig=evaluate_classifier(DT,X,Y,25)
print("Max Depth = No Limit: ", round(DTFullmu,3),
  round(DTFullsig,3))
```

```
Max Depth = No Limit:  0.403 0.01
```

We can visualize the single decision tree using **graphviz** (as we did on page

Figure 19.1: Decision tree with a maximum depth of five. The full decision tree can be found at `https://github.com/biomathman/machine-learning-in-python-book/blob/master/extras/white-wine-class.pdf`.



194). The code to do this is summarized below; the resulting depth-5 tree is illustrated in figure 19.1. A full tree is illustrated in the book web site.

```
import graphviz
import pydotplus
from IPython.display import Image
from sklearn.tree import export_graphviz
dot5= export_graphviz(DT5,
        out_file=None, rotate=True,
        feature_names=np.array(white.columns)[:-1].tolist(),
        class_names=list(map(str,np.unique(Y).tolist())),
        filled=True, rounded=True, special_characters=True)
g=pydotplus.graph_from_dot_data(dot5)
Image(g.create_png()) # to display to screen
g.write_pdf("mypicture.pdf") # to save file
```

A typical confusion matrix can be extracted after making a prediction on a single train/test split, e.g,

```
DT=DecisionTreeClassifier(max_depth=None)
XTRAIN,XTEST,YTRAIN,YTEST=train_test_split(X,Y)
DT.fit(XTRAIN,YTRAIN)
YP=DT.predict(XTEST)
print(confusion_matrix(YP,YTEST))
```

```
[[  0,   0,   1,   2,   0,   0,   0],
 [  0,  14,  13,  10,   6,   0,   0],
 [  1,  18, 218,  87,  14,   4,   0],
 [  1,   9, 116, 379,  57,  12,   0],
 [  0,   1,  17,  49, 129,  13,   0],
 [  0,   0,   1,  10,  16,  20,   0],
 [  0,   0,   0,   2,   4,   1,   0]]
```

The function **evaluate_classifier** tells the mean μ and standard deviation σ of the prediction error for a classifier, after it has been used, say, n times. To compare the different ensemble methods, we will plot the prediction error $\mu \pm 1.96\sigma$ as a function of forest size **ntrees**. For each given forest size **ntrees** we will do n (parameter **repeats**, below) train/test splits. To do this we will encapsulate **evaluate_classifier** in a loop over a list of forest sizes in the function **generate_forests**. The inputs to **generate_forests** include the classification function, the list of forest sizes, the number of train/test splits for each forest size, and the **X** and **Y** data arrays. The return value is a list of means and standard deviations, with one mean and standard deviations for each forest size.

```
def generate_forests(clf, X, Y, repeats, forestsizes):
    results=[]
    for ntrees in forestsizes:
        classifier=clf(n_estimators=ntrees)
        mu,sigma=evaluate_classifier(classifier,X,Y,repeats)
        results.append([mu,sigma])
        print(ntrees,mu,sigma)
    mus=np.array(results).T[0]
    sigs=np.array(results).T[1]
    return(mus,sigs)
```

Before we can run **generate_forests**, the classifier must be loaded via an **import** statement, and we need to define the list of forest sizes. Each classifier is instantiated (i.e., the Python class is initialized) within **generate_forests** so we do not need to do so before the function call, just pass in the classifier name. A tree ensemble classifier has a parameter **n_estimators** that specifiers the number of trees that should be used, and this number will be taken from **forestsizes**. The input parameter **repeats** is the number of train/test splits that is performed for each number of estimators as the input parameter **repeats** to **generate_forests**.

Boosting Algorithms

In **boosting**, a classification algorithm is iteratively improved by classifying on the residual errors. Two popular implementations are **gradient boosting** and **adaptive boosting**. In gradient boosting,[2] each model is classified by optimizing an objective function

$$\mathcal{E} = \frac{1}{2}\sum_i (y_{i,p}(\mathbf{x})_i - y_i(\mathbf{x})_i)^2 \tag{19.1}$$

where $y_{i,p}(\mathbf{x}_i)$ and $y_i(\mathbf{x}_i)$ are the predicted and exemplar values corresponding to feature vector $\mathbf{x}_i$. If $C_j(\mathbf{x}_i)$ is the classification of $\mathbf{x}_i$ by the jth model, then each new model is minimizing the residual function

$$\mathcal{R}_j(\mathbf{x}) = y(\mathbf{x}) - C_j(\mathbf{x}) \tag{19.2}$$

using gradient descent and then setting

$$C_{j+1}(\mathbf{x}_i) = C_j(\mathbf{x}_i) + \nu\mathcal{R}_j(\mathbf{x}_i) \tag{19.3}$$

[2]Friedman,J (2001) "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics.* **29**(5):1189-1232.

where ν is a constant.

In **adaptive boosting** (the most common implementation is called **adaboost**), classification trees are built and combined in a weighted algorithm where the weights decrease with time (iteration) so that successive iterations have less impact on the result.[3] We set

$$C_{j+1}(\mathbf{x}_i) = C_j(\mathbf{x}_i) + \alpha_j C_{j-1}(\mathbf{x}_i) \tag{19.4}$$

An objective function of the form

$$\mathcal{E} = \sum e^{y_i(C_{j-1}(\mathbf{x}_i) - \alpha_j C_{j-1}(\mathbf{x}_i)} \tag{19.5}$$

is minimized.[4]

Using our **generate_forests** function, we can run the gradient boost classifier in **sklearn** through 100 train/test splits (each) on forests of sizes 2, 3, 4, 5, 7, 10, 20, 40, 50, 70, 100, 300, and 500 trees with a single function call.

```
from sklearn.ensemble import GradientBoostingClassifier
forest_sizes=[2,3,4,5, 7,10,20,30,40,50,70,100,300,500]
BoostMeans, BoostSigmas=\
  generate_forests(GradientBoostingClassifier,X,Y,100,\
  forest_sizes)
```

```
2 0.47875102040816325 0.013943399597534543
3 0.47075102040816325 0.013394749292965804
4 0.4664326530612245 0.013105933683232481
5 0.46349387755102034 0.01186994917685235
7 0.4559428571428571 0.014189744112709474
10 0.4553714285714287 0.012687014353050224
20 0.44059591836734696 0.011396956271079483
30 0.4323918367346939 0.01581731609807552
40 0.42953469387755105 0.011709309901058097
50 0.42402448979591834 0.014190589416523132
70 0.4157061224489796 0.014030428050738999
100 0.4097795918367347 0.013007020178639313
300 0.3734857142857142 0.012996513164524244
500 0.36040000000000005 0.013970156804161556
```

Printing out the errors at each step is not strictly necessary but it is sometimes nice to watch what is going on. What is important, however, is that we save

[3] Y. Freund, R. Schapire (1999) "A Short Introduction to Boosting." *J. Jap. Soc. Art. Int.* **14**(5):771-780; Rojas,R. (2009) "Adaboost and the super bowl of classifiers: a tutorial introduction to adaptive boosting." Free University of Berlin, Technical Report. `http://www.inf.fu-berlin.de/inst/ag-ki/adaboost4.pdf`

[4] The exact solution and pseudocode are given in Freund,Y and Schapire,R (1997) "Generalization of On-Line Learning and an Application to Boosting." *Journal of Computer and System Sciences.* **55**(1):119-139.

the results in our two lists (e.g., **BoostMeans** and **BoostSigmas**) so that we can access the results later.

```
print(BoostMeans)
```

```
[0.47875102, 0.47075102, 0.46643265, 0.46349388, 0.45594286,
 0.45537143, 0.44059592, 0.43239184, 0.42953469, 0.42402449,
 0.41570612, 0.40977959, 0.37348571, 0.3604    ]
```

```
print(BoostSigmas)
```

```
[0.0139434 , 0.01339475, 0.01310593, 0.01186995, 0.01418974,
 0.01268701, 0.01139696, 0.01581732, 0.01170931, 0.01419059,
 0.01403043, 0.01300702, 0.01299651, 0.01397016]
```

We can run a single instance of boosting to illustrate a typical confusion matrix. The format for running the classifier is similar to that of other classification methods in **sklearn**.[5] There are **fit** and **predict** methods, for example, and a **predict_proba** method so that we could produce an ROC curve for a two-class data set (not appropriate in this example).

```
from sklearn.ensemble import GradientBoostingClassifier
GB=GradientBoostingClassifier()
GB.fit(XTRAIN,YTRAIN)
GP=GB.predict(XTEST)
confusion_matrix(YTEST,GP)
```

```
array([[  0,   2,   0,   0,   0,   0,   0],
       [  1,   6,  29,  11,   0,   0,   0],
       [  0,   3, 231, 135,   6,   2,   0],
       [  3,   1,  85, 403,  32,   2,   0],
       [  0,   0,  10, 114, 100,   2,   0],
       [  0,   0,   0,  28,   9,   6,   0],
       [  0,   0,   0,   2,   1,   1,   0]])
```

The **fit** and **predict** methods work as they do for other methods in **sklearn**.

Bagging Algorithms

Ideally we would like to build different models from different training sets, and then combine them. Often the training set is not large enough to do this, so we resort ot a trick to simulate the creation of multiple sets out of a single set. In **bagging**[6] we take multiple samples from the same training set and pretend that each sample is a different training set. Since the sampling is done with replacement, it is likely that the simulated training sets will have some overlap.

[5]For complete documentation of the gradient boosting classifier in Python see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`.

[6]Breiman, L (1996) "Bagging predictors." *Machine Learning.* **24**:123-140.

Some authors refer to bagging as **bootstrap aggregation**. A typical instance of bagging can be run as follows in Python.[7]

```
from sklearn.ensemble import BaggingClassifier
BC=BaggingClassifier()
BC.fit(XTRAIN,YTRAIN)
YP=BC.predict(XTEST)
print(1-accuracy_score(YTEST,YP))
print(confusion_matrix(YTEST,YP))
```

```
0.3453061224489796
[[  0,   0,   1,   1,   0,   0],
 [  0,  15,  20,   7,   0,   0],
 [  0,   8, 238, 117,   3,   0],
 [  0,   5,  95, 402,  35,   2],
 [  0,   2,  10,  83, 128,   3],
 [  0,   0,   0,  17,  14,  19]]
```

To explore the dependence of the bagging classifier on the number of trees we will use **generate_forests**. As with boosting, we will perform 100 train/test splits (each) on forests of sizes 2, 3, 4, 5, 7, 10, 20, 40, 50, 70, 100, 300, and 500 trees with a single function call.

```
from sklearn.ensemble import BaggingClassifier
BAGmu, BAGsig=\
  generate_forests(BaggingClassifier,X,Y,100,forest_sizes)
```

```
2 0.44858775510204085 0.0144558867383183
3 0.42013877551020407 0.01319727863104045
4 0.3969551020408163 0.014616966515689327
5 0.38026122448979593 0.01435592310209574
7 0.3678612244897959 0.013513424611524709
10 0.3537795918367347 0.012432725420883669
20 0.34132244897959174 0.011983762441976329
30 0.33408979591836735 0.012010613104889854
40 0.33131428571428573 0.011242816948816986
50 0.3295183673469388 0.011165616287904261
70 0.3303591836734694 0.012516016228180335
100 0.32861224489795915 0.011283259872606404
300 0.32581224489795924 0.01089866542082189
500 0.32310204081632654 0.01112842004111149
```

When we are doing a large number of slow runs it is useful to print out the values of the results; this way we can copy and paste them in to variables in *future* notebook runs, in case we don't want to wait and repeat the entire simulations. This could be helpful if we decide at a later time, for example, to reformat a plot for a report.

```
print(BAGmu)
```

[7]For full documentation of the bagging classifier in **sklearn** see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html`.

```
[0.44858776 0.42013878 0.3969551  0.38026122 0.36786122
 0.35377959 0.34132245 0.3340898  0.33131429 0.32951837
 0.33035918 0.32861224 0.32581224 0.32310204]
```

```
print(BAGsig)
```

```
[0.01445589 0.01319728 0.01461697 0.01435592 0.01351342
 0.01243273 0.01198376 0.01201061 0.01124282 0.01116562
 0.01251602 0.01128326 0.01089867 0.01112842]
```

Random Forests

In the **random forest** algorithm[8] an ensemble of decision trees is built based on the same data, similar to bagging. The individual trees are selected by choosing subsets of feature space. If there are n features in the data set, then each tree in the ensemble is built by randomly selecting a subset of $m \leqslant n$ features to split on. This is particularly helpful when there are a large number of features (e.g., in the hundreds). The optimal number of features used for each tree is $\sqrt{n}$. Bagging is a special case of random forests when $m = n$ is chosen. In a decision tree built using random forests, class predictions are determined by majority vote among trees.

Here is how a single instance of a random forests classifier can be run.[9]

```
from sklearn.ensemble import RandomForestClassifier
RF=RandomForestClassifier(n_estimators=500)
RF.fit(XTRAIN,YTRAIN)
YP=RF.predict(XTEST)
print(1-accuracy_score(YP,YTEST))
print(confusion_matrix)
```

```
0.3118367346938775
[[  0   0   0   0   0   0   0]
 [  0   8   0   0   0   0   0]
 [  3  22 260  76   6   0   0]
 [  2  11 115 433  87  14   1]
 [  0   1   3  31 123   8   1]
 [  0   0   0   2   1  17   0]
 [  0   0   0   0   0   0   0]]
```

[8] Ho, TK (1995) "Random Decision Forests." *Proceedings of the 3rd International Conference on Document Analysis and Recognition (IEEE).* **1**:278-282; Amit, Y, Geman, D (1997) "Shape Quantization and Recognition with Randomized Trees." *Neural Computation.* **9**:1545-1588; Breiman, L. (2001) "Random Forests." *Machine Learning.* **45**:5-32. The name *Random Forest* is a registered trademark of Leo Breiman and Andy Liaw.

[9] For full documentation of the random forest classifier in Python see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`.

We can see how important the random forest classifier has determined that each feature is by looking at the class attribute **feature_importances_**.

```
print(RF.feature_importances_)
```

```
[0.0750037  0.09881211 0.08138214 0.08818542 0.08488747
 0.09527255 0.09047286 0.10277019 0.08668786 0.0787098
 0.1178159 ]
```

The importances correspond to the columns of the feature matrix. To get an annotated list we can zip the feature importances to the column names, sort, and print. The importances sum to one.

```
names=np.array(white.columns)[:-1].tolist()
importances=RF.feature_importances_.tolist()
imp=list(zip(importances,names))
imp.sort(reverse=True)
for impt,name in imp:
     print("%20s %5.3f" %(name,impt/max(importances)))
```

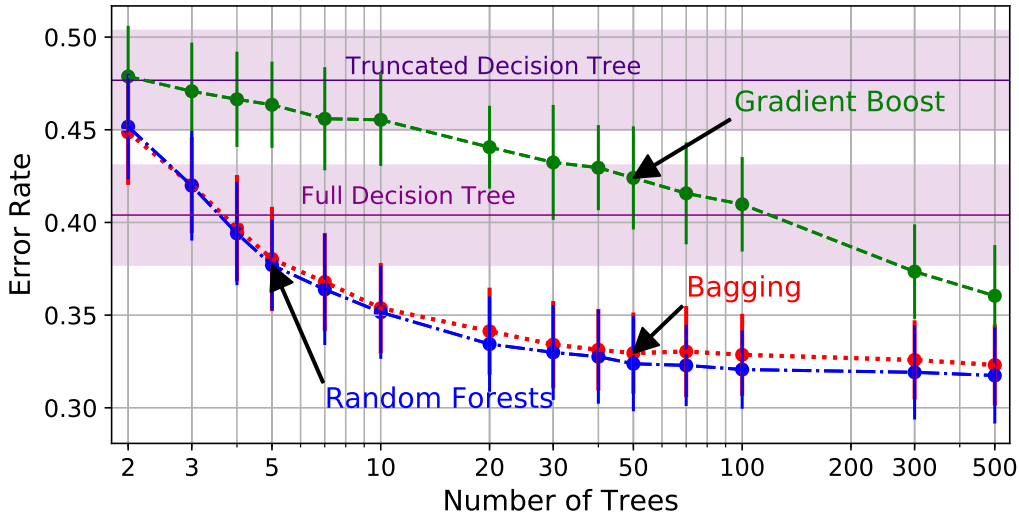
```
             alcohol 1.000
             density 0.896
    volatile acidity 0.859
 free sulfur dioxide 0.805
total sulfur dioxide 0.788
      residual sugar 0.755
                  pH 0.749
           chlorides 0.738
         citric acid 0.698
           sulphates 0.693
       fixed acidity 0.650
```

We'll explore random forests with **generate_forests** as we did for bagging and boosting, performing 100 train/test splits (each) on forests of sizes 2, 3, 4, 5, 7, 10, 20, 40, 50, 70, 100, 300, and 500 trees with a single function call.

```
from sklearn.ensemble import RandomForestClassifier
RFMeans, RFSigmas=generate_forests(RandomForestClassifier,
  X,Y,100,forest_sizes)
```

```
2 0.45168979591836733 0.014513119486674303
3 0.4197795918367347 0.015092928304275511
4 0.3940571428571429 0.01423224044397006
5 0.3770040816326531 0.012461247885965058
7 0.3638448979591837 0.01528397635960942
10 0.351526530612245 0.012808150258184626
20 0.3343755102040816 0.013125807168611453
30 0.3298857142857143 0.013076977744829866
40 0.3275591836734694 0.012925117364733224
50 0.3237632653061225 0.013077996884039956
```

Figure 19.2: Comparison of tree ensemble algorithms, error rate as a function of the size of the forest (the number of trees in the forest). The error bars show $\mu \pm 1.96\sigma$ for each of the three methods discussed in this chapter, and are calculated using the function **generate_forest**. The shaded ribbons show $\mu \pm 1.96\sigma$ as calculated using **evaluate_classifier** for a single decision tree that is either constrained to a maximum depth of 1 (truncated decision tree) or is unconstrained (full tree).




```
70 0.3228734693877551 0.011163980874726634
100 0.3206448979591836 0.010791759547718657
300 0.31915918367346935 0.012994449204209585
500 0.3174204081632653 0.013213258016927827
```

```
print(RFMeans)
```

```
[0.4516898  0.41977959 0.39405714 0.37700408 0.3638449
 0.35152653 0.33437551 0.32988571 0.32755918 0.32376327
 0.32287347 0.3206449  0.31915918 0.31742041]
```

```
print(RFSigmas)
```

```
[0.01451312 0.01509293 0.01423224 0.01246125 0.01528398
 0.01280815 0.01312581 0.01307698 0.01292512 0.013078
 0.01116398 0.01079176 0.01299445 0.01321326]
```

A plot of the prediction error as a function of forest size for random forests, bagging and boosting for this data set is shown in figure 19.2. In this case, there is not much difference between bagging and random forests; however, the relative placement of these three curves will vary with the data set.

Chapter 20

Comparing Classifiers

Chapters 10 through 19 have presented different methods of classification. Many of these techniques have variations of their own. This presents a newcomer to the subject with a bewildering plethora of choices, along with the question of where to begin. The simple fact is this: while some data sets are very easy to classify with any method,[1] other data sets can be quite challenging. Sometimes we will even get quite different results from different classifiers. We have already seen a little of that when we plotted the two dimensional decision boundaries predicted by different classifiers. Here we will work through a particular data set[2] that shows a wide variety of classification results. This data set has multiple features; we will use these features to predict into which of two classes the subjects fall.

The heart disease data[2] set, summarized in table 20.1, has 14 columns of data. We will use the 14th column, labeled **`"num"`**, as the prediction class. The remaining thirteen columns are the features, and are a mixture of numerical and categorical data. Categorical data is indicated by an asterisk in the table. The thirteen features expand to a total of 25 features when the categorical data is-represented by a 1-hot encoding (page 153). Overall, there are 297 complete

[1] For a data set that is very easy to classify and has consistently high accuracy and low error rate, regardless of the method used, see the breast cancer data set at `https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)`, University of Wisconsin Hospitals, Madison, donated by Dr. William H. Wolberg. Wolberg WH and Mangasarian OL, "Multisurface method of pattern separation for medical diagnosis applied to breast cytology" *Proceedings of the National Academy of Sciences, U.S.A.*, **87** (Dec. 1990), 9193-9196.

[2] The "processed Cleveland" data set from `https://archive.ics.uci.edu/ml/datasets/heart+Disease`; Detrano R et. al. (1989). "International application of a new probability algorithm for the diagnosis of coronary artery disease." *American Journal of Cardiology*, **64**, 304-310.

individual records in the file, after missing data has been excluded.

Read the data and remove all records with missing data.

```
DF=pd.read_csv(
"https://archive.ics.uci.edu/ml/machine-learning-databases/
    heart-disease/processed.cleveland.data",
    header=None,na_values="?")
DF=DF.dropna(axis=0)
```

For easy reference, we'll label the columns based on the names given by the author of the data set and then make a list of the ones we want to mark as categorical.

```
DF.columns=["age","sex","cp","trestbps","chol","fbs","restecg",
    "thalach","exang","oldpeak","slope","ca","thal","num"]
categorical=["sex","cp","fbs", "restecg","exang","slope",
    "thal"]
```

Here's what the first few lines of the data look like. The data wraps around because of the page width.

```
print(DF[:3])
```

```
    age  sex   cp  trestbps   chol  fbs  restecg  thalach  exang  oldpeak  slope
ca  thal  num
0  63.0  1.0  1.0     145.0  233.0  1.0      2.0    150.0    0.0      2.3    3.0
0.0   6.0    0
1  67.0  1.0  4.0     160.0  286.0  0.0      2.0    108.0    1.0      1.5    2.0
3.0   3.0    2
2  67.0  1.0  4.0     120.0  229.0  0.0      2.0    129.0    1.0      2.6    2.0
2.0   7.0    1
```

The **Y** variable is **"num"**, set to 1 if its value is nonzero, and 0 otherwise.

```
M=np.array(DF["num"])
Y=np.array([1 if x>0.5 else 0 for x in num])
```

The feature matrix **X** is built by dropping the column labeled **"num"** and keeping the rest of the data. Categorical features are encoded using a one-hot encoding. The easiest way to do this is with the function **get_dummies**.

```
X=np.array(pd.get_dummies(DF.drop("num",1),
                          columns=categorical))
print(X.shape)
X
```

```
(297,25)
array([[ 63., 145., 233., ...,   0.,   1.,   0.],
       [ 67., 160., 286., ...,   1.,   0.,   0.],
       [ 67., 120., 229., ...,   0.,   0.,   1.],
```

Table 20.1: Columns in the heart disease data set. The columns marked with asterisks(*) are treated as categorical data in the analysis; the remainder are treated as numerical data. Column 14 is the prediction class.

column	name	description
1	**age**	Age (years)
2*	**sex**	Sex, 1=male, 0 =female
3*	**cp**	Type of chest pain: 1=typical angina; 2=atypical angina; 3=non-anginal pain; 4=asymptomatic
4	**trestbps**	resting blood pressure in mm Hg on admission to hospital
5	**chol**	serum cholesterol, mg/dl
6*	**fbs**	fasting blood sugar: 1 means $>$ 120 mg/dl 0 means $\leqslant$ 120 mg/dl.
7*	**restecg**	resting electrocardiogram: 0 = normal; 1 = ST-T Wave abnormality; 2 = left ventricular abnormality
8	**thalach**	maximum elevated heart rate
9*	**exang**	exercise induced angina: 1 = yes; 2 = no
10	**oldpeak**	ST depression induced by exercise
11*	**slope**	Slope of peak of ST depression: 1 = positive; 2 = flat; 3 =negative
12	**ca**	Number of blood vessels colored by flourosopy, 0 to 3
13*	**thal**	Thalassemia: 3 = normal; 6 = fixed defect; 7 = reversible defect
14	**num**	Diagnosis: 0 = healthy nonzero = heart-disease

```
       ...,
       [ 68., 144., 193., ...,   0.,   0.,   1.],
       [ 57., 130., 131., ...,   0.,   0.,   1.],
       [ 57., 130., 236., ...,   1.,   0.,   0.]])
```

When we use **get_dummies**, if we do not supply a list of categorical variables using the **columns** option, it will look for columns that have an **object** or **category dtype** and encode them. Since some of the categories in this data set actually have integer data types we needed to indicate them specifically, or they would not be encoded.[3]

We will examine the distribution of classification errors, taking the classification error as one minus the accuracy score. The following function **evaluate** will perform a requested number of train/test splits and then return a list of all of the errors for further processing.

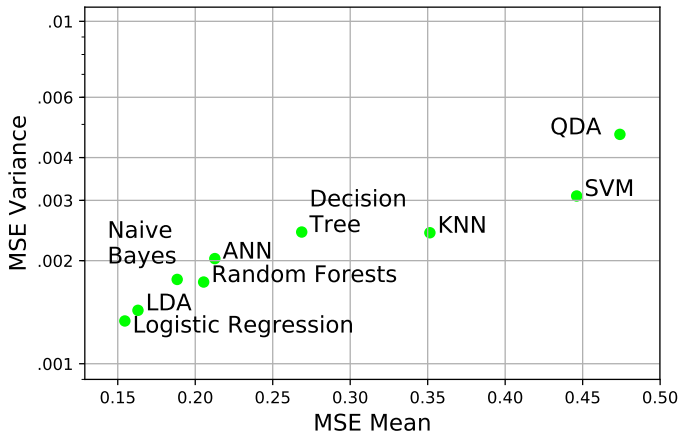
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
def evaluate(X, Y, MODEL, nsplits=100, PRINT=True):
    errs=[]
    for j in range(nsplits):
        XTRAIN,XTEST,YTRAIN,YTEST=train_test_split(X,Y)
        model=MODEL
        model.fit(XTRAIN,YTRAIN)
        YP=model.predict(XTEST)
        errs.append(1-accuracy_score(YTEST,YP))
    if PRINT:
        print("Mean error=%7.5f std=%7.5f" \
          %(np.mean(errs), np.std(errs)))
    return errs
```

We need to load all of the required methods.

```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.discriminant_analysis import \
  LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import \
  QuadraticDiscriminantAnalysis as QDA
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier as ANN
from sklearn.ensemble import RandomForestClassifier as RF
from sklearn.tree import DecisionTreeClassifier as DT
```

[3]For documentation see https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

Figure 20.1: Variance in classification error as a function classification error.



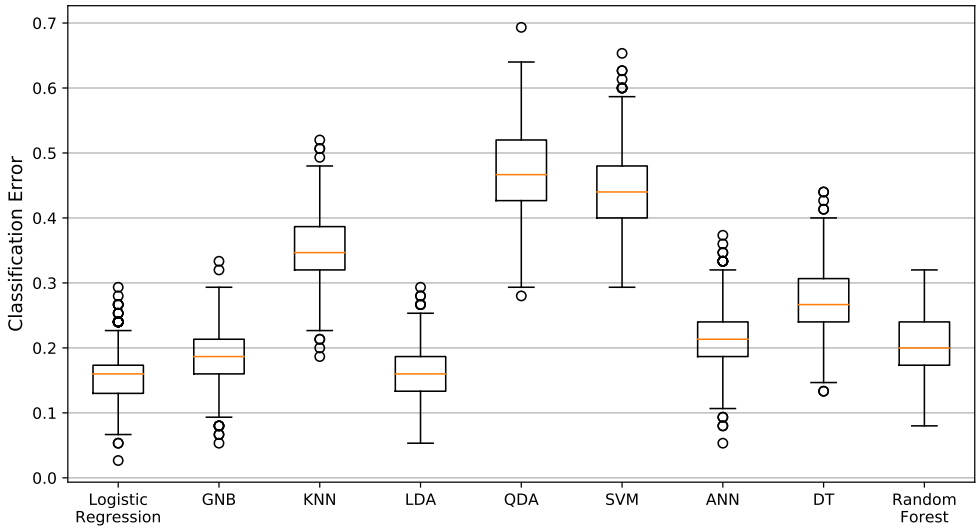
We'll do 1000 runs on each classifier. The **PRINT** option is not needed so we will turn it off; it just prints out the mean and standard deviation of the errors.

```
nsplits=1000
LRerrs=evaluate(X,Y,LR(),nsplits,PRINT=False)
NBerrs=evaluate(X,Y,GaussianNB(), nsplits, PRINT=False)
KNNerrs=evaluate(X,Y, KNN(), nsplits, PRiNt=False)
LDAerrs=evaluate(X,Y,LDA(), nsplits, PRINT=False)
QDAerrs=evaluate(X,Y,QDA(), nsplits, PRINT=False)
SVMerrs=evaluate(X,Y,SVC(kernel="rbf"), nsplits, PRINT=False)
ANNerrs=evaluate(X,Y,ANN(hidden_layer_sizes=(7,7,7),
    solver='lbfgs',random_state=1,
    alpha=1e-5), nsplits,PRINT=True)
RFerrs=evaluate(X,Y,RF(), nsplits, PRINT=False)
DTerrs = evaluate(X,Y,DT(), nsplits, PRINT=False)
```

To get an idea of the error distribution, we can plot the error variance as a function of the mean error (figure 20.1; the code to produce this is omitted from the text, but can be found in the accompanying Jupyter notebook). Ideally, we would like a method that has low classification error and low error variance, i.e., a method that has a scatter point in the lower left hand corner of figure 20.1. This would suggest that logistic regression, LDA, and naive Bayes' method perform the best, although none of the methods have less than 15% error. The neural network stands out as having a very high error variance.

To further compare the results we can do a boxplot (figure 20.2). Boxplots graphically compare the median and quartiles of the error, rather than the mean and variance. The circles on the boxplots are projected outliers and the whiskers extend to the maximum and minimum recorded error values. The boxplots are

Figure 20.2: Boxplots comparing distributions of classification error for different models using the heart disease data set. In a box plot, the box shows the central 50 percent of the data, with the horizontal line at the center indicating the median. Whiskers project to the maximum and minimum values and dots are estimated outliers.



consistent with the mean/variance plot.

```
plt.boxplot([LRerrs,NBerrs,KNNerrs, LDAerrs,QDAerrs, SVMerrs,
   ANNerrs, DTerrs,RFerrs])
plt.xticks(range(1,10),["Logistic\nRegression", "GNB", "KNN",
  "LDA", "QDA", "SVM", "ANN", "DT", "Random\nForest"]);
plt.ylabel("Classification Error", fontsize=12)
```

To get some more insight into the error distribution we plot the histograms of the error distributions in figure 20.3). The histograms give some more information than either the mean/variance plots or the boxplots, because they tell us that for each method the error distribution is unimodal and somewhat bell shaped. The boxplots do not tell us the shape of the error distribution, but they give us a very intuitive way to compare the center and spread of the distributions. The histograms tell us at a glance the shapes of the distributions.

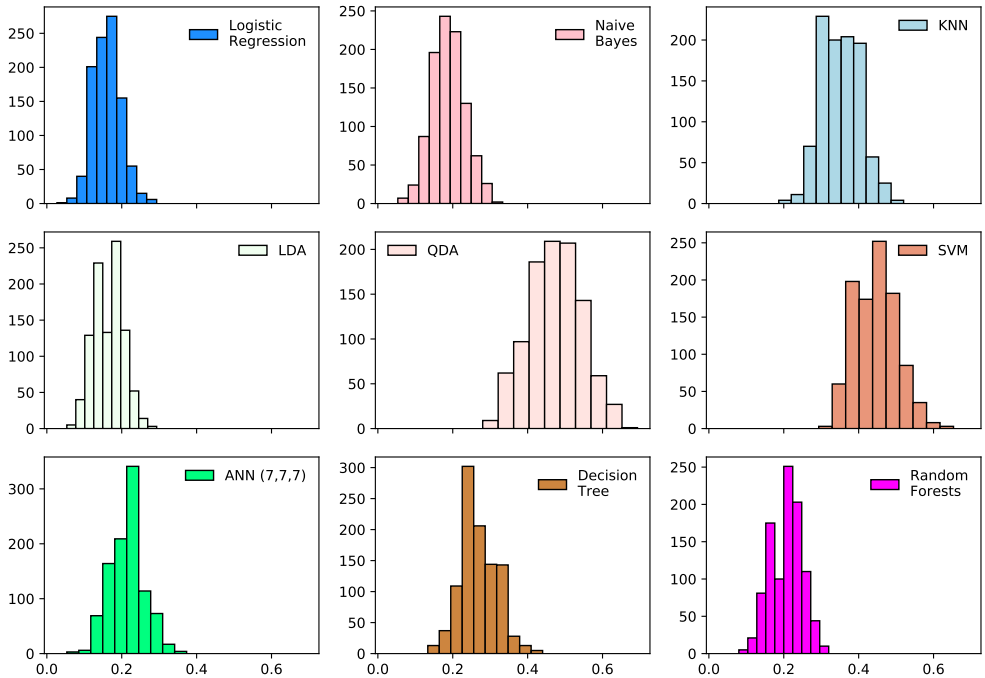
```
fig,ax=plt.subplots(nrows=3,ncols=3,sharex=True)
ax[0][0].hist(LRerrs,color="dodgerblue",edgecolor="k",
   label="Logistic\nRegression")
ax[0][1].hist(NBerrs,color="pink",edgecolor="k",
   label="Naive\nBayes")
ax[0][2].hist(KNNerrs,color="lightblue",edgecolor="k",
   label="KNN")
ax[1][0].hist(LDAerrs,color="honeydew",edgecolor="k",
   label="LDA")
ax[1][1].hist(QDAerrs,color="mistyrose",edgecolor="k",
   label="QDA")
ax[1][2].hist(SVMerrs,color="darksalmon",edgecolor="k",
   label="SVM")
ax[2][0].hist(ANNerrs,color="SpringGreen",edgecolor="k",
   label="ANN (7,7,7)")
ax[2][1].hist(DTerrs,color="Peru",edgecolor="k",
   label="Decision\nTree")
ax[2][2].hist(RFerrs,color="fuchsia",edgecolor="k",
   label="Random\nForests")
for j in range(3):
    for k in range(3): ax[j][k].legend(frameon=False)
```

As we can see, there is a noticeable difference in the error distributions. For example, logistic regression seems to have a low error rate and a very tight distribution, while the neural network has a very wide spread of errors. (One reason for this spread could be that we did not normalize our data.) The LDA gives much lower errors than the QDA results, although the classifier identified collinear variables for both of these methods (not shown; see notebook for chapter 20). Perhaps a dimensional reduction through PCA or LDA could lead to some additional improvement.

A particularly useful measure of a binary classifier is its ROC curve (see pages 119ff). Since we classified this data into precisely two categories, we can calculate ROC curves for our results. It is easiest to generate an ROC curve for a *typical* classification run, although we could average the results of our 1000 test/train splits. Here we define a new function **ROC** that fits a model (parameter **MODEL**) and returns lists containing the false positive rate and true positive rates. These are the x and y variables of the ROC plot.

```
def ROC(X, Y, MODEL):
    XTRAIN,XTEST,YTRAIN,YTEST=train_test_split(X,Y)
    model=MODEL
    model.fit(XTRAIN,YTRAIN)
    YPR=model.predict_proba(XTEST)[:,1]
    F,T,THRESH=roc_curve(YTEST,YPR)
    return(F,T)
```

Figure 20.3: Histograms showing variability of class prediction error for different models using heart diseasae data set. For each method 1000 train/test splits were performed. The y-axis gives the frequency counts for the number of results (in each bin) with a given error; the x-axis gives the classification error as a proportion of wrongly classified data.



We'll use **ROC** to obtain a single set of x and y data lists for subsequent plotting from each classifier.

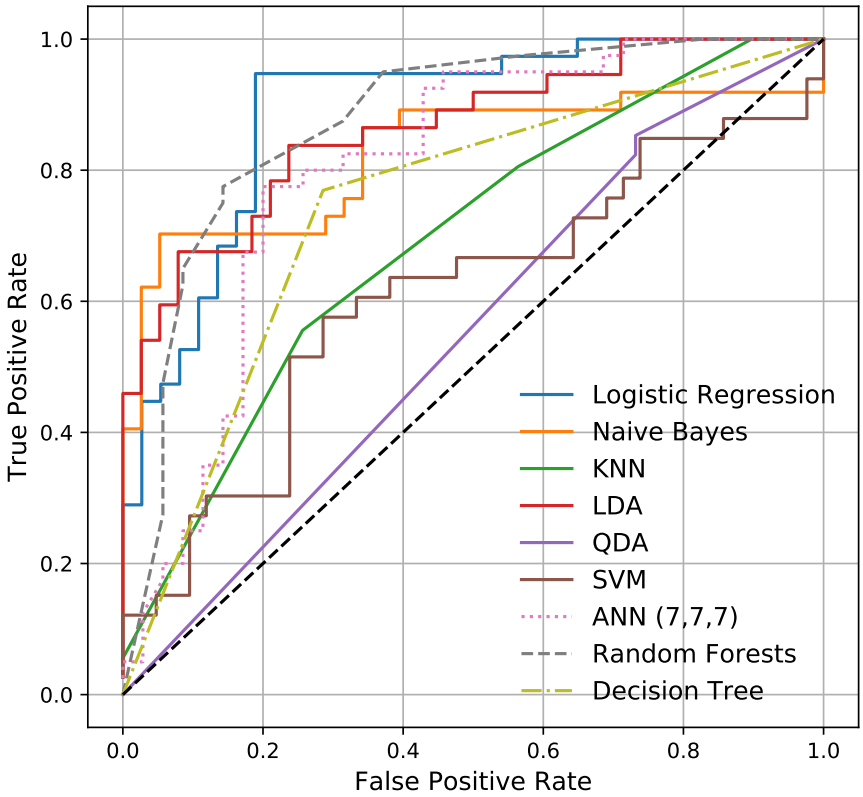
```
LRF,LRT=ROC(X,Y,LR())          # Logistic
NBF,NBT=ROC(X,Y,GaussianNB())  # Naive Bayes
KNNF,KNNT=ROC(X,Y,KNN())       # KNN
LDAF,LDAT=ROC(X,Y,LDA())       # LDA
QDAF,QDAT=ROC(X,Y,QDA())       # QDA
SVMF,SVMT=ROC(X,Y,SVC(probability=True)) # SVM
AF, AT = ROC(X,Y, ANN(hidden_layer_sizes=(7,7,7),
   solver='lbfgs',random_state=1,
   alpha=1e-5))                # ANN
RFF, RFT = ROC(X,Y, RF())      # Ran Forest
DTF, DTT = ROC(X,Y, DT())      # Dec Tree
```

The ROC curve can be produced with a sequence of **plot** calls. The result is shown in figure 20.4.

```
plt.plot(LRF,LRT,label="Logistic Regression")
plt.plot(NBF,NBT,label="Naive Bayes")
plt.plot(KNNF,KNNT,label="KNN")
plt.plot(LDAF,LDAT,label="LDA")
plt.plot(QDAF,QDAT,label="QDA")fig-heart-disease-roc}
plt.plot(SVMF,SVMT,label="SVM")
plt.plot(AF, AT, label="ANN (7,7,7)", ls=":")
plt.plot(RFF, RFT, label="Random Forests", ls="--")
plt.plot(DTF, DTT, label="Decision Tree", ls="-.")
plt.xlabel("False Positive Rate", fontsize=12)
plt.ylabel("True Positive Rate",fontsize=12)
```

The ROC curves show us that all of the methods do better than random guessing, but some of them do only slightly better. Logistic regression and LDA are the top curves, with random forests as the dashed line slightly below them. This is consistent with the box plots and histograms. It may well be possible to improve the classification by tweaking some of these methods, as we have used the default settings for all of them. As previously stated, normalization and dimensional reduction to avoid over-fitting and co-linearity could also improve performance.

Figure 20.4: ROC curves for different models on heart disease data set.



Chapter 21

Clustering Methods

Clustering is a form of **unsupervised learning**. In all of our learning methods so far, we have had a **Y** vector of **exemplar values**. These exemplar values are either known class labels (for classification) or known function values (for regression). These **Y** values have been used to train our classifier. In an unsupervised classifier, there are no exemplar values available to train the model. Instead we look for patterns that may reveal hidden groupings in the data. These emergent groupings are known as **clusters**. The purpose of the clustering algorithm is then to find these emergent (or hidden) patterns without the assistance of training values.

Quite a variety of clustering algorithms have been invented.[1] Each of these methods have their uses and advantages, and each have their failings. It is worthwhile becoming at least familiar with all of them so that you know what their capabilities and restrictions limitations are. While some algorithms work very well[2] with some data sets, they fail miserably on other data sets. It would not be surprising if one method that works well on data set A fails to cluster set B well, while a second method that succeeds on data set B fails on set A. Making things even more ambiguous, when multiple algorithms work well on the same data, they may produce conflicting clustering results, especially when the cluster boundaries are not visually clear.

In this chapter we will review three representative methods: **k-means**, **agglomerative hierarchical clustering**, and **DBscan**.

[1] For a comparison of the nine different methods implemented in **sklearn**, for example, see `https://scikit-learn.org/stable/modules/clustering.html`.

[2] In the sense of revealing hidden patterns that would otherwise be obvious to the naked eye once they were pointed out to the human observer.

K-Means Clustering

In the **k-means** clustering algorithm, each cluster is assumed to be associated with a *characteristic* point. After convergence, each point will represent the **centroid** of its cluster. These points are initially seeded (randomly located) by the algorithm, which iterates on their location.

The input to k-means is the number of clusters N. K-means will always find the requested number of clusters, even if the natural data groupings do not reflect the number requested.[3] Since k-means has no idea where the centroids actually are, the algorithm will initially place the N centroids $\mathbf{C}_1, \ldots, \mathbf{C}_N$, at random locations. Next, k-means calculates the distance between each point in the data set $\mathbf{p}_i$, $i = 1, \ldots, n$ and each center point $\mathbf{C}_j$, $j = 1, \ldots, N$. It determines which characteristic center point is closest to each data point. Then data point $\mathbf{p}_i$ is assigned to cluster k if $\mathbf{p}_i$ is closer to $\mathbf{C}_k$ than it is to any of the other $\mathbf{C}_j$ for $j \neq k$. Let κ_i be the cluster to which point $\mathbf{p}_i$ is assigned. Then[4]

$$\kappa_i = \arg\min_k \|\mathbf{C}_k - \mathbf{p}_i\|$$

The next step is to update the characteristic points. The cluster centers are adjusted to point to the new centroids.

$$\mathbf{C}_k = \frac{\sum_{j=1}^{n} I_{kj}\mathbf{p}_j}{\sum_{j=1}^{n} I_{kj}}$$

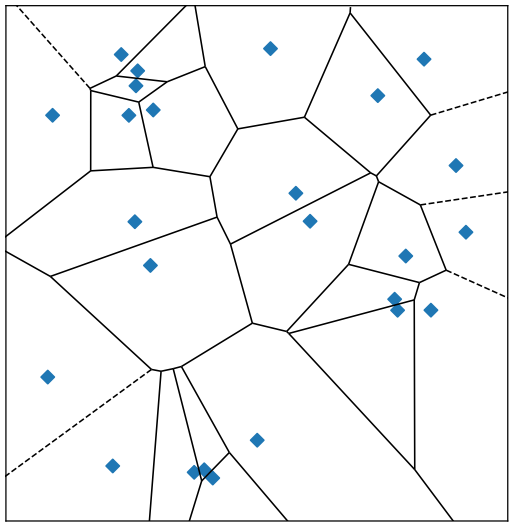
where I_{ij} is an **indicator function**, so that $I_{kj} = 1$ if $\mathbf{p}_j$ is in cluster k, and zero otherwise.

The whole process is then repeated until the cluster centers don't move any more. Because each data point is closer to its center than another center, k-means effectively finds the **Voronoi centers** (figure 21.1) of the clusters.

[3] Kmeans was invented by Steinhaus, H. (1957). "Sur la division des corps matériels en parties." [Bull. Acad. Polon. Sci. (in French).] **4**:801-804. It was named k-means by MacQueen, J. (1967). "Some methods for classification and analysis of multivariate observations." *Proc. Fifth Berkeley Symp. on Math. Stat. and Prob.*, eds Le Cam, L. Neyman, J **1**:281-297. The most common implementation is given by Lloyd, S. P. (1957) "Least squares quantization in PCM." Technical Note, Bell Laboratories (internal publication) and Forgy, E. W. (1965) "Cluster analysis of multivariate data: efficiency vs interpretability of classifications." *Biometrics.* **21**: 768-769. Lloyd's report, though it preceded Forgy, was not published publicly until (1982) *IEEE Trans. Inf. Theory.* **28**: 128-137.

[4] The arg min function is defined as follows: $\arg\min(f(x))$ returns the value of x that minimizes $f(X)$.

Figure 21.1: Voronoi diagram and Voronoi centers. The centers are shown as small filled diamonds. There is one cell corresponding to each center. The visualization of the Voronoi Diagram is truncated to a square, but cells around the edge either extend to infinity or edges join at some point off the boundary of the figure.



A **Voronoi diagram** is a partitioning of the plane into N regions such that if a point P is in a region R with Voronoi center R', then P is closer to R' then it is to any other Voronoi center in the plane. Voronoi diagrams are named after Georgy Voronoi[5] who described them in detail in 1907. They were first mentioned in the scientific literature by René Descartes in the 17th century in a discussion on the influences of the planets.[6] In 1854, the physician John Snow analyzed outbreaks of cholera in London by comparing the locations of individual deaths to the locations of water pumps. He determined that nearly all deaths occurred in the same Voronoi cell, meaning that they were tied to a common water pump. Snow's work is one of the earliest published applications of clustering. (Snow used pipe length rather than Euclidean distance as his metric.[7]) About the same time they were also studied by Dirichlet.[8] In two dimensions individual Voronoi cells are sometimes also called **Voronoi-Dirichlet polygons** or **Thiessen polygons**.[9]

[5] G. Voronoi, ["Nouvelles applications des paramẽtres continus à la théorie des formes quadratiques." (in French)] *Journal für die Reine und Angewandte Mathematik*, **133**:97-178 (1907)

[6] R.Descartes. [*Le Monde, ou Traite de la Lumiere* (in French)] (1644).

[7] J.Snow. *On the mode of communication of cholera.* London: John Churchill. (1855).

[8] Gustav Lejeune Dirichlet. [" Uber die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen." (in German)] *Journal für die Reine und Angewandte Mathematik*, **40**:209-227 (1850).

[9] Ater meteorologist After Alfred Thiessen (1872-1965) who used them in his analysis of the

While k-means will converge, it will not necessary converge to a *unique* solution. Thus any k-means algorithm should be run multiple times with different (randomized) starting points to verify that an optimal solution is reached (this is normally done by the built-in algorithms, e.g., in **scipy** or **sklearn**).

K-means works well at identifying well-separated Gaussian clouds, so long as the correct number of clusters is requested. Guessing the correct number of clusters is crucial. Using a value of k that is too small or too big will result in grouped or artificially split clusters, and if the clusters are too close, they may not be well separated. Clusters that are oddly shaped, or wrap around one another, will be artificially split by k-means, and thus other algorithms are better.

The k-means clustering algorithm is implemented in the **sklearn.clustering** package class **KMeans**.[10] As with classification algorithms, the class must first be instantiated. The parameter **n_clusters** specifies how many clusters the algorithm should look for.

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=5)
```

The algorithm is then fit with the **fit** method. Input to **fit** is a standard feature array **X**.

```
kmeans.fit(X)
```

The clustering result is given by the class attribute **kmeans.labels_**. If we asked for 5 clusters, then **labels_** will contain an array of integers of the same length as **X**, and the value of each integer will be in the range from 0 through 4. If we want to predict the cluster membership of another feature vector (or array of feature vectors) we can use the **kmeans.predict** method.

K-means works best with data that has near-Gaussian distributions. To demonstrate this, let's generate a few Gaussian clouds in the plane (see figure 21.2). We'll save the coordinate vectors to a feature matrix **X** that we pass to **KMeans**.

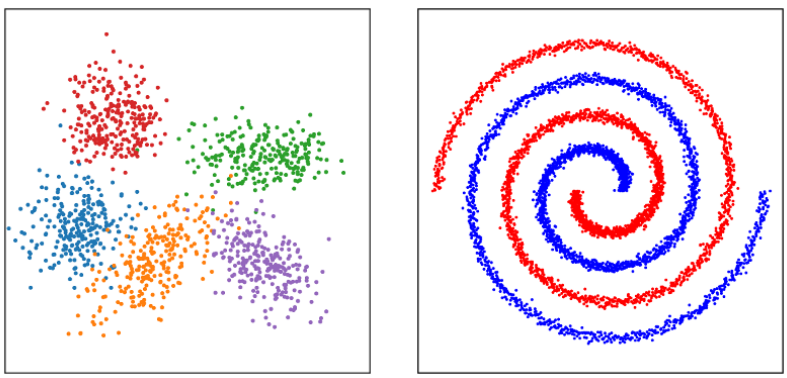
```
import numpy as np
import random
import matplotlib.pyplot as plt
nclouds=5
```

effects of weather on different land areas. See A. Thiessen, "Precipitation Averages for Large Areas." *Monthly Weather Review*, **39**(7):1082-1098 (1911).

[10] Full documentation on **KMeans** can be found online at `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`.

```
A=[[[.5,0],[0,.5]],
   [[.75,.5],[.5,.75]],
   [[1,0],[0,.25]],
   [[.5,0],[0,.5]],
   [[.5,-.2],[-.2,.5]]]
centers=np.array([[2,4],[4,3],[7,6],[3,7],[7,3]])
xcenters=centers.T[0]
ycenters=centers.T[1]
xx=[]; yy=[]
for j in range(nclouds):
    center=[xcenters[j],ycenters[j]]
    xy=np.random.multivariate_normal(center,A[j],250)
    x,y=zip(*xy)
    plt.scatter(x,y,marker=".", s=15)
    xx=xx+list(x); yy=yy+list(y)
X=np.array([xx,yy]).T
```

Figure 21.2: Two different input files used to compare the k-means algoorithm. Left: Gaussian clouds generated with multivariate normal random number generator. The k-means clustering results are illustrated in figure 21.3. Right: Inter-woven spirals. The k-means clustering results are illustratred in figure 21.4.

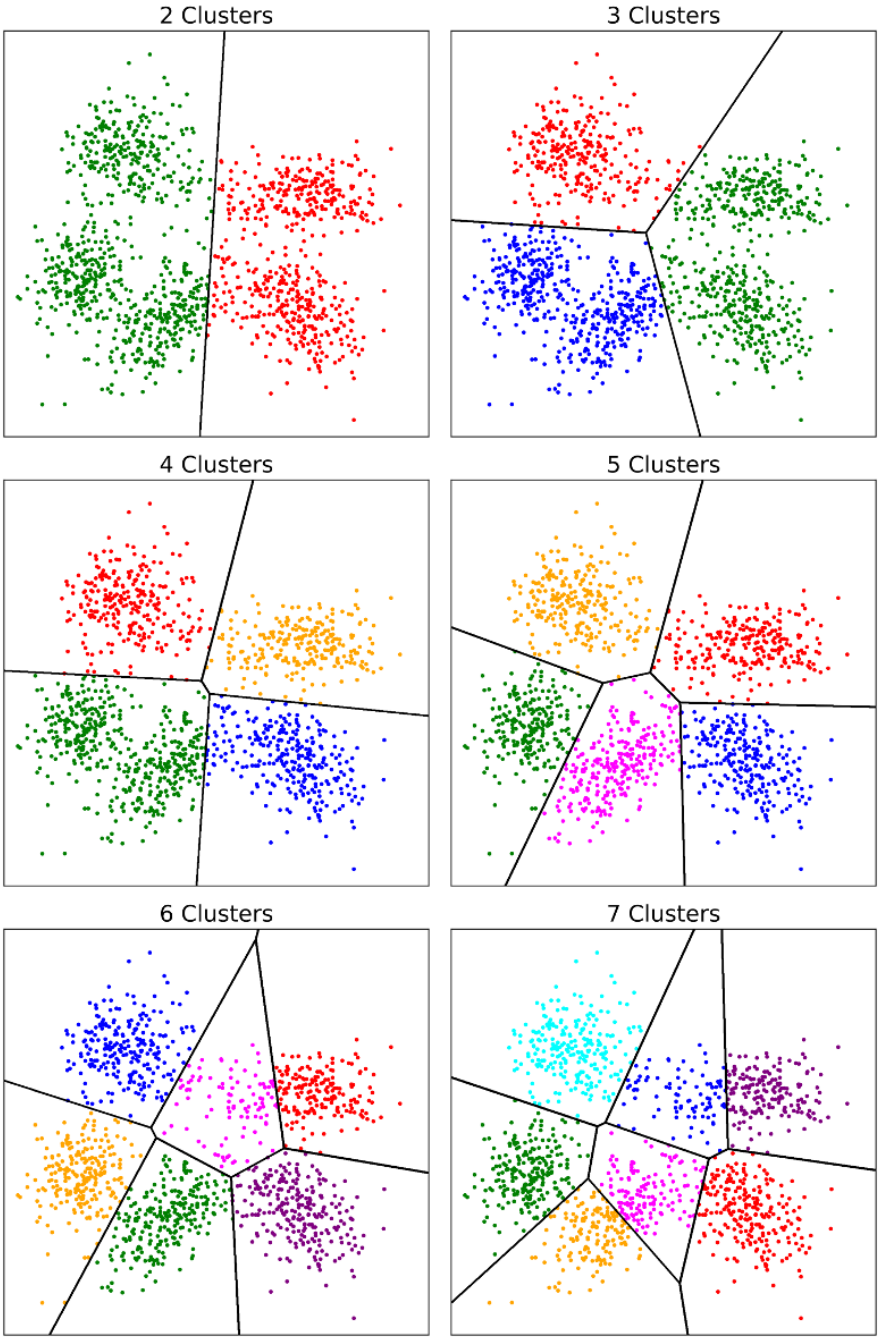


To perform the clustering and then make a scatter plot with each cluster a different color, we pull out each the coordinates for each data point, then determine which cluster it is in by looking the **labels_**. For example, to cluster and plot the results for five clusters,

```
kmeans5=KMeans(n_clusters=5)
kmeans5.fit(X)
for j in range(5):
    p,q=get_cluster_xy(X,j,kmeans5.labels_)
    plt.scatter(p,q,marker=".",s=25)
```

The results, when we ask k-means to identify from two through seven clusters,

Figure 21.3: Results of k-means clustering on a collection Gaussian clouds for various values of the parameter **n_cluseters**. The lines show the cluster boundaries (Voronoi cells).



are compared in figure 21.3. As we see in each frame of fig. 21.3, the cluster boundaries are always straight lines. These straight lines are the Voronoi cell boundaries of the cluster centers (not shown). For **n_clusters** too small (e.g., 2,3 or 4), multiple clusters are grouped together into a single cell. For **n_clusters** too large (e.g., for **n_clusters** larger than 5), the clusters are artificially split.

As an example of a data set that k-means works very poorly at, consider the inter-woven spirals shown on the right-hand side of figure 21.2. The function **make_spiral** shown here produces data points that are randomly placed about an Archimedian spiral $r(\theta) = 1 + (1 + \eta)\theta$.

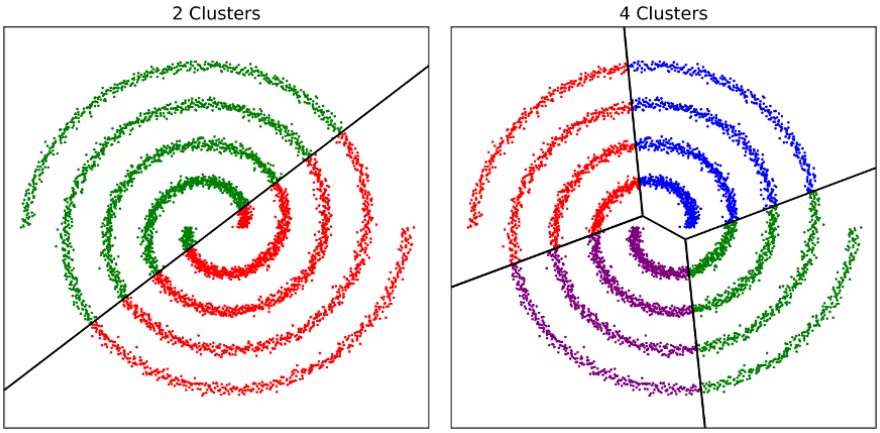
```
def make_spiral(rand, Eta, Phi):
    def f(theta,eta=.1, rnd=.1, phi=0):
        R=1+(1+eta)*theta/(2*pi)+rnd*np.random.normal(0,rnd)
        x=R*cos(theta+phi)
        y=R*sin(theta+phi)
        return(x,y)
    rvals=np.linspace(0,4*pi,1000)
    xvals,yvals=zip(*[f(r,rnd=rand, eta=Eta,phi=Phi)
                      for r in rvals])
    return(xvals,yvals)
```

We can use **make_spiral** to generate, for example, two spirals that are phase shifted from one another by 180°.

```
x,y=make_spiral(.5,2.1,0)
u,v=make_spiral(.5,2.1,pi)
X=np.array([x+u,y+v]).T
```

The natural groupings of the inter-woven spirals are the two separate spirals (i..e, red and blue spirals in figure 21.2). Such groupings can be easily separated by the human eye. There are points in either spiral that have nearer neighbors in the other spiral than the same spiral. This type of grouping cannot be determined by an algorithm like k-means because k-means is centroid-based. Clustering results on the spirals for two and four clusters are shown in figure 21.4. The clusters found by k-means split the spiral up like a pie, and do not find the natural clusters. An algorithm that works by examining very-near-neighbors and growing the clusters along a path will work better than an algorithm like k-means that builds clusters from global measures like the centroid. Examples of such algorithms are **spectral clustering** (not discussed in this book) and the **DBScan** algorithm.

Figure 21.4: Results of k-means clustering on two interwound spirals when either two (left) or four (right) clusters is requested. K-means is unable to determine the clusters correctly for this type of data.



Clustering Measures

When you don't know the number of clusters in advanced there are several different metrics for calculating a "goodness of fit." Two of these are the **silhouette index** and the **Calinski-Harabasz Index (CI Index)**. We use these indices to performing repeated clusterings and requesting a different number of clusters each time; then we look for a peak or elbow in a plot of the index versus the number of clusters. The silhouette essentially compares the centers, or means, of the clusters; the CI index compares the spread, or dispersion, of the data. We would expect well-defined, well-separated clusters to be well explained by such indices, but we would not expect clusters such as the inter-woven spirals to be explained as well.

To understand what the silhouette index means, suppose that $\mathbf{p}_i$ is in some cluster C. Then let

$$\begin{aligned}
\mu_i(\mathbf{p}_i) &= \text{mean distance from } \mathbf{p}_i \text{ to points } \mathbf{q}_j \neq \mathbf{p}_i, \forall \mathbf{q}_j \in C \\
\mu_i'(\mathbf{p}_i) &= \text{mean distance from } \mathbf{p}_i \text{ to points } \mathbf{q}_j, \forall \mathbf{q}_j \notin C \\
M_i(\mathbf{p}_i) &= \max_j(\mu_i, \mu_i')
\end{aligned}$$

We can define a silhouette score for each point as

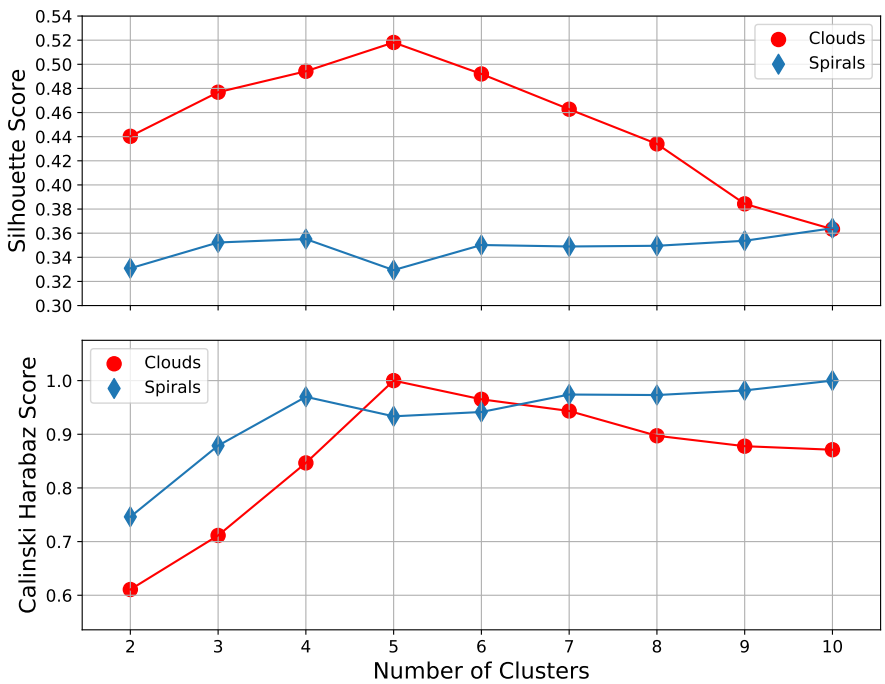
$$S_i = \frac{\mu_i'(\mathbf{p}_i) - \mu_i(\mathbf{p}_i)}{M_i(\mathbf{p}_i)} \tag{21.1}$$

Then the silhouette index is the average over all the scores,

$$S = \frac{1}{n} \sum_{i=1}^{n} S_i \tag{21.2}$$

Terms in the sum that are more negative come from points that are closer to other clusters. Thus clusterings with an average closer to positive one are better-defined. In other words, the closer the silhouette index is to 1, the better the overall clustering.

Figure 21.5: Comparison of silhouette (top) and Calinski-Harabaz measures(bottom) as a function of number of clusters requested for clustering on a collection of Gaussian clouds (shown in figure 21.3) and two interlocking spirals (such as those shown in figure 21.4). In the bottom figure, the CH scores are normalized to the maximum CH score for each plot, to make the plots easier to read.



The function **`silhouette_score`** from **`sklearn.metrics`** can be used to calculate the silhouette score in Python. A classifier must be instantiated and a data set fit before the silhouette score for the classification can be found. In general, one is interested in calculating the silhouette score as a function of a complexity index (e.g., the number of clusters). For example, to calculate the silhouette score for k-means clustering of 5-clusters of a data set with a feature matrix **`X`**,

```
from sklearn.metrics import silhouette_score
from sklearn.clustering import KMeans
kmeans=KMeans(n_clusters=5)
kmeans.fit(X)
print(silhouette_score(X,kmeans.labels_))
```

```
0.514783493314483
```

We can encapsulate the above code in a **for** loop, as in **for j in range(2,11)**, and then replace the **n_clusters=5** with **n_clusters=j** to explore the dependence of the silhouette score on the number of clusters. In the top frame of figure 21.5 we have plotted the silhouette score using k-means for both the Gaussian clouds data set (e.g., the data shown clustered in figure 21.3) and for the interwoven spirals (the data shown clustered in figure 21.4). As we can see from this figure, the silhouete score peaks at about 0.53 for five clusters for the Gaussian data, but is relatively flat at around 0.35 for the spiral data. This corroborates our observations with the actual clustering results.

The **Calinski-Harabasz Index** (**CI Index**), is defined as[11]

$$\mathrm{CH} = \left[\frac{n-k}{k-1}\right]\frac{\mathrm{tr}\,\mathbf{S}'}{\mathrm{tr}\,\mathbf{S}} \tag{21.3}$$

Here $\mathbf{S}$ and $\mathbf{S}'$ are the in-cluster and between-cluster scatter matrices (page 173), where n is the total number of data points and k is the number of clusters. To understand the meaning of the CH index, recall the definition of the scatter matrix (eq. 16.8). Since a scatter matrix is essentially the covariance of the centered feature matrix times the number of degrees of freedom, the diagonal elements of the scatter matrix contains the variances of the features. Thus the *trace* of the scatter matrix is the sum of variances in the features. This sum (divided by the number of degrees of freedom) is called the **dispersion** and is a measure of the overall variability of the feature data. Since there are n data points and k clusters, there are $n-k$ degrees of freedom in the in-cluster data and $k-1$ degrees of freedom in the between-cluster data.

$$\mathrm{tr}(\mathbf{S}') = \mathrm{tr}((k-1)\mathrm{cov}(\mathbf{X}'^{\mathrm{T}}\mathbf{X}')) = (k-1)(\sigma'^2_1 + \sigma'^2_2 + \cdots + \sigma'^2_k) \tag{21.4}$$

$$\mathrm{tr}(\mathbf{S}) = \mathrm{tr}((n-k)\mathrm{cov}(\mathbf{X}^{\mathrm{T}}\mathbf{X})) = (n-k)(\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2) \tag{21.5}$$

Hence

$$\mathrm{CH} = \frac{\sigma'^2_1 + \sigma'^2_2 + \cdots + \sigma'^2_k}{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2} \tag{21.6}$$

[11] Calinski,T., Harabasz, J. (1974) "A dendrite method for cluster analysis." *Commun. Statist.* **3**(1):127.

A high CH score means that the between-cluster variances are higher than the in-cluster variances, and the cluster variation is good. A low CH means that the cluster separation is poor. The idea here is to plot CH(k) as a function of the number of clusters k and look for the "elbow," the point beyond which there is no significant change. The CH index is also implemented in **sklearn**. To calculate the CH index for a single clustering instance, assuming the feature data is in the array **X**,

```
from sklearn.metrics import calinski_harabaz_score as CH
from sklearn.clustering import KMeans
kmeans=KMeans(n_clusters=5)
kmeans.fit(X)
print(CH(X,kemans.labels_))
```

```
1972.0630797994027
```

By encapsulating this code in a **for** loop we can explore the dependence of the CH index on the number of clusters for a given data set. Unlike the silhouette index, which has a maximum possible value of one, there is no upper limit for the CH index. In order to compare the results for two different examples (Gaussian clouds and spirals) we have normalize the CH indices to their relative maximum values in the bottom frame of figure 21.5. The knee in the CH (for the Gaussian cloud) occurs at the same number of clusters as it did for the silhouette index. While the CH index for the spirals seems to have a knee at 3 clusters, it still continues to grow for larger numbers of clusters, showing the ambiguity of this index in situations like this. Silhouette and CH are clear and consistent when the data is Gaussian, but are ambiguous and/or inconsistent when the data is not. This could be a signal that perhaps the clustering is not being successful, but it could also be a sign that the clustering measure is inappropriate for the data and method being applied.

Hierarchical Clustering

The name **hierarchical clustering** refers to the tree-like methods in which the clusters are built. Clustering trees are built either from the bottom up, or from the top down. The top-down methods begin by placing the entire data set into a single cluster (called the **root**) and then using some measure to divide the cluster. This process is repeated until every individual is encapsulated into its own cluster (called **leaves**). Top-down methods are called **divisive clustering** algorithms, and can be halted when the clusters reach a certain minimum size

or a desired tree depth is reached.[12] In bottom-up methods, each individual is initially assigned to its own cluster (a leaf), and then some distance measure is used to combine the points (or clusters) together into clusters (or larger clusters). The process is repeated until all clusters have been merged into a single cluster. Bottom-up methods are known as **agglomerative clustering** algorithms.[13] Once a data set has been clustered agglomeratively, a tree can be built and clusters formed at the desired depth as with divisive clustering. However, it is not unusual for the two approaches to yield different trees and hence different clustering results. Since divisive clustering is not implemented in the standard Python packages we will explore only agglomerative clustering.

Agglomerative clustering begins with each point assigned to its own cluster. The clusters are then successively merged to form larger clusters. This pairwise merging builds up a succession of binary trees, with two clusters represented as leaves and the pair represented by the root node of the binary tree. At each step, some metric or distance measure must be used to determine which pair of clusters (or trees) to merge next. The process continues until the entire data set is merged. The process is represented hierarchically by a single binary tree, with the entire data set represented by the root. The natural clusters are found by pruning the tree at a desired depth.

Two distances are important during this process: the distance between points within a cluster, and the distance between clusters. The distance between points is represented by a **distance rule** and the distance between clusters is measured by a **linkage measure**. Clusters are built, or agglomerated, according to a distance measure $d(\mathbf{p}, \mathbf{q})$, where $\mathbf{p}$ and $\mathbf{q}$ are points. Common distance measures include:

- *Euclidean* distance $d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum (p_i - q_i)^2}$. This may be best if all the clusters have Gaussian distributions.
- L_1 (Manhattan) norm $d(\mathbf{p}, \mathbf{q}) = \sum |p_i - q_i|$
- L_∞ (max) norm $d(\mathbf{p}, \mathbf{q}) = \max_i |p_i - q_i|$
- *Mahalanobis* distance $d(\mathbf{p}, \mathbf{q}) = \sqrt{(\mathbf{p} - \mathbf{q})^{\mathrm{T}} \mathbf{S}^{-1} (\mathbf{p} - \mathbf{q})}$, where $\mathbf{S}$ is a covariance matrix. This may be best if all the clusters have multivariate normal distributions.
- *Hamming* distance for strings, i.e., the number of bits (for bit strings) or characters (for character strings) for which the two strings are different.

[12] Kaufman L., Rousseeuw P.J. (1990). *Finding groups in data.* Wiley.

[13] Ward, J. H. (1963) "Hierarchical Grouping to Optimize an Objective Function." *Journal of the American Statistical Association.* **58**(301):236-244.

Initially, the distances between all points must be calculated using the chosen distance metric. The closest pair of points is joined to form a new cluster. Subsequently, all distances must be recomputed using a linkage measure between clusters, and the distance between all pairs of clusters must be updated after each iteration. At each iteration, two clusters are joined to form a new cluster, and the between-cluster distances recalculated.[14] The process is repeated until all clusters are joined into a single tree. If A and B are clusters, some common linkage measures are:

- The *Single Link* measure:$L(A,B) = \min\{d(\mathbf{p},\mathbf{q}) | \forall \mathbf{p} \in A, \mathbf{q} \in B\}$
- The *Complete Link* measure: $L(A,B) = \max\{d(\mathbf{p},\mathbf{q}) | \forall \mathbf{p} \in A, \mathbf{q} \in B\}$
- A *group average* $L(A,B) = \dfrac{1}{|A||B|} \sum_{\mathbf{p}\in A, \mathbf{q}\in B} d(\mathbf{p},\mathbf{q})$ where $|C|$ means the number of points in cluster C. This averages the distances between all the points in either cluster.
- A *mean distance*, $L(A,B) = d\left(\dfrac{1}{|A|}\sum_{\mathbf{p}\in A}\mathbf{p}, \dfrac{1}{|B|}\sum_{\mathbf{p}\in B}\mathbf{p}\right)$. This gives the distance between the centroids of the two clusters.
- *Ward's method*: $L(A,B) = \dfrac{|A||B|}{|A|+|B|}\left[L_2\left(\dfrac{1}{|A|}\sum_{\mathbf{p}\in A}\mathbf{p} - \dfrac{1}{|B|}\sum_{\mathbf{p}\in B}\mathbf{p}\right)\right]^2$, where $L_2(\mathbf{p}) = \sqrt{\sum p_i^2}$. Ward's method is popular because it minimizes the variance.

Example. Consider the following collection of data points (figure 21.6):

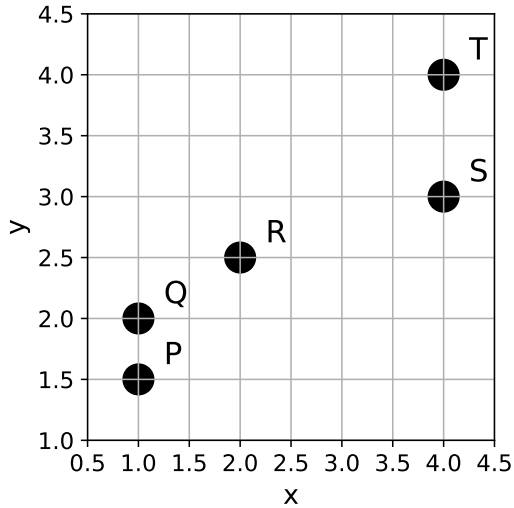
$$P = (1.0, 1.5),\ Q = (1.0, 2.0),\ R = (2.0, 2.5),\ S = (4.0, 3.0),\ T = (4.0, 4.0)$$

The distances between the points (using an Euclidean metric) are:

	P	Q	R	S
Q	0.50			
R	1.41	1.12		
S	3.35	3.16	2.06	
T	3.91	3.61	2.50	1.00

[14]The Lance-William formula provides a technique for updating these distances at each step that is more efficient that recalculating the entire matrix. Lance, G, Williams, W. (1967) "A General Theory of classificatory Sorting Strategies 1. Hierarchical Systems." *The Computer Journal.* **9**:373-380.

Figure 21.6: Points $\{\mathbf{P}, \mathbf{Q}, \mathbf{R}, \mathbf{S}, \mathbf{T}\}$ used in the clustering example.



Thus **P** and **Q** are the two closest points. We can join them into a cluster. Using an averaging metric, the center of this cluster $\{\mathbf{P}, \mathbf{Q}\}$ is at $(1, 1.75)$. The distances between the new clusters are now:

	PQ	R	S
R	1.25		
S	3.25	2.06	
T	3.75	2.50	1.00

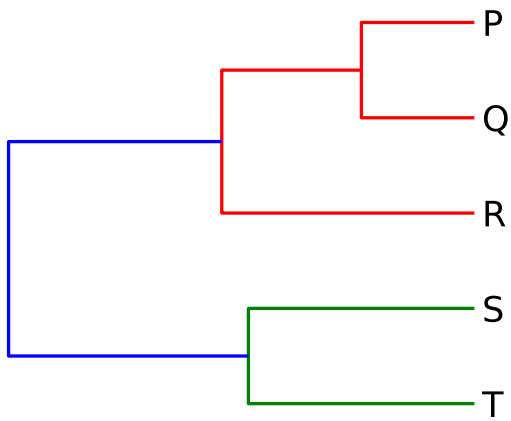
This time, the closest pair of clusters are **S** and **T**. We form a new clusters $\{\mathbf{S}, \mathbf{T}\}$ centered at $(4, 3.5)$. The new cluster distances are:

	ST	PQ
PQ	3.47	
R	2.24	1.25

The smallest distances is between **R** and $\{\mathbf{P}, \mathbf{Q}\}$. We form a new cluster $\{\mathbf{R}, \{\mathbf{P}, \mathbf{Q}\}\}$. Since there are only two clusters left, we join them together to form the entire tree, $\{\{\mathbf{S}, \mathbf{T}\}, \{\mathbf{R}, \{\mathbf{P}, \mathbf{Q}\}\}\}$. It is usually easier to visualize this as a **dendrogram** (figure 21.7). □

We build the dendrogram with **`scipy.cluster.hierarchy.dendrogram`**. Here we demonstrate using the Gaussian cloud data shown on the left hand side of figure 21.2 and that we clustered with k-means in figure 21.3. Assume that we have already generated the feature matrix and placed it into the **`numpy array XCLOUD`**. Here we will truncate the dendrogram at level 5.

Figure 21.7: Dendrogram for the data clustered in figure 21.6.




```
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
Z=linkage(XCLOUD, method="ward")
dendrogram(Z,5,truncate_mode="level",show_leaf_counts=True)
plt.show()
```

Additional features of **pyplot** such as the standard commands for ticks, labels, and figure size can be used. The **linkage** function builds up the linkage matrix between the clusters at each level. We have selected a **method="ward"** (this is not the default method) for the **linkage** function because the clustering method we will be using below from **sklearn**, called **agglomerative_clustering**, uses **ward** by default. The resulting truncated dendrogram is shown in figure 21.8.

Sometimes it is useful to wrap the dendrogram around in a circle (fig. 21.9) because the leaves are otherwise very tightly pushed together. This puts the leaves all at the outer edge of the circle. While neither **scipy** nor **sklearn** include functions for a circular dendrogram, we can map the standard dendrogram to a circle via a coordinate transformation. An example is given by the code used to produce the cover figure, and figure 21.9 (see the notebook on the **github** site).

There are agglomerative clustering functions in both **sklearn**[15] and **scipy**.[16]

[15] Full documentation of the agglomerative clustering in **sklearn** can be found at `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html`.

[16] See `https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html` for full documentation of the clustering sofware in **scipy**.

Figure 21.8: Dendrogram for agglomerative clustering on the Gaussian cloud data shown in figure 21.2. The horizontal line shows an arbitrary choice for truncating the dendrogram with five clusters. The entire dendrogram was not calculated; rather, it was also truncated after five splits. The red and green coloring, alternatively, show an arbitrary clustering into two clusters.

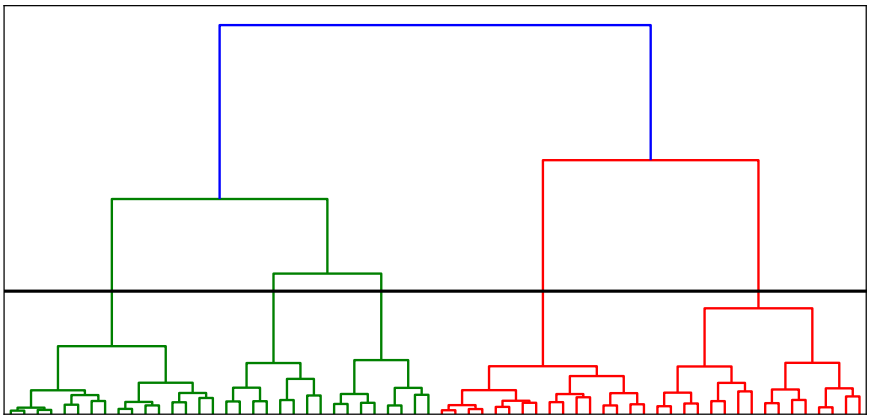
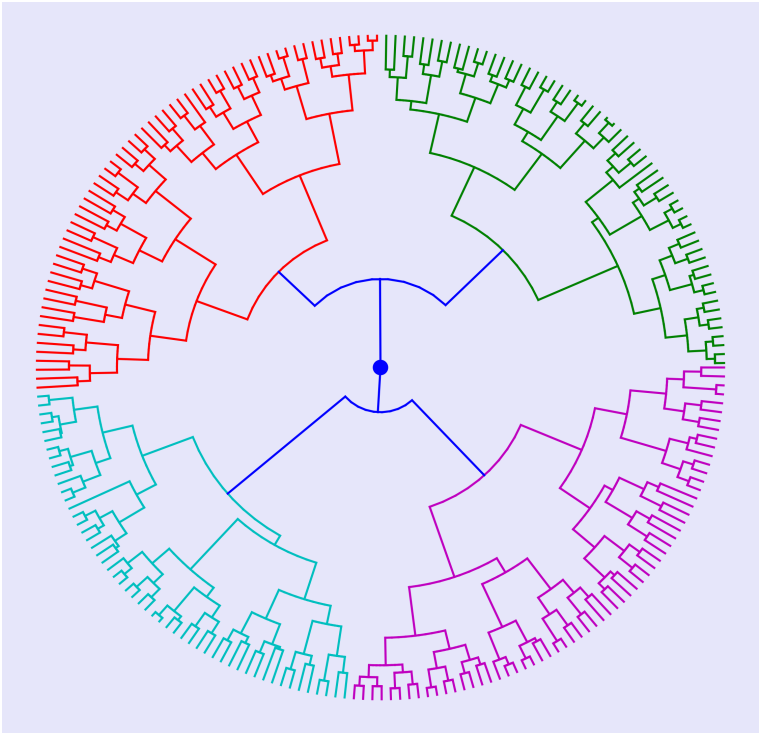


Figure 21.9: A circular dendrogram for hierarchical clustering with four clusters identified by colors.



We will demonstrate clustering using **sklearn**. To find the cluster labels for five clusters, we assume that the coordinates are placed in the feature matrix **XCLOUD** (e.g., for the Gaussian cloud).

```
from sklearn.cluster import AgglomerativeClustering
clustering = AgglomerativeClustering(
   n_clusters=5).fit_predict(XCLOUD)
```

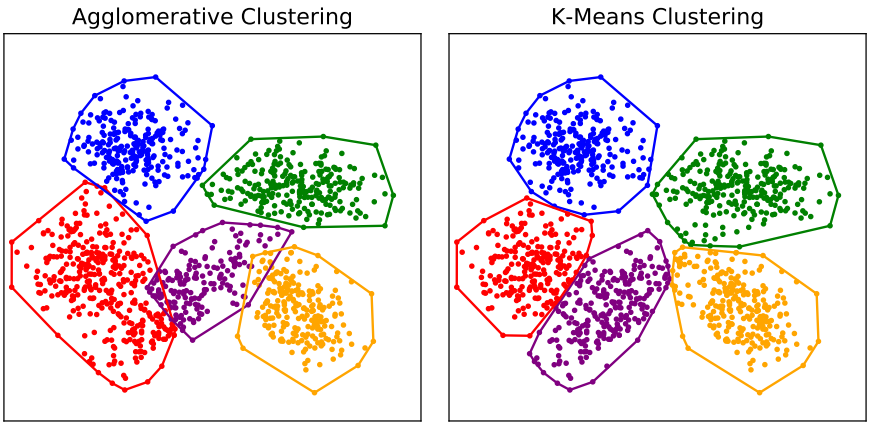
The labels are then given by the **array clustering**, which has the same length as **XCLOUD** and integer values between 0 and **n_clusters**.

To visualize the clusters for agglomerative clusters and compare them to the k-means clusters, we will draw the **convex hull** around each cluster. A convex hull is the smallest convex set that includes every point in the set. It can be determined using **scipy.spatial.ConvexHull**. The following will draw the clusters for agglometive clustering; a similar code block can be used for k-means. This code assumes that there are five clusters.

```
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull
colors=["red", "green","blue","purple","orange"]
for j in range(5):
    col=colors[j]
    xy=XCLOUD[clustering==j]
    x,y=zip(*xy)
    plt.scatter(x,y,marker=".",s=25,c=col)
    hull=ConvexHull(xy)
    for simplex in hull.simplices:
        plt.plot(xy[simplex,0], xy[simplex,1],c=col)
```

The results shown in figure 21.10 compare agglomerative clustering with k-means. It is interesting to observe that while the convex-hulls of the k-means clusters are non-overlapping, many of the convex-hulls for the agglomerative clusters do overlap. This is because the natural borders of the k-means clusters are convex Voronoi cells (figure 21.1). If we were to draw the actual borders between the agglomerative cells through the overlapping regions of their convex hulls, we would not end up with convex regions. For example, we would have to take a small purple chunk out of the red polygon, a larger orange chunk out of the purple polygon, while a very small part of the orange polygon would become purple, and a small part of the blue polygon red.

Figure 21.10: Comparison of the clustering results for agglomerative hierarchical clustering (on the left) and k-means clustering (right). The convex hulls of each cluster are drawn.



DBScan Clustering

The **DBSCAN**[17] Algorithm[18] is an example of topological clustering. It determines the overall shapes of clusters by looking at points in very small neighborhoods and then connecting those neighborhoods together bit by bit. There are two input parameters: ϵ, N. We use these to define the concept of an **ϵ-neighborhood** $N_\epsilon(\mathbf{p})$ of a point,

$$N_\epsilon(\mathbf{p}) = \{\mathbf{q} | \|\mathbf{p} - \mathbf{q}\| \leqslant \epsilon\}$$

There are three types of points in the algorithm: **core points** (those that fall inside a cluster); **border points** (on the edge of cluster; and **noise** (points that do not fall into any cluster).

The following concepts behind DBSCAN are illustrated in figure 21.11.

- $\mathbf{p}$ is a core point if $|N_\epsilon(\mathbf{p})| \geqslant N$
- $\mathbf{p}$ is **directly density reachable from** $\mathbf{q}$ if (a) $\mathbf{p} \in N_\epsilon(q)$ and (b) $|N_\epsilon(\mathbf{q})| \geqslant N$.[19]

[17] DBSCAN is an acronym for Density Based Spatial Clustering with (And) Noise

[18] Ester et. al (1996) "A density-based algorithm for discovering clusters in large spatial databases with noise," KDD96, `https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf` The paper is easily readable by students.

[19] The absolute value on the set denotes the number of points in the set.

Figure 21.11: **B** and **D** are core points; **A** and **E** are border points. **A** is directly directly density reachable from **B** but not vice-versa (using $N = 5$). **E** is density reachable from **B** but not from **A**. **A** and **E** are density connected to each other by **B**

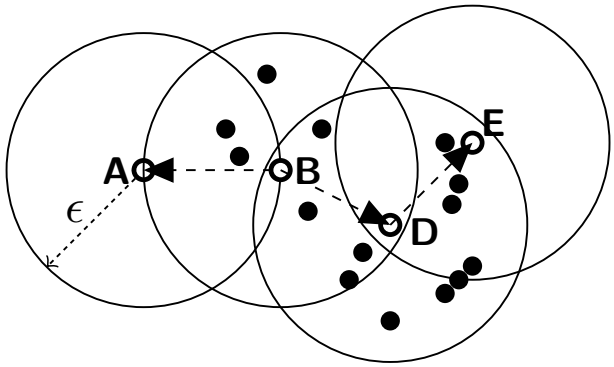
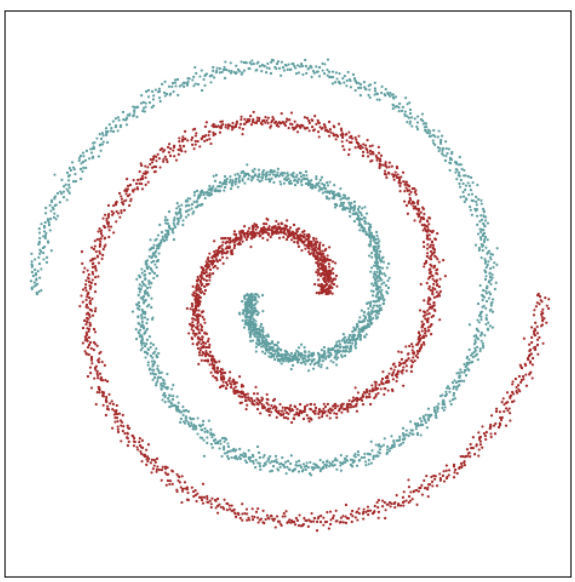


Figure 21.12: Clustering of spirals using DBSCAN algorithm.



- $\mathbf{p}$ is **density reachable** from $\mathbf{q}$ if there is a sequence of points $\mathbf{q} = \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n = \mathbf{q}$ such that each $\mathbf{p}_{i+1}$ is directly density reachable form $\mathbf{p}_i$.
- $\mathbf{p}$ is **density connected** to $\mathbf{q}$ if there is a point $\mathbf{x}$ such that both $\mathbf{p}$ and $\mathbf{q}$ are density reachable from $\mathbf{x}$.
- If S is a set of points, the C is a **cluster of points** in S (a) if $\mathbf{p} \in C$ and $\mathbf{q}$ is density reachable from $\mathbf{p}$, then $\mathbf{q} \in C$; and (b) $\mathbf{p}$ is density connected to $\mathbf{q}$.

The DBSCAN algorithm then proceeds through all the points $\mathbf{p}$ in set S. Let $C = \{\mathbf{p}\}$. Then C is expanded into the maximally density connected set containing $\mathbf{p}$, by first examining each point in $N_\epsilon(\mathbf{p})$, and then recursively considering the ϵ neighborhood of the added points. This process is continued until all points have been clustered.

DBSCAN works much better than k-means for the spiral clusters. We cluster with **`sklearn.cluster.DBSCAN`**.[20]

```
from sklearn.cluster import DBSCAN
clustering = DBSCAN(eps=.5, min_samples=5).fit(XSPIRAL)
clusteringlabels=list(set(clustering.labels_))
```

The clusters can be colored and plotted (figure 21.12) with the following block of code.

```
clusteringlabels=list(set(clustering.labels_))
for j in clusteringlabels:
    xy=XSPIRAL[clustering.labels_==j]
    if (len(xy)>0):
        x,y=zip(*xy)
        plt.scatter(x,y,marker=".",s=5)
```

[20]See the **`sklearn`** website at `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html` for full documentation.

Comparing Clustering Results

A numerical similarity measure of two cluster assignments that is often used in hierarchical clustering is the **Fowlkes-Mallows** index.[21] Alternatively, if we know the **ground truth** or correct cluster assignments, we can use an index such as the FM index to measure the accuracy of the clustering.

Let A and B be two different clustering assignments. Then the FM index is defined as

$$FM = \sqrt{\frac{a}{a+b}\frac{a}{a+c}} \tag{21.7}$$

where

- $a =$ Count of pairs of points in same cluster in A and B (analagous to *true positives*.
- $b =$ Count of pairs of points in same cluster in A but not in B (analogous to *false positives*.
- $c =$ Count of pairs of points in same cluster in B but not in A (analogous to *false negatives*.

If the two clusterings are identical then $FM = 1$ because $b = c = 0$.

We can measure the Fowlkes-Mallows using the function **fowlkes_mallows_score** in **sklearn.metrics.cluster**. We can compare the clustering results of k-means and hierarchical clustering, shown in figure 21.10 by using either of the classifications as ground truth, and the other as the classification score.

```
from sklearn.metrics.cluster import fowlkes_mallows_score as FM
print(FM(kmeans5.labels_,clustering))
```

```
0.9104708183976866
```

[21] Fowlkes, E, Mallows, C. (1983) "A Method for Comparing Two Hierarchical Clusterings." *J. Amer. Stat. Assoc.* **78**: 553-569. For a survey of different measures, as well as a more understandable description of the FM index, see Halkidi, M. et. al. (2001) "On Clustering Validation Techniques." *J. Intelligent Information Systems.* **17**:107-145.

Appendix A

Review of Probability

A **random variable** X is a variable whose value is determined by **chance**. This means that you cannot predict its value like you would be able to predict the value of a an algebraic variable. In algebra, if you are told that $y = 3x + 7$, then every time that $x = 3$ you know with absolute certainty that $y = 16$. The value of x precisely **determines** the value of the variable y. We call such algebraic variables **deterministic**. A random variable is the opposite of a deterministic variable; it keeps changing. A random variable X may depend on one or more parameters, but every time we attempt to find its value we get a different answer.

There are two general classes of random variables; **discrete** and **continuous**. A **discrete random variable** (DRV) can take on values from a discrete or enumerable sequence of values; while a **continuous random variable** (CRV) can take on an value in a continuous interval. These definitions can be expanded to higher dimensions.

The rules of probability are based on the rules of set theory. If you are already familiar with set theory (such as DeMorgan's laws) much of probability will come naturally to you. The outcome of an experiment that depends on a random variable is called an **event**. Events are usually denoted by upper case letters such as X or E. The term *event* may refer to a single outcome, or a set of outcomes. The collection of all possible outcomes from an experiment is called the **sample space** of the experiment.

Example A.1. Rolling two dice. Roll two dice, one red, and the other blue. This experiment has the following sample space. Each ordered pair represents the values (R, B) where R is the number of dots on the red die, and B is the number of dots on the blue die.

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1, 6)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2, 6)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3, 6)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4, 6)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5, 6)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6, 6)

We will consider the following events in subsequent examples.

$$E_1 = \text{The event that } R = 2 \text{ and } B = 3 \tag{A.1}$$
$$E_2 = \text{The event that } R = B \tag{A.2}$$
$$E_3 = \text{The event that } R = 2 \text{ or R=4} \tag{A.3}$$

When we use the words **and** and **or** in probability they correspond to set intersection and set union, respectively (compare with logic). Hence E_1 is an intersection formed by finding the intersection of a row and a column of the sample space.

$$E_1 = \{(R, B) \in S | R = 2 \wedge B = 3\} \tag{A.4}$$
$$= \{(R, B) \in S | R = 2\} \cap \{(R, B) \in S | B = 3\} \tag{A.5}$$
$$= \{(2, 3)\} \tag{A.6}$$

Similarly, E_2 gives the diagonal of the sample space,

$$E_2 = \{(R, B) \in S | (R = B)\} \tag{A.7}$$
$$= \{(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)\} \tag{A.8}$$

Finally, E_3 is a set union formed by joining two rows of the sample space.

$$E_3 = \{(R, B) \in S | R = 2 \vee R = 4\} \tag{A.9}$$
$$= \{(R, B) \in S | R = 2\} \cup \{(R, B) \in S | R = 4\} \tag{A.10}$$
$$= \{(2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6)\} \cup \{(4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6)\} \tag{A.11}$$
$$= \{(2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6)\} \quad \square \tag{A.12}$$

Example A.2. Baseball distances. During a game of baseball, the batter can either hit the baseball or not. If the player hits the ball, assign the distance d the ball travels to where it is either caught by another player, or hits the ground. If the player does not hit the ball, set $d = 0$. The the distance traveled by the baseball is a continuous random variable with a sample space given by the possible values of d.

Chance is described by the **distribution** of a variable. When the random variable is discrete, we can enumerate the **probability** that a random variable will take on any given value.

Definition A.1. Probability Mass Function. *Let X be a discrete random variable with sample space S and let $|S|$ be the total number of possible outcomes in S (the cardinality of S). Then the* ***Probability Mass Function*** *(PMF) is*

$$f(x) = P(X = x) = \frac{\textit{Number of ways that } X = x \textit{ in } S}{|S|} \tag{A.13}$$

where $P(X = x)$ is the probability of the event E that the random variable X takes on the value of x.

Equation A.13 is sometimes written as

$$\text{probability} = \frac{\text{Number of Successes}}{\text{Number of Possible Outcomes}} \tag{A.14}$$

The PMF has the property that

$$\sum_{x \in S} f(x) = 1 \tag{A.15}$$

where S is the sample space for X. Equation A.15 says that the probability that some event in S happening is 1, i.e., that all probabilities add up to 100%. Another way of writing this is

$$P(S) = 1 \tag{A.16}$$

This expresses the fact that the event $x \in S$ has probability 1.

Example A.3. Dice rolling (continued). Consider the three events in example A.1. We observe that the numerator of equation A.13 is the cardinality of each of the sets we enumerated, while the denominator is cardinality of the sample space, $|S| = 36$. Hence

$$P(E_1) = \frac{1}{36} \qquad \text{(Specific roll of dice)} \tag{A.17}$$

$$P(E_2) = \frac{1}{6} \qquad \text{(Doubles)} \tag{A.18}$$

$$P(E_3) = \frac{1}{3} \qquad \text{(First die is 2 or 4)} \quad \square \tag{A.19}$$

Probabilities for complex events are determined by the rules of set theory. Thus if A and B are events then

$$P(A \wedge B) = P(A \cap B) \tag{A.20}$$
$$P(A \vee B) = P(A \cup B) \tag{A.21}$$

If A are B are disjoint events (i.e., the sets A and B do not intersect $A \cap B = \varnothing$) then

$$|A \cap B| = |\varnothing| = 0 \tag{A.22}$$
$$|A \cup B| = |A| + |B| \tag{A.23}$$

Therefore

$$P(A \wedge B) = P(A \cap B) = P(\varnothing) = 0 \tag{A.24}$$
$$P(A \vee B) = P(A \cup B) = P(A) + P(B) \tag{A.25}$$

The second equation follows directly from equation A.13. Since there is no overlap, the number of possible successes is the sum of the successes in each set. If the events are not disjoint, then A and B overlap. In this case $A \cap B \neq \varnothing$. If $X \in A \cup B$, then it is possible that $A \in A \cap B$ as well, and when we count successes in the numerator of equation A.13 we have to be careful not to count it twice.

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{A.26}$$

Hence by equation A.13,

$$P(A \vee B) = P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{A.27}$$

Theorem A.1. Addition Rule. *Let A and B be events. Then*

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B) \tag{A.28}$$

We do not have to specify whether or not A and B are disjoint in the addition rule because if they are disjoint, the final term is zero; and if they are not disjoint, the final term is non-zero. Equation A.29 covers both cases.

If we set $B = \overline{A}$ in equation A.29, where $\overline{A}$ refers to the complement of A in the sample space S, then

$$1 = P(S) = P(A \cup \overline{A}) = P(A) + P(\overline{A}) - P(A \cap \overline{A}) = P(A) + P(\overline{A}) \tag{A.29}$$

because $A \cap \overline{A} = \varnothing$.

Theorem A.2. Complement Probability. *Let A be an event and $\overline{A}$ its complement in the sample space S. Then*

$$P(\overline{A}) = 1 - P(A) \tag{A.30}$$

Definition A.2. Conditional Probability. *Let A and B be events. Then the **conditional probability of** A **given that** B **has occurred**, read as the **probability of** A **given** B is*

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{A.31}$$

Example A.4. Dice, continued. Consider the sample space of example A.1. Let

$$E_1 = \text{Event that one die is less than or equal to 3} \tag{A.32}$$

$$E_2 = \text{Event that the roll doubles} \tag{A.33}$$

Then the probability that the roll is doubles given that one die is less than or equal to 3 is

$$P(E_2|E_1) = \frac{P(E_1 \cap E_2)}{P(E_1)} \tag{A.34}$$

Counting possibilities in the sample space in example A.1, $P(E_1) = 27/36 = 3/4$ (there are 27 events where one of the dice is less than or equal to 3). There are 3 events which are doubles where one of the dice is less than or equal to 3, so that $P(E_1 \cap E_2) = 3/36 = 1/12$. Thus

$$P(E_2|E_1) = \frac{3/36}{27/36} = \frac{1}{9} \quad \square \tag{A.35}$$

Definition A.3. Independent Events. *Let A and B be events. Then we say that A and B are **independent** if any of the following are true:*

$$P(B|A) = P(B) \text{ or} \tag{A.36}$$

$$P(A|B) = P(A) \text{ or} \tag{A.37}$$

$$P(A \cap B) = P(A)P(B) \tag{A.38}$$

Theorem A.3. Bayes' Rule. *Suppose that we can partition the sample space into non-overlapping events $E_1, E_2, \ldots, E_k$ such that $E_i \cap E_j = \varnothing$ for $i \neq j$ and*

$$S = \bigcap_{j=1}^{k} E_j \tag{A.39}$$

Then for any event $A \in S$

$$P(E_i|A) = \frac{P(A \cap E_i)}{P(A)} = \frac{P(A|E_i)P(E_i)}{\sum_{j=1}^{k} P(A|E_j)P(E_j)} \tag{A.40}$$

To see where the denominator in equation A.40 comes from, observe that from set theory we have

$$A = A \cap S = A \cap \left(\bigcap_{j=1}^{k} E_j \right) = \bigcup_{j=1}^{k} (A \cap E_j) \tag{A.41}$$

The denominator in equation A.40 follows by applying equation A.31 to each term on the right hand side of equation A.41.

Definition A.4. Fundamental Principle of Counting. *The number of possible outcomes of a combination of independent events is the product of the numbers of their individual outcomes.*

Example A.5. Multiple Choice Test. Suppose that you are taking a multiple choice test. Each question has four possible answers, labeled A, B, C, and D. There are 5 questions. Denote an answer sequences as the sequence of five answers that you give to the five questions, such as BADCA or ADDCB. The sample space for the test has $4^5 = 1024$ possible combinations of answer sequences. □

Definition A.5. Bernoulli Random Variable. *A **Bernoulli Random Variable** has two possible outcomes, which we call **success** and **failure**. If we denote the probability of success, in a single trial, by p then the probability of failure is $1 - p$.*

Theorem A.4. Bernouli Probability. *Let X be a Bernoulli Random Variable with a probability of success p. The the probability of k successes in n trials is given by*

$$P(X = k) = p^k (1 - p)^{n-k} \tag{A.42}$$

Example A.6. Coin Flipping. Suppose you flip a fair coin 5 times. A fair coin is one in which the probability of coming up heads is equal to the probability of coming up tails on any given flip. Then the probability of getting HHHTT (in that order) is

$$P(8) = 0.5^3 (1 - 0.5)^2 = 0.03125 \quad \square \tag{A.43}$$

Definition A.6. Combination. *A **combination** of k items from a set of n items is a selection of k items in which order does not matter.*

Theorem A.5. Combinations. *Suppose a set S contains n items. Then the number of combinations of k items that can be drawn from S, where $0 \leqslant k \leqslant n$, is given by*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 3 \cdot 2 \cdot 1} \tag{A.44}$$

The symbol $\binom{n}{k}$ can be read as "the number of combinations of n things chosen k at a time" or "n choose k" (for short), and is called the **binomial coefficient**. The notation ${}_nC_k$ is also sometimes used (for example, on the TI-84 calculator).

Example A.7. Poker Hands. A poker hand consists of 5 cards randomly assigned from a deck of 52 cards. This is a combination of 52 things taken 5 at a time:

$$ {}_{52}C_5 = \binom{52}{5} = \frac{52!}{5!47!} = 2,598,960 \tag{A.45} $$

Thus there are approximately two and a half million distinct poker hands. □

Combinations can be calculated in Python using either the **`binom`** or **`comb`** function in **`scipy.special`**

```
import scipy
print(scipy.special.binom(52,5))
print(scipy.special.comb(52,5))
```

```
2598960.0
2598960.0
```

The function **`itertools.combinations`** can be used to make a list of possible combinations. For example, the following will print the possible choices of 2 cards from a hand of 5.

```
import itertools
hand=["A-Hearts","Q-Spades","10-Diamonds","5-Clubs","4-Clubs"]
for cards in itertools.combinations(hand,2):
    print (cards)
```

```
('A-Hearts', 'Q-Spades')
('A-Hearts', '10-Diamonds')
('A-Hearts', '5-Clubs')
('A-Hearts', '4-Clubs')
('Q-Spades', '10-Diamonds')
('Q-Spades', '5-Clubs')
('Q-Spades', '4-Clubs')
('10-Diamonds', '5-Clubs')
('10-Diamonds', '4-Clubs')
('5-Clubs', '4-Clubs')
```

Definition A.7. Permutations. *A **permutation** of n items is a rearrangement of the items. A k-**permutation** of n items is an arrangement of k items selected from a list or set of n items. The notation ${}_nP_k$ is sometimes used to designate the number of k-permutations from a set of n items.*

Theorem A.6. Permutations. *There are $n!$ possible rearrangements or permutations of n items. There are*

$$ {}_nP_k = \frac{n!}{(n-k)!} = n(n-1)(n-2)\cdots(n-k+1) \tag{A.46} $$

possible k-permutations of n items.

Example A.8. Poker Hands (continued). Given any particular poker hand of 5 cards, they can rearranged ${}_5P_5 = 5! = 120$ different ways. If we consider the entire deck of cards, there are ${}_{52}P_5$ =52!/47!=311,875,200 possible permutations. □

The function **`scipy.special.perm(n,k)`** will return the total number of permutations ${}_nP_k$.

```
import scipy.special
print(scipy.special.perm(52,5))
print(scipy.special.perm(5,5))
```

```
311875200.0
120.0
```

The function **`itertools.permutations(L,k)`** will return all of of the ${}_nP_k$ permutations of the list **`L`**. For example, to list the different orders in which three different boys can stand in line,

```
import itertools
theboys=["Tom","Dick","Harry"]
list(itertools.permutations(theboys,3))
```

```
[('Tom', 'Dick', 'Harry'),
 ('Tom', 'Harry', 'Dick'),
 ('Dick', 'Tom', 'Harry'),
 ('Dick', 'Harry', 'Tom'),
 ('Harry', 'Tom', 'Dick'),
 ('Harry', 'Dick', 'Tom')]
```

Definition A.8. Binomial Distribution. *Suppose that an event has two possible outcomes, called success and failure, with probability p and $q = 1 - p$, respectively. If we perform n independent trials of the experiment then the probability of k successes (where $0 \leqslant k \leqslant n$) is given by*

$$ P(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{A.47} $$

where

$$ \binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{A.48} $$

Example A.9. Coin Flipping Revisited. In example A.6 we saw that the probability of getting 3 heads followed by 2 tails in 5 flips is 0.03125. The probability of getting exactly 3 heads out of 5 flips, in any order, is given by a binomial

$$P(3) = \binom{5}{3}(.5)^3(1-.5)^{(5-3)} = \frac{5!}{3!2!} \times (0.03125) = 0.3125 \quad \square \tag{A.49}$$

Example A.10. Multiple Choice Test (continued). If you answer the questions in the test in example A.5, what is the probability getting exactly 3 questions correct? This is a binomial outcome with 5 independent trials. Since there are four possible answers for each question, and only one of them is correct, we have $p = 1/4$. Thus

$$P(3) = \binom{5}{3}(1/4)^3(3/4)^2 = \frac{5!3^2}{2!3!4^5} \tag{A.50}$$

$$= \frac{(5 \cdot 4 \cdot 3 \cdot 2 \cdot 1) \cdot 9}{(2 \cdot 1) \cdot (3 \cdot 2 \cdot 1) \cdot 1024} \tag{A.51}$$

$$= \frac{90}{1024} = 0.088 \quad \square \tag{A.52}$$

The probability mass function for a Binomial random variable can be calculated using **`scipy.stats.binom.pdf(k,n,p)`**. For flipping a coin $n = 5$ times, and getting k heads, as in example A.9, the PMF is:

```
for k in range(0,6):
    print(k,round(scipy.stats.binom.pmf(k,5,.5),5))
```

```
0 0.03125
1 0.15625
2 0.3125
3 0.3125
4 0.15625
5 0.03125
```

For getting k correct answers out of 5 multiple choice questions with 4 choices to each questions as in example A.10,

```
for k in range(0,6):
    print(k,round(scipy.stats.binom.pmf(k,5,.25),5))
```

```
0 0.2373
1 0.39551
2 0.26367
3 0.08789
4 0.01465
5 0.00098
```

Definition A.9. Cumulative Distribution Function. *The **Cumulative Distribution Function (CDF)** a random variable X is a function $F(x)$ that gives the **probability** that $X < x$:*

$$F(x) = P(X < x) \tag{A.53}$$

The CDF is defined for both continuous and discrete random variables.

Theorem A.7. Properties of the CDF. *Let $F(x)$ be the CDF of a random variable. Then*

1. $\lim_{x \to -\infty} F(x) = 0$
2. $\lim_{x \to \infty} F(x) = 1$
3. *$F(x)$ is non decreasing: if $a < b$ then $F(a) \leqslant F(b)$.*

Example A.11. Multiple Choice Test (continued) . Suppose you need to get at least 60% correct to pass the multiple choice test in example A.5. Suppose that you guess at all the questions. What is the probability that you pass the test?
Since you need to get 60% you need to get at least 3 or more questions correct. This means you want to know

$$P(X \geqslant 3) = 1 - P(X < 2) = 1 - \sum_{k=0}^{2} \binom{5}{3} (1/4)^k (3/4)^{5-k} \tag{A.54}$$

$$= 1 - .896 = 0.104. \tag{A.55}$$

The term $P(X < 2)$ is the CDF for a binomial distribution with is implemented in software and easily calculated, e.g., with a TI-84 or in Python. Thus there is approximately 10% chance of passing by randomly guessing the answers. □

The CDF for a binomial distribution can be obtained from the Python **scipy** package using **scipy.stats.binom.cdf(k,n,p)**. The CDF for getting k correct answers out of 5 multiple choice questions with 4 choices to each questions as in example A.11,

```
for k in range(0,6):
    print(k,round(scipy.stats.binom.cdf(k,5,.25),5))
```

```
0 0.2373
1 0.63281
2 0.89648
3 0.98438
4 0.99902
5 1.0
```

We cannot define the probability that a continuous variable takes on a precise value; this is tantamount to measuring the length of a point. The best we can do is to measure the probability that that a continuous random variables falls between two values a and b. Let X be a continuous random variable with CDF $F(x)$.

Definition A.10. Probability Density Function. *Let X be a random variable with cumulative density function $F(x)$. Then the* ***Probability Density Function (PDF)*** *of X is defined by*

$$f(x) = F'(x) \tag{A.56}$$

The PDF is the generalization of the PMF to continuous random variables.

Theorem A.8. Properties of the PDF. *Let $f(x)$ be the probability distribution function of random variable. Then it has the following properties:*

1. *Non-negativity:* $f(x) \geqslant 0$
2. *All probability must sum to one:* $\displaystyle\int_{-\infty}^{\infty} f(x)\ dx = 1$
3. *The area under the curve of $f(x)$ and above the x axis on the interval $[a, b]$ is the probability* $P(a < x < b) = \displaystyle\int_{a}^{b} f(x)\ dx$

From equation A.56 and the fundamental theorem of integral calculus we also have the formula

$$F(x) = \int_{-\infty}^{x} f(z)dz \tag{A.57}$$

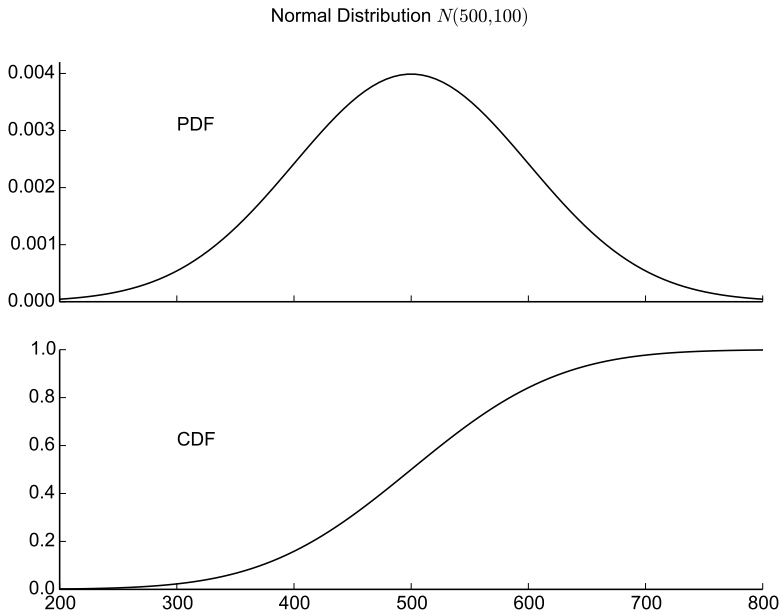
Unfortunately it is often not possible to solve the integral in equation A.57 analytically and so a numerical solver or software must be used.

Example A.12. Normal Distribution. A normal distribution has a probability density function given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \tag{A.58}$$

where μ and σ are positive, real valued parameters (constants), and x can be any real number. A population that has a normal distribution with parameters μ and σ is sometimes expressed as $N(\mu, \sigma)$. For example, the distribution of SAT scores is approximately $N(500, 100)$. This function is illustrated in the top frame of figure A.1.

Figure A.1: Probability density function (PDF) and cumulative density function (CDF) for a normal distribution with mean $\mu = 500$ and standard deviation $\sigma = 100$



Example A.13. Area under a normal distribution. It is not possible, in general, to find a formula for the cumulative distribution of a normal distribution. It is possible, however, to compute some definite integrals, such as proving that the total area under the curve is equal to 1. Let

$$I = \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} dx \tag{A.59}$$

With the substitution $z = x - \mu$,

$$I = \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-z^2/(2\sigma^2)} dz \tag{A.60}$$

With a second substitution of $u = z/(\sigma\sqrt{2})$, so that $du = dz/(\sigma\sqrt{2})$,

$$I = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-u^2} du \tag{A.61}$$

Therefore

$$I^2 = \left(dx\right)\left(\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-y^2} dy\right) \tag{A.62}$$

$$= \frac{1}{\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2+y^2)} dxdy \tag{A.63}$$

Changing variables to polar coordinates, with $r^2 = x^2 + y^2$, the area element becomes $rdrd\theta$ and we have

$$I^2 = \frac{1}{\pi} \int_0^{\infty} \int_0^{2\pi} re^{-r^2} drd\theta = 2 \int_0^{\infty} re^{-r^2} dr = -\int_0^{\infty} e^{-z} dz = 1 \tag{A.64}$$

In the last step we used the substitution $z = r^2$ hence $dz = 2dr$. Hence $I = 1$. This proves that the total area under a normal curve is equal to 1. □

Definition A.11. Expected Value. *Let X be a continuous random variable with a probability distribution function $f(x)$. Then its* ***expected value*** *is defined as*

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx \tag{A.65}$$

Furthermore, we define the ***expectation*** *or* ***expected value of a function*** *$u(x)$ to be*

$$E[u(X)] = \int_{-\infty}^{\infty} u(x)f(x)dx \tag{A.66}$$

There are several immediate consequences to this definition. In particular, the **expected value is linear**.

1. The expected value of a constant is a constant:

$$E(c) = c \tag{A.67}$$

2. The expectation is a linear operator: for any two functions f and g, and any two constants a and b, if X is a random variable, then

$$E[af(X) + g(X)] = aE[f(x)] + bE[g(X)] \tag{A.68}$$

The mean and variance examples of **moments**, which are special cases of expectations.

Definition A.12. Moment. *Let X be a random variable. Then the kth* ***moment about the origin*** *is defined as*

$$\mu'_k = E(X^k) \tag{A.69}$$

and the kth ***moment about the mean*** *is*

$$\mu_k = E((X - \mu)^k) \tag{A.70}$$

where μ is the mean. The ***mean*** *is the first moment about the origin, and the* ***variance*** *is the the second moment about the mean.*

Example A.14. Mean of Normal Distribution. To see that the mean of a normal distribution is μ, we make a change of variables $z = x - \mu$. Then

$$E[x] = \int_{-\infty}^{\infty} \frac{x}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} dx \tag{A.71}$$

$$= \mu \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-z^2/(2\sigma^2)} dx + \int_{-\infty}^{\infty} \frac{z}{\sigma\sqrt{2\pi}} e^{-z^2/(2\sigma^2)} dx \tag{A.72}$$

The first integral is equal to 1 by example A.13. The second integral is equal to zero because the integral is the product of an even function and an odd function. Hence $E[x] = \mu$. $\square$

Example A.15. Variance of a Normal Distribution. Since the variance is the second moment about the mean we need to calculate

$$E[(x-\mu)^2] = \int_{-\infty}^{\infty} \frac{(x-\mu)^2}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} dx \tag{A.73}$$

Make the substitution $z = (x - \mu)/(\sigma\sqrt{2})$.

$$E[(x-\mu)^2] = \int_{-\infty}^{\infty} \frac{2\sigma^2 z^2}{\sigma\sqrt{2\pi}} e^{-z^2} (\sigma\sqrt{2} dz) = \frac{2\sigma^2}{\sqrt{\pi}} \int_{-\infty}^{\infty} z^2 e^{-z^2} dz \tag{A.74}$$

Use the fact that

$$\int_{-\infty}^{\infty} e^{-\alpha z^2} dz = \sqrt{\frac{\pi}{\alpha}} \tag{A.75}$$

Hence

$$\int_{-\infty}^{\infty} z^2 e^{-\alpha z^2} dz = -\frac{\partial}{\partial\alpha} \int_{-\infty}^{\infty} e^{-\alpha z^2} dz = -\frac{\partial}{\partial\alpha} \sqrt{\frac{\pi}{\alpha}} = \frac{1}{2}\sqrt{\frac{\pi}{\alpha^3}} \tag{A.76}$$

Setting $\alpha = 1$ gives

$$\int_{-\infty}^{\infty} z^2 e^{-z^2} dz = \frac{\sqrt{\pi}}{2} \tag{A.77}$$

Substitution into equation A.74 gives

$$E[(x-\mu)^2] = \sigma^2 \quad \square \tag{A.78}$$

The **variance** operator is sometimes written as $\mathrm{var}[X]$, meaning *variance of* X. But

$$\mathrm{var}[X] = E[(X-\mu)^2] \tag{A.79}$$

$$= E[X^2 - 2\mu X + \mu^2] \tag{A.80}$$

$$= E[X^2] - 2\mu E[X] + \mu^2 \tag{A.81}$$

In the last line we have used the fact the E is linear in the middle term, and that the expectation of a constant is itself in the final term. Since $E[X] = \mu$, this gives the following result.

Theorem A.9. Variance/Expectation Formulas. *Let X be a random variable with variance* $\mathrm{var}[X]$. *Then*

$$\mathrm{var}[X] = E[X^2] - E[X]^2 \tag{A.82}$$

or equivalently

$$E[X^2] = \mathrm{var}[X] + E[X]^2 \tag{A.83}$$

The probability density function for a normal distribution with mean **mu** and standard deviation **sigma** can be found in Python using

```
scipy.stats.norm.pdf(x,mu,sigma)
```

The standard deviation is defined as $\sigma = \sqrt{\sigma^2}$ where σ^2 is the variance of the distribution. The corresponding cumulative density function is

```
scipy.stats.norm.cdf(x,mu,sigma)
```

An **array** of **n** random numbers from a normal distribution with mean **mu** and standard deviation **sigma** can be produced either with **scipy.stats.norm.rvs (mu,sigma,n)**, as in

```
import scipy
scipy.stats.norm.rvs(500,100,20)
```

```
array([293.79034369, 629.68515718, 477.44796018, 302.12916214,
       498.11534449, 540.34168405, 321.35409886, 730.98979059,
       282.79885082, 496.1819826 , 497.45128964, 506.05432368,
       670.96282882, 478.74027881, 219.77111424, 603.4961768 ,
       522.36658982, 590.96318169, 491.23851556, 614.67746531])
```

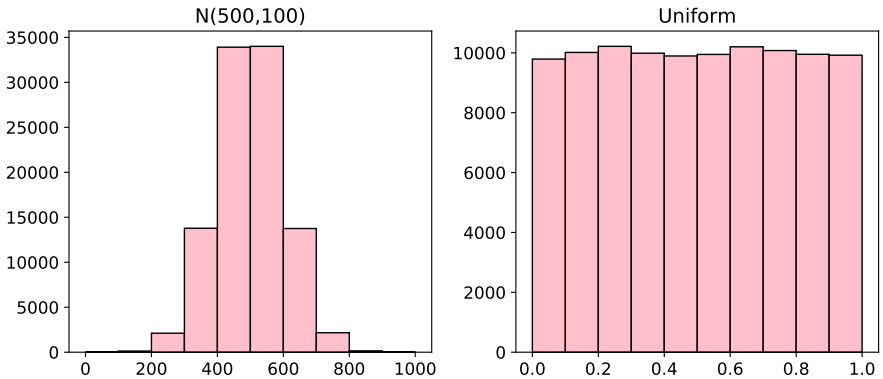
or with **numpy.random.normal(mu,sigma,n)**,

```
import numpy as np
np.random.normal(500,100,20)
```

```
array([769.32657823, 547.9489816 , 382.95918352, 602.15031874,
       619.75549511, 595.69897894, 473.46899878, 439.90433231,
       462.47114243, 554.75356831, 535.00745288, 668.35261561,
       552.69775135, 395.60449771, 309.71960172, 656.11477523,
       538.56600115, 504.58103886, 476.18629725, 339.68431345])
```

A single random number from a uniform distribution on [0,1] can be generated by **random.random()**. To generate an array of **n** uniform random numbers on [0,1] use **np.random.random(n)**. The following code will produce histograms such as those illustrated in figure A.2.

Figure A.2: Histograms random numbers generated by Python. Left; 100,000 random numbers in an N(500,100) normal distribution. Right: 100,000 random numbers in a uniform distribution on [0,1]. The y-axis in each plot gives the frequency count.




```
fig,ax=plt.subplots(nrows=1,ncols=2)
ax[0].hist(np.random.normal(500,100,100000),
    edgecolor="k",color="pink",bins=np.linspace(0,1000,11));
ax[1].hist(np.random.random(100000),
        edgecolor="k",color="pink", bins=np.linspace(0,1,11));
```

Definition A.13. Skewness. *The **skewness** of a distribution is the third moment about the mean:*

$$Skewness[X] = E\left[\left(\frac{x-\mu}{\sigma}\right)^3\right] \tag{A.84}$$

The skewness measures the deviation of a distribution from a central tendency. A perfectly symmetric distribution has zero skewness. In a negatively skewed distribution, or left-skewed distribution, there is a longer tail to the left (smaller x-values); the mean will be less than the median. In a positively skewed distribution, or right-skewed distribution, there is a longer tail to the right; the mean is larger than the median. In a symmetric distribution, such as a normal distribution, the median and mean are equal.

Definition A.14. Kurtosis. *The **Kurtosis** of a distribution is the fourth moment about the mean normalized to the square of the variance:*

$$Kurtosis[X] = \frac{E[(X-\mu)^4]}{(E[(X-\mu)^2])^2} \tag{A.85}$$

With a little manipulation,[1]

$$\text{Kurtosis}[X] = \frac{E[(X-\mu)^4]}{(\sigma^2)^2} = \frac{E[(\sigma z)^4]}{\sigma^4} \tag{A.86}$$

where $z = \dfrac{x-\mu}{\sigma}$ and $\sigma^2 = E[(X-\mu)^2]$ is the variance. Since the expectation is a linear operator,

$$\text{Kurtosis}[X] = E[Z^4] \tag{A.87}$$

Hence by equation A.83,

$$\text{Kurtosis}[X] = \text{var}[Z^2] + E[Z^2]^2 \tag{A.88}$$

But since the expectation is linear,

$$E[Z^2] = E\left[\left(\frac{X-\mu}{\sigma}\right)^2\right] = \frac{1}{\sigma^2}E[(x-\mu)^2] = \frac{\sigma^2}{\sigma^2} = 1 \tag{A.89}$$

Therefore

$$\text{Kurtosis}[X] = \text{var}[Z^2] + 1 \tag{A.90}$$

Kurtosis measures dispersion about the values $\mu \pm \sigma$. A high kurtosis can occur in two ways: high density near the mean; or high density in each of the two tails.[2] Centrally peaked distributions that are very "pointy," therefore, tend to have high kurtosis, while widely spread out or flattened distributions tend to have low or negative kurtosis.

The kurtosis of a normal distribution is 3.0. The **Fisher Kurtosis** is sometimes defined as

$$k_{\text{Fisher}} = \frac{\mu_4}{\sigma^4} - 3 \tag{A.91}$$

while the **Pearson Kurtosis** is defined by to definition A.14

$$k_{\text{Pearson}} = \frac{\mu_4}{\sigma^4} \tag{A.92}$$

Kurtosis can be calculated using **`scipy.stats.kurtosis`**.[3] The Python implementation will calculate either Pearson or Fisher kurtosis and works on either the rows or the columns of a **`numpy array`**.

[1]This derivation is based on JJA Moors (1986) "The Meaning of Kurtosis: Darlington Reexamined." *The American Statistician*, **40**(4):283-284.
[2]see Moors, *op.cit.*
[3]For complete documentation see `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kurtosis.html`.

Definition A.15. Sample Mean. *The mean $\overline{x}$ of a random sample $\{x_0, x_1, \ldots, x_{n-1}\}$ of n individuals taken from any population is given by*

$$\overline{x} = \frac{1}{n}\sum x_i \tag{A.93}$$

The expectation of $\overline{x}$ is μ, i.e., $E(\overline{x}) = \mu$.

Definition A.16. Sample Variance. *The variance S^2 of a random sample $\{x_0, x_1, \ldots, x_{n-1}\}$ of n individuals taken from any population is given by*

$$S^2 = \frac{1}{n-1}\sum (x_i - \overline{x})^2 \tag{A.94}$$

where $\overline{x}$ is the sample mean. The expectation of S is σ^2, i.e., $E(S^2) = \sigma^2$. We define the sample standard deviation of as

$$S = \sqrt{S^2} = \sqrt{\frac{1}{n-1}\sum (x_i - \overline{x})^2} \tag{A.95}$$

Everything we have discussed have multidimensional generalizations.

Definition A.17. Covariance. *Let $\mathbf{X}$ is a feature matrix with n features, such that μ_i and σ_i^2 are the mean and variance of the ith feature. Then the* ***covariance matrix*** $\boldsymbol{\Sigma}$ *has elements*

$$\boldsymbol{\Sigma}_{ij} = E[(\mathbf{X}_i - \mu_i)(\mathbf{X}_j - \mu_j)] \tag{A.96}$$

The diagonal elements of the covariance matrix are the variances σ_i^2 of the individual features.

The covariance is closely related to the correlation matrix $\mathbf{R}$.

Definition A.18. Correlation. *Let $\boldsymbol{\Sigma}$ be the covariance of a feature matrix $\mathbf{X}$, and further let σ_i^2 be the the variances of the ith feature. The we define the* ***correlation matrix*** *to be the matrix with elements*

$$\mathbf{R}_{ij} = \frac{\boldsymbol{\Sigma}_{ij}}{\sigma_i \sigma_j} \tag{A.97}$$

The correlation matrix is then the matrix of Pearson Correlations ρ_{ij}, where

$$\rho_{ij} = \frac{1}{\sigma_i \sigma_j} E[(\mathbf{X}_i - \mu_i)(\mathbf{X}_j - \mu_j)] \tag{A.98}$$

The correlation and covariance of a feature matrix can be calculated in Python using **`numpy.cov`** and **`numpy.corrcoef`**. Each returns a matrix and assumes that the features are given by the rows. Since **`pandas DataFrames`** typically give the features as columns rather than rows, it is necessary to either set the parameter **`rowvar=False`** or to take the transpose of the feature matrix before calculating these.

Definition A.19. Multivariate Normal. *Let* $\mathbf{X}$ *be an* n*-dimensional random variable with mean* $\boldsymbol{\mu}$ *and covariance* $\boldsymbol{\Sigma}$*. Then if* $\mathbf{X}$ *is normally distributed, its probability distribution function (pdf) is*

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \det \boldsymbol{\Sigma}}} e^{(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})/2} \tag{A.99}$$

Random values in a multivariate normal can be generated using the function **`multivariate_normal`** from **`numpy.random`** as illustrated on page 176.

Appendix B

Review of Linear Algebra

This appendix provides a brief of review of some basic concepts from linear algebra.[1]

Definition B.1. Vector. *A **vector** $\mathbf{v}$ is an object with a **magnitude** and **direction**.*

A vector, of and by itself, is not fixed to any particular location in space. There is a common visualization (that we will use) in which vectors are represented by arrows in $\mathbb{R}^n$. If we place the tail of the arrow at the origin then its head will be at a point $\mathbf{P}$. We represent $\mathbf{P}$ its coordinates. In 3 dimensions, for example, we typically represent the coordinates as a 3-tuple of distances along the x, y, and z axes.

$$\mathbf{P} = (x, y, z) \tag{B.1}$$

Vectors are often represented by the coordinates of the point $\mathbf{P}$, but as column vectors, rather than tuples.

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{B.2}$$

We will sometimes write the vector in its row form as

$$\mathbf{v}^{\mathrm{T}} = \begin{bmatrix} x & y & z \end{bmatrix} \tag{B.3}$$

The superscript T indicates that the row form gives the **vector transpose**. Because there is a one-to-one relationship between the set of all vectors in $\mathbb{R}^n$

[1]For a more detailed treatment, see Strang, G. (2006) *Linear Algebra and Its Applications.* Brooks Cole, and Franklin, J. (1968) *Matrix Theory.* Prentice Hall (available in Dover Reprint (2000).

and the points in $\mathbb{R}^n$, the row and column representations are often used interchangeably, although (technically) they represent different objects (one is a row matrix, the other is a column matrix). You will often have to determine from the context which one is intended as the transpose notation is frequently omitted. The word **transpose** means flip all the rows and columns of a matrix.

In higher dimensions the symbols $x, y, z, \ldots$ can be replaced with indices, such as $x_0, x_1, \ldots, x_{n-1}$. The the vector is written as

$$\mathbf{v}^{\mathrm{T}} = \begin{bmatrix} v_0 & v_1 & \cdots & v_{n-1} \end{bmatrix} \tag{B.4}$$

Mathematically, the indices can begin and end at any value.

Python vectors are most easily represented using single dimensional **numpy arrays**. When printed, a single dimensional array looks like a Python **list** without the commas. The transpose is obtained using the **array** method **T**. Row vectors are defined like lists, and **numpy** does not really distinguish between rows and columns of one-dimensional arrays.

```
import numpy as np
v=np.array([4,6,-7])
print(v)
```

```
[ 4  6 -7]
```

```
v_transpose=v.T
print(v_transpose)
```

```
[ 4  6 -7]
```

If it is important for you to easily distinguish between the row and column vectors in your code you should define your vectors as column matrices. Column vectors are printed as vertical columns. However, some functions in Python that are intended for vectors will not work as expected if you define a vector as a matrix. To use these functions you need to unwind the matrix using the **ravel** method.

```
w=np.array([[3,4,-5]])
print(w)
```

```
[[3 4 -5]]
```

```
w_transpose=w.T
print(w_transpose)
```

```
[[3]
 [4]
 [-5]]
```

The magnitude of **v** is defined as the Euclidean length of the tip-to-tail line segment that corresponds to the vector. In other words, if you place the tail of the vector at the origin, the magnitude of the **v** is the distance from the origin to the head of the vector.

Definition B.2. Vector Magnitude. *The **magnitude**, **absolute value**, **Euclidean Length**, **length**, or **L-2 Norm** of a vector* $\mathbf{v} = [v_1, \ldots, v_n]^T$ *is given by the positive square root*

$$v = |\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \tag{B.5}$$

Definition B.3. P-Norm. *The **p-norm** or L-p norm of a vector* **v** *is given by*

$$|\mathbf{v}|_p = \left[|x_0|^p + |x_1|^p + \cdots + |x_{n-1}|^p \right]^{1/p} \tag{B.6}$$

When $p = 2$, the p-norm is sometimes called the Euclidean Norm. When $p = 1$, the p-norm returns the sum of the absolute values of the vector components, and is called the taxicab norm. In the limit as $p \to \infty$, the p-norm becomes the **sup**-norm; it picks out the component with the largest magnitude.

Example. Let $\mathbf{v} = [3, 4, -5]^{\mathrm{T}}$. Then

$$|\mathbf{v}|_1 = |3| + |4| + |-5| = 12 \tag{B.7}$$

$$|\mathbf{v}|_2 = |3|^2 + |4|^2 + |-5|^2 = 50 \tag{B.8}$$

$$|\mathbf{v}|_\infty = \max\{|3|, |4|, |-5|\} = 5 \quad \square \tag{B.9}$$

The vector norm is calculated in Python with the function **norm** in **numpy.linalg**. By default, the Euclidean norm is returned. Any p-norm can be calculated with the option **ord**.

```
u=w.ravel()
print("The norm is",np.linalg.norm(u))
print("The 1-norm is", np.linalg.norm(u,ord=1))
print("The 4-norm is",np.linalg.norm(u,ord=4))
print("The inf-norm is",np.linalg.norm(u,ord=float("inf")))
```

```
The norm is 7.0710678118654755
The 1-norm is 12.0
The 4-norm is 5.569212227823757
The inf-norm is 5.0
```

We can visualize the angle between two vectors by placing their tails at the same point in space. Since the two arrows will fall in a single plane (or $n-1$ dimensional subspace), we define the angle between the vectors as the angle between the arrows in that plane.

Definition B.4. Angle Between Vectors. *If we place the vectors* $\mathbf{v}$ *and* $\mathbf{w}$ *at the origin so that their tips are at points* $\mathbf{P}$ *and* $\mathbf{Q}$*, respectively, then the angle between* $\mathbf{v}$ *and* $\mathbf{w}$ *is defined to be the angle between the line segments* $\overline{\mathbf{OP}}$ *and* $\overline{\mathbf{OQ}}$*. The angle* θ *between two vectors is then defined in terms of the normal trigonometric cosine function.*

$$\mathbf{v} \cdot \mathbf{w} = |\mathbf{v}||\mathbf{w}| \cos\theta \tag{B.10}$$

We will define angles in higher dimensions in terms of the dot product.

Definition B.5. Dot Product. *Let* $\mathbf{v}$ *and* $\mathbf{w}$ *be* n*-dimensional vectors. Their* ***dot product*** *is given by*

$$\mathbf{v} \cdot \mathbf{w} = v_0 w_0 + v_1 w_1 + v_2 w_2 + \cdots + v_{n-1} w_{n-1} \tag{B.11}$$

Therefore the angle between two vectors in $\mathbb{R}^n$ is determined by

$$|\mathbf{v}||\mathbf{w}| \cos\theta = \sum v_i w_i \tag{B.12}$$

Example. Let $\mathbf{v} = [1, 2, 4, -3]^{\mathrm{T}}$ and $\mathbf{w} = [3, 2, 1, -2]^{\mathrm{T}}$. Then

$$\cos\theta = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{17}{\sqrt{30}\sqrt{18}} \approx 0.7315 \tag{B.13}$$

Thus the angle between the two vectors is approximately 43°. $\square$

The **`array`** method **`dot`** is used to compute the dot product between vectors.

```
v=np.array([1,2,4,-3])
w=np.array([3,2,1,-2])
v.dot(w)
```

```
17
```

```
nv=np.linalg.norm(v)
nw=np.linalg.norm(w)
costheta = v.dot(w)/(nv*nw)
print("Cosine(angle)=",costheta)
from math import pi, acos
theta = (180/pi)*acos(costheta)
print(theta, "Degrees")
```

```
Cosine(angle)= 0.7315635209502899
42.982369831394024 Degrees
```

Trigonometric functions and π are not included in Python by default and need to be added by importing the **`math`** package.

Definition B.6. Vector Addition. *Let* **`v`** *and* **`w`** *be vectors. Their sum is given by*

$$\mathbf{x} = \begin{bmatrix} v_1 + w_1 & v_2 + w_2 & \cdots & v_n + w_n \end{bmatrix}^T \tag{B.14}$$

Vector addition works as you would expect it to using **`numpy array`**'s. This is in contrast to Python **`list`**'s where there addition sign represents list concatenation. When used between arrays, the addition sign is always interpreted as vector addition, and is never interpreted as concatenation.

```
print(v+w)
```

```
[ 4  4  5 -5]
```

Definition B.7. Scalar Product. *Let* $k \in \mathbb{R}$, *and* $\mathbf{v} \in \mathbb{R}^n$ *be a vector. Then their* ***scalar product***

$$k\mathbf{v} = \begin{bmatrix} kv_1 & kv_2 & \cdots & kv_n \end{bmatrix}^T \tag{B.15}$$

Like vector addition, scalar multiplication also works between a Python scalar and a **`numpy array`**. This is also in contrast to multiplication between a Python **`list`** and a Python **`int`**, where the multiplication symbol represents list repetition and concatenation. When placed between a Python scalar (**`int`**, **`float`**, or **`complex`**), the symbol **`*`** is always interpreted as scalar multiplication.

```
print{3w}
```

```
[ 9  6  3 -6]
```

A set is said to be **closed** under an operation if you can perform that operation on any combination of set elements and still get another element of the original set. Since any sum of vectors is a vector, the set of all vectors is said to be closed under vector addition. Similarly, since scalar multiplication always produces another vector, the set of all vectors is closed under scalar multiplication. This is called the closure property of vector addition and scalar multiplication.

Theorem B.1. Closure. *The set of all vectors in* $\mathbb{R}^n$ *is closed under vector addition and scalar multiplication.*

Definition B.8. Linear Dependence. *The vectors* $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ *are said to be* ***linearly dependent*** *if there exist numbers* $a_1, a_2, \ldots, a_n$*, not all zero, such that*

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = 0 \tag{B.16}$$

If no such numbers exist the vectors are said to be ***linearly independent****.*

Definition B.9. Spanning Set. *A set of vectors* $\{\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-1}\}$ *is said to* ***span*** *the vector space* $\mathcal{V}$ *if for every vector* $\mathbf{v} \in \mathcal{V}$ *there exists some linear combination*

$$\mathbf{v} = a_0\mathbf{e}_0 + a_1\mathbf{e}_1 + \cdots + a_{n-1}\mathbf{e}_{n-1} \tag{B.17}$$

Definition B.10. Basis Set. *A set of vectors* $\{\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-1}\}$ *is called a* ***basis*** *of the vector space* $\mathcal{V}$ *if it is (a) linearly independent; and (b) spans* $\mathcal{V}$*.*

In three dimensions, the set of **standard basis vectors** is written as

$$\mathbf{i} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^{\mathrm{T}} \tag{B.18}$$

$$\mathbf{j} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^{\mathrm{T}} \tag{B.19}$$

$$\mathbf{k} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^{\mathrm{T}} \tag{B.20}$$

We can write any vector $\mathbf{v}$ as a combination of basis vectors:

$$\mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ v_1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ v_2 \end{bmatrix} = v_0\mathbf{i} + v_1\mathbf{j} + v_2\mathbf{k} \tag{B.21}$$

In higher dimensions we typically use the notation $\mathbf{e}_1$, $\mathbf{e}_2$, ... for the basis vectors, where

$$\mathbf{e}_0 = [1, 0, 0, \ldots, 0] \tag{B.22}$$

$$\mathbf{e}_1 = [0, 1, 0, \ldots, 0] \tag{B.23}$$

$$\vdots$$

$$\mathbf{e}_{n-1} = [0, 0, 0, \ldots, 1] \tag{B.24}$$

With this notation an n-dimensional vector can be written as

$$\mathbf{v} = v_0\mathbf{e}_0 + v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots + v_{n-1}\mathbf{e}_{n-1} \tag{B.25}$$

A matrix is a bunch of numbers written as a grid in a box. Individual elements are referenced by row and column, where the row index comes first and the column index comes second.

Definition B.11. Matrix. *An $m \times n$ (read as "m by n") **matrix** $\mathbf{A}$ is a rectangular array of number with m rows and n columns*

If we denote the number in the ith row and j^{th} column as a_{ij} then we can write a matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ a_{20} & a_{21} & & a_{2,n-1} \\ \vdots & & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{B.26}$$

We will sometimes denote the matrix $\mathbf{A}$ by $[a_{ij}]$. (Here we have followed the Python notation, as usual, of starting all indices at zero. Most mathematics books number matrices starting from 1, so that the upper left-hand corner has an index 1,1, and the lower right hand corner will have an index m, n.)

Definition B.12. Transpose. *The **transpose** $\mathbf{A}^T$ of the matrix $\mathbf{A}$ is the matrix obtained by interchanging the row and column indices.*

If $\mathbf{A}$ is as given above, then

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} a_{00} & a_{10} & \cdots & a_{m-1,0} \\ a_{01} & a_{11} & \cdots & a_{m-1,1} \\ a_{02} & a_{12} & & a_{m-1,2} \\ \vdots & & & \vdots \\ a_{0,n-1} & a_{1,n-1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{B.27}$$

Note that $\left(\mathbf{A}^{\mathrm{T}}\right)_{ij} = \mathbf{A}_{ji}$

The transpose of an $m \times n$ matrix is an $n \times m$ matrix.

```
M=np.array([[1,2],[3,4]])
print("Matrix:")
print(M)
print("Transpose:")
print(M.T)
```

```
Matrix:
[[1 2]
 [3 4]]
Transpose:
[[1 3]
 [2 4]]
```

```
N=np.array([[3,4,5],[6,7,8]])
print("Matrix:")
print(N)
print("Transpose:")
print(N.T)
```

```
Matrix:
[[3 4 5]
 [6 7 8]]
Transpose:
[[3 6]
 [4 7]
 [5 8]]
```

Definition B.13. Matrix Addition. *Let* $\mathbf{A} = [a]_{ij}$ *and* $\mathbf{B} = [b]_{ij}$ *be two matrices of the same size. Then their sum is obtained by adding corresponding elements.*

$$\begin{bmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \cdots \\ \vdots & & \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} & \cdots \\ b_{10} & b_{11} & \cdots \\ \vdots & & \end{bmatrix} = \begin{bmatrix} a_{00}+b_{00} & b_{01}+b_{01} & \cdots \\ a_{10}+b_{10} & a_{11}+b_{11} & \cdots \\ \vdots & & \end{bmatrix} \tag{B.28}$$

Matrices that have different sizes cannot be added.

Definition B.14. Square Matrix. *A* ***square matrix*** *is any matrix with the same number of rows as columns. The* ***order*** *of the square matrix is the number of rows (or columns).*

Definition B.15. Submatrix. *A* ***submatrix*** *of* $\mathbf{A}$ *is the matrix* $\mathbf{A}$ *with one (or more) rows and/or one (or more) columns deleted.*

Definition B.16. Determinant. *The* ***determinant*** $\det \mathbf{A}$ ***of a square*** $n \times n$ ***matrix*** $\mathbf{A}$ *is calculated as follows.*

1. *If* $n = 1$ *then* $\mathbf{A} = [a]$ *and* $\det \mathbf{A} = a$.

2. *If* $n \geqslant 2$ *then*

$$\det \mathbf{A} = \sum_{i=1}^{n} a_{ki}(-1)^{i+k} \det(\mathbf{A}'_{ik}) \tag{B.29}$$

where $\mathbf{A}'_{ik}$ *is the submatrix of* A *with the* i^{th} *row and* k^{th} *column deleted. The choice of which* k *does not matter because the result will be the same.*

The determinant is sometimes written as

$$\det \mathbf{A} = \begin{vmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \cdots \\ \vdots & & \end{vmatrix} \tag{B.30}$$

In particular, for a 2×2 matrix,

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{B.31}$$

and for a 3×3 matrix,

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} = A \begin{vmatrix} E & F \\ H & I \end{vmatrix} - B \begin{vmatrix} D & F \\ G & I \end{vmatrix} + C \begin{vmatrix} D & E \\ G & H \end{vmatrix} \tag{B.32}$$

Example. Find the determinant of

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 3 & 1 \end{bmatrix} \tag{B.33}$$

Then following equation B.32,

$$\det \mathbf{A} = \begin{vmatrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 3 & 1 \end{vmatrix} = (1) \begin{vmatrix} 5 & 0 \\ 3 & 1 \end{vmatrix} - 0 \begin{vmatrix} 4 & 0 \\ 0 & 1 \end{vmatrix} + 3 \begin{vmatrix} 4 & 5 \\ 0 & 3 \end{vmatrix} = 41 \quad \square \tag{B.34}$$

Determinants of a matrix can be found using **`det`** in the **`numpy.linalg`** package.

```
A=np.array([[1,0,3],[4,5,0],[0,3,1]])
np.linalg.det(A)
```

```
41.00000000000001
```

The long string of 0's ending with a 1 is due to rounding in the least significant bit of the calculation.

A important property of a matrix is whether or not it is singular. A matrix is singular if and only if its determinant is zero. The matrix has an inverse (discussed below) if and only if it is nonsingular. Invertibile (and hence non-singular) matrices are important because they represent the coefficients of linear systems that can be solved. Linear systems of with singular matrices do not have unique solutions, and may not have any solutions at all.

Definition B.17. Singular Matrix. *A square matrix* $\mathbf{A}$ *is said to be* ***singular*** *if* $\det \mathbf{A} = 0$*, and* ***non-singular*** *if* $\det \mathbf{A} \neq 0$*.*

Theorem B.2. Determinant of a Non-singular Matrix. *The* n *columns (or rows) of an* $n \times n$ *square matrix* $\mathbf{A}$ *are linearly independent if and only if* $\det \mathbf{A} \neq 0$*.*

Definition B.18. Matrix Multiplication. *The product of an* $m \times r$ *matrix* $\mathbf{A}$ *and an* $r \times n$ *matrix* $\mathbf{B}$ *is the* $m \times n$ *matrix* $\mathbf{AB}$ *with elements*

$$\mathbf{AB}_{ij} = \sum_{k=0}^{r-1} a_{ik} b_{kj} = \sum_{k=0}^{r-1} \mathbf{a}_i \cdot \mathbf{b}_j \tag{B.35}$$

where $\mathbf{a}_i$ *is the* i^{th} *row of* $\mathbf{A}$ *and* $\mathbf{b}_j$ *is the* j^{th} *column of* $\mathbf{B}$*.*

Example. the product of a 2×3 and a 3×2 matrix is a 2×2 matrix.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 8 & 9 \\ 10 & 11 \\ 12 & 13 \end{bmatrix} = \begin{bmatrix} (1,2,3) \cdot (8,10,12) & (1,2,3) \cdot (9,11,13) \\ (4,5,6) \cdot (8,10,12) & (4,5,6) \cdot (9,11,13) \end{bmatrix} \tag{B.36}$$

$$= \begin{bmatrix} 64 & 70 \\ 156 & 169 \end{bmatrix} \quad \square \tag{B.37}$$

If we reverse the order of multiplication, the product $\mathbf{BA}$ multiplies a 3×2 matrix by a 2×3 resulting in a 3×3 matrix. In Python, matrix products are computed using the **`array dot`** method.

```
A=np.array([[1,2,3],[4,5,6]])
B=np.array([[8,9],[10,11],[12,13]])
C=A.dot(B)
print(C)
```

```
[[ 64  70]
 [154 169]]
```

```
D=B.dot(A)
print(D)
```

```
[[ 44  61  78]
 [ 54  75  96]
 [ 64  89 114]]
```

Matrix multiplication is not commutative: we cannot reverse the order of operations. Even though we can calculate $\mathbf{AB}$, the operation $\mathbf{BA}$ might not even be defined, and when it is defined, the result is not necessarily the same. When

we multiply matrices the dimensions must match. In the example, we multiplied a $[2 \times 3] \times [3 \times 2]$ matrix. The middle dimension here matched (3) so we are able to multiply. If the middle dimensions do not match, we cannot multiple. Thus, for example, we cannot multiply a $[2 \times 3]$ matrix by itself because $[2 \times 3] \times [2 \times 3]$ does not nave matching middle dimensions.

Theorem B.3. Determinant of Product. *If* $\mathbf{A}$ *and* $\mathbf{B}$ *are square matrices then*

$$\det \mathbf{AB} = (\det \mathbf{A})(\det \mathbf{B}) \tag{B.38}$$

Definition B.19. Main Diagonal and Diagonal Matrices. *The* ***main diagonal*** *of a square matrix* $\mathbf{A}$ *is the sequence of numbers* $a_{11}, a_{22}, \ldots, a_{nn}$. *A square matrix is said to be* ***a diagonal matrix*** *if its only non-zero elements lie on its main diagonal.*

An identity matrix of degree n is a special $n \times n$ diagonal matrix that has 1's on its main diagonal and zeros everywhere else.

Definition B.20. Identity Matrix. *The* $n \times n$ ***identity matrix*** *is the matrix* $\mathbf{I}$ *that satisfies the equation*

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A} \tag{B.39}$$

for all $n \times n$ *matrices* $\mathbf{A}$.

The identity matrix is a diagonal matrix with 1's in the diagonal and zero's everywhere else. In general we use the same symbol $\mathbf{I}$ regardless of the dimension of $\mathbf{I}$, so its dimension will have to be determined from the context. If it is necessary to distinguish between identity matrices, a subscript is sometimes used, e.g., $\mathbf{I}_m$ vs $\mathbf{I}_n$ to distinguish the $m \times m$ identity from the $n \times n$ identity matrix.

Definition B.21. Matrix Inverse. *A square matrix* $\mathbf{A}$ *is said to be* ***invertible*** *if there exists a matrix* $\mathbf{A}^{-1}$, *called the* ***inverse*** *of* $\mathbf{A}$, *such that*

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \tag{B.40}$$

The inverse of a matrix can be found with **`np.linalg.inv`**.

```
IP = np.linalg.inv(P)
print("Matrix:")
print(P)
print("Inverse:")
print(IP)
```

```
Matrix:
[[ 1  2  3]
 [-4  5 -6]
 [ 7  8  9]]
Inverse:
[[-0.775       -0.05         0.225      ]
 [ 0.05         0.1          0.05       ]
 [ 0.55833333 -0.05         -0.10833333]]
```

We'll round the matrix products to 14 digits precision because there is some numerical error in the least significant bit.

```
np.round(P.dot(IP),14)
```

```
array([[ 1.,  0., -0.],
       [ 0.,  1.,  0.],
       [ 0., -0.,  1.]])
```

```
np.round(IP.dot(P),14)
```

```
array([[ 1.,  0., -0.],
       [-0.,  1.,  0.],
       [-0., -0.,  1.]])
```

Theorem B.4. Requirement for Invertibility. *A square matrix is invertible if and only if it is nonsingular.*

Definition B.22. Matrix Cofactor. *Let* $\mathbf{A}$ *be a square* $n \times n$ *matrix. Then the* ij ***cofactor****, is*

$$cof\, a_{ij} = (-1)^{i+j} \det \mathbf{M}_{ij} \tag{B.41}$$

where $\mathbf{M}_{ij}$ *is the submatrix of* $\mathbf{A}$ *with row* i *and column* j *removed.*

Example. Find the cofactor of $a_{01} = 2$ in

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{B.42}$$

Following definition B.22, the required formula is

$$\text{cof } a_{01} = (-1)^{1+2} \begin{vmatrix} 4 & 6 \\ 7 & 9 \end{vmatrix} = (-1)(36 - 42) = 6 \quad \square \tag{B.43}$$

Definition B.23. Classical Adjoint Matrix. *If* $\mathbf{A}$ *is an* $n \times n$ *matrix then its* ***classical adjoint****, denoted by adj*$\mathbf{A}$*, is the transpose of the matrix that results when every element of* $\mathbf{A}$ *is replaced by its cofactor.*

$$\text{adj } \mathbf{A} = [\text{cof } a_{ij}]^T = (\text{cof } \mathbf{A})^T \tag{B.44}$$

Example. Find the classical adjoint of

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 3 & 1 \end{bmatrix} \tag{B.45}$$

The matrix of cofactors is

$$\text{cof } \mathbf{A} = \begin{bmatrix} (1)\begin{vmatrix} 5 & 0 \\ 3 & 1 \end{vmatrix} & (-1)\begin{vmatrix} 4 & 0 \\ 0 & 1 \end{vmatrix} & (1)\begin{vmatrix} 4 & 5 \\ 0 & 3 \end{vmatrix} \\ (-1)\begin{vmatrix} 0 & 3 \\ 3 & 1 \end{vmatrix} & (1)\begin{vmatrix} 1 & 3 \\ 0 & 1 \end{vmatrix} & (-1)\begin{vmatrix} 1 & 0 \\ 0 & 3 \end{vmatrix} \\ (1)\begin{vmatrix} 0 & 3 \\ 5 & 0 \end{vmatrix} & (-1)\begin{vmatrix} 1 & 3 \\ 4 & 0 \end{vmatrix} & (1)\begin{vmatrix} 1 & 0 \\ 4 & 5 \end{vmatrix} \end{bmatrix} = \begin{bmatrix} 5 & -4 & 12 \\ 9 & 1 & -3 \\ -15 & 12 & 5 \end{bmatrix} \tag{B.46}$$

Therefore the classical adjoint is then given by its transpose:

$$\text{adj } \mathbf{A} = (\text{cof } \mathbf{A})^{\mathrm{T}} = \begin{bmatrix} 5 & -4 & 12 \\ 9 & 1 & -3 \\ -15 & 12 & 5 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 5 & 9 & -15 \\ -4 & 1 & 12 \\ 12 & -3 & 5 \end{bmatrix} \quad \square \tag{B.47}$$

The cofactors and classical adjoint are important because they are required in order to calculate the inverse of a matrix.

Theorem B.5. Matrix Inverse. *If* $\mathbf{A}$ *is square and non-singular then*

$$\mathbf{A}^{-1} = \frac{1}{\det \mathbf{A}} \text{adj } \mathbf{A} \tag{B.48}$$

Example. Continue working with the same matrix $\mathbf{A}$ and find its inverse $\mathbf{A}^{-1}$. First we must calculate its determinant:

$$\det \mathbf{A} = \begin{vmatrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 3 & 1 \end{vmatrix} = (1)\begin{vmatrix} 5 & 0 \\ 3 & 1 \end{vmatrix} - 0\begin{vmatrix} 4 & 0 \\ 0 & 1 \end{vmatrix} + 3\begin{vmatrix} 4 & 5 \\ 0 & 3 \end{vmatrix} = 41 \tag{B.49}$$

Next, we use the formula for the inverse given in theorem B.5 and the adjoint matrix we calculated previously. Then we find that

$$\mathbf{A}^{-1} = \frac{1}{\det \mathbf{A}} \text{adj } \mathbf{A} = \frac{1}{41} \begin{bmatrix} 5 & 9 & -15 \\ -4 & 1 & 12 \\ 12 & -3 & 5 \end{bmatrix} \quad \square \tag{B.50}$$

Suppose that $\mathbf{A}$ be is any $n \times n$ square matrix and $\mathbf{0}$ be an $n \times 1$ vector of zeros, and suppose that

$$\mathbf{Av} = \mathbf{0} \tag{B.51}$$

for some vector $\mathbf{v} \neq \mathbf{0}$. Then it is not possible for $\mathbf{A}$ to have an inverse $\mathbf{A}^{-1}$, because if such an inverse were to exist, we would have

$$\mathbf{0} = \mathbf{A}^{-1}\mathbf{0} = \mathbf{A}^{-1}\mathbf{Av} = \mathbf{Iv} = \mathbf{v} \tag{B.52}$$

The same result follows if we multiply on the right. Together with theorem B.4, this proves the following theorem.

Theorem B.6. Singular Systems. *If* $\mathbf{v} \neq \mathbf{0}$ *then*

$$\mathbf{Av} = \mathbf{0} \iff \det \mathbf{A} = 0 \tag{B.53}$$

In practical terms, computation of the determinant is computationally inefficient, and there are faster ways to calculate the inverse, such as Gaussian Elimination. A particularly effective method of Gaussian Elimination involves converting the matrix into a product of an upper and lower triangular matrices (LU decomposition using partial pivoting). In fact, determinants and matrix inverses are very rarely used computationally because there is almost always a faster method in terms of number of computations required.

Example. Suppose we want to solve the following linear system.

$$\begin{aligned} x \qquad\quad - z &= 1 \\ -2y - z &= 2 \\ 2x + y + z &= 3 \end{aligned} \tag{B.54}$$

One way to do this begins by forming the matrix equation $\mathbf{Av}$=$\mathbf{b}$,

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & -2 & -1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{B.55}$$

Then, so long as the matrix $\mathbf{A}$ is non-singular (invertible), the solution is given by $\mathbf{v} = \mathbf{A}^{-1}\mathbf{b}$.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.2 & 0.2 & 0.4 \\ 0.4 & -0.6 & -0.2 \\ -0.8 & 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1.8 \\ -1.4 \\ 0.8 \end{bmatrix} \tag{B.56}$$

The matrix in equation B.56 is the inverse of the matrix in the previous equation. Thus the solution is $x = 1.8$, $y = -1.4$, and $z = -0.8$. $\square$

```
Q=np.array([[1,0,-1],[0,-2,-1],[2,1,1]])
b=np.array([1,2,3])
print(Q)
```

```
[[ 1  0 -1]
 [ 0 -2 -1]
 [ 2  1  1]]
```

```
IQ=np.linalg.inv(Q)
print(IQ)
```

```
[[ 0.2  0.2  0.4]
 [ 0.4 -0.6 -0.2]
 [-0.8  0.2  0.4]]
```

```
print(IQ.dot(b))
```

```
[ 1.8 -1.4  0.8]
```

It is better not to solve linear equations this way, however, because numerical instabilities can occur when the inverse is calculated. Instead, you should use the **`np.linalg.solve`** function. This program performs an LU decomposition with partial pivoting, which is a more efficient algorithm.[2] Furthermore, using **`solve`** only requires a single function call and does not require the calculation of the matrix inverse.

```
print(np.linalg.solve(Q,b))
```

```
[ 1.8 -1.4  0.8]
```

Definition B.24. Eigenvalue/Eigenvector. *Let* $\mathbf{A}$ *be a square matrix. Then we say that* λ *is an* ***eigenvalue*** *of* $\mathbf{A}$ *with corresponding* ***eigenvector*** $\mathbf{v}$ *if*

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{B.57}$$

The pair $(\lambda, \mathbf{v})$ *is called an* ***eigenvalue-eigenvector pair****.*

We can re-write B.57 by substituting an identity matrix,

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{I}\mathbf{v} \tag{B.58}$$

because $\mathbf{I}\mathbf{v} = \mathbf{v}$. Rearranging the terms gives

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0} \tag{B.59}$$

where $\mathbf{0}$ is an $n \times 1$ vector of zeros. Thus by theorem B.6, $\det(\mathbf{A} - \lambda\mathbf{I}) = \mathbf{0}$

[2] It is a wrapper for the FORTRAN LAPACK function **`dgesv`**. See `http://www.netlib.org/lapack/double/dgesv` for more information.

Theorem B.7. Characteristic equation. *Let* $\mathbf{A}$ *be a square matrix. Then its eigenvalues are the roots of its* ***characteristic equation***

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \mathbf{0} \tag{B.60}$$

Example. Find the eigenvalues of the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 3 & 1 \end{bmatrix} \tag{B.61}$$

The characteristic equation is formed by subtracting λ from each diagonal element and setting the resulting determinant equal to zero.

$$0 = \begin{vmatrix} 1-\lambda & 0 & 3 \\ 4 & 5-\lambda & 0 \\ 0 & 3 & 1-\lambda \end{vmatrix} = 41 - 11\lambda + 7\lambda^2 - \lambda^3 \tag{B.62}$$

The eigenvalues are $\lambda \approx 6.28761$, $\lambda \approx 0.356196 - 2.52861i$ and $\lambda \approx 0.356196 + 2.52861i$. $\square$

In Python we can calculate the Eigenvalues with **`eigvals`**, and we can calculate the Eigenvalue/Eigenvector pairs with **`eig`**. Both functions are in **`numpy.linalg`**

```
A=np.array([[1,0,3],[4,5,0],[0,3,1]])
print(np.linalg.eigvals(A))
[6.28760866+0.j 0.35619567+2.52861359j 0.35619567-2.52861359j]
```

The **`eig`** function returns two lists: first, a list of the eigenvalues, and second, a matrix with the corresponding eigenvectors in its columns.

```
vals,vecs=np.linalg.eig(A)
for val,vec in zip(vals,vecs.T):
    print("The eigenvector correspsonding to ",
       np.around(val,3),"is")
    print(np.around(vec,3))
    print()
```

```
The eigenvector correspsonding to  (6.288+0j) is
[-0.27 +0.j -0.838+0.j -0.475+0.j]

The eigenvector correspsonding to  (0.356+2.529j) is
[ 0.655+0.j    -0.435-0.237j -0.141+0.552j]

The eigenvector correspsonding to  (0.356-2.529j) is
[ 0.655-0.j    -0.435+0.237j -0.141-0.552j]
```

Theorem B.8. Non-uniqueness of Eigenvectors . *If* $\mathbf{v}$ *is an eigenvector of* $\mathbf{A}$ *with eigenvalue* λ*, then so is any multiple of* $\mathbf{v}$*.*

Example. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -2 & 3 \\ 1 & 1 & 1 \\ 1 & 3 & -1 \end{bmatrix} \tag{B.63}$$

With some work we can find the characteristic equation. It is

$$0 = \begin{vmatrix} 2-\lambda & -2 & 3 \\ 1 & 1-\lambda & 1 \\ 1 & 3 & -1-\lambda \end{vmatrix} \tag{B.64}$$

$$= \cdots \text{ algebra omitted } \cdots$$

$$= -(\lambda+2)(\lambda-3)(\lambda-1) \tag{B.65}$$

Therefore the eigenvalues are -2, 3, 1. To find the eigenvector corresponding to -2 we must solve the following matrix equation for x, y, and z:

$$\begin{bmatrix} 2 & -2 & 3 \\ 1 & 1 & 1 \\ 1 & 3 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = -2 \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{B.66}$$

This is equivalent to solving the following system of three equations in three unknowns x, y, and z:

$$2x - 2y + 3z = -2x \tag{B.67}$$

$$x + y + z = -2y \tag{B.68}$$

$$x + 3y - z = -2z \tag{B.69}$$

Rearranging by combining the terms on the right with those on the left,

$$4x - 2y + 3z = 0 \tag{B.70}$$

$$x + 3y + z = 0 \tag{B.71}$$

$$x + 3y + z = 0 \tag{B.72}$$

The second and third equation are identical. This is because (as we were told in theorem B.8) the eigenvector is only determined in direction but not in magnitude. We are free to arbitrarily set any of x, y, and z. If we choose a nonzero value for $\mathbf{v}_j$ and it turns out that the remaining system has no solution, that means that $\mathbf{v}$ does not have a projection in the $\mathbf{e}_j$ direction and we should have chosen that component to be zero.

Suppose we try $y = 1$. The system becomes

$$4x - 2 + 3z = 0 \tag{B.73}$$
$$x + 3 + z = 0 \tag{B.74}$$

or moving the constants back to the right hand side of the equations,

$$4x + 3z = 2, \tag{B.75}$$
$$x + z = -3 \tag{B.76}$$

Solving the second equation for $x = -z - 3$ and substituting into the first gives

$$4(-z - 3) + 3z = 2 \tag{B.77}$$

This simplifies to $z = -14$. Thus $x = -z-3 = -(-14)-3 = 11$. An eigenvector corresponding to $\lambda = 2$ is

$$\mathbf{v} = [11, 1, -14]^{\mathrm{T}}$$

or any multiple thereof. A similar procedure can be used to find the two remaining eigenvectors. □

Here we repeat the previous example in Python.

```
A=np.array([[2,-2,3],[1,1,1],[1,3,-1]])
vals,vecs=np.linalg.eig(A)
A
```

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

```
print{vals}
```

```
[ 3.  1. -2.]
```

The eigenvalue -2 is the third eigenvalue returned. Thus we computed the third eigenvector. This corresponds to a Python index of 2 in the transpose of the matrix **vecs**, i.e., our eigenvalue should be **vecs.T[2]**.

```
print{vecs.T[2]}
```

```
[-0.61684937 -0.05607722  0.78508102]
```

This is not equal to the expected result of [11, 1, -14] because we set the y-component equal to 1. Any vector parallel to this vector will also be an eigenvector, so we can divide by the middle component (which is the y-component, or **vecs.T[2][1]** to get the following.

```
yvalue=vecs.T[2][1]
vecs.T[2]/yvalue
```

```
array([ 11.,   1., -14.])
```

This is the expected result. Python returns normalized eigenvectors, i.e., eigenvectors with unit length.

Sometimes matrices will have a large number of zeros in them; if all of the elements of a square matrix fall on the main diagonal, the matrix is called a diagonal matrix. Diagonal matrices are particularly easy to work with because the eigenvalues can be read off and the determinant is the product of the diagonal elements.

Theorem B.9. Eigenvalues of a Diagonal Matrix. *The eigenvalues of a diagonal matrix are the elements of the diagonal.*

Even if a square matrix only has non zero elements on or above (or on or below) the main diagonal it is still very easy to work with, because the determinant is still the product of the diagonal elements.

Definition B.25. Triangular Matrices. *An **upper (lower) triangular matrix** is a square matrix that only has its only nonzero entries on or above (below) the main diagonal.*

Theorem B.10. Eigenvalues of Triangular Matrix. *The eigenvalues of an upper (lower) triangular matrix are the numbers on the main diagonal.*

Many efficient implementations of matrix operations (such as solving large systems of linear equations) involving reducing matrices to a triangular form.

The **Singular Value Decomposition** is a factoring of a matrix $\mathbf{X}$ into three parts, written as

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}} \tag{B.78}$$

While all matrices have a singular value decomposition (SVD), when $\mathbf{X}$ is square, there is an intuitive description of the process: a rotation, followed by a scaling, followed by another rotation. In any decomposition, the first and third matrix are orthogonal, that is, they are composed of mutually perpendicular unit vectors; and the middle matrix is a diagonal matrix whose entries are called the **singular values** of $\mathbf{X}$. This SVD is widely used in statistical and data analysis applications.

Theorem B.11. Singular Value Decomposition. *Any $n \times m$ matrix $\mathbf{X}$ can be uniquely decomposed as a product*

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T \tag{B.79}$$

where $\mathbf{U}$ is an $n \times k$ orthogonal matrix, Σ is a $k \times k$ diagonal matrix, and $\mathbf{V}$ is an $m \times k$ orthogonal matrix.

The SVD has a geometric interpretation when $\mathbf{X}$ is a real, square 2×2 matrix. Think of the matrix $\mathbf{X}$ as an operator that transforms a vector $\mathbf{v}$, as in $\mathbf{v} \mapsto \mathbf{X}\mathbf{v}$. Then the SVD writes this transformation as

$$\mathbf{v} \mapsto \mathbf{X}\mathbf{v} = \mathbf{R}_2\mathbf{S}\mathbf{R}_1\mathbf{v} \tag{B.80}$$

where $\mathbf{R}_1$ and $\mathbf{R}_2$ are orthogonal matrices, and $\mathbf{S}$ is diagonal. Since $\mathbf{R}_1$ is orthogonal, it first rotates $\mathbf{v} \mapsto \mathbf{v}' = \mathbf{R}_1\mathbf{v}$ in a new coordinate system. Then the diagonal matrix expands (if $s_i > 1$) or compresses (if $s_i < 1$) the components of $\mathbf{v}'$ along the x' and y' axis. Finally, $\mathbf{R}_2$ is like an "unwinding" rotation, that brings the the expanded vector back, without changing its length.

Definition B.26. Singular Values. *Let $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T$ be the Singular Value Decomposition of $\mathbf{X}$. Then the values on the diagonal of Σ are the **singular values** of $\mathbf{X}$.*

Definition B.27. Square Root of a Matrix. *Let $\mathbf{A}$ be any matrix. Then its square root is any matrix $\sqrt{\mathbf{A}}$ such that*

$$\mathbf{A} = (\sqrt{\mathbf{A}})(\sqrt{\mathbf{A}}) \tag{B.81}$$

In general $\sqrt{\mathbf{A}}$ will not be the matrix of square roots of $\mathbf{A}$. Furthermore, the square root is not unique.

Example. Let $\mathbf{A} = \begin{bmatrix} 13 & 9 \\ 12 & 16 \end{bmatrix}$. Then

$$A = \begin{bmatrix} 13 & 9 \\ 12 & 16 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \tag{B.82}$$

hence

$$\begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} = \sqrt{\begin{bmatrix} 13 & 9 \\ 12 & 16 \end{bmatrix}} \quad \square \tag{B.83}$$

To demonstrate the non-uniqueness of the square root function, there is the function **`scipy.linalg.sqrtm`**, which will return a single matrix square root.

```
W=np.array([[1,3],[4,2]])
A=W.dot(W)
print(A)
```

```
[[13  9]
 [12 16]]
```

```
print(sqrtm(A))
```

```
[[3.28571429 1.28571429]
 [1.71428571 3.71428571]]
```

Theorem B.12. Singular Value Decomposition Algorithm. *Let* $\mathbf{X}$ *be any matrix over* $\mathbb{R}$*. Then there exist orthogonal matrices* $\mathbf{V}$ *and* $\mathbf{U}$ *such that*

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T \tag{B.84}$$

here Σ *is a diagonal matrix of singular values of* $\mathbf{X}$*, and in particular, for any orthonormal bases,* $(\mathbf{v}_1, \ldots, \mathbf{v}_n)$ *and* $(\mathbf{u}_1, \ldots, \mathbf{u}_n)$*, the matrices given by*

$$\mathbf{U} = \left[\begin{array}{c|c|c} \mathbf{u}_1 & \ldots & \mathbf{u}_n \end{array}\right] \tag{B.85}$$

$$\mathbf{V} = \left[\begin{array}{c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_n \end{array}\right] \tag{B.86}$$

where the $\mathbf{v}_j$ *are normalized eigenvectors of* $\mathbf{S} = \sqrt{\mathbf{X}^T\mathbf{X}}$ *and* $\mathbf{u}_j = \Sigma\mathbf{v}_j$*, and*

$$\mathbf{X} = \Sigma\mathbf{S} = \Sigma\sqrt{\mathbf{X}^T\mathbf{X}} \tag{B.87}$$

and the elements of Σ *are the square roots of the eigenvalues of* $\mathbf{S}$*.*

Calculation of the SVD. Here is an algorithm to compute the singular value decomposition.

1. Determine the matrix product $\mathbf{M}^{\mathrm{T}}\mathbf{M}$
2. Find the eigenvectors and eigenvalues $\mathbf{v}_1, \mathbf{v}_2, \ldots$ of $\mathbf{M}^{\mathrm{T}}\mathbf{M}$
3. Determine the matrix product $\mathbf{M}\mathbf{M}^{\mathrm{T}}$
4. Find the eigenvectors $\mathbf{u}_1, \mathbf{u}_2, \ldots$ of $\mathbf{M}\mathbf{M}^{\mathrm{T}}$
5. Define $\mathbf{S}$ as the diagonal matrix consisting of the eigenvalues of $\mathbf{M}^{\mathrm{T}}\mathbf{M}$
6. Let $\mathbf{V} = \left[\begin{array}{c|c|c} \mathbf{v}_1^{\mathrm{T}} & \mathbf{v}_2^{\mathrm{T}} & \cdots \end{array}\right]$
7. Let $\mathbf{U} = \left[\begin{array}{c|c|c} \mathbf{u}_1^{\mathrm{T}} & \mathbf{u}_2^{\mathrm{T}} & \cdots \end{array}\right]$
8. Repeat the following until all permutations have been checked:
   (a) If $\mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}} = \mathbf{M}$ then the singular value decomposition is $(\mathbf{U}, \mathbf{S}, \mathbf{V})$.
   (b) Otherwise, flip the direction of one of the column vectors of $\mathbf{U}$ or $\mathbf{V}$ and try again.

The following result has significant applications in many fields of analysis, and is particularly useful for image compression (see, for example, page 180).

Theorem B.13. SVD Expansion. *The SVD can be expanded as*

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T = s_1\mathbf{u}_1\mathbf{v}_1^T + s_2\mathbf{u}_2\mathbf{v}_2^T + \cdots + s_n\mathbf{u}_n\mathbf{v}_n^T \tag{B.88}$$

where $\mathbf{U}$ *and* $\mathbf{V}$ *are as previously defined and the the* $\mathbf{u}_i$ *and* $\mathbf{v}_i$ *and the column and row vectors of* $\mathbf{U}$ *and* $\mathbf{V}$.

The singular value decomposition is found with **SVD** from package **numpy.linalg**. It returns the triple **(U, S, V)** where the SVD is the product **USV**. Thus **V** is returned in its transposed form. Here we demonstrate how to find the SVD of a $[4 \times 4]$ matrix in Python.

```
M=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12], [13,14,15,16]])
print("M=")
print(M)
U,S,V=np.linalg.svd(M)
print("U=")
print(U)
print("S=")
print(S)
print("V=")
print(V)
```

```
U=
[[-0.13472212 -0.82574206  0.38124745 -0.39325613]
 [-0.3407577  -0.4288172  -0.21521411  0.80850658]
 [-0.54679327 -0.03189234 -0.71331411 -0.43724476]
 [-0.75282884  0.36503251  0.54728078  0.02199431]]
S=
[3.86226568e+01 2.07132307e+00 7.60977226e-16 3.86063773e-16]
V Transpose=
[[-0.4284124  -0.47437252 -0.52033264 -0.56629275]
 [ 0.71865348  0.27380781 -0.17103786 -0.61588352]
 [ 0.28255951 -0.72647617  0.6052738  -0.16135714]
 [-0.46921223  0.41500889  0.57761893 -0.52341558]]
```

To save space, the middle matrix is returned as a simple array of single values, rather than a full diagonal matrix, with all of the zeros. To make it compatible with matrix multiplication, we must identify it as a diagonal matrix using the **diag** function.

```
print(U.dot(np.diag(S).dot(V)))
```

```
U.dot(np.diag(S).dot(V))
```

Theorem B.14. Spectral Theorem. *Let* $\mathbf{M}$ *be a positive definite matrix. Then*

1. *The eigenvalues of* $\mathbf{M}$ *are positive.*
2. *The eigenvectors of* $\mathbf{M}$ *are orthogonal.*
3. *Let* $\mathbf{D}$ *be the diagonal matrix with the eigenvalues of* $\mathbf{M}$ *on its diagonal, and let* $\mathbf{U}$ *be a matrix with column vectors composed of the eigenvectors of* $\mathbf{M}$*, arranged in the same order as the corresponding eigenvalues are arranged in* $\mathbf{D}$*. Then*
$$\mathbf{M} = \mathbf{U}\mathbf{D}\mathbf{U}^T \tag{B.89}$$

all of its eigenvalues are positive, and its eigenvectors are orthogonal.

Corollary B.1. Principal Axis Theorem *The equation of an ellipsoid can be written as*
$$\mathbf{x}^T\mathbf{A}\mathbf{x} = 1 \tag{B.90}$$
for some symmetric positive definite matrix $\mathbf{A}$*. Alternatively, if* $\mathbf{A}$ *is a symmetric positive definite matrix, then equation B.90 is the equation of an an ellipsoid. Furthermore, the principal axes of the ellipsoid lie along the eigenvectors of* $\mathbf{A}$*, and the semi-axes of the ellipsoid are equal to* $1/\sqrt{\lambda_i}$*.*

Example. Let $\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$. Then the equation $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} = 1$ becomes the two dimensional equation
$$1 = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 3x + y \\ x + 3y \end{bmatrix} = 3x^2 + 2xy + 3y^2 \tag{B.91}$$
The eigenvalues of $\mathbf{A}$ are the solutions of its characteristic equation,
$$0 = (3 - \lambda)(3 - \lambda) - 1 = \lambda^2 - 6\lambda + 8 \tag{B.92}$$
Thus $\lambda_1 = 2$, $\lambda_2 = 4$. These correspond to eigenvectors
$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \qquad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{B.93}$$
which are orthogonal. To verify that
$$3x^2 + 2xy + 3y^2 = 1 \tag{B.94}$$
represents a 45 degree rotation about the origin of the ellipse
$$1 = \frac{x^2}{a^2} + \frac{y^2}{b^2} = 2x^2 + 4y^2 \tag{B.95}$$

where the last step follows from $a = 1/\sqrt{\lambda_1} = 1\sqrt{2}$, $b = 1/\sqrt{\lambda_2} = 1/2$, we construct the rotated coordinates with $\theta = -45$ degrees

$$x = \xi \cos\theta + \eta \sin\theta = \frac{\xi}{\sqrt{2}} - \frac{\eta}{\sqrt{2}} \tag{B.96}$$

$$y = -\xi \sin\theta + \eta \cos\theta = \frac{\xi}{\sqrt{2}} + \frac{\eta}{\sqrt{2}} \tag{B.97}$$

Therefore

$$1 = 2x^2 + 4y^2 \tag{B.98}$$

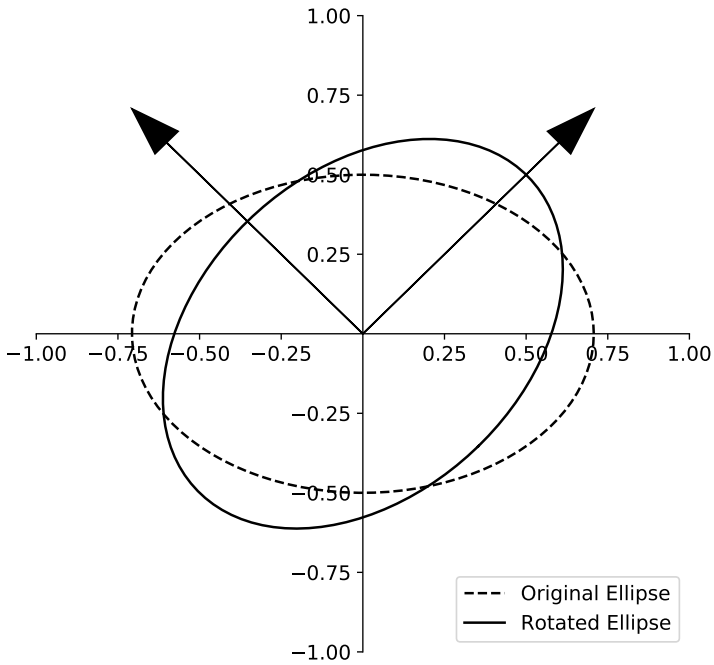
$$= 2\left(\frac{\xi}{\sqrt{2}} - \frac{\eta}{\sqrt{2}}\right)^2 + 4\left(\frac{\xi}{\sqrt{2}} + \frac{\eta}{\sqrt{2}}\right)^2 \tag{B.99}$$

$$= 2\left(\frac{\xi^2}{2} - \xi\eta + \frac{\eta^2}{2}\right) + 4\left(\frac{\xi^2}{2} + \xi\eta + \frac{\eta^2}{2}\right) \tag{B.100}$$

$$= 3\xi^2 + 2\xi\eta + 3\eta^2 \tag{B.101}$$

The original and rotated ellipse and eigenvectors are illustrated in figure B.1. □

Figure B.1: A plot of the ellipse $1 = 2x^2 + 4y^2$ is shown with a dashed line. This ellipse, which is centered at the origin, has semi-major axes parallel to the coordinate axes of $1/\sqrt{2}$ (along the x-axis) and $1/2$ (along the y axis). Unit eigenvectors of the matrix $\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$ are illustrated by the arrows. The rotated matrix, whose equation is given by $3x^2 + 2xy + 2y^2 = 1$, is shown by the solid line.



Literature Citations

Abu-Mostafa, Y, et. al. (2012) *Learning from Data.* AMLbook.com.

Amit, Y, Geman, D (1997) "Shape Quantization and Recognition with Randomized Trees." *Neural Computation.* **9**:1545-1588.

Blum A, Hopcroft J, Kannan R (2018) *Foundations of Data Science* (draft) `https://www.cs.cornell.edu/jeh/book.pdf`, accessed 23 March 2019.

Bradley, AP (1997) "The Use of the Area Under the Curve in the Evaluation of Machine Learning Algorithms." *Pattern Recognition.* **30**(7):1145-1159.

Breiman, L, et. al. (1984) *Classification and Regression Trees.* Brooks/Cole.

Breiman, L (1996) "Bagging predictors." *Machine Learning.* **24**:123-140.

Breiman, L. (2001) "Random Forests." *Machine Learning.* **45**:5-32.

Broyden, CG. "A Class of Methods for Solving Nonlinear Simultaneous Equations." *Mathematics of Computation. American Mathematical Society.* **19**(92): 577-593 (1965).

Burden, R.L. and Faires, J.D. (2001) *Numerical Analysis*, Brooks-Cole.

Calinski,T. and Harabasz, J. (1974) "A dendrite method for cluster analysis." *Commun. Statist.* **3**(1):127.

Cortes, C and Vapnik, V (1995) "Support Vector Networks." *Machine Learning.* **20**:273-297.

Cortez, P, et. al. (2009) "Modeling wine preferences by data mining from physicochemical properties." *Decision Support Systems.* **47**(4):547-553.

Cox, D. (1958) "The regression analysis of binary sequences." *Journal of the Royal Statistical Society B.* **20**: 215-242.

Davidson, WC "Variable metric method for minimization". *SIAM Journal on Optimization.* **1**: 1-17 (1991).

Descartes ,R. (1644) [*Le Monde, ou Traite de la Lumiere* (in French)].

Detrano R et. al. (1989). "International application of a new probability algorithm for the diagnosis of coronary artery disease." *American Journal of Cardiology*, **64**, 304-310.

Dirichlet, G.L. (1850) [" Uber die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen." (in German)] *Journal für die Reine und Angewandte Mathematik*, **40**:209-227.

Dua, D. and Karra Taniskidou, E. (2017). *UCI Machine Learning Repository.* `http://archive.ics.uci.edu/ml.` Irvine, CA: University of California, School of Information and Computer Science.

Ester et. al (1996) "A density-based algorithm for discovering clusters in large spatial databases with noise," KDD96, `https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf`

Fawcett,T (2006) "An introduction to ROC analysis." *Pattern Recognition Letters.* **27**:861-

874.

Fletcher, R "A New Approach to Variable Metric Algorithms," *Computer Journal*, **13** (3): 317-322 (1970)

Forgy, E. W. (1965) "Cluster analysis of multivariate data: efficiency vs interpretability of classifications." *Biometrics.* **21**: 768-769.

Fowlkes, E, Mallows, C. (1983) "A Method for Comparing Two Hierarchical Clusterings." *J. Amer. Stat. Assoc.* **78**: 553-569.

Franklin, J. (1968) *Matrix Theory.* Prentice Hall (available in Dover Reprint (2000)).

Freund, Y. and Schapire, R. (1997) "Generalization of On-Line Learning and an Application to Boosting." *Journal of Computer and System Sciences.* **55**(1):119-139.

Freund, Y. and Schapire, R. (1999) "A Short Introduction to Boosting." *J. Jap. Soc. Art. Int.* **14**(5):771-780.

Goldfarb, D "A Family of Variable Metric Updates Derived by Variational Means," *Mathematics of Computation*, **24** (109): 23-26 (1970)

Green, PJ and Silverman, B (1993) *Nonparametric Regression and Generalized Linear Models.* CRC Press.

Halkidi, M. et. al. (2001) "On Clustering Validation Techniques." *J. Intelligent Information Systems.* **17**:107-145.

Hanley,JA and McNeil,BJ (1982) "The meaning and use of The Area under a Receiver Operating Characteristic (ROC) Curve." *Radiology.* **143**:29-36.

Hassoun, MH (1995) *Fundamentals of Artificial Neural Networks.* MIT Press.

Hastie,T et al (2009) *The Elements of Statistical Learning, 2nd Ed.* Springer.

Heorl, A.E. and Kennard, R.W. (1970) "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, **12**(1):55-67.

Ho, TK (1995) "Random Decision Forests." *Proceedings of the 3rd International Conference on Document Analysis and Recognition (IEEE).* **1**:278-282.

Hyvärinen,A and Oja, E (2000) "Independent Component Analysis: Algorithms and Applications," *Neural Networks*, **13**:411-430.

James, G. et al (2013) *Introduction to Statistical Learning.* Springer.

Kaufman L., Rousseeuw P.J. (1990). *Finding groups in data.* Wiley.

Lance, G, Williams, W. (1967) "A General Theory of classificatory Sorting Strategies 1. Hierarchical Systems." *The Computer Journal.* **9**:373-380.

LeCun, Y et. al. (1998) "Gradient-based learning applied to document recognition." *Proceedings of the IEEE.* **86**(11):2278-2324.

Levenberg, K. (1944) "A Method for the Solution of Certain Non-Linear Problems in Least Squares. *Quarterly of Applied Mathematics.* **2**:164-168.

Lloyd, S. P. (1957). "Least squares quantization in PCM." Technical Note, Bell Laboratories (internal publication)

Lloyd, S. P. (1982). "Least squares quantization in PCM." *IEEE Trans. Inf. Theory.* **28**: 128-137.

Lutz, M. (2016) *Learning Python. 5th Ed.* O'Reilly.

Marquardt, D. (1963) "An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics.* **11**(2):431-441.

MacQueen, J. (1967). "Some methods for classification and analysis of multivariate observations." *Proc. Fifth Berkeley Symp. on Math. Stat. and Prob.*, eds Le Cam, L. Neyman, J

1:281-297.

Martelli, A (2017) *Python in a Nutshell. 3rd Ed.* O'Reilly.

McCulloch, W., Pitts, W. (1943) "A logical calculus of the ideas immanent in nervous activity." *Bulletin of Mathematical Biophysics. 5*:115-133.

Metropolis, N. et. al. (1953) "Equation of State Calculations by Fast Computing Machines." *J. Chem. Phys.* **21**:1087-1092.

Moors, JJA (1986) "The Meaning of Kurtosis: Darlington Reexamined." *The American Statistician*, **40**(4):283-284.

Pedregosa, F et al (2011) "Scikit-learn: Machine Learning in Python." *JMLR.* **12**:2825-2830.

Pollock, D. (1999). "Smoothing with Cubic Splines," in *Handbook of Time Series Analysis, Signal Processing, and Dynamics.*

Quinlan, R (1993) "Combining Instance-Based and Model-Based Learning." *Proceedings on the Tenth International Conference of Machine Learning.* pp. 236- 243, University of Massachusetts, Amherst.

Reinsch, C (1967) "Smoothing by Spline Functions." *Numerische Mathematick.* **10**:177-183.

Robbins, H and Monro, S. (1951) "A Stochastic Approximation Method." *Ann. Math. Statist.* **22**(3):400-407.

Rojas,R. (2009) "Adaboost and the super bowl of classifiers: a tutorial introduction to adaptive boosting." Free University of Berlin, Technical Report. `http://www.inf.fu-berlin.de/inst/ag-ki/adaboost4.pdf`

Rokach, L, and Maimon, O. (2005) "Top-Down Induction of Decision Trees. Classifiers – A Survey." *IEEE Trans. on Systems, Man, and Cybernetics – Part C.* **35**(4): 476-486.

Rosenblatt, F (1957) "The Perceptrona perceiving and recognizing automaton." Report 85-460- 1, Cornell Aeronautical Laboratory.

Rumelhart, D.E. (1986) "Hinton, G. E. et al "Learning representations by back-propagating errors". *Nature.* **323** (6088): 533-536.

Shanno, DF "Conditioning of quasi-Newton methods for function minimization," *Mathematics of Computation*, **24** (111): 647-656 (1970)

Snow, J. (1855) *On the mode of communication of cholera.* London: John Churchill.

Steinhaus, H. (1957). 'Sur la division des corps matériels en parties.' [Bull. Acad. Polon. Sci. (in French).] **4**:801-804.

Strang, G. (2006) *Linear Algebra and Its Applications.* Brooks Cole.

Strang, G. (2007) *Computational Science and Engineering.* Wellesley-Cambridge Press.

Summerfield, M (2010) *Programming in Python 3.* Addison Wesley.

Swets,JA (2000) "Better Decisions Through Science." *Scientific American.* **283**(4):82-87.

Tharwat, A. (2016) "Linear vs. quadratic discriminant analysis classifier: a tutorial." *International Journal of Applied Pattern Recognition.* **3**:145-10.

Thiessen, A. (1911) "Precipitation Averages for Large Areas." *Monthly Weather Review*, **39**(7):1082-1098.

Tibshirani, R. (1996). "Regression Shrinkage and Selection via the lasso." *Journal of the Royal Statistical Society.* Series B (methodological). **58**:267-88.

Tikhonov, A. N. (1943). [On the stability of inverse problems (in Russion)]. *Doklady Akademii Nauk SSSR.* **39**:195-198.

Voronoi, G. (1907) ["Nouvelles applications des paramẽtres continus à la théorie des formes quadratiques." (in French)] *Journal für die Reine und Angewandte Mathematik.* **133**:97-178.

Ward, J. H. (1963) "Hierarchical Grouping to Optimize an Objective Function." *Journal of the American Statistical Association.* **58**(301):236-244.

Wolberg WH and Mangasarian OL, "Multisurface method of pattern separation for medical diagnosis applied to breast cytology" *Proceedings of the National Academy of Sciences, U.S.A.*, **87** (Dec. 1990), 9193-9196.

Zaki, MJ and Meira, W (2014) *Data Mining and Analysis.* Cambridge.

Zelle, J (2010) *Python Programming: An Introduction to Computer Science.* Beedle.

Zhou, Q., et al. (2014) "A Reduction of the Elastic Net to Support Vector Machines with an Application to GPU Computing." *arXiv.* `https://arxiv.org/abs/1409.1976`.

Index

accuracy, 112, 113, 117
addition rule, 243
argmin, 220
AUC, 123
auto data, 179
auto mpg data, 66, 102, 110, 137, 147, 160, 187, 193

backpropagation, 129
bagging, 197, 204
basis vectors, 264
Bayes' Rule, 145, 244
Bayesian Method, 146
Bernoulli variable, 245
binomial distribution, 247
boosting, 197, 202

Calnski-Harabaszi Index, 228
CART, 97
categorical features, 152
chapter-lasso, 73
chapter-spline-regression, 79
chapter-stepwise-regression, 51
characteristic equation, 273
classification
  clustering
    DBSCAN, 236
    hierarchical, 230
    K-Means, 220
  KNN, 160
  LDA, 165
  logistic regression, 110
  naive Bayes', 147
  neural networks, 131
  QDA, 169
  SVM, 187
  trees, 193
closure, 263
cofactor, 270
columns, 6
combination, 245
conditional probability, 244
confusion matrix, 116
continuous random variable, 240
convex hull, 235
correlation, 13, 257
covariance, 257
covariance matrix, 174
cross validation, 16
csv file, 1
cumulative distribution function, 249
curse of dimensionality, 156, 183
cut point, 191

data point, 1
data set, 1
DataFrame, 2
DBSCAN, 236
decision tree, 191
dendrogram, 232
determinant, 266, 269
diagonal matrix, 277
dimensionality reduction, 75
discrete random variable, 240
display_MNIST, 142
dot product, 262
dropna, 67

eigenvalue, 273, 274
eigenvector, 273, 274

energy function, 7
entropy, 191
error rate, 117
event, 240
exemplar, 115
expected value, 252
explanatory variable, 28

fallout, 116
false alarm rate, 116
false positive rate, 116
FastICA, 155
feature, 23
feature matrix, 28, 173
feature reduction, 75
feature vector, 28, 61
figsize, 19
fit, 23
fitsets, 42
Fowlkes-Mallows Index, 239

gamma function, 183
geomspace, 65
gradient descent, 85
graphviz, 135, 194
gzip, 142

header, 67
hierarchical clustering, 230
Hill function, 85, 92
hypersphere volume, 183

ICA (Independent Component Analysis), 147, 149, 155
ill conditioned matrix, 62
imputation, 110
inline, 4
IPython.display.Image, 194

K-Means, 220
KNN (K-Nearest Neighbors), 159
kurtosis, 149, 255, 256

lambda expressions, 53
lasso regression, 73
LDA (Linear Discriminant Analysis), 165
learning rate, 127, 129
least squares, 1, 6
linear dependence, 264
linear regression
    evaluation, 16
    linregress, 11
    numpy, 11
    sklearn, 22
    statistical form, 12
    theory, 9
log axis, 44

machine.data (UCI), 17
magic commands, 4
Mahalanobis distance, 231
marker, 1
marker, 19
math
    .acos, 263
    .pi, 263
matplotlib.pyplot, 4
matrix, 265
matrix addition, 266
matrix adjoint, 270
matrix identity, 269
matrix inverse, 269, 271
matrix multiplication, 268
matrix transpose, 265
maximum likelihood estimator, 109
mean, 13
mean sum of squares, 17
MNIST digits data, 139
moment, 252
MRSS, 21
MSS, 17, 34
multicollinearity, 61
multilinear regression
    numpy, 31
    sklearn, 36
    theory, 28

multivariate normal, 258

na_values, 67
naive Bayes, 145
neural networks, 125
normal equations, 30
normal matrix, 31
numpy
  .array, 260
    .dot, 263
  .corrcoef, 157, 179, 258
  .cov, 258
  .linalg
    .det, 267
    .dot, 268
    .eig, 274
    .eigvals, 274
    .inv, 269
    .norm, 261
    .solve, 31, 273
    .svd, 280
  .logspace, 70, 76
  .meshgrid, 150
  .piecewise, 53
  .polyfit, 40
  .polyval, 40
  .random
    .multivariate_normal, 176
    .random, 39
    .seed, 39
  .ravel, 260, 261
  .reshape, 23, 33, 67, 100
  .round, 179

objective function, 7
one-hot encoding, 153, 163, 210

p value, 12
pandas, 2
  .get_dummies, 153, 210
  .plotting
    .scatter_matrix, 19, 67
  .read_csv, 2, 94, 100, 132
  .read_fwf, 67, 102, 110, 179, 187, 193
PCA, 156, 171
Pearson correlation, 13
perceptron, 125
permutation, 246
polynomial regression, 37
population, 94
posterior probability, 146
precision, 117
predict, 23
principal components, 174, 175
prior probability, 146
probability density function, 250
probability mass function, 242
pydotplus, 194
pyplot, 4
  .contour, 150
  .fill_between, 44

QDA (Quadratic Discrimination Analysis), 169
quantitative features, 152

r value, 12, 13
random forests, 197, 206
random variable, 240
random.shuffle, 21
read_MNIST_images, 142
read_MNIST_labels, 140
recall, 116
regression
  lasso, 73
  linear, 9
  logistic, 107
  multilinear, 28
  nonlinear, 85
  polynomial, 37
  ridge, 61
  spline, 79
  stepwise, 51
  tree, 98
regression tree, 97

residual error, 6
response variable, 28
ridge regression, 61
ROC curve, 119, 121, 135
RSS, 34
RSS error, 7

sample space, 240
scalar product, 264
scatter, 4
scatter matrix, 173, 174
scatter plot, 1
scikit-learn, 22
scipy
  .interpolate.LSQUnivariateSpline, 81
  .interpolate.UnivariateSpline, 83
  .interpolate.interp1d, 122
  .linalg
    .sqrtm, 278
  .optimize.curve_fit, 53, 54, 92, 94
  .stats.kurtosis, 256
  .stats.linregress, 11
sensitivity, 116
sep (in **read_csv**), 48, 132
show, 4
silhouette index, 226
singular linear system, 272
singular matrix, 267
singular value decomposition, 279
singular values, 278
skewness, 255
skiprows (in **read_csv**), 48
sklearn, 22
  .cluster
    .DBSCAN, 238
  .clustering
    .KMeans, 222
  .decomposition
    .FastICA, 155
    .PCA, 176, 179, 188
  .discriminant_analysis
    .LinearDiscriminantAnalysis, 167
    .QuadraticDiscriminantAnalysis, 169
  .impute
    .SimpleImputer, 110
  .linear_model, 22
    .Lasso, 75
    .LinearRegression, 22
    .LogisticRegression, 111
    .Ridge, 69
  .metrics, 22
    .accuracy_score, 119, 133, 143, 148, 161, 189, 194
    .auc, 124
    .confusion_matrix, 118, 133, 143, 148, 161, 194
    .f1_score, 119
    .mean_squared_error, 22, 23
    .precision_score, 119, 133
    .r2_score, 22, 24
    .recall_score, 119, 133
    .roc_auc_score, 124
    .roc_curve, 121
    .silhouette_score, 227
  .model_selection
    .train_test_split, 132, 137, 148, 161, 188, 193
  .naive_bayes.GaussianNB, 148
  .neighbors.KNeighborsClassifier, 161
  .neural_network
    .MLPClassifier, 133
    .MLPRegressor, 137
  .preprocessing
    .MinMaxScaler, 132, 137, 143, 188
    .MinMaxScaler.fit_transform, 132, 137, 143
    .MinMaxScaler.transform, 132, 137, 143
  .svm.SVC, 188
  .tree

.DecisionTreeClassifier, 194
.DecisionTreeRegressor, 100
softmax, 109
solar flux, 58, 100
specificity, 116
spines, 5
spline regression, 79
splitting criteria, 191
square root of a matrix, 278
standard deviation, 13
standard error, 12
stepwise regression, 51
sunspots, 58
symbol, 1

test and training sets, 17, 21, 32, 33, 41, 67, 132
threshold function, 125, 126
train_test_split, 41, 111, 113
trapezoidal rule, 124
true positive rate, 116
TSS, 35
tuple, 3

variance, 13
variance/bias trade-off, 47
vector, 259, 260
vector addition, 263
vector magnitude, 261
visualize_classifier_contours, 150
Voronoi diagram, 163, 220

wine data, 197
wine data file, 131

zero centering, 173

About the Author

Isabella Romeo is a pseudonym for Bruce E. Shapiro. Bella is a blue nose American Staffordshire Terrier. Romeo, a red nose pit, was her life partner and constant companion. Bella's hobbies include barking at trees, chasing grasshoppers, and sleeping on the living room couch.

Since she does not have opposable thumbs, she dictated this book to her human, who, in turn, did all the typing for her. Not only does he have opposable thumbs, sometimes it seems like he has ten of them (for example, he totally ignores the little red lines under words – what is this thing you call "spell check?").

Said human with opposable thumbs and extraordinary typing skills has taught undergraduate mathematics at a California State University campus for the past two decades. In his alternate and sometime prior incarnations he was a senior researcher in the machine learning group at JPL, a computational scientist at Caltech, and a satellite orbital design and mission engineer under contract at the NASA/Goddard Space Flight Center.