José Unpingco

Python for Probability, Statistics, and Machine Learning

EXTRAS ONLINE

Springer

Python for Probability, Statistics, and Machine Learning

José Unpingco

Python for Probability, Statistics, and Machine Learning

Springer

José Unpingco
San Diego, CA
USA

Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-3-319-30715-2 ISBN 978-3-319-30717-6 (eBook)
DOI 10.1007/978-3-319-30717-6

Library of Congress Control Number: 2016933108

Printed on acid-free paper

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

To Irene, Nicholas, and Daniella, for all their patient support.

Preface

This book will teach you the fundamental concepts that underpin probability and statistics and illustrates how they relate to machine learning via the Python language and its powerful extensions. This is not a good *first* book in any of these topics because we assume that you already had a decent undergraduate-level introduction to probability and statistics. Furthermore, we also assume that you have a good grasp of the basic mechanics of the Python language itself. Having said that, this book is appropriate if you have this basic background and want to learn how to use the scientific Python toolchain to investigate these topics. On the other hand, if you are comfortable with Python, perhaps through working in another scientific field, then this book will teach you the fundamentals of probability and statistics and how to use these ideas to interpret machine learning methods. Likewise, if you are a practicing engineer using a commercial package (e.g., Matlab, IDL), then you will learn how to effectively use the scientific Python toolchain by reviewing concepts with which you are already familiar.

The most important feature of this book is that everything in it is reproducible using Python. Specifically, all of the code, all of the figures, and (most of) the text is available in the downloadable supplementary materials that correspond to this book as IPython Notebooks. IPython Notebooks are *live* interactive documents that allow you to change parameters, recompute plots, and generally tinker with all of the ideas and code in this book. I *urge* you to download these IPython Notebooks and follow along with the text to experiment with the topics covered. I guarantee doing this will boost your understanding because the IPython Notebooks allow for interactive widgets, animations, and other intuition-building features that help make many of these abstract ideas concrete. As an open-source project, the entire scientific Python toolchain, including the IPython Notebook, is freely available. Having taught this material for many years, I am convinced that the only way to learn is to experiment as you go. The text provides instructions on how to get started installing and configuring your scientific Python environment.

This book is not designed to be exhaustive and reflects the author's eclectic background in industry. The focus is on fundamentals and intuitions for day-to-day

work, especially when you must explain the results of your methods to a nontechnical audience. We have tried to use the Python language in the most expressive way possible while encouraging good Python coding practices.

Acknowledgments

I would like to acknowledge the help of Brian Granger and Fernando Perez, two of the originators of the Jupyter/IPython Notebook, for all their great work, as well as the Python community as a whole, for all their contributions that made this book possible. Additionally, I would also like to thank Juan Carlos Chavez for his thoughtful review. Hans Petter Langtangen is the author of the Doconce [19] document preparation system that was used to write this text. Thanks to Geoffrey Poore [31] for his work with PythonTeX and LATEX.

San Diego, California
February 2016

Contents

Notation

Symbol	Meaning
σ	Standard deviation
μ	Mean
$\mathbb{V}$	Variance
$\mathbb{E}$	Expectation
$f(x)$	Function of x
$x \rightarrow y$	Mapping from x to y
(a, b)	Open interval
$[a, b]$	Closed interval
$(a, b]$	Half-open interval
Δ	Differential of
Π	Product operator
Σ	Summation of
$\|x\|$	Absolute value of x
$\|\|x\|\|$	Norm of x
$\#A$	Number of elements in A
$A \cap B$	Intersection of sets A, B
$A \cup B$	Union of sets A, B
$A \times B$	Cartesian product of sets A, B
$\in$	Element of
$\wedge$	Logical conjunction
$\neg$	Logical negation
$\{\}$	Set delimiters
$\mathbb{P}(X\|Y)$	Probability of X given Y
$\forall$	For all
$\exists$	There exists
$A \subseteq B$	A is a subset of B
$A \subset B$	A is a proper subset of B
$f_X(x)$	Probability density function of random variable X
$F_X(x)$	Cumulative density function of random variable X
$\sim$	Distributed according to

$\propto$	Proportional to
$\triangleq$	Equal by definition
$:=$	Equal by definition
$\perp$	Perpendicular to
$\therefore$	Therefore
$\Longrightarrow$	Implies
$\equiv$	Equivalent to
$\mathbf{X}$	Matrix X
$\mathbf{x}$	Vector x
$\text{sgn}(x)$	Sign of x
$\mathbb{R}$	Real line
$\mathbb{R}^n$	n-dimensional vector space
$\mathbb{R}^{m\times n}$	$m \times n$-dimensional matrix space
$\mathcal{U}_{(a,b)}$	Uniform distribution on the interval (a, b)
$\mathcal{N}_{(\mu,\sigma^2)}$	Normal distribution with mean μ and variance σ^2
$\xrightarrow{as}$	Converges almost surely
$\xrightarrow{d}$	Converges in distribution
$\xrightarrow{P}$	Converges in probability

About the Author

Dr. José Unpingco earned his PhD from the University of California, San Diego in 1998 and has since worked in industry as an engineer, consultant, and instructor on a wide-variety of advanced data processing and analysis topics, with a rich experience in multiple machine learning technologies. He has been the onsite technical director for large-scale signal and image processing for the Department of Defense (DoD), where he also spearheaded the DoD-wide adoption of the Scientific Python. As the primary Scientific Python instructor for the DoD, he has taught Python to over 600 scientists and engineers. He is currently the technical director for data science for a non-profit medical research organization in San Diego, California.

Chapter 1
Getting Started with Scientific Python

Python went mainstream years ago. It is now part of many undergraduate curricula in engineering and computer science. Great books and interactive on-line tutorials are easy to find. In particular, Python is well-established in web programming with frameworks such as Django and CherryPy, and is the back-end platform for many high-traffic sites.

Beyond web programming, there is an ever-expanding list of third-party extensions that reach across many scientific disciplines, from linear algebra to visualization to machine learning. For these applications, Python is the software *glue* that permits easy exchange of methods and data across core routines typically written in Fortran or C. Scientific Python has been fundamental for almost two decades in government, academia, and industry. For example, NASA's Jet Propulsion Laboratory uses it for interfacing Fortran/C++ libraries for planning and visualization of spacecraft trajectories. The Lawrence Livermore National Laboratory uses scientific Python for a wide variety of computing tasks, some involving routine text processing, and others involving advanced visualization of vast data sets (e.g. VISIT [1]). Shell Research, Boeing, Industrial Light and Magic, Sony Entertainment, and Procter & Gamble use scientific Python on a daily basis for data processing and analysis. Python is thus well-established and continues to extend into many different fields.

Python is a language geared towards scientists and engineers who may not have formal software development training. It is used to prototype, design, simulate, and test without *getting in the way* because Python provides an inherently easy and incremental development cycle, interoperability with existing codes, access to a large base of reliable open source codes, and a hierarchical compartmentalized design philosophy. It is known that productivity is strongly influenced by the workflow of the user, (e.g., time spent running versus time spent programming) [2]. Therefore, Python can dramatically enhance user-productivity.

Python is an *interpreted* language. This means that Python codes run on a Python *virtual machine* that provides a layer of abstraction between the code and the platform it runs on, thus making codes portable across different platforms. For

J. Unpingco, *Python for Probability, Statistics, and Machine Learning*,
DOI 10.1007/978-3-319-30717-6_1

example, the same script that runs on a Windows laptop can also run on a Linux-based supercomputer or on a mobile phone. This makes programming easier because the virtual machine handles the low-level details of implementing the business logic of the script on the underlying platform.

Python is a dynamically typed language, which means that the interpreter itself figures out the representative types (e.g., floats, integers) interactively or at run-time. This is in contrast to a language like Fortran that have compilers that study the code from beginning to end, perform many compiler-level optimizations, link intimately with the existing libraries on a specific platform, and then create an executable that is henceforth liberated from the compiler. As you may guess, the compiler's access to the details of the underlying platform means that it can utilize optimizations that exploit chip-specific features and cache memory. Because the virtual machine abstracts away these details, it means that the Python language does not have programmable access to these kinds of optimizations. So, where is the balance between the ease of programming the virtual machine and these key numerical optimizations that are crucial for scientific work?

The balance comes from Python's native ability to bind to compiled Fortran and C libraries. This means that you can send intensive computations to compiled libraries directly from the interpreter. This approach has two primary advantages. First, it give you the fun of programming in Python, with its expressive syntax and lack of visual clutter. This is a particular boon to scientists who typically want to *use* software as a tool as opposed to developing software as a product. The second advantage is that you can mix-and-match different compiled libraries from diverse research areas that were not otherwise designed to work together. This works because Python makes it easy to allocate and fill memory in the interpreter, pass it as input to compiled libraries, and then retrieve the output back at the interpreter.

Moreover, Python provides a multiplatform solution for scientific codes. As an open-source project, Python itself is available anywhere you can build it, even though it typically comes standard nowadays, as part of many operating systems. This means that once you have written your code in Python, you can just transfer the script to another platform and run it, as long as the compiled libraries are also available there. What if the compiled libraries are absent? Building and configuring compiled libraries across multiple systems used to be a painstaking job, but as scientific Python has matured, a wide range of libraries have now become available across all of the major platforms (i.e., Windows, MacOS, Linux, Unix) as prepackaged distributions.

Finally, scientific Python facilitates maintainability of scientific codes because Python syntax is clean, free of semi-colon litter and other visual distractions that makes code hard to read and easy to obfuscate. Python has many built-in testing, documentation, and development tools that ease maintenance. Scientific codes are usually written by scientists unschooled in software development, so having solid software development tools built into the language itself is a particular boon.

1.1 Installation and Setup

The easiest way to get started is to download the freely available Anaconda distribution provided by Continuum Analytics (`continuum.io`), which is available for all of the major platforms. On Linux, even though most of the toolchain is available via the built-in Linux package manager, it is still better to install the Anaconda distribution because it provides its own powerful package manager (i.e., `conda`) that can keep track of changes in the software dependencies of the packages that it supports. Note that if you do not have administrator privileges, there is also a corresponding `miniconda` distribution that does not require these privileges.

Regardless of your platform, we recommend Python version 2.7. Python 2.7 is the *last* of the Python 2.x series and guarantees backwards compatibility with legacy codes. Python 3.x makes no such guarantees. Although all of the key components of scientific Python are available in version 3.x, the safest bet is to stick with version 2.7. Alternatively, one compromise is to write in a hybrid dialect of Python that is the intersection of elements of versions 2.7 and 3.x. The `six` module enables this transition by providing utility functions for 2.5 and newer codes. There is also a Python 2.7 to 3.x converter available as the `2to3` module but it may be hard to debug or maintain the so-converted code; nonetheless, this might be a good option for small, self-contained libraries that do not require further development or maintenance.

You may have encountered other Python variants on the web, such as `IronPython` (Python implemented in `C#`) and `Jython` (Python implemented in `Java`). In this text, we focus on the C-implementation of Python (i.e., known as *CPython*), which is, by far, the most popular implementation. These other Python variants permit specialized, native interaction with libraries in `C#` or `Java` (respectively), which is still possible (but clunky) using the CPython. Even more Python variants exist that implement the low-level machinery of Python differently for various reasons, beyond interacting with native libraries in other languages. Most notable of these is `Pypy` that implements a just-in-time compiler (JIT) and other powerful optimizations that can substantially speed up *pure* Python codes. The downside of `Pypy` is that its coverage of some popular scientific modules (e.g., Matplotlib, Scipy) is limited or non-existent which means that you cannot use those modules in code meant for `Pypy`.

You may later want to use a Python module that is not maintained by Anaconda's `conda` manager. Because Anaconda comes with the `pip` package manager, which is the main one used outside of scientific Python, you can simply do

```
Terminal> pip install package_name
```

and `pip` will run out to the web and download the package you want and its dependencies and install them in the existing Anaconda directory tree. This works beautifully in the case where the package in question is pure-Python, without any system-specific dependencies. Otherwise, this can be a real nightmare, especially on Windows, which lacks freely available Fortran compilers. If the module in question is a C-library, one way to cope is to install the freely available Visual Studio Community Edition,

which usually has enough to compile many C-codes. This platform dependency is the problem that `conda` was designed to solve by making the binary dependencies of the various platforms available instead of attempting to compile them. On a Windows system, if you installed Anaconda and registered it as the default Python installation (it asks during the install process), then you can use the high-quality Python *wheel* files on Christoph Gohlke's laboratory site at the University of California, Irvine where he kindly makes a long list of scientific modules available.[1] Failing this, you can try the `binstar.org` site, which is a community-powered repository of modules that `conda` is capable of installing, but which are not formally supported by Anaconda. Note that `binstar` allows you to share scientific Python configurations with your remote colleagues using authentication so that you can be sure that you are downloading and running code from users you trust.

Again, if you are on Windows, and none of the above works, then you may want to consider installing a full virtual machine solution, as provided by VMWare's `Player` or Oracle's `VirtualBox` (both freely available under liberal terms). Using either of these, you can set up a Linux machine running on top of Windows, which should cure these problems entirely! The great part of this approach is that you can share directories between the virtual machine and the Windows system so that you don't have to maintain duplicate data files. Anaconda Linux images are also available on the cloud by IAAS providers like Amazon Web Services and Microsoft Azure. Note that for the vast majority of users, especially newcomers to Python, the Anaconda distribution should be more than enough on any platform. It is just worth highlighting the Windows-specific issues and associated workarounds early on. Note that there are other well-maintained scientific Python Windows installers like `WinPython` and `PythonXY`. These provide the `spyder` integrated development environment, which is very Matlab-like environment for transitioning Matlab users.

1.2 Numpy

As we touched upon earlier, to use a compiled scientific library, the memory allocated in the Python interpreter must somehow reach this library as input. Furthermore, the output from these libraries must likewise return to the Python interpreter. This two-way exchange of memory is essentially the core function of the Numpy (numerical arrays in Python) module. Numpy is the de-facto standard for numerical arrays in Python. It arose as an effort by Travis Oliphant and others to unify the numerical arrays in Python. In this section, we provide an overview and some tips for using Numpy effectively, but for much more detail, Travis' book [3] is a great place to start and is available for free online.

[1] Wheel files are a Python distribution format that you download and install using `pip` as in `pip install file.whl`. Christoph names files according to Python version (e.g., `cp27` means Python 2.7) and chipset (e.g., `amd32` vs. Intel `win32`).

Numpy provides specification of byte-sized arrays in Python. For example, below we create an array of three numbers, each of four-bytes long (32 bits at 8 bits per byte) as shown by the `itemsize` property. The first line imports Numpy as `np`, which is the recommended convention. The next line creates an array of 32 bit floating point numbers. The `itemize` property shows the number of bytes per item.

```
>>> import numpy as np # recommended convention
>>> x = np.array([1,2,3],dtype=np.float32)
>>> x
array([ 1.,  2.,  3.], dtype=float32)
>>> x.itemsize
4
```

In addition to providing uniform containers for numbers, Numpy provides a comprehensive set of unary functions (i.e., *ufuncs*) that process arrays element-wise without additional looping semantics. Below, we show how to compute the element-wise sine using Numpy,

```
>>> np.sin(np.array([1,2,3],dtype=np.float32) )
array([ 0.84147096,  0.90929741,  0.14112   ], dtype=float32)
```

This computes the sine of the input array `[1,2,3]`, using Numpy's unary function, `np.sin`. There is another sine function in the built-in `math` module, but the Numpy version is faster because it does not require explicit looping (i.e., using a `for` loop) over each of the elements in the array. That looping happens in the compiled `np.sin` function itself. Otherwise, we would have to do looping explicitly as in the following:

```
>>> from math import sin
>>> [sin(i) for i in [1,2,3]] # list comprehension
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
```

Numpy uses common-sense casting rules to resolve the output types. For example, if the inputs had been an integer-type, the output would still have been a floating point type. In this example, we provided a Numpy array as input to the sine function. We could have also used a plain Python list instead and Numpy would have built the intermediate Numpy array (e.g., `np.sin([1,1,1])`). The Numpy documentation provides a comprehensive (and very long) list of available *ufuncs*.

Numpy arrays come in many dimensions. For example, the following shows a two-dimensional 2×3 array constructed from two conforming Python lists.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x.shape
(2, 3)
```

Note that Numpy is limited to 32 dimensions unless you build it for more.[2] Numpy arrays follow the usual Python slicing rules in multiple dimensions as shown below where the `:` colon character selects all elements along a particular axis.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x[:,0] # 0th column
array([1, 4])
>>> x[:,1] # 1st column
array([2, 5])
```

[2] See `arrayobject.h` in the Numpy source code.

```
>>> x[0,:] # 0th row
array([1, 2, 3])
>>> x[1,:] # 1st row
array([4, 5, 6])
```

You can also select sub-sections of arrays by using slicing as shown below.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x[:,1:] # all rows, 1st thru last column
array([[2, 3],
       [5, 6]])
>>> x[:,::2] # all rows, every other column
array([[1, 3],
       [4, 6]])
>>> x[:,::-1] # reverse order of columns
array([[3, 2, 1],
       [6, 5, 4]])
```

1.2.1 Numpy Arrays and Memory

Some interpreted languages implicitly allocate memory. For example, in Matlab, you can extend a matrix by simply tacking on another dimension as in the following Matlab session:

```
>> x=ones(3,3)
x =
     1     1     1
     1     1     1
     1     1     1
>> x(:,4)=ones(3,1) % tack on extra dimension
x =
     1     1     1     1
     1     1     1     1
     1     1     1     1
>> size(x)
ans =
     3     4
```

This works because Matlab arrays use pass-by-value semantics so that slice operations actually copy parts of the array as needed. By contrast, Numpy uses pass-by-reference semantics so that slice operations are *views* into the array without implicit copying. This is particularly helpful with large arrays that already strain available memory. In Numpy terminology, *slicing* creates views (no copying) and advanced indexing creates copies. Let's start with advanced indexing.

If the indexing object (i.e., the item between the brackets) is a non-tuple sequence object, another Numpy array (of type integer or boolean), or a tuple with at least one sequence object or Numpy array, then indexing creates copies. For the above example, to accomplish the same array extension in Numpy, you have to do something like the following

```
>>> x = np.ones((3,3))
>>> x
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> x[:,[0,1,2,2]] # notice duplicated last dimension
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> y=x[:,[0,1,2,2]] # same as above, but do assign it to y
```

Because of advanced indexing, the variable `y` has its own memory because the relevant parts of `x` were copied. To prove it, we assign a new element to `x` and see that `y` is not updated.

```
>>> x[0,0]=999 # change element in x
>>> x                          # changed
array([[ 999.,    1.,    1.],
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y                          # not changed!
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

However, if we start over and construct `y` by slicing (which makes it a view) as shown below, then the change we made *does* affect `y` because a view is just a window into the same memory.

```
>>> x = np.ones((3,3))
>>> y = x[:2,:2] # view of upper left piece
>>> x[0,0] = 999 # change value
>>> x
array([[ 999.,    1.,    1.], # see the change?
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y
array([[ 999.,    1.], # changed y also!
       [   1.,    1.]])
```

Note that if you want to explicitly force a copy without any indexing tricks, you can do `y=x.copy()`. The code below works through another example of advanced indexing versus slicing.

```
>>> x = np.arange(5) # create array
>>> x
array([0, 1, 2, 3, 4])
>>> y=x[[0,1,2]] # index by integer list to force copy
>>> y
array([0, 1, 2])
>>> z=x[:3]      # slice creates view
>>> z            # note y and z have same entries
array([0, 1, 2])
>>> x[0]=999     # change element of x
>>> x
array([999,   1,   2,   3,   4])
>>> y            # note y is unaffected,
array([0, 1, 2])
>>> z            # but z is (it's a view).
array([999,   1,   2])
```

In this example, `y` is a copy, not a view, because it was created using advanced indexing whereas `z` was created using slicing. Thus, even though `y` and `z` have the same entries, only `z` is affected by changes to `x`. Note that the `flags.ownsdata` property of Numpy arrays can help sort this out until you get used to it.

Manipulating memory using views is particularly powerful for signal and image processing algorithms that require overlapping fragments of memory. The following is an example of how to use advanced Numpy to create overlapping blocks that do not actually consume additional memory,

```
>>> from numpy.lib.stride_tricks import as_strided
>>> x = arange(16)
>>> y=as_strided(x,(7,4),(8,4)) # overlapped entries
>>> y
array([[ 0,  1,  2,  3],
       [ 2,  3,  4,  5],
       [ 4,  5,  6,  7],
       [ 6,  7,  8,  9],
       [ 8,  9, 10, 11],
       [10, 11, 12, 13],
       [12, 13, 14, 15]])
```

The above code creates a range of integers and then overlaps the entries to create a 7×4 Numpy array. The final argument in the `as_strided` function are the strides, which are the steps in bytes to move in the row and column dimensions, respectively. Thus, the resulting array steps four bytes in the column dimension and eight bytes in the row dimension. Because the integer elements in the Numpy array are four bytes, this is equivalent to moving by one element in the column dimension and by two elements in the row dimension. The second row in the Numpy array starts at eight bytes (two elements) from the first entry (i.e., `2`) and then proceeds by four bytes (by one element) in the column dimension (i.e., `2,3,4,5`). The important part is that memory is re-used in the resulting 7×4 Numpy array. The code below demonstrates this by reassigning elements in the original `x` array. The changes show up in the `y` array because they point at the same allocated memory.

```
>>> x[::2]=99 # assign every other value
>>> x
array([99,  1, 99,  3, 99,  5, 99,  7, 99,  9, 99, 11, 99, 13, 99, 15])
>>> y # the changes appear because y is a view
array([[99,  1, 99,  3],
       [99,  3, 99,  5],
       [99,  5, 99,  7],
       [99,  7, 99,  9],
       [99,  9, 99, 11],
       [99, 11, 99, 13],
       [99, 13, 99, 15]])
```

Bear in mind that `as_strided` does not check that you stay within memory block bounds. So, if the size of the target matrix is not filled by the available data, the remaining elements will come from whatever bytes are at that memory location. In other words, there is no default filling by zeros or other strategy that defends memory block bounds. One defense is to explicitly control the dimensions as in the following code,

```
>>> n = 8 # number of elements
>>> x = arange(n) # create array
>>> k = 5 # desired number of rows
>>> y = as_strided(x,(k,n-k+1),(x.itemsize,)*2)
>>> y
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5],
       [3, 4, 5, 6],
       [4, 5, 6, 7]])
```

1.2.2 Numpy Matrices

Matrices in Numpy are similar to Numpy arrays but they can only have two dimensions. They implement row-column matrix multiplication as opposed to element-wise multiplication. If you have two matrices you want to multiply, you can either create them directly or convert them from Numpy arrays. For example, the following shows how to create two matrices and multiply them.

```
>>> import numpy as np
>>> A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> x=np.matrix([[1],[0],[0]])
>>> A*x
matrix([[1],
        [4],
        [7]])
```

This can also be done using arrays as shown below,

```
>>> A=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> x=np.array([[1],[0],[0]])
>>> A.dot(x)
array([[1],
       [4],
       [7]])
```

Numpy arrays support elementwise multiplication, not row-column multiplication. You must use Numpy matrices for this kind of multiplication unless use the inner product `np.dot`, which also works in multiple dimensions (see `np.tensordot` for more general dot products).

It is unnecessary to cast everything to matrices for multiplication. In the next example, everything until last line is a Numpy array and thereafter we cast the array as a matrix with `np.matrix` which then uses row-column multiplication. Note that it is unnecessary to cast the `x` variable as a matrix because the left-to-right order of the evaluation takes care of that automatically. If we need to use `A` as a matrix elsewhere in the code then we should bind it to another variable instead of re-casting it every time. If you find yourself casting back and forth for large arrays, passing the `copy=False` flag to `matrix` avoids the expense of making a copy.

```
>>> A=np.ones((3,3))
>>> type(A) # array not matrix
<type 'numpy.ndarray'>
>>> x=np.ones((3,1)) # array not matrix
>>> A*x
```

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.matrix(A)*x # row-column multiplication
matrix([[ 3.],
        [ 3.],
        [ 3.]])
```

1.2.3 Numpy Broadcasting

Numpy broadcasting is a powerful way to make implicit multidimensional grids for expressions. It is probably the single most powerful feature of Numpy and the most difficult to grasp. Proceeding by example, consider the vertices of a two-dimensional unit square as shown below,

```
>>> X,Y=np.meshgrid(np.arange(2),np.arange(2))
>>> X
array([[0, 1],
       [0, 1]])
>>> Y
array([[0, 0],
       [1, 1]])
```

Numpy's `meshgrid` creates two-dimensional grids. The `X` and `Y` arrays have corresponding entries match the coordinates of the vertices of the unit square (e.g., (0, 0), (0, 1), (1, 0), (1, 1)). To add the x and y-coordinates, we could use `X` and `Y` as in `X+Y` shown below, The output is the sum of the vertex coordinates of the unit square.

```
>>> X+Y
array([[0, 1],
       [1, 2]])
```

Because the two arrays have compatible shapes, they can be added together element-wise. It turns out we can skip a step here and not bother with `meshgrid` to implicitly obtain the vertex coordinates by using broadcasting as shown below

```
>>> x = np.array([0,1])
>>> y = np.array([0,1])
>>> x
array([0, 1])
>>> y
array([0, 1])
>>> x + y[:,None] # add broadcast dimension
array([[0, 1],
       [1, 2]])
>>> X+Y
array([[0, 1],
       [1, 2]])
```

On line 7 the `None` Python singleton tells Numpy to make copies of `y` along this dimension to create a conformable calculation. Note that `np.newaxis` can be used instead of `None` to be more explicit. The following lines show that we obtain the same output as when we used the `X+Y` Numpy arrays. Note that without broadcasting

`x+y=array([0, 2])` which is not what we are trying to compute. Let's continue with a more complicated example where we have differing array shapes.

```
>>> x = np.array([0,1])
>>> y = np.array([0,1,2])
>>> X,Y = np.meshgrid(x,y)
>>> X
array([[0, 1], # duplicate by row
       [0, 1],
       [0, 1]])
>>> Y
array([[0, 0], # duplicate by column
       [1, 1],
       [2, 2]])
>>> X+Y
array([[0, 1],
       [1, 2],
       [2, 3]])
>>> x+y[:,None] # same as with meshgrid
array([[0, 1],
       [1, 2],
       [2, 3]])
```

In this example, the array shapes are different, so the addition of `x` and `y` is not possible without Numpy broadcasting. The last line shows that broadcasting generates the same output as using the compatible array generated by `meshgrid`. This shows that broadcasting works with different array shapes. For the sake of comparison, on line 3, `meshgrid` creates two conformable arrays, `X` and `Y`. On the last line, `x+y[:,None]` produces the same output as `X+Y` without the `meshgrid`. We can also put the `None` dimension on the `x` array as `x[:,None]+y` which would give the transpose of the result.

Broadcasting works in multiple dimensions also. The output shown has shape `(4,3,2)`. On the last line, the `x+y[:,None]` produces a two-dimensional array which is then broadcast against `z[:,None,None]`, which duplicates itself along the *two* added dimensions to accommodate the two-dimensional result on its left (i.e., `x + y[:,None]`). The caveat about broadcasting is that it can potentially create large, memory-consuming, intermediate arrays. There are methods for controlling this by re-using previously allocated memory but that is beyond our scope here. Formulas in physics that evaluate functions on the vertices of high dimensional grids are great use-cases for broadcasting.

```
>>> x = np.array([0,1])
>>> y = np.array([0,1,2])
>>> z = np.array([0,1,2,3])
>>> x+y[:,None]+z[:,None,None]
array([[[0, 1],
        [1, 2],
        [2, 3]],
       [[1, 2],
        [2, 3],
        [3, 4]],
       [[2, 3],
        [3, 4],
        [4, 5]],
       [[3, 4],
        [4, 5],
        [5, 6]]])
```

1.2.4 Numpy Masked Arrays

Numpy provides a powerful method to temporarily hide array elements without changing the shape of the array itself,

```
>>> from numpy import ma # import masked arrays
>>> x = np.arange(10)
>>> y = ma.masked_array(x, x<5)
>>> print y
[-- -- -- -- -- 5 6 7 8 9]
>>> print y.shape
(10,)
```

Note that the elements in the array for which the logical condition (`x < 5`) is true are masked, but the size of the array remains the same. This is particularly useful in plotting categorical data, where you may only want those values that correspond to a given category for part of the plot. Another common use is for image processing, wherein parts of the image may need to be excluded from subsequent processing. Note that creating a masked array does not force an implicit copy operation unless `copy=True` argument is used. For example, changing an element in `x` *does* change the corresponding element in `y`, even though `y` is a masked array,

```
>>> x[-1] = 99 # change this
>>> print x
>>> print y # masked array changed!
[ 0  1  2  3  4  5  6  7  8 99]
[-- -- -- -- -- 5 6 7 8 99]
```

1.2.5 Numpy Optimizations and Prospectus

The scientific Python community continues to push the frontier of scientific computing. Several important extensions to Numpy are under active development. First, Numba is a compiler that generates optimized machine code from pure Python code using the LLVM compiler infrastructure. LLVM started as a research project at the University of Illinois to provide a target-independent compilation strategy for arbitrary programming languages and is now a well-established technology. The combination of LLVM and Python via Numba means that accelerating a block of Python code can be as easy as putting a `@numba.jit` decorator above the function definition, but this doesn't work for all situations. Numba can target general graphics processing units (GPGPUs) also.

Blaze is considered the next generation of Numpy and generalizes the semantics of Numpy for very large data sets that exist on a variety of backend filesystems. This means that Blaze is designed to handle out-of-core (i.e., too big to fit in a single workstation's RAM) data manipulations and computations using the familiar operations from Numpy. Further, Blaze offers tight integration with Pandas (see Sect. 1.6) dataframes. Roughly speaking, Blaze understands how to unpack Python expressions and translate them for a variety of distributed backend data services upon which the computing will actually happen (i.e., using `blaze.compute`).

This means that Blaze separates the expression of the computation from the particular implementation on a given backend.

Building on his amazing work on PyTables, Francesc Alted has been working on the `bcolz` module which is a compressed columnar data container. Also motivated by out-of-core data and computing, `bcolz` tries to relieve the stress of the memory subsystem by compressing data in memory and then interleaving computations on the compressed data in an intelligent way. This approach takes advantage of emerging architectures that have more cores and wider vector units.

1.3 Matplotlib

Matplotlib is the primary visualization tool for scientific graphics in Python. Like all great open-source projects, it originated to satisfy a personal need. At the time of its inception, John Hunter primarily used Matlab for scientific visualization, but as he began to integrate data from disparate sources using Python, he realized he needed a Python solution for visualization, so he single-handedly wrote Matplotlib. Since those early years, Matplotlib has displaced the other competing methods for two-dimensional scientific visualization and today is a very actively maintained project, even without John Hunter, who sadly passed away in 2012.

John had a few basic requirements for Matplotlib:

- Plots should look publication quality with beautiful text.
- Plots should output Postscript for inclusion within LaTeX documents and publication quality printing.
- Plots should be embeddable in a Graphical User Interface (GUI) for application development.
- The code should be mostly Python to allow for users to become developers.
- Plots should be easy to make with just a few lines of code for simple graphs.

Each of these requirements has been completely satisfied and Matplotlib's capabilities have grown far beyond these requirements. In the beginning, to ease the transition from Matlab to Python, many of the Matplotlib functions were closely named after the corresponding Matlab commands. The community has moved away from this style and, even though you will still find the old Matlab-esque style used in the on-line Matplotlib documentation.

The following shows the quickest way to draw a plot using Matplotlib and the plain Python interpreter. Later, we'll see how to do this even faster using IPython. The first line imports the requisite module as `plt` which is the recommended convention. The next line plots a sequence of numbers generated using Python's `range` function. Note the output list contains a `Line2D` object. This is an *artist* in Matplotlib parlance. Finally, the `plt.show()` function draws the plot in a GUI figure window.

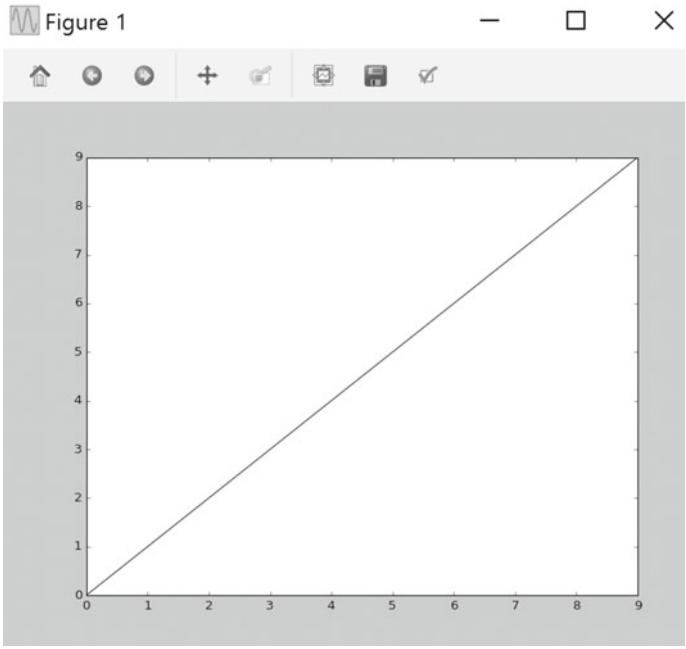

Fig. 1.1 The Matplotlib figure window. The icons on the *bottom* allow some limited plot-editing tools

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(10))
[<matplotlib.lines.Line2D object at 0x00CB9770>]
>>> plt.show() # unnecessary in IPython (discussed later)
```

If you try this in your own plain Python interpreter (and you should), you will see that you cannot type in anything further in the interpreter until the figure window (i.e., something like Fig. 1.1) is closed. This is because the `plt.show()` function preoccupies the interpreter with the controls in the GUI and *blocks* further interaction. As we discuss below, IPython provides ways to get around this blocking so you can simultaneously interact with the interpreter and the figure window.[3]

As shown in Fig. 1.1, the `plot` function returns a list containing the `Line2D` object. More complicated plots yield larger lists filled with *artists*. The suggestion is that artists draw on the *canvas* contained in the Matplotlib figure. The final line is the `plt.show` function that provokes the embedded artists to render on the Matplotlib canvas. The reason this is a separate function is that plots may have dozens of complicated artists and rendering may be a time-consuming task to only

[3] You can also do this in the plain Python interpreter by doing `import matplotlib; matplotlib.interactive(True)`.

be undertaken at the end, when all the artists have been mustered. Matplotlib supports plotting images, contours, and many others that we cover in detail in the following chapters.

Even though this is the quickest way to draw a plot in Matplotlib, it is not recommended because there are no handles to the intermediate products of the plot such as the plot's axis. While this is okay for a simple plot like this, later on we will see how to construct complicated plots using the recommended method. There is a close working relationship between Numpy and Matplotlib and you can load Matplotlib's plotting functions and Numpy's functions simultaneously using `pylab` as `from matplotlib.pylab import *`. Although importing everything this way as a standard practice is not recommended because of namespace pollution.

One of the best ways to get started with Matplotlib is to browse the extensive online gallery of plots on the main Matplotlib site. Each plot comes with corresponding source code that you can use as a starting point for your own plots. In Sect. 1.4, we discuss special *magic* commands that make this particularly easy. The annual *John Hunter: Excellence in Plotting Contest* provides fantastic, compelling examples of scientific visualizations that are possible using Matplotlib.

1.3.1 Alternatives to Matplotlib

Even though Matplotlib is unbeatable for script-based plotting, there are some alternatives for specialized scientific graphics that may be of interest.

Chaco is part of the Enthought Tool-Suite (ETS) and implements many real-time data visualization concepts and corresponding widgets. It is available on all of the major platforms and is also actively maintained and well-documented. Chaco is geared towards GUI application development, rather than script-based data visualization. It depends on the `Traits` package, which is also available in ETS and in the Enthought Canopy. If you don't want to use Canopy, then you have to build Chaco and its dependencies separately. On Linux, this should be straight-forward, but potentially a nightmare on Windows if not for Christoph Gohlke's installers or Anaconda's `conda` package manager.

If you require real-time data display and tools for volumetric data rendering and complicated 3D meshes with isosurfaces, then PyQtGraph is an option. `PyQtGraph` is a pure-Python graphics and GUI library that depends on Python bindings for the Qt GUI library (i.e., `PySide` or `PyQt4`) and Numpy. This means that the `PyQtGraph` relies on these other libraries (especially Qt's `GraphicsView` framework) for the heavy-duty numbercrunching and rendering. This package is actively maintained, but is still pretty new, with good (but not comprehensive) documentation. You also need to grasp a few Qt-GUI development concepts to use this effectively. Mayavi is another Enthought-supported 3D visualization package that sits on VTK (open-source C++ library for 3D visualization). Like Chaco, it is a toolkit for scientific GUI development as opposed to script-based plotting. To use it effectively, you need

to already know (or be willing to learn) about graphics pipelines. This package is actively supported and well-documented.

An alternative that comes from the `R` community is `ggplot` which is a Python port of the `ggplot2` package that is fundamental to statistical graphics in `R`. From the Python standpoint, the main advantage of `ggplot` is the tight integration with the Pandas dataframe, which makes it easy to draw beautifully formatted statistical graphs. The downside of this package is that it applies un-Pythonic semantics based on the *Grammer of Graphics* [4], which is nonetheless a well-thought-out method for articulating complicated graphs. Of course, because there are two-way bridges between Python and `R` via the `R2Py` module (among others), it is workable to send Numpy arrays to `R` for native `ggplot2` rendering and then retrieve the so-computed graphic back into Python. This is a workflow that is lubricated by the IPython Notebook via the `rmagic` extension. Thus, it is quite possible to get the best of both worlds via the IPython Notebook and this kind of multi-language workflow is quite common in data analysis communities.

1.3.2 Extensions to Matplotlib

Initially, to encourage adoption of Matplotlib from Matlab, many of the graphical sensibilities were adopted from Matlab to preserve the look and feel for transitioning users. Modern sensibilities and prettier default plots are possible because Matplotlib provides the ability to drill down and tweak just about every element on the canvas. However, this can be tedious to do and several alternatives offer relief. For statistical plots, the first place to look is the `seaborn` module that includes a vast array of beautifully formatted plots including violin plots, kernel density plots, and bivariate histograms. The `seaborn` gallery includes samples of available plots and the corresponding code that generates them. Note that importing `seaborn` hijacks the default settings for all plots, so you have to coordinate this if you only want to use `seaborn` for some (not all) of your visualizations in a given session. Note that you can find the defaults for Matplotlib in the `matplotlib.rcParams` dictionary.

The `prettyplotlib` module, like `seaborn`, provides an intelligent default color palate based on Cynthia Brewer's work on color perception (c.f. `colorbrewer2.org`). Unfortunately, this work is no longer supported by the author, but still provides a great set of plotting tools and designs for building beautiful data visualizations.

1.4 IPython

IPython [5] originated as a way to enhance Python's basic interpreter for smooth interactive scientific development. In the early days, the most important enhancement was tab-completion for dynamic introspection of workspace variables. For example,

you can start IPython at the commandline by typing `ipython` and then you should see something like the following in your terminal:

```
Python 2.7.11 |Continuum Analytics, Inc.| (default, Dec  7 2015, 14:00
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Next, creating a string as shown and hitting the `TAB` key after the `dot` character initiates the introspection, showing all the functions and attributes of the `string` object in `x`.

```
In [1]: x = 'this is a string'

In [2]: x.<TAB>
x.capitalize x.format     x.isupper    x.rindex     x.strip
x.center     x.index      x.join       x.rjust      x.swapcase
x.count      x.isalnum    x.ljust      x.rpartition x.title
x.decode     x.isalpha    x.lower      x.rsplit     x.translate
x.encode     x.isdigit    x.lstrip     x.rstrip     x.upper
x.endswith   x.islower    x.partition  x.split      x.zfill
x.expandtabs x.isspace    x.replace    x.splitlines
x.find       x.istitle    x.rfind      x.startswith
```

To get help about any of these, you simply add the `?` character at the end as shown below,

```
In [2]: x.center?
Type:       builtin_function_or_method
String Form:<built-in method center of str object at 0x03193390>
Docstring:
S.center(width[, fillchar]) -> string

Return S centered in a string of length width. Padding is
done using the specified fill character (default is a space)
```

and IPython provides the built-in help documentation. Note that you can also get this documentation with `help(x.center)` which works in the plain Python interpreter as well.

The combination of dynamic tab-based introspection and quick interactive help accelerates development because you can keep your eyes and fingers in one place as you work. This was the original IPython experience, but IPython has since grown into a complete framework for delivering a rich scientific computing workflow that retains and enhances these fundamental features.

1.4.1 IPython Notebook

As you may have noticed investigating Python on the web, most Python users are web-developers, not scientific programmers, meaning that the Python stack is *very* well developed for web technologies. The genius of the IPython development team was to leverage these technologies for scientific computing by embedding IPython in modern web-browsers. In fact, this strategy has been so successful that IPython has moved into other languages beyond Python such as Julia and R as the *Jupyter* project.[4]

You can start the `IPython Notebook` with the following commandline: `'jupyter notebook'`. After starting the notebook, you should see something like the following in the terminal,

```
[W 10:26:55.332 NotebookApp] ipywidgets package not installed.  Widgets
[I 10:26:55.348 NotebookApp] Serving notebooks from local directory: D:\
[I 10:26:55.351 NotebookApp] 0 active kernels
[I 10:26:55.351 NotebookApp] The IPython Notebook is running at: http://
[I 10:26:55.351 NotebookApp] Use Control-C to stop this server and shut
```

The first line reveals where IPython looks for default settings. The next line shows where it looks for documents in the IPython Notebook format. The third line shows that the IPython Notebook started a web-server on the local machine (i.e., `127.0.0.1`) on port number `8888`. This is the address your browser needs to connect to the IPython session although your default browser should have opened automatically to this address. The port number and other configuration options are available either on the commandline or in the profile file shown in the first line. If you are on a Windows platform and you do not get this far, then the Window's firewall is probably blocking the port. For additonal configuration help, see the main IPython site (`www.ipython.org`) or e-mail the very responsive IPython mailing list (`ipython-dev@scipy.org`).

When IPython starts, it initiates many small Python processes that use the blazing-fast `ZeroMQ` message passing framework for interprocess-communication, along with the web-sockets protocol for back-and-forth communication with the browser. To start IPython and get around your default browser, you can use the additonal `–no-browser` flag and then manually type in the local host address `http://127.0.0.1:8888` into your favorite browser to get started. Once all that is settled, you should see something like the following Fig. 1.2.

You can create a new document by clicking the `New Notebook` button shown in Fig. 1.2. Then, you should see something like Fig. 1.3. To start using the IPython Notebook, you just start typing code in the shaded textbox and then hit `SHIFT+ENTER` to execute the code in that IPython *cell*. Figure 1.4 shows the dynamic introspection in the pulldown menu when you type the `TAB` key after the `x.`. Context-based help is also available as before by using the `?` suffix which opens a help panel at the bottom of the browser window. There are many amazing features including the ability to

[4]Because we are primarily focused on Python in this text we will continue to refer to IPython and the IPython Notebook instead of to the more general Jupyter project. At the time of this writing, the re-factorization of IPython into Jupyter has not been completed.

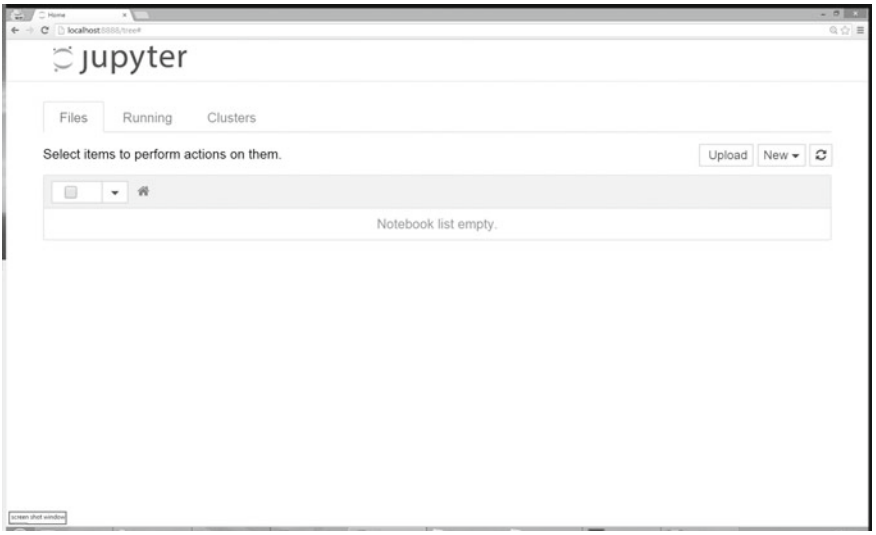

Fig. 1.2 The IPython Notebook dashboard

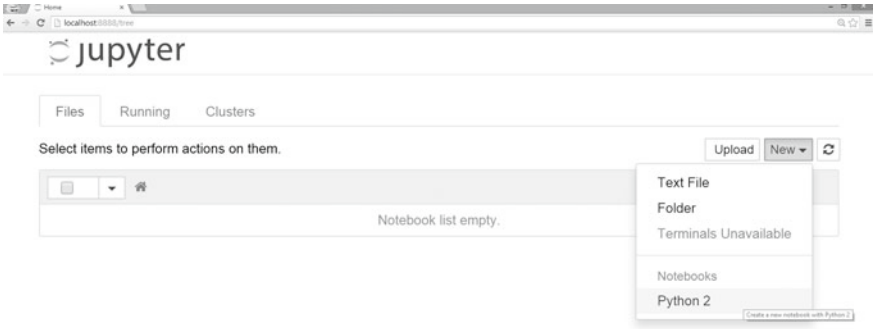

Fig. 1.3 A new IPython Notebook

share notebooks between different users and to run IPython Notebooks in the Amazon cloud, but these features go beyond our scope here. Check the `ipython.org` website or peek at the mailing list for the lastest work on these fronts.

The IPython Notebook supports high-quality mathematical typesetting using MathJaX, which is a JavaScript implementation of most of LaTeX, as well as video and other rich content. The concept of consolidating mathematical algorithm descriptions and the code that implements those algorithms into a shareable document is more important than all of these amazing features. There is no understating the importance of this in practice because the algorithm documentation (if it exists) is usually in one format and completely separate from the code that implements it. This common practice leads to un-synchronized documentation and code that renders one or the

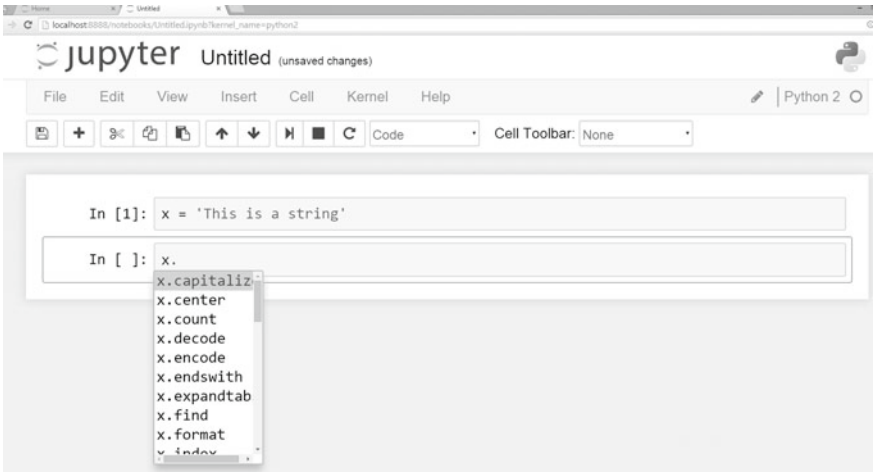

Fig. 1.4 IPython Notebook pulldown completion menu

other useless. The IPython Notebook solves this problem by putting everything into a living shareable document based upon open standards and freely available software. IPython Notebooks can even be saved as static HTML documents for those without Python!

Finally, IPython provides a large set of `magic` commands for creating macros, profiling, debugging, and viewing codes. A full list of these can be found by typing in `%lsmagic` in IPython. Help on any of these is available using the `?` character suffix. Some frequently used commands include the `%cd` command that changes the current working directory, the `%ls` command that lists the files in the current directory, and the `%hist` command that shows the history of previous commands (including optional searching). The most important of these for new users is probably the `%loadpy` command that can load scripts from the local disk or from the web. Using this to explore the Matplotlib gallery is a great way to experiment with and re-use the plots there.

1.5 Scipy

Scipy was the first consolidated module for a wide range of compiled libraries, all based on Numpy arrays. Scipy includes numerous special functions (e.g., Airy, Bessel, elliptical) as well as powerful numerical quadrature routines via the QUADPACK Fortran library (see `scipy.integrate`), where you will also find other quadrature methods. Note that some of the same functions appear in multiple places within Scipy itself as well as in Numpy. Additionally, Scipy provides access to the ODEPACK library for solving differential equations. Lots of statistical

functions, including random number generators, and a wide variety of probability distributions are included in the `scipy.stats` module. Interfaces to the Fortran MINPACK optimization library are provided via `scipy.optimize`. These include methods for root-finding, minimization and maximization problems, with and without higher-order derivatives. Methods for interpolation are provided in the `scipy.interpolate` module via the FITPACK Fortran package. Note that some of the modules are so big that you do not get all of them with `import scipy` because that would take too long to load. You may have to load some of these packages individually as `import scipy.interpolate`, for example.

As we discussed, the Scipy module is already packed with an extensive list of scientific codes. For that reason, the `scikits` modules were originally established as a way to stage candidates that could eventually make it into the already stuffed Scipy module, but it turns out that many of these modules became so successful on their own that they will never be integrated into Scipy proper. Some examples include `sklearn` for machine learning and `scikit-image` for image processing.

1.6 Pandas

Pandas [6] is a powerful module that is optimized on top of Numpy and provides a set of data structures particularly suited to time-series and spreadsheet-style data analysis (think of pivot tables in Excel). If you are familiar with the `R` statistical package, then you can think of Pandas as providing a Numpy-powered dataframe for Python.

1.6.1 Series

There are two primary data structures in Pandas. The first is the `Series` object which combines an index and corresponding data values.

```
>>> import pandas as pd # recommended convention
>>> x=pd.Series(index = range(5),data=[1,3,9,11,12])
>>> x
0     1
1     3
2     9
3    11
4    12
```

The main thing to keep in mind with Pandas is that these data structures were originally designed to work with time-series data. In that case, the `index` in the data structures corresponds to a sequence of ordered time-stamps. In the general case, the index must be a sort-able array-like entity. For example,

```
>>> x=pd.Series(index = ['a','b','d','z','z'],data=[1,3,9,11,12])
>>> x
```

```
a     1
b     3
d     9
z    11
z    12
```

Note the duplicated `z` entries in the `index`. We can get at the entries in the `Series` in a number of ways. First, we can used the *dot* notation to select as in the following:

```
>>> x.a
1
>>> x.z
z    11
z    12
```

We can also use the indexed-position of the entries with `iloc` as in the following:

```
>>> x.iloc[:3]
a    1
b    3
d    9
```

which uses the same slicing syntax as Numpy arrays. You can also slice across the `index`, even if it is not numeric with `loc` as in the following:

```
>>> x.loc['a':'d']
a    1
b    3
d    9
```

which you can get directly from the usual slicing notation:

```
>>> x['a':'d']
a    1
b    3
d    9
```

Note that, unlike Python, slicing this way includes the endpoints. While that is very interesting, the main power of Pandas comes from its power to aggregate and group data. In the following, we build a more interesting `Series` object,

```
>>> x = pd.Series(range(5),[1,2,11,9,10])
1     0
2     1
11    2
9     3
10    4
```

and then group it in the following:

```
>>> grp=x.groupby(lambda i:i%2) # odd or even
>>> grp.get_group(0) # even group
2    1
10    4
>>> grp.get_group(1) # odd group
1     0
11    2
9     3
```

The first line groups the elements of the `Series` object by whether or not the index is even or odd. The `lambda` function returns `0` or `1` depending on whether or not the corresponding index is even or odd, respectively. The next line shows the `0` (even)

group and then the one after shows the `1` (odd) group. Now, that we have separate groups, we can perform a wide-variety of summarizations on the group. You can think of these as reducing each group into a single value. For example, in the following, we get the maximum value of each group:

```
>>> grp.max() # max in each group
0    4
1    3
```

Note that the operation above returns another `Series` object with an `index` corresponding to the `[0,1]` elements.

1.6.2 Dataframe

The Pandas `DataFrame` is an encapsulation of the `Series` that extends to two-dimensions. One way to create a `DataFrame` is with dictionaries as in the following:

```
>>> df = pd.DataFrame({'col1': [1,3,11,2], 'col2': [9,23,0,2]})
   col1  col2
0     1     9
1     3    23
2    11     0
3     2     2
```

Note that the keys in the input dictionary are now the column-headings (labels) of the `DataFrame`, with each corresponding column matching the list of corresponding values from the dictionary. Like the `Series` object, the `DataFrame` also has in `index`, which is the `[0,1,2,3]` column on the far-left. We can extract elements from each column using the `iloc` method as discussed earlier as shown below:

```
>>> df.iloc[:2,:2] # get section
   col1  col2
0     1     9
1     3    23
```

or by directly slicing or by using the *dot* notation as shown below:

```
>>> df['col1'] # indexing
0     1
1     3
2    11
3     2
>>> df.col1 # use dot notation
0     1
1     3
2    11
3     2
```

Subsequent operations on the `DataFrame` preserve its column-wise structure as in the following:

```
>>> df.sum()
col1    17
col2    34
```

where each column was totaled. Grouping and aggregating with the dataframe is even more powerful than with Series. Let's construct the following dataframe,

```
>>> df = pd.DataFrame({'col1': [1,1,0,0], 'col2': [1,2,3,4]})
   col1  col2
0     1     1
1     1     2
2     0     3
3     0     4
```

In the above dataframe, note that the `col1` column has only two entries. We can group the data using this column as in the following:

```
>>> grp=df.groupby('col1')
>>> grp.get_group(0)
   col1  col2
2     0     3
3     0     4
>>> grp.get_group(1)
   col1  col2
0     1     1
1     1     2
```

Note that each group corresponds to entries for which `col1` was either of its two values. Now that we have grouped on `col1`, as with the Series object, we can also functionally summarize each of the groups as in the following:

```
>>> grp.sum()
      col2
col1
0        7
1        3
```

where the `sum` is applied across each of the Dataframes present in each group. Note that the `index` of the output above is each of the values in the original `col1`.

The Dataframe can compute new columns based on existing columns using the `eval` method as shown below:

```
>>> df['sum_col']=df.eval('col1+col2')
>>> df
   col1  col2  sum_col
0     1     1        2
1     1     2        3
2     0     3        3
3     0     4        4
```

Note that you can assign the output to a new column to the Dataframe as shown.[5] We can group by multiple columns as shown below:

```
>>> grp=df.groupby(['sum_col','col1'])
```

Doing the `sum` operation on each group gives the following:

```
>>> res=grp.sum()
>>> res
                col2
sum_col col1
2       1          1
3       0          3
        1          2
4       0          4
```

[5]Note this kind of on-the-fly memory extension is not possible in regular Numpy. For example, `x = np.array([1,2]); x[3]=3` generates an error.

This output is much more complicated than anything we have seen so far, so let's carefully walk through it. Below the headers, the first row `2 1 1` indicates that for `sum_col=2` and for all values of `col1` (namely, just the value `1`), the value of `col2` is `1`. For the next row, the same pattern applies except that for `sum_col=3`, there are now two values for `col1`, namely `0` and `1`, which each have their corresponding two values for the `sum` operation in `col2`. This layered display is one way to look at the result. Note that the layers above are not uniform. Alternatively, we can `unstack` this result to obtain the following tabular view of the previous result:

```
>>> res.unstack()
         col2
col1        0   1
sum_col
2         NaN   1
3           3   2
4           4 NaN
```

The `NaN` values indicate positions in the table where there is no entry. For example, for the pair `(sum_col=2,col2=0)`, there is no corresponding value in the Dataframe, as you may verify by looking at the penultimate code block. There is also no entry corresponding to the `(sum_col=4,col2=1)` pair. Thus, this shows that the original presentation in the penultimate code block is the same as this one, just without the above-mentioned missing entries indicated by `NaN`.

We have barely scratched the surface of what Pandas is capable of and we have completely ignored its powerful features for managing dates and times. The text by Mckinney [6] is a very complete and happily readable introduction to Pandas. The online documentation and tutorials at the main Pandas site are also great for diving deeper into Pandas.

1.7 Sympy

Sympy [7] is the main computer algebra module in Python. It is a pure-Python package with no platform-dependencies. With the help of multiple *Google Summer of Code* sponsorships, it has grown into a powerful computer algebra system with many collateral projects that make it faster and integrate it tighter with Numpy and IPython (among others). Sympy's on-line tutorial is excellent and allows interacting with its embedded code samples in the browser by running the code on the Google App Engine behind the scenes. This provides an excellent way to interact and experiment with Sympy.

If you find Sympy too slow or need algorithms that it does not implement, then SAGE is your next stop. The SAGE project is a consolidation of over 70 of the best open source packages for computer algebra and related computation. Although Sympy and SAGE share code freely between them, SAGE is a specialized build of the Python kernel to facilitate deep integration with the underlying libraries. Thus, it is not a pure-Python solution for computer algebra (i.e., not as portable) and it is a proper superset of Python with its own extended syntax. The choice between SAGE

and Sympy really depends on whether or not you intend *primarily* work in SAGE or just need *occasional* computer algebra support in your existing Python code.

An important new development regarding SAGE is the freely available SAGE Cloud (https://cloud.sagemath.com/), sponsored by University of Washington that allows you to use SAGE entirely in the browser with no additional setup. Both SAGE and Sympy offer tight integration with the IPython Notebook for mathematical typesetting in the browser using MathJaX.

To get started with Sympy, you must import the module as usual,

```
>>> import sympy as S # might take awhile
```

which may take a bit because it is a big package. The next step is to create a Sympy variable as in the following:

```
>>> x = S.symbols('x')
```

Now we can manipulate this using Sympy functions and Python logic as shown below:

```
>>> p=sum(x**i for i in range(3)) # 2nd order polynomial
>>> p
x**2 + x + 1
```

Now, we can find the roots of this polynomial using Sympy functions,

```
>>> S.solve(p) # solves p == 0
[-1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

There is also a `sympy.roots` function that provides the same output but as a dictionary.

```
>>> S.roots(p)
{-1/2 - sqrt(3)*I/2: 1, -1/2 + sqrt(3)*I/2: 1}
```

We can also have more than one symbolic element in any expression as in the following:

```
>>> from sympy.abc import a,b,c  # quick way to get common symbols
>>> p = a* x**2 + b*x + c
>>> S.solve(p,x) # specific solving for x-variable
[(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

which is the usual quadratic formula for roots. Sympy also provides many mathematical functions designed to work with Sympy variables. For example,

```
>>> S.exp(S.I*a) #using Sympy exponential
```

We can expand this using `expand_complex` to obtain the following:

```
>>> S.expand_complex(S.exp(S.I*a))
I*exp(-im(a))*sin(re(a)) + exp(-im(a))*cos(re(a))
```

which gives us Euler's formula for the complex exponential. Note that Sympy does not know whether or not `a` is itself a complex number. We can fix this by making that fact part of the construction of `a` as in the following:

```
>>> a = S.symbols('a',real=True)
>>> S.expand_complex(S.exp(S.I*a))
I*sin(a) + cos(a)
```

Note the much simpler output this time because we have forced the additional condition on `a`.

A powerful way to use Sympy is to construct complicated expressions that you can later evaluate using Numpy via the `lambdify` method. For example,

```
>>> y = S.tan(x) * x + x**2
>>> yf= S.lambdify(x,y,'numpy')
>>> y.subs(x,.1) # evaluated using Sympy
0.0200334672085451
>>> yf(.1) # evaluated using Numpy
0.0200334672085451
```

After creating the Numpy function with `lambdify`, you can use Numpy arrays as input as shown:

```
>>> yf(np.arange(3)) # input is Numpy array
array([ 0.        ,  2.55740772, -0.37007973])
>>> [ y.subs(x,i).evalf() for i in range(3) ] # need extra work for Sympy
[0, 2.55740772465490, -0.370079726523038]
```

We can get the same output using Sympy, but that requires the extra programming logic shown to do the vectorizing that Numpy performs natively.

Once again, we have merely scratched the surface of what Sympy is capable of and the on-line interactive tutorial is the best place to learn more. Sympy also allows automatic mathematical typesetting within the IPython Notebook using LaTeX so the so-constructed notebooks look almost publication-ready (see `sympy.latex`) and can be made so with the `ipython nbconvert` command. This makes it easier to jump the cognitive gap between the Python code and the symbology of traditional mathematics.

1.8 Interfacing with Compiled Libraries

As we have discussed, Python for scientific computing really consists of gluing together different scientific libraries written in a compiled language like C or Fortran. Ultimately, you may want to use libraries not available with existing Python bindings. There are many, many options for doing this. The most direct way is to use the built-in `ctypes` module which provides tools for providing input/output pointers to the library's functions just as if you were calling them from a compiled language. This means that you have to know the function signatures in the library *exactly*—how many bytes for each input and how many bytes for the output. You are responsible for building the inputs exactly the way the library expects and collecting the resulting outputs. Even though this seems tedious, Python bindings for vast libraries have been built this way.

If you want an easier way, then `SWIG` is an automatic wrapper generating tool that can provide bindings to a long list of languages, not just Python; so if you need

bindings for multiple languages, then this is your best and only option. Using SWIG consists of writing an interface file so that the compiled Python dynamically linked library (Python PYD) can be readily imported into the Python interpreter. Huge and complex libraries like Trilinos (Sandia National Labs) have been interfaced to Python using SWIG, so it is a well-tested option. SWIG also supports Numpy arrays.

However, the SWIG model assumes that you want to continue developing primarily in C/Fortran and you are hooking into Python for usability or other reasons. On the other hand, if you start developing algorithms in Python and then want to speed them up, then Cython is an excellent option because it provides a mixed language that allows you to have both C-language and Python code intermixed. Like SWIG, you have to write additional files in this hybrid Python/C dialect to have Cython generate the C-code that you will ultimately compile. The best part of Cython is the profiler that can generate an HTML report showing where the code is slow and could benefit from translation to Cython. The IPython Notebook integrates nicely with Cython via its %cython magic command. This means you can write Cython code in a cell in IPython Notebook and the notebook will handle all of the tedious details like setting up the intermediate files to actually compile the Cython extension. Cython also supports Numpy arrays.

Cython and SWIG are just two of the ways to create Python bindings for your favorite compiled libraries. Other notable (but less popular) options include FWrap, f2py, CFFI, and weave. It is also possible to use Python's own API directly, but this is a difficult undertaking that is hard to justify given the existence of so many well-developed alternatives.

1.9 Integrated Development Environments

For those who prefer Integrated Development Environments (IDEs), there is a lot to choose from. The most comprehensive is Enthought Canopy, which includes a rich, syntax-highlighted editor, integrated help, debugger, and even integrated training. If you are already familiar with Eclipse from other projects, or do mixed-language programming, then there is a Python plugin called PyDev that contains all usual features from Eclipse with a Python debugger. Wingware provides an affordable professional-level IDE with multi-project management support and unusually clairvoyant code-completion that works even in debug-mode. Another favorite is PyCharm, which also supports multiple languages and is particularly popular among Python web-developers because it provides powerful templates for popular web frameworks like Django. NinjaIDE is relatively new, but has quickly developed a strong following among Python newcomers because of its beautiful interface and easy-to-get-started framework. If you are a VIM user, then the Jedi plugin provides excellent code-completion that works well with pylint, which provides static code analysis (i.e., identifies missing modules and typos). Naturally, emacs has many related plugins for developing in Python. Note that are many other options, but I have tried to emphasize those most suitable for Python beginners.

1.10 Quick Guide to Performance and Parallel Programming

There are many options available to improve the performance of your Python codes. The first thing to determine is what is limiting your computation. It could be CPU speed (unlikely), memory limitations (out-of-core computing), or it could be data transfer speed (waiting on data to arrive for processing). If your code is pure-Python, then you can try running it with `Pypy`, which is is an alternative Python implementation that employs a just-in-time compiler. If your code does not experience a massive speed-up with `Pypy`, then there is probably something external to the code that is slowing it down (e.g., disk access or network access). If `Pypy` doesn't make any sense because you are using many compiled modules that `Pypy` does not support, then there are many diagnostic tools available.

Python has its own built-in profiler `cProfile` you can invoke from the command line as in the following

```
>>> python -m cProfile -o program.prof my_program.py
```

The output of the profiler is saved to the `program.prof` file. This file can be visualized in `runsnakerun` to get a nice graphical picture of where the code is spending the most time. The task manager on your operating system can also provide clues as your program runs to see how it is consuming resources. The `line_profiler` by Robert Kern provides an excellent way to see how the code is spending its time by annotating each line of the code by its timings. In combination with `runsnakerun`, this narrows down problems to the line-level from the function-level.

The most common situation is that your program is waiting on data from disk or from some busy network resource. This is a common situation in web programming and there are lots of well-established tools to deal with this. Python has a `multiprocessing` module that is part of the standard library. This makes it easy to spawn child worker processes that can break off and individually process small parts of a big job. However, it is still your responsibility as the programmer to figure out how to distribute the data for your algorithm. Using this module means that the individual processes are to be managed by the operating system, which will be in charge of balancing the load.

The basic template for using `multiprocessing` is the following:

```
# filename multiprocessing_demo.py
import multiprocessing
import time
def worker(k):
    'worker function'
    print 'am starting process %d' % (k)
    time.sleep(10) # wait ten seconds
    print 'am done waiting!'
    return

if __name__ == '__main__':
    for i in range(10):
        p = multiprocessing.Process(target=worker, args=(i,))
        p.start()
```

Then, you run this program at the terminal as in the following,

```
Terminal> python multiprocessing_demo.py
```

It is crucially important that you run the program from the terminal this way. It is not possible to do this interactively from within IPython, say. If you look at the process manager on the operating system, you should see a number of new Python processes loitering for ten seconds. You should also see the output of the `print` statements above. Naturally, in a real application, you would be assigning some meaningful work for each of the workers and figuring out how to send partially finished pieces between individual workers. Doing this is complex and easy to get wrong, so Python 3.2 has the helpful `concurrent.futures` module that has thankfully been back-ported to Python 2.7 and is available on `pypi`.

```
# filename: concurrent_demo.py
import futures
import time

def worker(k):
   'worker function'
   print 'am starting process %d' % (k)
   time.sleep(10) # wait ten seconds
   print 'am done waiting!'
   return

def main():
   with futures.ProcessPoolExecutor(max_workers=3) as executor:
      list(executor.map(worker,range(10)))

if __name__ == '__main__':
    main()
```

```
Terminal> python concurrent_demo.py
```

You should see something like the following in the terminal. Note that we explicitly restricted the number of processes to three.

```
am starting process 0
am starting process 1
am starting process 2
am done waiting!
am done waiting!
...
```

The `futures` module is built on top of `multiprocessing` and makes it easier to use for this kind of simple task. Note that there are also versions of both that use threads instead of processes while maintaining the same usage pattern. The main difference between threads and processes is that processes have their own compartmentalized resources. The C-language Python (i.e., CPython) implementation uses a Global Interpreter Lock (GIL) that prevents threads from locking up on internal data structures. This is a course-grained locking mechanism where one thread may individually run faster because it does not have to keep track of all the bookkeeping involved in running multiple threads simultaneously. The downside is that you cannot run multiple threads simultaneously to speed up certain tasks.

There is no corresponding locking problem with processes but these are somewhat slower to start up because each process has to create its own private workspace for data structures that may be transferred between them. However, each process can certainly run independently and simultaneously once all that is set up. Note that certain alternative implementations of Python like IronPython use a finer-grain threading design rather than a GIL approach. As a final comment, on modern systems with multiple cores, it could be that multiple threads actually slow things down because the operating system may have to switch threads between different cores. This creates additional overheads in the thread switching mechanism that ultimately slow things down.

IPython itself has a parallel programming framework built into it that is powerful and easy-to-use. The first step is to fire up separate IPython engines at the terminal as in the following,

```
Terminal> ipcluster start --n=4
```

Then, in an IPython window, you can get the client,

```
In [1]: from IPython.parallel import Client
   ...: rc = Client()
```

The client has a connection to each of the processes we started before using `ipcluster`. To use all of the engines, we assign the `DirectView` object from the client as in the following,

```
In [2]: dview = rc[:]
```

Now, we can apply functions for each of the engines. For example, we can get the process identifiers using the `os.getpid` function,

```
In [3]: import os
In [4]: dview.apply_sync(os.getpid)
Out[4]: [6824, 4752, 8836, 3124]
```

Once the engines are up and running, data can be distributed to them using `scatter`,

```
In [5]: dview.scatter('a',range(10))
Out[5]: <AsyncResult: finished>
In [6]: dview.execute('print a').display_outputs()
[stdout:0] [0, 1, 2]
[stdout:1] [3, 4, 5]
[stdout:2] [6, 7]
[stdout:3] [8, 9]
```

Note that the `execute` method evaluates the given string in each engine. Now that the data have been sprinkled among the active engines, we can do further computing on them,

```
In [7]: dview.execute('b=sum(a)')
Out[7]: <AsyncResult: finished>
In [8]: dview.execute('print b').display_outputs()
[stdout:0] 3
[stdout:1] 12
[stdout:2] 13
[stdout:3] 17
```

In this example, we added up the individual `a` sub-lists available on each of the engines. We can gather up the individual results into a single list as in the following,

```
In [9]: dview.gather('b').r
Out[9]: [3, 12, 13, 17]
```

This is one of the simplest mechanisms for distributing work to the individual engines and collecting the results. Unlike the other methods we discussed, you can do this iteratively, which makes it easy to experiment with how you want to distribute and compute with the data. The IPython documentation has many more examples of parallel programming styles that include running the engines on cloud resources, supercomputer clusters, and across disparate networked computing resources. Although there are many other specialized parallel programming packages, IPython provides the best trade-off for generality against complexity across all of the major platforms.

1.11 Other Resources

The Python community is filled with super-smart and amazingly helpful people. One of the best places to get help with scientific Python is the `www.stackoverflow.com` site which hosts a competitive Q&A forum that is particularly welcoming for Python newbies. Several of the key Python developers regularly participate there and the quality of the answers is very high. The mailing lists for any of the key tools (e.g., Numpy, IPython, Matplotlib) are also great for keeping up with the newest developments. Anything written by Hans Petter Langtangen [8] is excellent, especially if you have a physics background. The Scientific Python conference held annually in Austin is also a great place to see your favorite developers in person, ask questions, and participate in the many interesting sub-groups organized around niche topics. The `PyData` workshop is a semi-annual meeting focused on Python for large-scale data-intensive processing. The `PyVideo` site provides links to videos of talks and tutorials related to Python from around the world. A great article that summarizes best practices in Python for science is [9].

References

1. H. Childs, E.S. Brugger, K.S. Bonnell, J.S. Meredith, M. Miller, B.J. Whitlock, N. Max, A contract-based system for large data visualization. IEEE Vis. **2005**, 190–198 (2005)
2. MIT Graduate Class Experimental Data. Interactive supercomputings star-p platform: Parallel MATLAB and MPI homework classroom study on high level language productivity (HPEC, 2006)
3. T.E. Oliphant, *A Guide to NumPy* (Trelgol Publishing, 2006)
4. L. Wilkinson, D. Wills, D. Rope, A. Norton, R. Dubbs, *The Grammar of Graphics. Statistics and Computing* (Springer, 2006)
5. F. Perez, B.E. Granger et al., *IPython Software Package for Interactive Scientific Computing*. http://ipython.org/

6. W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (O'Reilly, 2012)
7. O. Certik et al., *SymPy: Python Library for Symbolic Mathematics*. http://sympy.org/
8. H.P. Langtangen, Texts in Computational Science and Engineering, in *Python Scripting for Computational Science*, vol. 3, 3rd edn. (Springer, 2009)
9. D.A. Aruliah, C.T. Brown, N.P.C. Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K. Huff, I. Mitchell, M.D. Plumbley, B. Waugh, E.P. White, G. Wilson, P. Wilson, Best practices for scientific computing. CoRR, (2012). arXiv:abs/1210.0530

Chapter 2
Probability

2.1 Introduction

This chapter takes a geometric view of probability theory and relates it to familiar concepts in linear algebra and geometry. This approach connects your natural geometric intuition to the key abstractions in probability that can help guide your reasoning. This is particularly important in probability because it is easy to be misled. We need a bit of rigor and some intuition to guide us.

In grade school, you were introduced to the natural numbers (i.e., `1,2,3,...`) and you learned how to manipulate them by operations like addition, subtraction, and multiplication. Later, you were introduced to positive and negative numbers and were again taught how to manipulate them. Ultimately, you were introduced to the calculus of the real line, and learned how to differentiate, take limits, and so on. This progression provided more abstractions, but also widened the field of problems you could successfully tackle. The same is true of probability. One way to think about probability is as a new number concept that allows you to tackle problems that have a special kind of *uncertainty* built into them. Thus, the key idea is that there is some number, say x, with a traveling companion, say, $f(x)$, and this companion represents the uncertainties about the value of x as if looking at the number x through a frosted window. The degree of opacity of the window is represented by $f(x)$. If we want to manipulate x, then we have to figure out what to do with $f(x)$. For example if we want $y = 2x$, then we have to understand how $f(x)$ generates $f(y)$.

Where is the *random* part? To conceptualize this, we need still another analogy: think about a beehive with the swarm around it representing $f(x)$, and the hive itself, which you can barely see through the swarm, as x. The random piece is you don't know *which* bee in particular is going to sting you! Once this happens the uncertainty evaporates. Up until that happens, all we have is a concept of a swarm (i.e., density of bees) which represents a *potentiality* of which bee will ultimately sting. In summary, one way to think about probability is as a way of carrying through mathematical reasoning (e.g., adding, subtracting, taking limits) with a notion of potentiality that is so-transformed by these operations.

J. Unpingco, *Python for Probability, Statistics, and Machine Learning*,
DOI 10.1007/978-3-319-30717-6_2

2.1.1 Understanding Probability Density

In order to understand the heart of modern probability, which is built on the Lesbesgue theory of integration, we need to extend the concept of integration from basic calculus. To begin, let us consider the following piecewise function

$$f(x) = \begin{cases} 1 & \text{if } 0 < x \leq 1 \\ 2 & \text{if } 1 < x \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

as shown in Fig. 2.1. In calculus, you learned Riemann integration, which you can apply here as

$$\int_0^2 f(x)dx = 1 + 2 = 3$$

which has the usual interpretation as the area of the two rectangles that make up $f(x)$. So far, so good.

With Lesbesgue integration, the idea is very similar except that we focus on the y-axis instead of moving along the x-axis. The question is given $f(x) = 1$, what is the set of x values for which this is true? For our example, this is true whenever $x \in (0, 1]$. So now we have a correspondence between the values of the function (namely, `1` and `2`) and the sets of x values for which this is true, namely, $\{(0, 1]\}$ and $\{(1, 2]\}$, respectively. To compute the integral, we simply take the function values (i.e., `1 , 2`) and some way of measuring the size of the corresponding interval (i.e., μ) as in the following:

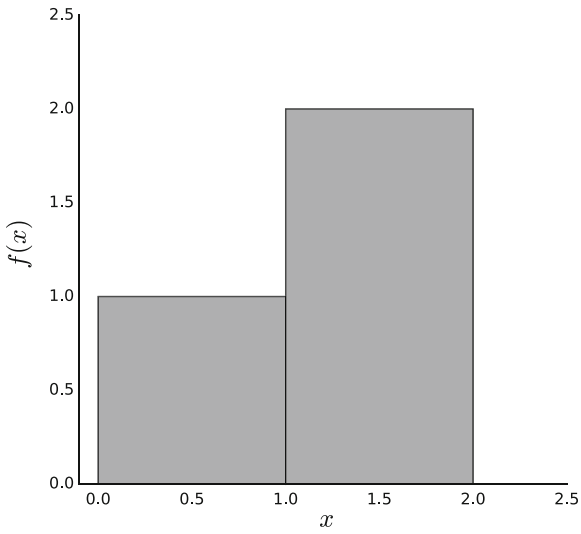

Fig. 2.1 Simple piecewise-constant function

$$\int_0^2 f d\mu = 1\mu(\{(0,1]\}) + 2\mu(\{(1,2]\})$$

We have suppressed some of the notation above to emphasize generality. Note that we obtain the same value of the integral as in the Riemann case when $\mu((0,1]) = \mu((1,2]) = 1$. By introducing the μ function as a way of measuring the intervals above, we have introduced another degree of freedom in our integration. This accommodates many weird functions that are not tractable using the usual Riemann theory, but we refer you to a proper introduction to Lesbesgue integration for further study [1]. Nonetheless, the key step in the above discussion is the introduction of the μ function, which we will encounter again as the so-called probability density function.

2.1.2 Random Variables

Most introductions to probability jump straight into *random variables* and then explain how to compute complicated integrals. The problem with this approach is that it skips over some of the important subtleties that we will now consider. Unfortunately, the term *random variable* is not very descriptive. A better term is *measurable function*. To understand why this is a better term, we have to dive into the formal constructions of probability by way of a simple example.

Consider tossing a fair six-sided die. There are only six outcomes possible,

$$\Omega = \{1,2,3,4,5,6\}$$

As we know, if the die is fair, then the probability of each outcome is $1/6$. To say this formally, the measure of each set (i.e., $\{1\}, \{2\}, \ldots, \{6\}$) is $\mu(\{1\}) = \mu(\{2\}) \ldots = \mu(\{6\}) = 1/6$. In this case, the μ function we discussed earlier is the usual *probability* mass function, denoted by $\mathbb{P}$. The measurable function maps a set into a number on the real line. For example, $\{1\} \mapsto 1$ is one such uninteresting function.

Now, here's where things get interesting. Suppose you were asked to construct a fair coin from the fair die. In other words, we want to throw the die and then record the outcomes as if we had just tossed a fair coin. How could we do this? One way would be to define a measurable function that says if the die comes up 3 or less, then we declare *heads* and otherwise declare *tails*. This has some strong intuition behind it, but let's articulate it in terms of formal theory. This strategy creates two different non-overlapping sets $\{1,2,3\}$ and $\{4,5,6\}$. Each set has the same probability *measure*,

$$\mathbb{P}(\{1,2,3\}) = 1/2$$
$$\mathbb{P}(\{4,5,6\}) = 1/2$$

And the problem is solved. Everytime the die comes up $\{1,2,3\}$, we record heads and record tails otherwise.

Is this the only way to construct a fair coin experiment from a fair die? Alternatively, we can define the sets as {1}, {2}, {3, 4, 5, 6}. If we define the corresponding measure for each set as the following

$$\begin{aligned}\mathbb{P}(\{1\}) &= 1/2\\ \mathbb{P}(\{2\}) &= 1/2\\ \mathbb{P}(\{3,4,5,6\}) &= 0\end{aligned}$$

then, we have another solution to the fair coin problem. To implement this, all we do is ignore every time the die shows `3,4,5,6` and throw again. This is wasteful, but it solves the problem. Nonetheless, we hope you can see how the interlocking pieces of the theory provide a framework for carrying the notion of uncertainty/potentiality from one problem to the next (e.g., from the fair die to the fair coin).

Let's consider a slightly more interesting problem where we toss two dice. We assume that each throw is *independent*, meaning that the outcome of one does not influence the other. What are the sets in this case? They are all pairs of possible outcomes from two throws as shown below,

$$\Omega = \{(1,1),(1,2),\ldots,(5,6),(6,6)\}$$

What are the measures of each of these sets? By virtue of the independence claim, the measure of each is the product of the respective measures of each element. For instance,

$$\mathbb{P}((1,2)) = \mathbb{P}(\{1\})\mathbb{P}(\{2\}) = \frac{1}{6^2}$$

With all that established, we can ask the following question: what is the probability that the sum of the dice equals seven? As before, the first thing to do is characterize the measurable function for this as $X : (a,b) \mapsto (a+b)$. Next, we associate all of the (a,b) pairs with their sum. We can create a Python dictionary for this as shown,

```
d={(i,j):i+j for i in range(1,7) for j in range(1,7)}
```

The next step is to collect all of the (a,b) pairs that sum to each of the possible values from two to twelve.

```
from collections import defaultdict
dinv = defaultdict(list)
for i,j in d.iteritems():
    dinv[j].append(i)
```

Programming Tip

The `defaultdict` object from the built-in collections module creates dictionaries with default values when it encounters a new key. Otherwise, we would have had to create default values manually for a regular dictionary.

For example, `dinv[7]` contains the following list of pairs that sum to seven,

```
[(1, 6), (2, 5), (5, 2), (6, 1), (4, 3), (3, 4)]
```

The next step is to compute the probability measured for each of these items. Using the independence assumption, this means we have to compute the sum of the products of the individual item probabilities in `dinv`. Because we know that each outcome is equally likely, the probability of every term in the sum equals 1/36. Thus, all we have to do is count the number of items in the corresponding list for each key in `dinv` and divide by `36`. For example, `dinv[11]` contains `[(5, 6), (6, 5)]`. The probability of `5+6=6+5=11` is the probability of this set which is composed of the sum of the probabilities of the individual elements `(5,6),(6,5)`. In this case, we have $\mathbb{P}(11) = \mathbb{P}(\{(5,6)\}) + \mathbb{P}(\{(6,5)\}) = 1/36 + 1/36 = 2/36$. Repeating this procedure for all the elements, we derive the probability mass function as shown below,

```
X={i:len(j)/36. for i,j in dinv.iteritems() }
print X
{2: 0.027777777777777776,
 3: 0.05555555555555555,
 4: 0.08333333333333333,
 5: 0.1111111111111111,
 6: 0.1388888888888889,
 7: 0.16666666666666666,
 8: 0.1388888888888889,
 9: 0.1111111111111111,
 10: 0.08333333333333333,
 11: 0.05555555555555555,
 12: 0.027777777777777776}
```

Programming Tip

In the preceding code note that `36.` is written with the trailing decimal mark. This is a good habit to get into because division in Python 2.x is integer division by default, which is not what we want here. This can be fixed with a top-level `from__future__ import division`, but that's easy to forget to do, especially when you are passing code around and others may not reflexively do the future import.

The above example exposes the elements of probability theory that are in play for this simple problem while deliberately suppressing some of the gory technical details. With this framework, we can ask other questions like what is the probability that half the product of three dice will exceed the their sum? We can solve this using the same method as in the following. First, let's create the first mapping,

```
d={(i,j,k):((i*j*k)/2>i+j+k) for i in range(1,7)
                                 for j in range(1,7)
                                    for k in range(1,7)}
```

The keys of this dictionary are the triples and the values are the logical values of whether or not half the product of three dice exceeds their sum. Now, we do the inverse mapping to collect the corresponding lists,

```
dinv = defaultdict(list)
for i,j in d.iteritems(): dinv[j].append(i)
```

Note that `dinv` contains only two keys, `True` and `False`. Again, because the dice are independent, the probability of any triple is $1/6^3$. Finally, we collect this for each outcome as in the following,

```
X={i:len(j)/6.0**3 for i,j in dinv.iteritems() }
print X
{False: 0.37037037037037035, True: 0.6296296296296297}
```

Thus, the probability of half the product of three dice exceeding their sum is `136/(6.0**3) = 0.63`. The set that is induced by the random variable has only two elements in it, `True` and `False`, with $\mathbb{P}(\text{True}) = 136/216$ and $\mathbb{P}(\text{False}) = 1 - 136/216$.

As a final example to exercise another layer of generality, let is consider the first problem with the two dice where we want the probability of a seven, but this time one of the dice is no longer fair. The distribution for the unfair die is the following:

$$\mathbb{P}(\{1\}) = \mathbb{P}(\{2\}) = \mathbb{P}(\{3\}) = \frac{1}{9}$$
$$\mathbb{P}(\{4\}) = \mathbb{P}(\{5\}) = \mathbb{P}(\{6\}) = \frac{2}{9}$$

From our earlier work, we know the elements corresponding to the sum of seven are the following:

$$\{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)\}$$

Because we still have the independence assumption, all we need to change is the probability computation of each of elements. For example, given that the first die is the unfair one, we have

$$\mathbb{P}((1, 6)) = \mathbb{P}(1)\mathbb{P}(6) = \frac{1}{9} \times \frac{1}{6}$$

and likewise for $(2, 5)$ we have the following:

$$\mathbb{P}((2, 5)) = \mathbb{P}(2)\mathbb{P}(5) = \frac{1}{9} \times \frac{1}{6}$$

and so forth. Summing all of these gives the following:

$$\mathbb{P}_X(7) = \frac{1}{9} \times \frac{1}{6} + \frac{1}{9} \times \frac{1}{6} + \frac{1}{9} \times \frac{1}{6} + \frac{2}{9} \times \frac{1}{6} + \frac{2}{9} \times \frac{1}{6} + \frac{2}{9} \times \frac{1}{6} = \frac{1}{6}$$

Let's try computing this using Pandas instead of Python dictionaries. First, we construct a `DataFrame` object with an index of tuples consisting of all pairs of possible dice outcomes.

```
>>> from pandas import DataFrame
>>> d=DataFrame(index=[(i,j) for i in range(1,7) for j in range(1,7)],
...             columns=['sm','d1','d2','pd1','pd2','p'])
```

Now, we can populate the columns that we set up above where the outcome of the first die is the `d1` column and the outcome of the second die is `d2`,

```
>>> d.d1=[i[0] for i in d.index]
>>> d.d2=[i[1] for i in d.index]
```

Next, we compute the sum of the dices in the `sm` column,

```
>>> d.sm=map(sum,d.index)
```

With that established, the DataFrame now looks like the following:

```
>>> d.head(5) # show first five lines
        sm  d1  d2  pd1  pd2    p
(1, 1)   2   1   1  NaN  NaN  NaN
(1, 2)   3   1   2  NaN  NaN  NaN
(1, 3)   4   1   3  NaN  NaN  NaN
(1, 4)   5   1   4  NaN  NaN  NaN
(1, 5)   6   1   5  NaN  NaN  NaN
```

Next, we fill out the probabilities for each face of the unfair die (`d1`) and the fair die (`d2`),

```
>>> d.loc[d.d1<=3,'pd1']=1/9.
>>> d.loc[d.d1 > 3,'pd1']=2/9.
>>> d.pd2=1/6.
>>> d.head(10)
        sm  d1  d2        pd1       pd2    p
(1, 1)   2   1   1  0.1111111  0.166667  NaN
(1, 2)   3   1   2  0.1111111  0.166667  NaN
(1, 3)   4   1   3  0.1111111  0.166667  NaN
(1, 4)   5   1   4  0.1111111  0.166667  NaN
(1, 5)   6   1   5  0.1111111  0.166667  NaN
(1, 6)   7   1   6  0.1111111  0.166667  NaN
(2, 1)   3   2   1  0.1111111  0.166667  NaN
(2, 2)   4   2   2  0.1111111  0.166667  NaN
(2, 3)   5   2   3  0.1111111  0.166667  NaN
(2, 4)   6   2   4  0.1111111  0.166667  NaN
```

Finally, we can compute the joint probabilities for the sum of the shown faces as the following:

```
>>> d.p = d.pd1 * d.pd2
>>> d.head(5)
        sm  d1  d2        pd1       pd2           p
(1, 1)   2   1   1  0.1111111  0.166667  0.01851852
(1, 2)   3   1   2  0.1111111  0.166667  0.01851852
(1, 3)   4   1   3  0.1111111  0.166667  0.01851852
(1, 4)   5   1   4  0.1111111  0.166667  0.01851852
(1, 5)   6   1   5  0.1111111  0.166667  0.01851852
```

With all that established, we can compute the density of all the dice outcomes by using `groupby` as in the following,

```
>>> d.groupby('sm')['p'].sum()
sm
2     0.018519
3     0.037037
4     0.055556
5     0.092593
6     0.129630
7     0.166667
8     0.148148
9     0.129630
10    0.111111
11    0.074074
12    0.037037
Name: p, dtype: float64
```

These examples have shown how the theory of probability breaks down sets and measurements of those sets and how these can be combined to develop the probability mass functions for new random variables.

2.1.3 Continuous Random Variables

The same ideas work with continuous variables but managing the sets becomes trickier because the real line, unlike discrete sets, has many limiting properties already built into it that have to be handled carefully. Nonetheless, let's start with an example that should illustrate the analogous ideas. Suppose a random variable X is uniformly distributed on the unit interval. What is the probability that the variable takes on values less than 1/2?

In order to build intuition onto the discrete case, let's go back to our dice-throwing experiment with the fair dice. The sum of the values of the dice is a measurable function,

$$Y : \{1, 2, \ldots, 6\}^2 \mapsto \{2, 3, \ldots, 12\}$$

That is, Y is a mapping of the cartesian product of sets to a discrete set of outcomes. In order to compute probabilities of the set of outcomes, we need to derive the probability measure for Y, $\mathbb{P}_Y$, from the corresponding probability measures for

each die. Our previous discussion went through the mechanics of that. This means that

$$\mathbb{P}_Y : \{2, 3, \ldots, 12\} \mapsto [0, 1]$$

Note there is a separation between the function definition and where the target items of the function are measured in probability. More bluntly,

$$Y : A \mapsto B$$

with,

$$\mathbb{P}_Y : B \mapsto [0, 1]$$

Thus, to compute $\mathbb{P}_Y$, which is derived from other random variables, we have to express the equivalence classes in B in terms of their progenitor A sets.

The situation for continuous variables follows the same pattern, but with many more deep technicalities that we are going to skip. For the continuous case, the random variable is now,

$$X : \mathbb{R} \mapsto \mathbb{R}$$

with corresponding probability measure,

$$\mathbb{P}_X : \mathbb{R} \mapsto [0, 1]$$

But where are the corresponding sets here? Technically, these are the *Borel* sets, but we can just think of them as intervals. Returning to our question, what is the probability that a uniformly distributed random variable on the unit interval takes values less than 1/2? Rephrasing this question according to the framework, we have the following:

$$X : [0, 1] \mapsto [0, 1]$$

with corresponding,

$$\mathbb{P}_X : [0, 1] \mapsto [0, 1]$$

To answer the question, by the definition of the uniform random variable on the unit interval, we compute the following integral,

$$\mathbb{P}_X([0, 1/2]) = \mathbb{P}_X(0 < X < 1/2) = \int_0^{1/2} dx = 1/2$$

where the above integral's dx sweeps through intervals of the B-type. The measure of any dx interval (i.e., A-type set) is equal to dx, by definition of the uniform random variable. To get all the moving parts into one notationally rich integral, we can also write this as,

$$\mathbb{P}_X(0 < X < 1/2) = \int_0^{1/2} d\mathbb{P}_X(dx) = 1/2$$

Now, let's consider a slightly more complicated and interesting example. As before, suppose we have a uniform random variable, X and let us introduce another random variable defined,

$$Y = 2X$$

Now, what is the probability that $0 < Y < \frac{1}{2}$? To express this in our framework, we write,

$$Y : [0, 1] \mapsto [0, 2]$$

with corresponding,

$$\mathbb{P}_Y : [0, 2] \mapsto [0, 1]$$

To answer the question, we need to measure the set [0,1/2], with the probability measure for Y, $\mathbb{P}_Y([0, 1/2])$. How can we do this? Because Y is derived from the X random variable, as with the fair-dice throwing experiment, we have to create a set of equivalences in the target space (i.e., B-type sets) that reflect back on the input space (i.e., A-type sets). That is, what is the interval [0,1/2] equivalent to in terms of the X random variable? Because, functionally, $Y = 2X$, then the B-type interval [0,1/2] corresponds to the A-type interval [0,1/4]. From the probability measure of X, we compute this with the integral,

$$\mathbb{P}_Y([0, 1/2]) = \mathbb{P}_X([0, 1/4]) = \int_0^{1/4} dx = 1/4$$

Now, let's up the ante and consider the following random variable,

$$Y = X^2$$

where now X is still uniformly distributed, but now over the interval $[-1/2, 1/2]$. We can express this in our framework as,

$$Y : [-1/2, 1/2] \mapsto [0, 1/4]$$

with corresponding,

$$\mathbb{P}_Y : [0, 1/4] \mapsto [0, 1]$$

What is the $\mathbb{P}_Y(Y < 1/8)$? In other words, what is the measure of the set $B_Y = [0, 1/8]$? As before, because X is derived from our uniformly distributed random variable, we have to reflect the B_Y set onto sets of the A-type. The thing to recognize is that because X^2 is symmetric about zero, all B_Y sets reflect back into two sets. This means that for any set B_Y, we have the correspondence $B_Y = A_X^+ \cup A_X^-$. So, we have,

$$B_Y = \left\{0 < Y < \frac{1}{8}\right\} = \left\{0 < X < \frac{1}{\sqrt{8}}\right\} \bigcup \left\{ -\frac{1}{\sqrt{8}} < X < 0\right\}$$

From this perspective, we have the following solution,

$$\mathbb{P}_Y(B_Y) = \mathbb{P}(A_X^+)/2 + \mathbb{P}(A_X^-)/2$$

where the $\frac{1}{2}$ comes from normalizing the $\mathbb{P}_Y$ to one. Also,

$$A_X^+ = \left\{0 < X < \frac{1}{\sqrt{8}}\right\}$$
$$A_X^- = \left\{-\frac{1}{\sqrt{8}} < X < 0\right\}$$

Therefore,

$$\mathbb{P}_Y(B_Y) = \frac{1}{2\sqrt{8}} + \frac{1}{2\sqrt{8}}$$

because $\mathbb{P}(A_X^+) = \mathbb{P}(A_X^-) = 1/\sqrt{8}$. Let's see if this comes out using the usual transformation of variables method from calculus. Using this method, the density $f_Y(y) = f_X(\sqrt{y})/(2\sqrt{y}) = \frac{1}{2\sqrt{y}}$. Then, we obtain,

$$\int_0^{\frac{1}{8}} \frac{1}{2\sqrt{y}} dy = \frac{1}{\sqrt{8}}$$

which is what we got using the sets method. Note that you would favor the calculus method in practice, but it is important to understand the deeper mechanics, because sometimes the usual calculus method fails, as the next problem shows.

2.1.4 Transformation of Variables Beyond Calculus

Suppose X and Y are uniformly distributed in the unit interval and we define Z as

$$Z = \frac{X}{Y - X}$$

What is the $f_Z(z)$? If you try this using the usual calculus method, you will fail (try it!). The problem is one of the technical prerequisites for the calculus method is not in force.

The key observation is that $Z \notin [-1, 0]$. If this were possible, the X and Y would have different signs, which cannot happen, given that X and Y are uniformly distributed over $[0, 1]$. Now, let's consider when $Z > 0$. In this case, $Y > X$ because Z cannot be positive otherwise. For the density function, we are interested in the set $\{0 < Z < z\}$. We want to compute

$$\mathbb{P}(Z < z) = \int \int B_1 dX dY$$

with,

$$B_1 = \{0 < Z < z\}$$

Now, we have to translate that interval into an interval relevant to X and Y. For $0 < Z$, we have $Y > X$. For $Z < z$, we have $Y > X(1/z + 1)$. Putting this together gives

$$A_1 = \{\max(X, X(1/z + 1)) < Y < 1\}$$

Integrating this over Y as follows,

$$\int_0^1 \{\max(X, X(1/z + 1)) < Y < 1\} dY = \frac{z - X - Xz}{z} \text{ where } z > \frac{X}{1 - X}$$

and integrating this one more time over X gives

$$\int_0^{\frac{z}{1+z}} \frac{-X + z - Xz}{z} dX = \frac{z}{2(z + 1)} \text{ where } z > 0$$

Note that this is the computation for the *probability* itself, not the probability density function. To get that, all we have to do is differentiate the last expression to obtain

$$f_Z(z) = \frac{1}{(z + 1)^2} \text{ where } z > 0$$

Now we need to compute this density using the same process for when $z < -1$. We want the interval $Z < z$ for when $z < -1$. For a fixed z, this is equivalent to $X(1 + 1/z) < Y$. Because z is negative, this also means that $Y < X$. Under these terms, we have the following integral,

$$\int_0^1 \{X(1/z + 1) < Y < X\} dY = -\frac{X}{z} \text{ where } z < -1$$

and integrating this one more time over X gives the following

$$-\frac{1}{2z} \text{ where } z < -1$$

To get the density for $z < -1$, we differentiate this with respect to z to obtain the following,

$$f_Z(z) = \frac{1}{2z^2} \text{ where } z < -1$$

Putting this all together, we obtain,

$$f_Z(z) = \begin{cases} \frac{1}{(z+1)^2} & \text{if } z > 0 \\ \frac{1}{2z^2} & \text{if } z < -1 \\ 0 & \text{otherwise} \end{cases}$$

We will leave it as an exercise to show that this integrates out to one.

2.1.5 Independent Random Variables

Independence is a standard assumption. Mathematically, the necessary and sufficient condition for independence between two random variables X and Y is the following:

$$\mathbb{P}(X, Y) = \mathbb{P}(X)\mathbb{P}(Y)$$

Two random variables X and Y are *uncorrelated* if,

$$\mathbb{E}(X - \overline{X})\mathbb{E}(Y - \overline{Y}) = 0$$

where $\overline{X} = \mathbb{E}(X)$ Note that uncorrelated random variables are sometimes called *orthogonal* random variables. Uncorrelatedness is a weaker property than independence, however. For example, consider the discrete random variables X and Y uniformly distributed over the set $\{1, 2, 3\}$ where

$$X = \begin{cases} 1 & \text{if } \omega = 1 \\ 0 & \text{if } \omega = 2 \\ -1 & \text{if } \omega = 3 \end{cases}$$

and also,

$$Y = \begin{cases} 0 & \text{if } \omega = 1 \\ 1 & \text{if } \omega = 2 \\ 0 & \text{if } \omega = 3 \end{cases}$$

Thus, $\mathbb{E}(X) = 0$ and $\mathbb{E}(XY) = 0$, so X and Y are uncorrelated. However, we have

$$\mathbb{P}(X = 1, Y = 1) = 0 \neq \mathbb{P}(X = 1)\mathbb{P}(Y = 1) = \frac{1}{9}$$

So, these two random variables are *not* independent. Thus, uncorrelatedness does not imply independence, generally, but there is the important case of Gaussian random variables for which it does. To see this, consider the probability density function for two zero-mean, unit-variance Gaussian random variables X and Y,

$$f_{X,Y}(x, y) = \frac{e^{\frac{x^2 - 2\rho xy + y^2}{2(\rho^2 - 1)}}}{2\pi\sqrt{1 - \rho^2}}$$

where $\rho := \mathbb{E}(XY)$ is the correlation coefficient. In the uncorrelated case where $\rho = 0$, the probability density function factors into the following,

$$f_{X,Y}(x, y) = \frac{e^{-\frac{1}{2}(x^2 + y^2)}}{2\pi} = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \frac{e^{-\frac{y^2}{2}}}{\sqrt{2\pi}} = f_X(x) f_Y(y)$$

which means that X and Y are independent.

Independence and conditional independence are closely related, as in the following:

$$\mathbb{P}(X, Y|Z) = \mathbb{P}(X|Z)\mathbb{P}(Y|Z)$$

which says that X and Y and independent conditioned on Z. Conditioning independent random variables can break their independence. For example, consider two independent Bernoulli-distributed random variables, $X_1, X_2 \in \{0, 1\}$. We define $Z = X_1 + X_2$. Note that $Z \in \{0, 1, 2\}$. In the case where $Z = 1$, we have,

$$\mathbb{P}(X_1|Z = 1) > 0$$
$$\mathbb{P}(X_2|Z = 1) > 0$$

Even though X_1, X_2 are independent, after conditioning on Z, we have the following,

$$\mathbb{P}(X_1 = 1, X_2 = 1|Z = 1) = 0 \neq \mathbb{P}(X_1 = 1|Z = 1)\mathbb{P}(X_2 = 1|Z = 1)$$

Thus, conditioning on Z breaks the independence of X_1, X_2. This also works in the opposite direction — conditioning can make dependent random variables independent. Define $Z_n = \sum_i^n X_i$ with X_i independent, integer-valued random variables. The Z_n variables are dependent because they stack the same telescoping set of X_i variables. Consider the following,

$$\mathbb{P}(Z_1 = i, Z_3 = j|Z_2 = k) = \frac{\mathbb{P}(Z_1 = i, Z_2 = k, Z_3 = j)}{\mathbb{P}(Z_2 = k)} \tag{2.1.5.1}$$
$$= \frac{\mathbb{P}(X_1 = i)\mathbb{P}(X_2 = k - i)\mathbb{P}(X_3 = j - k)}{\mathbb{P}(Z_2 = k)} \tag{2.1.5.2}$$

where the factorization comes from the independence of the X_i variables. Using the definition of conditional probability,

$$\mathbb{P}(Z_1 = i|Z_2) = \frac{\mathbb{P}(Z_1 = i, Z_2 = k)}{\mathbb{P}(Z_2 = k)}$$

We can continue to expand Eq. 2.1.5,

$$\begin{aligned}\mathbb{P}(Z_1 = i, Z_3 = j|Z_2 = k) &= \mathbb{P}(Z_1 = i|Z_2)\frac{\mathbb{P}(X_3 = j - k)\mathbb{P}(Z_2 = k)}{\mathbb{P}(Z_2 = k)} \\ &= \mathbb{P}(Z_1 = i|Z_2)\mathbb{P}(Z_3 = j|Z_2)\end{aligned}$$

where $\mathbb{P}(X_3 = j - k)\mathbb{P}(Z_2 = k) = \mathbb{P}(Z_3 = j, Z_2)$. Thus, we see that dependence between random variables can be broken by conditioning to create conditionally independent random variables. As we have just witnessed, understanding how conditioning influences independence is important and is the main topic of study in Probabilistic Graphical Models, a field with many algorithms and concepts to extract these notions of conditional independence from graph-based representations of random variables.

2.1.6 *Classic Broken Rod Example*

Let's do one last example to exercise fluency in our methods by considering the following classic problem: given a rod of unit-length, broken independently and randomly at two places, what is the probability that you can assemble the three remaining pieces into a triangle? The first task is to find a representation of a triangle as an easy-to-apply constraint. What we want is something like the following:

$$\mathbb{P}(\text{ triangle exists }) = \int_0^1 \int_0^1 \{\text{ triangle exists }\} dX dY$$

where X and Y are independent and uniformly distributed in the unit-interval. Heron's formula for the area of the triangle,

$$\text{area } = \sqrt{(s-a)(s-b)(s-c)s}$$

where $s = (a + b + c)/2$ is what we need. The idea is that this yields a valid area only when each of the terms under the square root is greater than or equal to zero. Thus, suppose that we have

$$\begin{aligned}a &= X \\ b &= Y - X \\ c &= 1 - Y\end{aligned}$$

assuming that $Y > X$. Thus, the criterion for a valid triangle boils down to

$$\{(s > a) \wedge (s > b) \wedge (s > c) \wedge (X < Y)\}$$

After a bit of manipulation, this consolidates into:

$$\left\{\frac{1}{2} < Y < 1 \bigwedge \frac{1}{2}(2Y - 1) < X < \frac{1}{2}\right\}$$

which we integrate out by dX first to obtain

$$\mathbb{P}(\text{ triangle exists }) = \int_0^1 \int_0^1 \left\{\frac{1}{2} < Y < 1 \bigwedge \frac{1}{2}(2Y - 1) < X < \frac{1}{2}\right\} dX dY$$

$$\mathbb{P}(\text{ triangle exists }) = \int_{\frac{1}{2}}^1 (1 - Y) dY$$

and then by dY to obtain finally,

$$\mathbb{P}(\text{ triangle exists }) = \frac{1}{8}$$

when $Y > X$. By symmetry, we get the same result for $X > Y$. Thus, the final result is the following:

$$\mathbb{P}(\text{ triangle exists }) = \frac{1}{8} + \frac{1}{8} = \frac{1}{4}$$

We can quickly check using this result using Python for the case $Y > X$ using the following code:

```
>>> import numpy as np
>>> x,y = np.random.rand(2,1000) # uniform rv
>>> a,b,c = x,(y-x),1-y # 3 sides
>>> s = (a+b+c)/2
>>> np.mean((s>a) & (s>b)  & (s>c) & (y>x)) # approx 1/8=0.125
0.115
```

Programming Tip

The chained logical & symbols above tell Numpy that the logical operation should be considered element-wise.

2.2 Projection Methods

The concept of projection is key to developing an intuition about conditional probability. We already have a natural intuition of projection from looking at the shadows of objects on a sunny day. As we will see, this simple idea consolidates many abstract

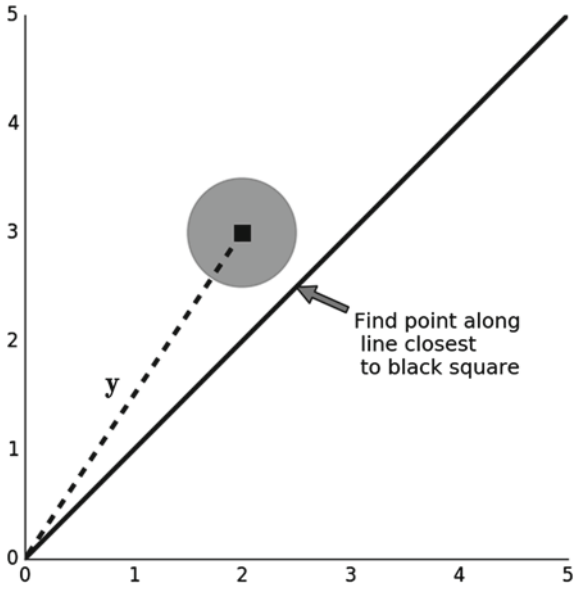

Fig. 2.2 Given the point **y** (*black square*) we want to find the **x** along the line that is closest to it. The *gray circle* is the locus of points within a fixed distance from **y**

ideas in optimization and mathematics. Consider Fig. 2.2 where we want to find a point along the blue line (namely, **x**) that is closest to the black square (namely, **y**). In other words, we want to inflate the gray circle until it just touches the black line. Recall that the circle boundary is the set of points for which

$$\sqrt{(\mathbf{y}-\mathbf{x})^T(\mathbf{y}-\mathbf{x})} = \|\mathbf{y}-\mathbf{x}\| = \epsilon$$

for some value of ϵ. So we want a point **x** along the line that satisfies this for the smallest ϵ. Then, that point will be the closest point on the black line to the black square. It may be obvious from the diagram, but the closest point on the line occurs where the line segment from the black square to the black line is perpedicular to the line. At this point, the gray circle just touches the black line. This is illustrated below in Fig. 2.3.

Programming Tip

Figure 2.2 uses the `matplotlib.patches` module. This module contains primitive shapes like circles, ellipses, and rectangles that can be assembled into complex graphics. As shown in the code in the IPython Notebook corresponding to this chapter, after importing a particular shape, you can apply that shape to an existing axis using the `add_patch` method. The patches themselves can by styled using the usual formatting keywords like `color` and `alpha`.

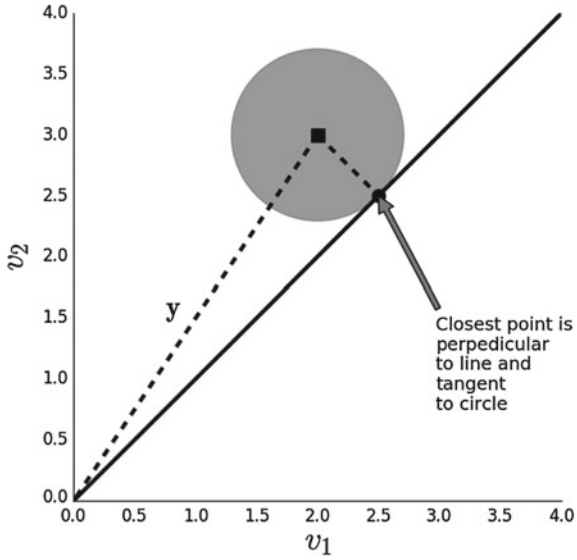

Fig. 2.3 The closest point on the line occurs when the line is tangent to the *circle*. When this happens, the *black line* and the line (minimum distance) are perpedicular

Now that we can see what's going on, we can construct the solution analytically. We can represent an arbitrary point along the black line as:

$$\mathbf{x} = \alpha \mathbf{v}$$

where $\alpha \in \mathbb{R}$ slides the point up and down the line with

$$\mathbf{v} = [1, 1]^T$$

Formally, $\mathbf{v}$ is the *subspace* onto which we want to *project* $\mathbf{y}$. At the closest point, the vector between $\mathbf{y}$ and $\mathbf{x}$ (the *error* vector above) is perpedicular to the line. This means that

$$(\mathbf{y} - \mathbf{x})^T \mathbf{v} = 0$$

and by substituting and working out the terms, we obtain

$$\alpha = \frac{\mathbf{y}^T \mathbf{v}}{\|\mathbf{v}\|^2}$$

The *error* is the distance between $\alpha\mathbf{v}$ and $\mathbf{y}$. This is a right triangle, and we can use the Pythagorean theorem to compute the squared length of this error as

$$\epsilon^2 = \|(\mathbf{y} - \mathbf{x})\|^2 = \|\mathbf{y}\|^2 - \alpha^2 \|\mathbf{v}\|^2 = \|\mathbf{y}\|^2 - \frac{\|\mathbf{y}^T \mathbf{v}\|^2}{\|\mathbf{v}\|^2}$$

where $\|\mathbf{v}\|^2 = \mathbf{v}^T\mathbf{v}$. Note that since $\epsilon^2 \geq 0$, this also shows that

$$\|\mathbf{y}^T\mathbf{v}\| \leq \|\mathbf{y}\|\|\mathbf{v}\|$$

which is the famous and useful Cauchy-Schwarz inequality which we will exploit later. Finally, we can assemble all of this into the *projection* operator

$$\mathbf{P}_v = \frac{1}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{v}^T$$

With this operator, we can take any $\mathbf{y}$ and find the closest point on $\mathbf{v}$ by doing

$$\mathbf{P}_v\mathbf{y} = \mathbf{v}\left(\frac{\mathbf{v}^T\mathbf{y}}{\|\mathbf{v}\|^2}\right)$$

where we recognize the term in parenthesis as the α we computed earlier. It's called an *operator* because it takes a vector ($\mathbf{y}$) and produces another vector ($\alpha\mathbf{v}$). Thus, projection unifies geometry and optimization.

2.2.1 Weighted Distance

We can easily extend this projection operator to cases where the measure of distance between $\mathbf{y}$ and the subspace $\mathbf{v}$ is weighted. We can accommodate these weighted distances by re-writing the projection operator as

$$\mathbf{P}_v = \mathbf{v}\frac{\mathbf{v}^T\mathbf{Q}^T}{\mathbf{v}^T\mathbf{Q}\mathbf{v}} \tag{2.2.1.1}$$

where $\mathbf{Q}$ is positive definite matrix. In the previous case, we started with a point $\mathbf{y}$ and inflated a circle centered at $\mathbf{y}$ until it just touched the line defined by $\mathbf{v}$ and this point was closest point on the line to $\mathbf{y}$. The same thing happens in the general case with a weighted distance except now we inflate an ellipse, not a circle, until the ellipse touches the line.

Note that the error vector ($\mathbf{y} - \alpha\mathbf{v}$) in Fig. 2.4 is still perpendicular to the line (subspace $\mathbf{v}$), but in the space of the weighted distance. The difference between the first projection (with the uniform circular distance) and the general case (with the elliptical weighted distance) is the inner product between the two cases. For example, in the first case we have $\mathbf{y}^T\mathbf{v}$ and in the weighted case we have $\mathbf{y}^T\mathbf{Q}^T\mathbf{v}$. To move from the uniform circular case to the weighted ellipsoidal case, all we had to do was change all of the vector inner products. Before we finish, we need a formal property of projections:

$$\mathbf{P}_v\mathbf{P}_v = \mathbf{P}_v$$

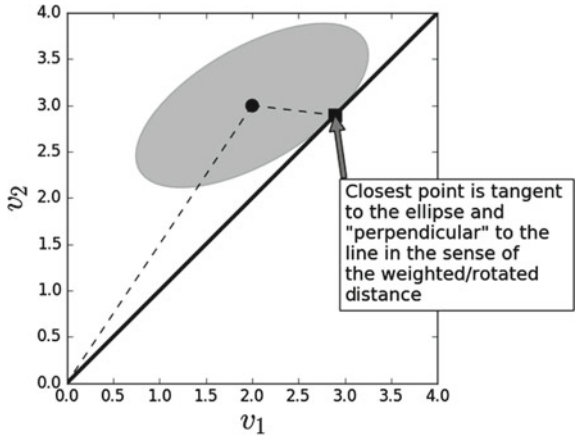

Fig. 2.4 In the weighted case, the closest point on the line is tangent to the *ellipse* and is still perpedicular in the sense of the weighted distance

known as the *idempotent* property which basically says that once we have projected onto a subspace, subsequent projections leave us in the same subspace. You can verify this by computing Eq. 2.2.1.

Thus, projection ties a minimization problem (closest point to a line) to an algebraic concept (inner product). It turns out that these same geometric ideas from linear algebra [2] can be translated to the conditional expectation. How this works is the subject of our next section.

2.3 Conditional Expectation as Projection

Now that we understand projection methods geometrically, we can apply them to conditional probability. This is the *key* concept that ties probability to geometry, optimization, and linear algebra.

Inner Product for Random Variables. From our previous work on projection for vectors in $\mathbb{R}^n$, we have a good geometric grasp on how projection is related to Minimum Mean Squared Error (MMSE). By one abstract step, we can carry all of our geometric interpretations to the space of random variables. For example, we previously noted that at the point of projection, we had the following orthogonal (i.e., perpendicular vectors) condition,

$$(\mathbf{y} - \mathbf{v}_{opt})^T \mathbf{v} = 0$$

which by noting the inner product slightly more abstractly as $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y}$, we can express as

$$\langle \mathbf{y} - \mathbf{v}_{opt}, \mathbf{v} \rangle = 0$$

and by defining the inner product for the random variables X and Y as

$$\langle X, Y \rangle = \mathbb{E}(XY)$$

we have the same relationship:

$$\langle X - h_{opt}(Y), Y \rangle = 0$$

which holds not for vectors in $\mathbb{R}^n$, but for random variables X and Y and functions of those random variables. Exactly why this is true is technical, but it turns out that one can build up the *entire theory of probability* this way [3], by using the expectation as an inner product.

Furthermore, by abstracting out the inner product concept, we have connected minimum-mean-squared-error (MMSE) optimization problems, geometry, and random variables. That's a lot of mileage to get a out of an abstraction and it enables us to shift between these interpretations to address real problems. Soon, we'll do this with some examples, but first we collect the most important result that flows naturally from this abstraction.

Conditional Expectation as Projection. The conditional expectation is the minimum mean squared error (MMSE) solution to the following problem[1]:

$$\min_h \int_{\mathbb{R}} (x - h(y))^2 dx$$

with the minimizing $h_{opt}(Y)$ as

$$h_{opt}(Y) = \mathbb{E}(X|Y)$$

which is another way of saying that among all possible functions $h(Y)$, the one that minimizes the MSE is $\mathbb{E}(X|Y)$. From our previous discussion on projection, we noted that these MMSE solutions can be thought of as projections onto a subspace that characterizes Y. For example, we previously noted that at the point of projection, we have perpendicular terms,

$$\langle X - h_{opt}(Y), Y \rangle = 0 \tag{2.3.0.2}$$

but since we know that the MMSE solution

$$h_{opt}(Y) = \mathbb{E}(X|Y)$$

we have by direct substitution,

$$\mathbb{E}(X - \mathbb{E}(X|Y), Y) = 0 \tag{2.3.0.3}$$

[1] See appendix for proof using the Cauchy-Schwarz inequality.

That last step seems pretty innocuous, but it ties MMSE to conditional expectation to the inner project abstraction, and in so doing, reveals the conditional expectation to be a projection operator for random variables. Before we develop this further, let's grab some quick dividends. From the previous equation, by linearity of the expectation, we obtain,

$$\mathbb{E}(XY) = \mathbb{E}(Y\mathbb{E}(X|Y))$$

which is the so-called *tower property* of the expectation. Note that we could have found this by using the formal definition of conditional expectation,

$$\mathbb{E}(X|Y) = \int_{\mathbb{R}^2} x \frac{f_{X,Y}(x, y)}{f_Y(y)} dxdy$$

and brute-force direct integration,

$$\begin{aligned}\mathbb{E}(Y\mathbb{E}(X|Y)) &= \int_{\mathbb{R}} y \int_{\mathbb{R}} x \frac{f_{X,Y}(x, y)}{f_Y(y)} f_Y(y) dxdy \\ &= \int_{\mathbb{R}^2} xy f_{X,Y}(x, y) dxdy \\ &= \mathbb{E}(XY)\end{aligned}$$

which is not very geometrically intuitive. This lack of geometric intuition makes it hard to apply these concepts and keep track of these relationships.

We can keep pursuing this analogy and obtain the length of the error term from the orthogonality property of the MMSE solution as,

$$\langle X - h_{opt}(Y), X - h_{opt}(Y)\rangle = \langle X, X\rangle - \langle h_{opt}(Y), h_{opt}(Y)\rangle$$

and then by substituting all the notation we obtain

$$\mathbb{E}(X - \mathbb{E}(X|Y))^2 = \mathbb{E}(X)^2 - \mathbb{E}(\mathbb{E}(X|Y))^2$$

which would be tough to compute by direct integration.

To formally establish that $\mathbb{E}(X|Y)$ *is* in fact *a projection operator* we need to show idempotency. Recall that idempotency means that once we project something onto a subspace, further projections do nothing. In the space of random variables, $\mathbb{E}(X|\cdot)$ is the idempotent projection as we can show by noting that

$$h_{opt} = \mathbb{E}(X|Y)$$

is purely a function of Y, so that

$$\mathbb{E}(h_{opt}(Y)|Y) = h_{opt}(Y)$$

because Y is fixed, this verifies idempotency. Thus, conditional expectation is the corresponding projection operator for random variables. We can continue to carry over our geometric interpretations of projections for vectors ($\mathbf{v}$) into random variables (X). With this important result, let's consider some examples of conditional expectations obtained by using brute force to find the optimal MMSE function h_{opt} as well as by using our new perspective on conditional expectation.

Example Suppose we have a random variable, X, then what constant is closest to X in the sense of the mean-squared-error (MSE)? In other words, which $c \in \mathbb{R}$ minimizes the following mean squared error:

$$\text{MSE} = \mathbb{E}(X - c)^2$$

we can work this out many ways. First, using calculus-based optimization,

$$\mathbb{E}(X - c)^2 = \mathbb{E}(c^2 - 2cX + X^2) = c^2 - 2c\mathbb{E}(X) + \mathbb{E}(X^2)$$

and then take the first derivative with respect to c and solve:

$$c_{opt} = \mathbb{E}(X)$$

Remember that X may potentially take on many values, but this says that the closest number to X in the MSE sense is $\mathbb{E}(X)$. This is intuitively pleasing. Coming at this same problem using our inner product, from Eq. 2.3.0.3 we know that at the point of projection

$$\mathbb{E}((X - c_{opt})1) = 0$$

where the 1 represents the space of constants we are projecting onto. By linearity of the expectation, gives

$$c_{opt} = \mathbb{E}(X)$$

Using the projection approach, because $\mathbb{E}(X|Y)$ is the projection operator, with $Y = \Omega$ (the entire underlying probability space), we have, using the definition of conditional expectation:

$$\mathbb{E}(X|Y = \Omega) = \mathbb{E}(X)$$

This is because of the subtle fact that a random variable over the entire Ω space can only be a constant. Thus, we just worked the same problem three ways (optimization, orthogonal inner products, projection).

Example Let's consider the following example with probability density $f_{X,Y} = x+y$ where $(x, y) \in [0, 1]^2$ and compute the conditional expectation straight from the definition:

$$\mathbb{E}(X|Y) = \int_0^1 x \frac{f_{X,Y}(x, y)}{f_Y(y)} dx = \int_0^1 x \frac{x + y}{y + 1/2} dx = \frac{3y + 2}{6y + 3}$$

That was pretty easy because the density function was so simple. Now, let's do it the hard way by going directly for the MMSE solution $h(Y)$. Then,

$$\begin{aligned}\text{MSE} &= \min_h \int_0^1 \int_0^1 (x - h(y))^2 f_{X,Y}(x, y) dx dy \\ &= \min_h \int_0^1 y h^2(y) - y h(y) + \frac{1}{3} y + \frac{1}{2} h^2(y) - \frac{2}{3} h(y) + \frac{1}{4} dy\end{aligned}$$

Now we have to find a function h that is going to minimize this. Solving for a function, as opposed to solving for a number, is generally very, very hard, but because we are integrating over a finite interval, we can use the Euler-Lagrange method from variational calculus to take the derivative of the integrand with respect to the function $h(y)$ and set it to zero. Using Euler-Lagrange methods, we obtain the following result,

$$2yh(y) - y + h(y) - \frac{2}{3} = 0$$

Solving this gives

$$h_{opt}(y) = \frac{3y + 2}{6y + 3}$$

which is what we obtained before. Finally, we can solve this using our inner product in Eq. 2.3.0.2 as

$$\mathbb{E}((X - h(Y))Y) = 0$$

Writing this out gives,

$$\int_0^1 \int_0^1 (x - h(y)) y (x + y) dx dy = \int_0^1 \frac{1}{6} y(-3(2y + 1)h(y) + 3y + 2) dy = 0$$

and the integrand must be zero,

$$2y + 3y^2 - 3yh(y) - 6y^2 h(y) = 0$$

and solving this for $h(y)$ gives the same solution:

$$h_{opt}(y) = \frac{3y + 2}{6y + 3}$$

Thus, doing it by the brute force integration from the definition, optimization, or inner product gives us the same answer; but, in general, no method is necessarily easiest because they both involve potentially difficult or impossible integration, optimization, or functional equation solving. The point is that now that we have a deep toolbox, we can pick and choose which tools we want to apply for different problems.

Before we leave this example, let's use Sympy to verify the length of the error function we found earlier for this example:

$$\mathbb{E}(X - \mathbb{E}(X|Y))^2 = \mathbb{E}(X)^2 - \mathbb{E}(\mathbb{E}(X|Y))^2$$

that is based on the Pythagorean theorem. First, we need to compute the marginal densities,

```
>>> from sympy.abc import y,x
>>> from sympy import integrate, simplify
>>> fxy = x + y                 # joint density
>>> fy = integrate(fxy,(x,0,1)) # marginal density
>>> fx = integrate(fxy,(y,0,1)) # marginal density
```

Then, we need to write out the conditional expectation,

```
>>> EXY = (3*y+2)/(6*y+3) # conditional expectation
```

Next, we can compute the left side, $\mathbb{E}(X - \mathbb{E}(X|Y))^2$, as the following,

```
>>> # from the definition
>>> LHS=integrate((x-EXY)**2*fxy,(x,0,1),(y,0,1))
>>> LHS # left-hand-side
-log(216)/144 + log(72)/144 + 1/12
```

We can similarly compute the right side, $\mathbb{E}(X)^2 - \mathbb{E}(\mathbb{E}(X|Y))^2$, as the following,

```
>>> # using Pythagorean theorem
>>> RHS=integrate((x)**2*fx,(x,0,1))-integrate((EXY)**2*fy,(y,0,1))
>>> RHS # right-hand-side
-log(216)/144 + log(72)/144 + 1/12
```

Finally, we can verify that the left and right sides match,

```
>>> print simplify(LHS-RHS)==0
True
```

In this section, we have pulled together all the projection and least-squares optimization ideas from the previous sections to connect geometric notions of projection from vectors in $\mathbb{R}^n$ to random variables. This resulted in the remarkable realization that the conditional expectation is in fact a projection operator for random variables. Knowing this allows to approach difficult problems in multiple ways, depending on which way is more intuitive or tractable in a particular situation. Indeed, finding the right problem to solve is the hardest part, so having many ways of looking at the same concepts is crucial.

For much more detailed development, the book by Mikosch [4] has some excellent sections covering much of this material with a similar geometric interpretation. Kobayashi [5] does too. Nelson [3] also has a similar presentation based on hyper-real numbers.

2.3.1 Appendix

We want to prove that we the conditional expectation is the minimum mean squared error minimizer of the following:

$$J = \min_h \int_{\mathbb{R}^2} |X - h(Y)|^2 f_{X,Y}(x, y) dx dy$$

We can expand this as follows,

$$J = \min_h \int_{\mathbb{R}^2} |X|^2 f_{X,Y}(x, y) dx dy + \int_{\mathbb{R}^2} |h(Y)|^2 f_{X,Y}(x, y) dx dy - \int_{\mathbb{R}^2} 2Xh(Y) f_{X,Y}(x, y) dx dy$$

To minimize this, we have to maximize the following:

$$A = \max_h \int_{\mathbb{R}^2} Xh(Y) f_{X,Y}(x, y) dx dy$$

Breaking up the integral using the definition of conditional expectation

$$A = \max_h \int_{\mathbb{R}} \left(\int_{\mathbb{R}} X f_{X|Y}(x|y) dx \right) h(Y) f_Y(y) dy \tag{2.3.1.1}$$

$$= \max_h \int_{\mathbb{R}} \mathbb{E}(X|Y) h(Y) f_Y(Y) dy \tag{2.3.1.2}$$

From properties of the Cauchy-Schwarz inequality, we know that the maximum happens when $h_{opt}(Y) = \mathbb{E}(X|Y)$, so we have found the optimal $h(Y)$ function as:

$$h_{opt}(Y) = \mathbb{E}(X|Y)$$

which shows that the optimal function is the conditional expectation.

2.4 Conditional Expectation and Mean Squared Error

In this section, we work through a detailed example using conditional expectation and optimization methods. Suppose we have two fair six-sided die (X and Y) and we want to measure the sum of the two variables as $Z = X + Y$. Further, let's suppose that given Z, we want the best estimate of X in the mean-squared-sense. Thus, we want to minimize the following:

$$J(\alpha) = \sum (x - \alpha z)^2 \mathbb{P}(x, z)$$

where $\mathbb{P}$ is the probability mass function for this problem. The idea is that when we have solved this problem, we will have a function of Z that is going to be the minimum MSE estimate of X. We can substitute in for Z in J and get:

$$J(\alpha) = \sum (x - \alpha(x + y))^2 \mathbb{P}(x, y)$$

Let's work out the steps in Sympy in the following:

```
>>> import sympy as S
>>> from sympy.stats import density, E, Die

>>> x=Die('D1',6)      # 1st six sided die
>>> y=Die('D2',6)      # 2nd six sides die
>>> a=S.symbols('a')
>>> z = x+y            # sum of 1st and 2nd die
>>> J = E((x-a*(x+y))**2) # expectation
>>> print S.simplify(J)
329*a**2/6 - 329*a/6 + 91/6
```

With all that setup we can now use basic calculus to minimize the objective function J,

```
>>> sol,=S.solve(S.diff(J,a),a) # using calculus to minimize
>>> print sol # solution is 1/2
1/2
```

Programming Tip

Sympy has a `stats` module that can do some basic work with expressions involving probability densities and expectations. The above code uses its `E` function to compute the expectation.

This says that $z/2$ is the MSE estimate of X given Z which means geometrically (interpreting the MSE as a squared distance weighted by the probability mass function) that $z/2$ is as *close* to x as we are going to get for a given z.

Let's look at the same problem using the conditional expectation operator $\mathbb{E}(\cdot|z)$ and apply it to our definition of Z. Then

$$\mathbb{E}(z|z) = \mathbb{E}(x + y|z) = \mathbb{E}(x|z) + \mathbb{E}(y|z) = z$$

using the linearity of the expectation. Now, since by the symmetry of the problem (i.e., two identical die), we have

$$\mathbb{E}(x|z) = \mathbb{E}(y|z)$$

we can plug this in and solve

$$2\mathbb{E}(x|z) = z$$

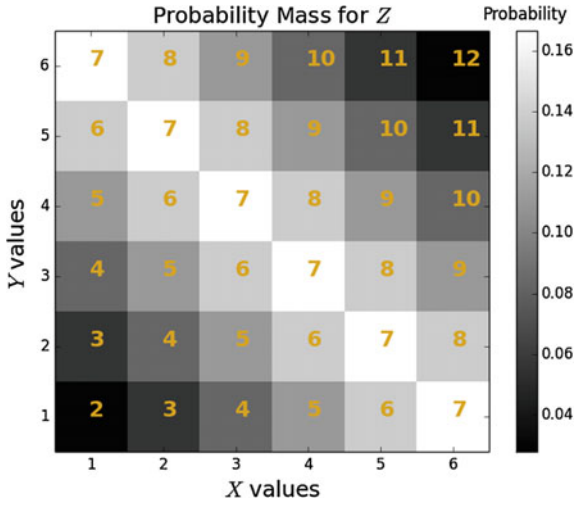

Fig. 2.5 The values of Z are in *yellow* with the corresponding values for X and Y on the axes. The *gray scale* colors indicate the underlying joint probability density

which once again gives,

$$\mathbb{E}(x|z) = \frac{z}{2}$$

which is equal to the estimate we just found by minimizing the MSE. Let's explore this further with Fig. 2.5. Figure 2.5 shows the values of Z in yellow with the corresponding values for X and Y on the axes. Suppose $z = 2$, then the closest X to this is $X = 1$, which is what $\mathbb{E}(x|z) = z/2 = 1$ gives. What happens when $Z = 7$? In this case, this value is spread out diagonally along the X axis so if $X = 1$, then Z is 6 units away, if $X = 2$, then Z is 5 units away and so on.

Now, back to the original question, if we had $Z = 7$ and we wanted to get as close as we could to this using X, then why not choose $X = 6$ which is only one unit away from Z? The problem with doing that is $X = 6$ only occurs 1/6 of the time, so we are not likely to get it right the other 5/6 of the time. So, 1/6 of the time we are one unit away but 5/6 of the time we are much more than one unit away. This means that the MSE score is going to be worse. Since each value of X from 1 to 6 is equally likely, to play it safe, we choose 7/2 as the estimate, which is what the conditional expectation suggests.

We can check this claim with samples using Sympy below:

```
>>> import numpy as np
>>> from sympy import stats
>>> # Eq constrains Z
>>> samples_z7 = lambda : stats.sample(x, S.Eq(z,7))
>>> #using 6 as an estimate
>>> mn= np.mean([(6-samples_z7())**2 for i in range(100)])
>>> #7/2 is the MSE estimate
>>> mn0= np.mean([(7/2.-samples_z7())**2 for i in range(100)])
>>> print 'MSE=%3.2f using 6 vs MSE=%3.2f using 7/2 ' % (mn,mn0)
MSE=10.85 using 6 vs MSE=2.29 using 7/2
```

Programming Tip

The `stats.sample(x, S.Eq(z,7))` function call samples the `x` variable subject to a condition on the `z` variable. In other words, it generates random samples of `x` die, given that the sum of the outcomes of that die and the `y` die add up to `z==7`.

Please run the above code repeatedly in the Jupyter/IPython Notebook corresponding to this section until you have convinced yourself that the $\mathbb{E}(x|z)$ gives the lower MSE every time. To push this reasoning, let's consider the case where the die is so biased so that the outcome of `6` is ten times more probable than any of the other outcomes. That is,

$$\mathbb{P}(6) = 2/3$$

whereas $\mathbb{P}(1) = \mathbb{P}(2) = \ldots = \mathbb{P}(5) = 1/15$. We can explore this using Sympy as in the following:

```
>>> # here 6 is ten times more probable than any other outcome
>>> x=stats.FiniteRV('D3',{1:1/15., 2:1/15.,
...                        3:1/15., 4:1/15.,
...                        5:1/15., 6:2/3.})
```

As before, we construct the sum of the two dice, and plot the corresponding probability mass function in Fig. 2.6. As compared with Fig. 2.5, the probability mass has been shifted away from the smaller numbers.

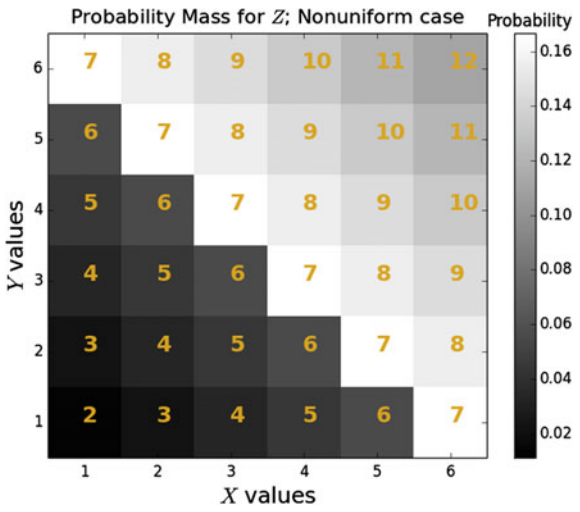

Fig. 2.6 The values of Z are in *yellow* with the corresponding values for X and Y on the axes

Let's see what the conditional expectation says about how we can estimate X from Z.

```
>>> E(x, S.Eq(z,7)) # conditional expectation E(x|z=7)
5.00000000000000
```

Now that we have $\mathbb{E}(x|z=7)=5$, we can generate samples as before and see if this gives the minimum MSE.

```
>>> samples_z7 = lambda : stats.sample(x, S.Eq(z,7))
>>> #using 6 as an estimate
>>> mn= np.mean([(6-samples_z7())**2 for i in range(100)])
>>> #5 is the MSE estimate
>>> mn0= np.mean([(5-samples_z7())**2 for i in range(100)])
>>> print 'MSE=%3.2f using 6 vs MSE=%3.2f using 5 ' % (mn,mn0)
MSE=2.69 using 6 vs MSE=2.33 using 5
```

Using a simple example, we have emphasized the connection between minimum mean squared error problems and conditional expectation. Hopefully, the last two figures helped expose the role of the probability density. Next, we'll continue revealing the true power of the conditional expectation as we continue to develop corresponding geometric intuition.

2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

Brzezniak [6] is a great book because it approaches conditional expectation through a sequence of exercises, which is what we are trying to do here. The main difference is that Brzezniak takes a more abstract measure-theoretic approach to the same problems. Note that you *do* need to grasp measure theory for advanced areas in probability, but for what we have covered so far, working the same problems in his text using our methods is illuminating. It always helps to have more than one way to solve *any* problem. I urge you to get a copy of his book or at least look at some pages on Google Books. I have numbered the examples corresponding to the book and tried to follow its notation.

2.5.1 Example

This is Example 2.1 from Brzezniak. Three coins, 10p, 20p and 50p are tossed. The values of the coins that land heads up are totaled. What is the expected total given that two coins have landed heads-up. In this case we have we want to compute $\mathbb{E}(\xi|\eta)$ where

$$\xi := 10X_{10} + 20X_{20} + 50X_{50}$$

where $X_i \in \{0, 1\}$ and where X_{10} is the Bernoulli-distributed random variable corresponding to the 10p coin (and so on). Thus, ξ represents the total value of the heads-up coins. The η represents the condition that only two of the three coins are heads-up,

$$\eta := X_{10}X_{20}(1 - X_{50}) + (1 - X_{10})X_{20}X_{50} + X_{10}(1 - X_{20})X_{50}$$

and is a function that is non-zero *only* when two of the three coins lands heads-up. Each triple term catches each of these three possibilities. For example, the first term equals one when the 10p and 20p are heads up and the 50p is heads down. The remaining terms are zero.

To compute the conditional expectation, we want to find a function h of η that minimizes the mean-squared-error (MSE),

$$\text{MSE} = \sum_{X \in \{0,1\}^3} \frac{1}{2^3}(\xi - h(\eta))^2$$

where the sum is taken over all possible triples of outcomes for $\{X_{10}, X_{20}, X_{50}\}$ because each of the three coins has a $\frac{1}{2}$ chance of coming up heads.

Now, the question boils down to how can we characterize the function $h(\eta)$? Note that $\eta \mapsto \{0, 1\}$ so h takes on only two values. So, the orthogonal inner product condition is the following:

$$\langle \xi - h(\eta), \eta \rangle = 0$$

But, because are only interested in $\eta = 1$, this simplifies to

$$\begin{aligned} \langle \xi - h(1), 1 \rangle &= 0 \\ \langle \xi, 1 \rangle &= \langle h(1), 1 \rangle \end{aligned}$$

This doesn't look so hard to evaluate but we have to compute the integral over the set where $\eta = 1$. In other words, we need the set of triples $\{X_{10}, X_{20}, X_{50}\}$ where $\eta = 1$. That is, we can compute

$$\int_{\{\eta=1\}} \xi dX = h(1) \int_{\{\eta=1\}} dX$$

which is what Brzezniak does. Instead, we can define $h(\eta) = \alpha\eta$ and then find α. Re-writing the orthogonal condition gives

$$\begin{aligned} \langle \xi - \eta, \alpha\eta \rangle &= 0 \\ \langle \xi, \eta \rangle &= \alpha \langle \eta, \eta \rangle \\ \alpha &= \frac{\langle \xi, \eta \rangle}{\langle \eta, \eta \rangle} \end{aligned}$$

where

$$\langle \xi, \eta \rangle = \sum_{X \in \{0,1\}^3} \frac{1}{2^3}(\xi\eta)$$

Note that we can just sweep over all triples $\{X_{10}, X_{20}, X_{50}\}$ because the definition of $h(\eta)$ zeros out when $\eta = 0$ anyway. All we have to do is plug everything in and solve. This tedious job is perfect for Sympy.

```
>>> import sympy as S
>>> X10,X20,X50 = S.symbols('X10,X20,X50',real=True)
>>> xi  = 10*X10+20*X20+50*X50
>>> eta = X10*X20*(1-X50)+X10*(1-X20)*(X50)+(1-X10)*X20*(X50)
>>> num=S.summation(xi*eta,(X10,0,1),(X20,0,1),(X50,0,1))
>>> den=S.summation(eta*eta,(X10,0,1),(X20,0,1),(X50,0,1))
>>> alpha=num/den
>>> print alpha # alpha=160/3
160/3
```

This means that

$$\mathbb{E}(\xi|\eta) = \frac{160}{3}\eta$$

which we can check with a quick simulation

```
>>> import pandas as pd
>>> d = pd.DataFrame(columns=['X10','X20','X50'])
>>> d.X10 = np.random.randint(0,2,1000)
>>> d.X10 = np.random.randint(0,2,1000)
>>> d.X20 = np.random.randint(0,2,1000)
>>> d.X50 = np.random.randint(0,2,1000)
```

Programming Tip

The code above creates an empty Pandas data frame with the named columns. The next four lines assigns values to each of the columns.

The code above simulates flipping the three coins 1000 times. Each column of the dataframe is either `0` or `1` corresponding to heads-down or heads-up, respectively. The condition is that two of the three coins have landed heads-up. Next, we can group the columns according to their sums. Note that the sum can only be in $\{0, 1, 2, 3\}$ corresponding to `0` heads-up, `1` heads-up, and so on.

```
>>> grp=d.groupby(d.eval('X10+X20+X50'))
```

Programming Tip

The `eval` function of the Pandas data frame takes the named columns and evaluates the given formula. At the time of this writing, only simple formulas involving primitive operations are possible.

Next, we can get the `2` group, which corresponds to exactly two coins having landed heads-up, and then evaluate the sum of the values of the coins. Finally, we can take the mean of these sums.

```
>>> grp.get_group(2).eval('10*X10+20*X20+50*X50').mean()
52.719999999999999
```

The result is close to `160/3=53.33` which supports the analytic result. The following code shows that we can accomplish the same simulation using pure Numpy.

```
>>> import numpy as np
>>> from numpy import array
>>> x=np.random.randint(0,2,(3,1000))
>>> print np.dot(x[:,x.sum(axis=0)==2].T,array([10,20,50])).mean()
52.698998418555611
```

In this case, we used the Numpy dot product to compute the value of the heads-up coins. The `sum(axis=0)==2` part selects the columns that correspond to two heads-up coins.

Still another way to get at the same problem is to forego the random sampling part and just consider all possibilities exhaustively using the `itertools` module in Python's standard library.

```
>>> import itertools as it
>>> list(it.product((0,1),(0,1),(0,1)))
[(0, 0, 0),
 (0, 0, 1),
 (0, 1, 0),
 (0, 1, 1),
 (1, 0, 0),
 (1, 0, 1),
 (1, 1, 0),
 (1, 1, 1)]
```

Note that we need to call `list` above in order to trigger the iteration in `it.product`. This is because the `itertools` module is generator-based so does not actually *do* the iteration until it is iterated over (by `list` in this case). This shows all possible triples (X_{10}, X_{20}, X_{50}) where `0` and `1` indicate heads-down and heads-up, respectively. The next step is to filter out the cases that correspond to two heads-up coins.

```
>>> list(it.ifilter(lambda i:sum(i)==2,it.product((0,1),(0,1),(0,1))))
[(0, 1, 1), (1, 0, 1), (1, 1, 0)]
```

Next, we need to compute the sum of the coins and combine the prior code.

```
>>> map(lambda k:10*k[0]+20*k[1]+50*k[2],
...                   it.ifilter(lambda i:sum(i)==2,
...                              it.product((0,1),(0,1),(0,1))))
[70, 60, 30]
```

The mean of the output is `53.33`, which is yet another way to get the same result. For this example, we demonstrated the full spectrum of approaches made possible using Sympy, Numpy, and Pandas. It is always valuable to have multiple ways of approaching the same problem and cross-checking the result.

2.5.2 Example

This is Example 2.2 from Brzezniak. Three coins, 10p, 20p and 50p are tossed as before. What is the conditional expectation of the total amount shown by the three coins given the total amount shown by the 10p and 20p coins only? For this problem,

$$\begin{aligned}\xi :=& 10X_{10} + 20X_{20} + 50X_{50}\\ \eta :=& 30X_{10}X_{20} + 20(1 - X_{10})X_{20} + 10X_{10}(1 - X_{20})\end{aligned}$$

which takes on four values $\eta \mapsto \{0, 10, 20, 30\}$ and only considers the 10p and 20p coins. In contrast to the last problem, here we are interested in $h(\eta)$ for all of the values of η. Naturally, there are only four values for $h(\eta)$ corresponding to each of these four values. Let's first consider $\eta = 10$. The orthogonal condition is then

$$\langle \xi - h(10), 10 \rangle = 0$$

The domain for $\eta = 10$ is $\{X_{10} = 1, X_{20} = 0, X_{50}\}$ which we can integrate out of the expectation below,

$$\begin{aligned}\mathbb{E}_{\{X_{10}=1,X_{20}=0,X_{50}\}}(\xi - h(10))10 &= 0\\ \mathbb{E}_{\{X_{50}\}}(10 - h(10) + 50X_{50}) &= 0\\ 10 - h(10) + 25 &= 0\end{aligned}$$

which gives $h(10) = 35$. Repeating the same process for $\eta \in \{20, 30\}$ gives $h(20) = 45$ and $h(30) = 55$, respectively. This is the approach Brzezniak takes. On the other hand, we can just look at affine functions, $h(\eta) = a\eta + b$ and use brute-force calculus.

```
>>> from sympy.abc import a,b
>>> h = a*eta + b
>>> eta = X10*X20*30 + X10*(1-X20)*(10)+ (1-X10)*X20*(20)
>>> MSE=S.summation((xi-h)**2*S.Rational(1,8),(X10,0,1),
...                 (X20,0,1),
...                 (X50,0,1))
>>> sol=S.solve([S.diff(MSE,a),S.diff(MSE,b)],(a,b))
>>> print sol
{b: 25, a: 1}
```

Programming Tip

The `Rational` function from Sympy code expresses a rational number that Sympy is able to manipulate as such. This is different that specifying a fraction like `1/8.`, which Python would automatically compute as a floating point number (i.e., `0.125`). The advantage of using `Rational` is that Sympy can later produce rational numbers as output, which are sometimes easier to make sense of.

This means that

$$\mathbb{E}(\xi|\eta) = 25 + \eta \tag{2.5.2.1}$$

since η takes on only four values, $\{0, 10, 20, 30\}$, we can write this out explicitly as

$$\mathbb{E}(\xi|\eta) = \begin{cases} 25 & \text{for } \eta = 0 \\ 35 & \text{for } \eta = 10 \\ 45 & \text{for } \eta = 20 \\ 55 & \text{for } \eta = 30 \end{cases} \tag{2.5.2.2}$$

Alternatively, we can use orthogonal inner products to write out the following conditions:

$$\langle \xi - h(\eta), \eta \rangle = 0 \tag{2.5.2.3}$$

$$\langle \xi - h(\eta), 1 \rangle = 0 \tag{2.5.2.4}$$

Writing these out and solving for a and b is tedious and a perfect job for Sympy. Starting with Eq. 2.5.2.3,

```
>>> expr = expr=S.expand((xi-h)*eta)
>>> print expr
-100*X10**2*a + 100*X10**2 - 400*X10*X20*a + 400*X10*X20 + 500*X10*X50
- 10*X10*b - 400*X20**2*a + 400*X20**2 + 1000*X20*X50 - 20*X20*b
```

and then because $\mathbb{E}(X_i^2) = 1/2 = \mathbb{E}(X_i)$, we make the following substitutions

```
>>> expr.xreplace({X10**2:0.5, X20**2:0.5,X10:0.5,X20:0.5,X50:0.5})
-350.0*a - 15.0*b + 725.0
```

We can do this for the other orthogonal inner product in Eq. 2.5.2.4 as follows,

Programming Tip

Because Sympy symbols are hashable, they can be used as keys in Python dictionaries as in the `xreplace` function above.

```
>>> print S.expand((xi-h)*1).xreplace({X10**2:0.5,
...                                    X20**2:0.5,
...                                    X10:0.5,
...                                    X20:0.5,
...                                    X50:0.5})
-15.0*a - b + 40.0
```

Then, combining this result with the previous one and solving for `a` and `b` gives,

```
>>> print S.solve([-350.0*a-15.0*b+725.0,-15.0*a-b+40.0])
{b: 25.0000000000000, a: 1.00000000000000}
```

which again gives us the final solution,

$$\mathbb{E}(\xi|\eta) = 25 + \eta$$

The following is a quick simulation to demonstrate this. We can build on the Pandas dataframe we used for the last example and create a new column for the sum of the 10p and 20p coins, as shown below.

```
>>> d['sm'] = d.eval('X10*10+X20*20')
```

We can group this by the values of this sum,

```
>>> d.groupby('sm').mean()
    X10  X20       X50
sm
0     0    0  0.484000
10    1    0  0.479839
20    0    1  0.503876
30    1    1  0.524590
```

But we want the expectation of the value of the coins

```
>>> d.groupby('sm').mean().eval('10*X10+20*X20+50*X50')
sm
0     24.200000
10    33.991935
20    45.193798
30    56.229508
```

which is very close to our analytical result in Eq. 2.5.2.2.

2.5.3 Example

This is Example 2.3 paraphrased from Brzezniak. Given X uniformly distributed on $[0, 1]$, find $\mathbb{E}(\xi|\eta)$ where

$$\xi(x) = 2x^2$$

$$\eta(x) = \begin{cases} 1 & \text{if } x \in [0, 1/3] \\ 2 & \text{if } x \in (1/3, 2/3) \\ 0 & \text{if } x \in (2/3, 1] \end{cases}$$

Note that this problem is different from the previous two because the sets that characterize η are intervals instead of discrete points. Nonetheless, we will eventually have three values for $h(\eta)$ because $\eta \mapsto \{0, 1, 2\}$. For $\eta = 1$, we have the orthogonal conditions,

$$\langle \xi - h(1), 1 \rangle = 0$$

which boils down to

$$\mathbb{E}_{\{x\in[0,1/3]\}}(\xi - h(1)) = 0$$

$$\int_0^{\frac{1}{3}} (2x^2 - h(1))dx = 0$$

and then by solving this for $h(1)$ gives $h(1) = 2/24$. This is the way Brzezniak works this problem. Alternatively, we can use $h(\eta) = a + b\eta + c\eta^2$ and brute force calculus. Note the `Piecewise` object in `sympy` is not complete at this point in its development, so we'll have to be exceptionally verbose in the following,

```
x,c,b,a=S.symbols('x,c,b,a')
xi = 2*x**2

eta=S.Piecewise((1,S.And(S.Gt(x,0),
                         S.Lt(x,S.Rational(1,3)))),  #  0 < x < 1/3
                (2,S.And(S.Gt(x,S.Rational(1,3)),
                         S.Lt(x,S.Rational(2,3)))), # 1/3 < x < 2/3,
                (0,S.And(S.Gt(x,S.Rational(2,3)),
                         S.Lt(x,1)))) # 1/3 < x < 2/3
h = a + b*eta + c*eta**2
J=S.integrate((xi-h)**2,(x,0,1))
sol=S.solve([S.diff(J,a),
             S.diff(J,b),
             S.diff(J,c),
            ],
            (a,b,c))

>>> print sol
{c: 8/9, b: -20/9, a: 38/27}
>>> print S.piecewise_fold(h.subs(sol))
Piecewise((2/27,And(x<1/3,x>0)),
          (14/27,And(x<2/3,x>1/3)),(38/27,And(x<1,x>2/3)))
```

Thus, collecting this result gives:

$$\mathbb{E}(\xi|\eta) = \frac{38}{27} - \frac{20}{9}\eta + \frac{8}{9}\eta^2$$

which can be re-written as a piecewise function of x,

$$\mathbb{E}(\xi|\eta(x)) = \begin{cases} \frac{2}{27} & \text{for } 0 < x < \frac{1}{3} \\ \frac{14}{27} & \text{for } \frac{1}{3} < x < \frac{2}{3} \\ \frac{38}{27} & \text{for } \frac{2}{3} < x < 1 \end{cases} \tag{2.5.3.1}$$

Alternatively, we can use the orthogonal inner product conditions directly by choosing $h(\eta) = c + \eta b + \eta^2 a$,

$$\langle \xi - h(\eta), 1 \rangle = 0$$
$$\langle \xi - h(\eta), \eta \rangle = 0$$
$$\langle \xi - h(\eta), \eta^2 \rangle = 0$$

and then solving for a,b, and c.

```
>>> x,a,b,c,eta = S.symbols('x,a,b,c,eta',real=True)
>>> xi  = 2*x**2
>>> eta=S.Piecewise((1,S.And(S.Gt(x,0),
...                          S.Lt(x,S.Rational(1,3)))),  #  0 < x < 1/3
...                (2,S.And(S.Gt(x,S.Rational(1,3)),
...                          S.Lt(x,S.Rational(2,3)))), # 1/3 < x < 2/3,
...                (0,S.And(S.Gt(x,S.Rational(2,3)),
...                          S.Lt(x,1)))) # 1/3 < x < 2/3
>>> h = c+b*eta+a*eta**2
```

Then, the orthogonal conditions become,

```
>>> S.integrate((xi-h)*1,(x,0,1))
-5*a/3 - b - c + 2/3
>>> S.integrate((xi-h)*eta,(x,0,1))
-3*a - 5*b/3 - c + 10/27
>>> S.integrate((xi-h)*eta**2,(x,0,1))
-17*a/3 - 3*b - 5*c/3 + 58/81
```

Now, we just combine the three equations and solve for the parameters,

```
>>> eqs=[ -5*a/3 - b - c + 2/3,
...    -3*a - 5*b/3 - c + 10/27,
...    -17*a/3 - 3*b - 5*c/3 + 58/81]
>>> sol=S.solve(eqs)
>>> print sol
{a: 0.888888888888889, c: 1.40740740740741, b: -2.22222222222222}
```

We can assemble the final result by substituting in the solution,

```
>>> print S.piecewise_fold(h.subs(sol))
Piecewise((0.0740740740740740, And (x < 1/3, x > 0)),
           (0.518518518518518, And (x < 2/3, x > 1/3)),
            (1.40740740740741, And (x < 1, x > 2/3)))
```

which is the same as our analytic result in Eq. 2.5.3.1, just in decimal format.

Programming Tip

The definition of Sympy's piecewise function is verbose because of the way Python parses inequality statements. As of this writing, this has not been reconciled in Sympy, so we have to use the verbose declaration.

To reinforce our result, let's do a quick simulation using Pandas.

```
>>> d = pd.DataFrame(columns=['x','eta','xi'])
>>> d.x = np.random.rand(1000)
>>> d.xi = 2*d.x**2
```

Now, we can use the `pd.cut` function to group the `x` values in the following,

```
>>> pd.cut(d.x,[0,1/3,2/3,1]).head()
0         (0.667, 1]
1         (0, 0.333]
2         (0.667, 1]
3    (0.333, 0.667]
4    (0.333, 0.667]
Name: x, dtype: category
Categories (3, object): [(0, 0.333] < (0.333, 0.667] < (0.667, 1]]
```

Note that the `head()` call above is only to limit the printout shown. The categories listed are each of the intervals for `eta` that we specified using the `[0,1/3,2/3,1]` list. Now that we know how to use `pd.cut`, we can just compute the mean on each group as shown below,

```
>>> d.groupby(pd.cut(d.x,[0,1/3,2/3,1])).mean()['xi']
x
(0, 0.333]        0.069240
(0.333, 0.667]    0.520154
(0.667, 1]        1.409747
Name: xi, dtype: float64
```

which is pretty close to our analytic result in Eq. 2.5.3.1. Alternatively, `sympy.stats` has some limited tools for the same calculation.

```
>>> from sympy.stats import E, Uniform
>>> x=Uniform('x',0,1)
>>> E(2*x**2,S.And(x < S.Rational(1,3), x > 0))
2/27
>>> E(2*x**2,S.And(x < S.Rational(2,3), x > S.Rational(1,3)))
14/27
>>> E(2*x**2,S.And(x < 1, x > S.Rational(2,3)))
38/27
```

which again gives the same result still another way.

2.5.4 Example

This is Example 2.4 from Brzezniak. Find $\mathbb{E}(\xi|\eta)$ for

$$\xi(x) = 2x^2$$

$$\eta = \begin{cases} 2 & \text{if } 0 \le x < \frac{1}{2} \\ x & \text{if } \frac{1}{2} < x \le 1 \end{cases}$$

Once again, X is uniformly distributed on the unit interval. Note that η is no longer discrete for every domain. For the domain $0 < x < 1/2$, $h(2)$ takes on only one value, say, h_0. For this domain, the orthogonal condition becomes,

$$\mathbb{E}_{\{\eta=2\}}((\xi(x) - h_0)2) = 0$$

which simplifies to,

$$\begin{aligned}\int_0^{1/2} 2x^2 - h_0 dx &= 0\\ \int_0^{1/2} 2x^2 dx &= \int_0^{1/2} h_0 dx\\ h_0 &= 2\int_0^{1/2} 2x^2 dx\\ h_0 &= \frac{1}{6}\end{aligned}$$

For the other domain where $\{\eta = x\}$ in Eq. 2.5.4, we again use the orthogonal condition,

$$\begin{aligned}\mathbb{E}_{\{\eta=x\}}((\xi(x) - h(x))x) &= 0\\ \int_{1/2}^{1} (2x^2 - h(x))x dx &= 0\\ h(x) &= 2x^2\end{aligned}$$

Assembling the solution gives,

$$\mathbb{E}(\xi|\eta(x)) = \begin{cases}\frac{1}{6} & \text{for } 0 \le x < \frac{1}{2}\\ 2x^2 & \text{for } \frac{1}{2} < x \le 1\end{cases}$$

although this result is not explicitly written as a function of η.

2.5.5 *Example*

This is Exercise 2.6 in Brzezniak. Find $\mathbb{E}(\xi|\eta)$ where

$$\xi(x) = 2x^2$$

$$\eta(x) = 1 - |2x - 1|$$

and X is uniformly distributed in the unit interval. We can write this out as a piecewise function in the following,

$$\eta = \begin{cases} 2x & \text{for } 0 \leq x < \frac{1}{2} \\ 2-2x & \text{for } \frac{1}{2} < x \leq 1 \end{cases}$$

The discontinuity is at $x = 1/2$. Let's start with the $\{\eta = 2x\}$ domain.

$$\mathbb{E}_{\{\eta=2x\}}((2x^2 - h(2x))2x) = 0$$
$$\int_0^{1/2} (2x^2 - h(2x))2x dx = 0$$

We can make this explicitly a function of η by a change of variables ($\eta = 2x$) which gives

$$\int_0^1 (\eta^2/2 - h(\eta))\frac{\eta}{2} d\eta = 0$$

Thus, for this domain, $h(\eta) = \eta^2/2$. Note that due to the change of variables, $h(\eta)$ is valid defined over $\eta \in [0, 1]$.

For the other domain where $\{\eta = 2 - 2x\}$, we have

$$\mathbb{E}_{\{\eta=2-2x\}}((2x^2 - h(2-2x))(2-2x)) = 0$$
$$\int_{1/2}^1 (2x^2 - h(2-2x))(2-2x) dx = 0$$

Once again, a change of variables makes the η dependency explicit using $\eta = 2-2x$ which gives

$$\int_0^1 ((2-\eta)^2/2 - h(\eta))\frac{\eta}{2} d\eta = 0$$
$$h(\eta) = (2-\eta)^2/2$$

Once again, the change of variables means this solution is valid over $\eta \in [0, 1]$. Thus, because both pieces are valid over the same domain ($\eta \in [0, 1]$), we can just add them to get the final solution,

$$h(\eta) = \eta^2 - 2\eta + 2$$

A quick simulation can help bear this out.

```
>>> from pandas import DataFrame
>>> import numpy as np
>>> d = DataFrame(columns=['xi','eta','x','h','h1','h2'])
>>> # 100 random samples
```

```
>>> d.x = np.random.rand(100)
>>> d.xi = d.eval('2*x**2')
>>> d.eta =1-abs(2*d.x-1)
>>> d.h1=d[(d.x<0.5)].eval('eta**2/2')
>>> d.h2=d[(d.x>=0.5)].eval('(2-eta)**2/2')
>>> d.fillna(0,inplace=True)
>>> d.h = d.h1+d.h2
>>> d.head()
         xi       eta         x         h        h1        h2
0  1.728372  0.140768  0.929616  1.728372  0.000000  1.728372
1  0.200187  0.632751  0.316376  0.200187  0.200187  0.000000
2  0.067652  0.367838  0.183919  0.067652  0.067652  0.000000
3  0.083690  0.409121  0.204560  0.083690  0.083690  0.000000
4  0.644623  0.864550  0.567725  0.644623  0.000000  0.644623
```

Note that we have to be careful where we apply the individual solutions using the slice `(d.x<0.5)` index. The `fillna` part ensures that the default `NaN` that fills out the empty row-etries is replaced with zero before combining the individual solutions. Otherwise, the `NaN` values would circulate through the rest of the computation. The following is the essential code that draws Fig. 2.7.

```
from matplotlib.pyplot import subplots
fig,ax=subplots()
ax.plot(d.xi,d.eta,'.',alpha=.3,label='$\eta$')
ax.plot(d.xi,d.h,'k.',label='$h(\eta)$')
ax.legend(loc=0,fontsize=18)
ax.set_xlabel('$2 x^2$',fontsize=18)
ax.set_ylabel('$h(\eta)$',fontsize=18)
```

Programming Tip

Basic LaTeX formatting works for the labels in Fig. 2.7. The `loc=0` in the `legend` function is the code for the *best* placement for the labels in the legend. The individual labels should be specified when the elements are drawn individually, otherwise they will be hard to separate out later. This is accomplished using the `label` keyword in the `plot` commands.

Figure 2.7 shows the ξ data plotted against η and $h(\eta) = \mathbb{E}(\xi|\eta)$. Points on the diagonal are points where ξ and $\mathbb{E}(\xi|\eta)$ match. As shown by the dots, there is no agreement between the raw η data and ξ. Thus, one way to think about the conditional expectation is as a functional transform that bends the curve onto the diagonal line. The black dots plot ξ versus $\mathbb{E}(\xi|\eta)$ and the two match everywhere along the diagonal line. This is to be expected because the conditional expectation is the MSE best estimate for ξ among all functions of η.

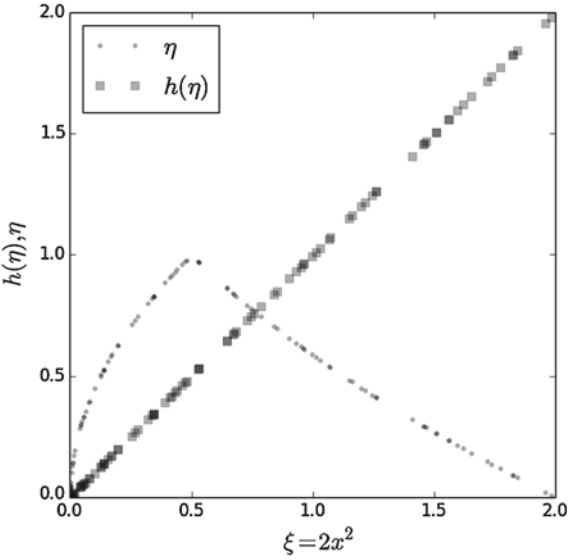

Fig. 2.7 The *diagonal line* shows where the conditional expectation equals the ξ function

2.5.6 Example

This is Exercise 2.14 from Brzezniak. Find $\mathbb{E}(\xi|\eta)$ where

$$\xi(x) = 2x^2$$

$$\eta = \begin{cases} 2x & \text{if } 0 \le x < \frac{1}{2} \\ 2x - 1 & \text{if } \frac{1}{2} < x \le 1 \end{cases}$$

and X is uniformly distributed in the unit interval. This is the same as the last example and the only difference here is that η is not continuous at $x = \frac{1}{2}$, as before. The first part is exactly the same as the first part of the prior example so we will skip it here. The second part follows the same reasoning as the last example, so we will just write the answer for the $\{\eta = 2x - 1\}$ case as the following

$$h(\eta) = \frac{(1+\eta)^2}{2}, \ \forall \eta \in [0, 1]$$

and then adding these up as before gives the full solution:

$$h(\eta) = \frac{1}{2} + \eta + \eta^2$$

The interesting part about this example is shown in Fig. 2.8. The dots show where η is discontinuous and yet the $h(\eta) = \mathbb{E}(\xi|\eta)$ solution is equal to ξ (i.e., matches

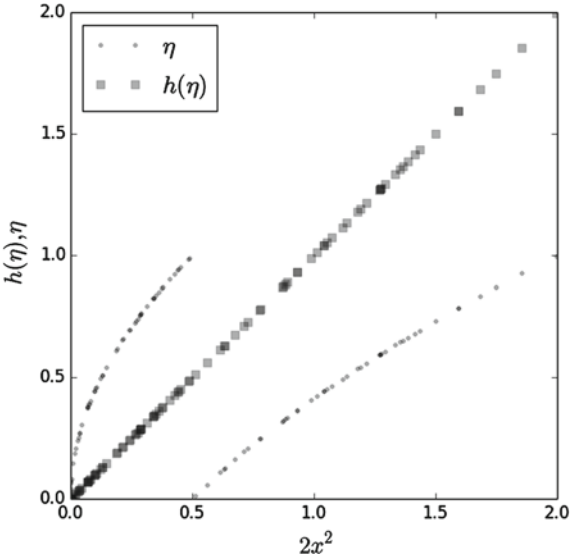

Fig. 2.8 The *diagonal line* shows where the conditional expectation equals the ξ function

the diagonal). This illustrates the power of the orthogonal inner product technique, which does not need continuity or complex set-theoretic arguments to calculate solutions. By contrast, I urge you to consider Brzezniak's solution to this problem which requires such methods.

Extending projection methods to random variables provides multiple ways for calculating solutions to conditional expectation problems. In this section, we also worked out corresponding simulations using a variety of Python modules. It is always advisable to have more than one technique at hand to cross-check potential solutions. We worked out some of the examples in Brzezniak's book using our methods as a way to show multiple ways to solve the same problem. Comparing Brzezniak's measure-theoretic methods to our less abstract techniques is a great way to get a handle on both concepts, which are important for advanced study in stochastic process.

2.6 Information Entropy

We are in a position to discuss information entropy. This will give us a powerful perspective on how information passes between experiments, and will prove important in certain machine learning algorithms.

There used to be a TV game show where the host would hide a prize behind one of three doors and the contestant would have to pick one of the doors. However, before opening the door of the contestant's choice, the host would open one of the other doors and ask the contestant if she wanted to change her selection. This is the classic *Monty Hall* problem. The question is should the contestant stay with her original

choice or switch after seeing what the host has revealed? From the information theory perspective, does the information environment change when the host reveals what is behind one of the doors? The important detail here is that the host *never* opens the door with the prize behind it, regardless of the contestant's choice. That is, the host *knows* where the prize is, but he does not reveal that information directly to the contestant. This is the fundamental problem information theory addresses—how to aggregate and reason about partial information. We need a concept of information that can accommodate this kind of question.

2.6.1 Information Theory Concepts

The Shannon *information content* of an outcome x is defined as,

$$h(x) = \log_2 \frac{1}{P(x)}$$

where $P(x)$ is the probability of x. The *entropy* of the ensemble X is defined to be the Shannon information content of

$$H(X) = \sum_x P(x) \log_2 \frac{1}{P(x)}$$

It is no accident that the entropy has this functional form as the expectation of $h(x)$. It leads to a deep and powerful theory of information.

To get some intuition about what information entropy means, consider a sequence of three-bit numbers where each individual bit is equally likely. Thus, the individual information content of a single bit is $h(x) = \log_2(2) = 1$. The units of entropy are *bits* so this says that information content of a single bit is one bit. Because the three-bit number has elements that are mutually independent and equally likely, the information entropy of the three-bit number is $h(X) = 2^3 \times \log_2(2^3)/8 = 3$. Thus, the basic idea of information content at least makes sense at this level.

A better way to interpret this question is as how much information would I have to provide in order to uniquely encode an arbitrary three-bit number? In this case, you would have to answer three questions: *Is the first bit zero or one? Is the second bit zero or one? Is the third bit zero or one?* Answering these questions uniquely specifies the unknown three-bit number. Because the bits are mutually independent, knowing the state of any of the bits does not inform the remainder.

Next, let's consider a situation that lacks this mutual independence. Suppose in a group of nine otherwise identical balls there is a heavier one. Furthermore, we also have a measuring scale that indicates whether one side is heavier, lighter, or equal to the other. How could we identify the heavier ball? At the outset, the information content, which measures the uncertainty of the situation is $\log_2(9)$ because one of the nine balls is heavier. Figure 2.9 shows one strategy. We could arbitrarily select out

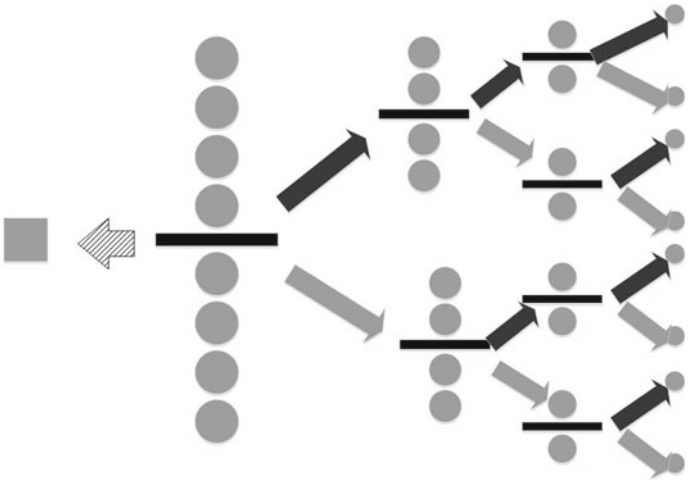

Fig. 2.9 One heavy ball is hidden among eight identical balls. By weighing groups sequentially, we can determine the heavy ball

one of the balls (shown by the square), leaving the remaining eight to be balanced. The thick, black horizontal line indicates the scale. The items below and above this line indicate the counterbalanced sides of the scale.

If we get lucky, the scale will report that the group of four walls on either side of the balance are equal in weight. This means that the ball that was omitted is the heavier one. This is indicated by the hashed left-pointing arrow. In this case, all the uncertainty has evaporated, and the *informational value* of that one weighing is equal to $\log_2(9)$. In other words, the scale has reduced the uncertainty to zero (i.e., found the heavy ball). On the other hand, the scale could report that the upper group of four balls is heavier (black, upward-pointing arrow) or lighter (gray, downward-pointing arrow). In this case, we cannot isolate the heavier ball until we perform all of the indicated weighings, moving from left-to-right. Specifically, the four balls on the heavier side have to be split by a subsequent weighing into two balls and then to one ball before the heavy ball can be identified. Thus, this process takes three weighings. The first one has information content $\log_2(9/8)$, the next has $\log_2(4)$, and the final one has $\log_2(2)$. Adding all these up sums to $\log_2(9)$. Thus, whether or not the heavier ball is isolated in the first weighing, the strategy consumes $\log_2(9)$ bits, as it must, to find the heavy ball.

However, this is not the only strategy. Figure 2.10 shows another. In this approach, the nine balls are split up into three groups of three balls apiece. Two groups are weighed. If they are of equal weight, then this means the heavier ball is in the group that was left out (dashed arrow). Then, this group is split into two groups, with one element left out. If the two balls on the scale weigh the same, then it means the excluded one is the heavy one. Otherwise, it is one of the balls on the scale. The same process follows if one of the initially weighed groups is heavier (black upward-facing

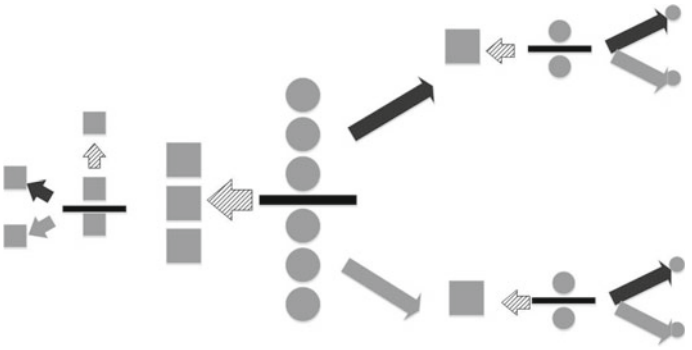

Fig. 2.10 For this strategy, the balls are broken up into three groups of equal size and subsequently weighed

arrow) or lighter (gray lower-facing arrow). As before the information content of the situation is $\log_2(9)$. The first weighing reduces the uncertainty of the situation by $\log_2(3)$ and the subsequent weighing reduces it by another $\log_2(3)$. As before, these sum to $\log_2(9)$, but here we only need two weighings whereas the first strategy in Fig. 2.9 takes an average of $1/9+3*8/9 \approx 2.78$ weighings, which is more than two.

Why does the second strategy use fewer weighings? To reduce weighings, we need each weighing to adjudicate equally probable situations as many times as possible. Choosing one of the nine balls at the outset (i.e., first strategy in Fig. 2.9) does not do this because the probability of selecting the correct ball is 1/9. This does not create a equiprobable situation in the process. The second strategy leaves an equally probable situation at every stage (see Fig. 2.10), so it extracts the most information out of each weighing as possible. Thus, the information content tells us how many bits of information have to be resolved using *any* strategy (i.e., $\log_2(9)$ in this example). It also illuminates how to efficiently remove uncertainty; namely, by adjudicating equiprobable situations as many times as possible.

2.6.2 Properties of Information Entropy

Now that we have the flavor of the concepts, consider the following properties of the information entropy,

$$H(X) \geq 0$$

with equality if and only if $P(x) = 1$ for exactly one x. Intuitively, this means that when just one of the items in the ensemble is known absolutely (i.e., with $P(x) = 1$), the uncertainty collapses to zero. Also note that entropy is maximized when P is uniformly distributed across the elements of the ensemble. This is illustrated in Fig. 2.11 for the case of two outcomes. In other words, information entropy is

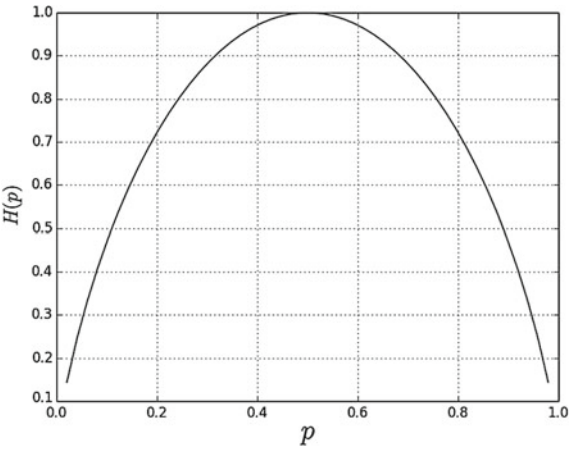

Fig. 2.11 The information entropy is maximized when $p = 1/2$

maximized when the two conflicting alternatives are equally probable. This is the mathematical reason why using the scale in the last example to adjudicate equally probable situations was so useful for abbreviating the weighing process.

Most importantly, the concept of entropy extends jointly as follows,

$$H(X, Y) = \sum_{x,y} P(x, y) \log_2 \frac{1}{P(x, y)}$$

If and only if X and Y are independent, entropy becomes additive,

$$H(X, Y) = H(X) + H(Y)$$

2.6.3 Kullback-Leibler Divergence

Notions of information entropy lead to notions of distance between probability distributions that will become important for machine learning methods. The Kullback-Leibler divergence between two probability distributions P and Q that are defined over the same set is defined as,

$$D_{KL}(P, Q) = \sum_{x} P(x) \log_2 \frac{P(x)}{Q(x)}$$

Note that $D_{KL}(P, Q) \geq 0$ with equality if and only if $P = Q$. Sometimes the Kullback-Leibler divergence is called the Kullback-Leibler distance, but it is not formally a distance metric because it is asymmetrical in P and Q. The Kullback-Leibler divergence defines a relative entropy as the loss of information if P is modeled

in terms of Q. There is an intuitive way to interpret the Kullback-Leibler divergence and understand its lack of symmetry. Suppose we have a set of messages to transmit, each with a corresponding probability $\{(x_1, P(x_1)), (x_2, P(x_2)), \ldots, (x_n, P(x_n))\}$. Based on what we know about information entropy, it makes sense to encode the length of the message by $\log_2 \frac{1}{p(x)}$ bits. This parsimonious strategy means that more frequent messages are encoded with fewer bits. Thus, we can rewrite the entropy of the situation as before,

$$H(X) = \sum_k P(x_k) \log_2 \frac{1}{P(x_k)}$$

Now, suppose we want to transmit the same set of messages, but with a different set of probability weights, $\{(x_1, Q(x_1)), (x_2, Q(x_2)), \ldots, (x_n, Q(x_n))\}$. In this situation, we can define the cross-entropy as

$$H_q(X) = \sum_k P(x_k) \log_2 \frac{1}{Q(x_k)}$$

Note that only the purported length of the encoded message has changed, not the probability of that message. The difference between these two is the Kullback-Leibler divergence,

$$D_{KL}(P, Q) = H_q(X) - H(X) = \sum_x P(x) \log_2 \frac{P(x)}{Q(x)}$$

In this light, the Kullback-Leibler divergence is the average difference in the encoded lengths of the same set of messages under two different probability regimes. This should help explain the lack of symmetry of the Kullback-Leibler divergence — left to themselves, P and Q would provide the optimal-length encodings separately, but there can be no necessary symmetry in how each regime would rate the informational value of each message ($Q(x_i)$ versus $P(x_i)$). Given that each encoding is optimal-length in its own regime means that it must therefore be at least sub-optimal in another, thus giving rise to the Kullback-Leibler divergence. In the case where the encoding length of all messages remains the same for the two regimes, then the Kullback-Leibler divergence is zero.[2]

2.7 Moment Generating Functions

Generating moments usually involves integrals that are extremely difficult to compute. Moment generating functions make this much, much easier. The moment generating function is defined as,

[2]The best, easy-to-understand presentation of this material is chapter four of Mackay's text [7]. Another good reference is chapter four of [8].

$$M(t) = \mathbb{E}(\exp(tX))$$

The first moment is the mean, which we can easily compute from $M(t)$ as,

$$\begin{aligned}\frac{dM(t)}{dt} &= \frac{d}{dt}\mathbb{E}(\exp(tX)) = \mathbb{E}\frac{d}{dt}(\exp(tX)) \\ &= \mathbb{E}(X\exp(tX))\end{aligned}$$

Now, we have to set $t = 0$ and we have the mean,

$$M^{(1)}(0) = \mathbb{E}(X)$$

continuing this derivative process again, we obtain the second moment as,

$$\begin{aligned}M^{(2)}(t) &= \mathbb{E}(X^2\exp(tX)) \\ M^{(2)}(0) &= \mathbb{E}(X^2)\end{aligned}$$

With this in hand, we can easily compute the variance as,

$$\mathbb{V}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2 = M^{(2)}(0) - M^{(1)}(0)^2$$

Example Returning to our favorite binomial distribution, let's compute some moments using Sympy.

```
>>> import sympy as S
>>> from sympy import stats
>>> p,t = S.symbols('p t',positive=True)
>>> x=stats.Binomial('x',10,p)
>>> mgf = stats.E(S.exp(t*x))
```

Now, let's compute the first moment (aka, mean) using the usual integration method and using moment generating functions,

```
>>> print S.simplify(stats.E(x))
10*p
>>> print S.simplify(S.diff(mgf,t).subs(t,0))
10*p
```

Otherwise, we can compute this directly as follows,

```
>>> print S.simplify(stats.moment(x,1)) # mean
10*p
>>> print S.simplify(stats.moment(x,2)) # 2nd moment
10*p*(9*p + 1)
```

In general, the moment generating function for the binomial distribution is the following,

$$M_X(t) = \left(p\left(e^t - 1\right) + 1\right)^n$$

A key aspect of moment generating functions is that they are unique identifiers of probability distributions. By the uniqueness theorem, given two random variables X and Y, if their respective moment generating functions are equal, then the corresponding probability distribution functions are equal.

Example Let's use the uniqueness theorem to consider the following problem. Suppose we know that the probability distribution of X given $U = p$ is binomial with parameters n and p. For example, suppose X represents the number of heads in n coin flips, given the probability of heads is p. We want to find the unconditional distribution of X. Writing out the moment generating function as the following,

$$\mathbb{E}(e^{tX}|U = p) = (pe^t + 1 - p)^n$$

Because U is uniform over the unit interval, we can integrate this part out

$$\begin{aligned}\mathbb{E}(e^{tX}) &= \int_0^1 (pe^t + 1 - p)^n dp \\ &= \frac{1}{n+1}\frac{e^{t(n+1)-1}}{e^t - 1} \\ &= \frac{1}{n+1}(1 + e^t + e^{2t} + e^{3t} + \cdots + e^{nt})\end{aligned}$$

Thus, the moment generating function of X corresponds to that of a random variable that is equally likely to be any of the values $0, 1, \ldots, n$. This is another way of saying that the distribution of X is discrete uniform over $\{0, 1, \ldots, n\}$. Concretely, suppose we have a box of coins whose individual probability of heads is unknown and that we dump the box on the floor, spilling all of the coins. If we then count the number of coins facing heads-up, that distribution is uniform.

Moment generating functions are useful for deriving distributions of sums of independent random variables. Suppose X_1 and X_2 are independent and $Y = X_1 + X_2$. Then, the moment generating function of Y follows from the properties of the expectation,

$$\begin{aligned}M_Y(t) &= \mathbb{E}(e^{tY}) = \mathbb{E}(e^{tX_1 + tX_2}) \\ &= \mathbb{E}(e^{tX_1}e^{tX_2}) = \mathbb{E}(e^{tX_1})\mathbb{E}(e^{tX_2}) \\ &= M_{X_1}(t)M_{X_2}(t)\end{aligned}$$

Example Suppose we have two normally distributed random variables, $X_1 \sim \mathcal{N}(\mu_1, \sigma_1)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2)$. We can save some tedium by exploring this in Sympy,

```
>>> S.var('x:2',real=True)
(x0, x1)
>>> S.var('mu:2',real=True)
(mu0, mu1)
>>> S.var('sigma:2',positive=True)
(sigma0, sigma1)
>>> S.var('t',positive=True)
t
>>> x0=stats.Normal(x0,mu0,sigma0)
>>> x1=stats.Normal(x1,mu1,sigma1)
```

Programming Tip

The `S.var` function defines the variable and injects it into the global namespace. This is sheer laziness. It is more expressive to define variables explicitly as in `x = S.symbols('x')`. Also notice that we used the Greek names for the `mu` and `sigma` variables. This will come in handy later when we want to render the equations in the Jupyter/IPython notebook which understands how to typeset these symbols in LaTeX. The `var('x:2')` creates two symbols, `x0` and `x1`. Using the colon this way makes it easy to generate array-like sequences of symbols.

In the next block we compute the moment generating functions

```
>>> mgf0=S.simplify(stats.E(S.exp(t*x0)))
>>> mgf1=S.simplify(stats.E(S.exp(t*x1)))
>>> mgfY=S.simplify(mgf0*mgf1)
```

The moment generating functions an individual normally distributed random variable is the following,

$$e^{\mu_0 t + \frac{\sigma_0^2 t^2}{2}}$$

Note the coefficients of t. To show that Y is normally distributed, we want to match the moment generating function of Y to this format. The following is the form of the moment generating function of Y,

$$M_Y(t) = e^{\frac{t}{2}(2\mu_0 + 2\mu_1 + \sigma_0^2 t + \sigma_1^2 t)}$$

We can extract the exponent using Sympy and collect on the t variable using the following code,

```
>>> S.collect(S.expand(S.log(mgfY)),t)
t**2*(sigma0**2/2 + sigma1**2/2) + t*(mu0 + mu1)
```

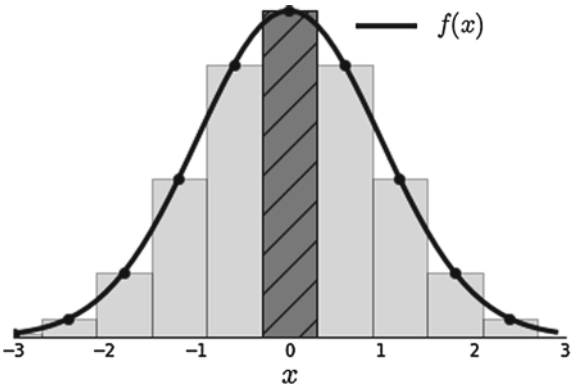

Fig. 2.12 The histogram approximates the target probability density

Thus, by the uniqueness theorem, Y is normally distributed with $\mu_Y = \mu_0 + \mu_1$ and $\sigma_Y^2 = \sigma_0^2 + \sigma_1^2$.

Programming Tip

When using the Jupyter/IPython notebook, you can do `S.init_printing` to get the mathematical typesetting to work in the browser. Otherwise, if you want to keep the raw expression and to selectively render to LaTeX, then you can `from IPython.display import Math`, and then use `Math(S.latex(expr))` to see the typeset version of the expression.

2.8 Monte Carlo Sampling Methods

So far, we have studied analytical ways to transform random variables and how to augment these methods using Python. In spite of all this, we frequently must resort to purely numerical methods to solve real-world problems. Hopefully, now that we have seen the deeper theory, these numerical methods feel more concrete. Suppose we want to generate samples of a given density, $f(x)$, given we already can generate samples from a uniform distribution, $\mathcal{U}[0, 1]$. How do we know a random sample v comes from the $f(x)$ distribution? One approach is to look at how a histogram of samples of v approximates $f(x)$. Specifically,

$$\mathbb{P}(v \in N_\Delta(x)) = f(x)\Delta x \tag{2.8.0.1}$$

which says that the probability that a sample is in some N_Δ neighborhood of x is approximately $f(x)\Delta x$. Figure 2.12 shows the target probability density function $f(x)$ and a histogram that approximates it. The histogram is generated from samples v. The hatched rectangle in the center illustrates Eq. 2.8.0.1. The area of this rectangle is approximately $f(x)\Delta x$ where $x = 0$, in this case. The width of the rectangle is $N_\Delta(x)$ The quality of the approximation may be clear visually, but to know that v samples are characterized by $f(x)$, we need the statement of Eq. 2.8.0.1, which says that the proportion of samples v that fill the hatched rectangle is approximately equal to $f(x)\Delta x$.

Now that we know how to evaluate samples v that are characterized by the density $f(x)$, let's consider how to create these samples for both discrete and continuous random variables.

2.8.1 *Inverse CDF Method for Discrete Variables*

Suppose we want to generate samples from a fair six-sided die. Our workhouse uniform random variable is defined continuously over the unit interval and the fair six-sided die is discrete. We must first create a mapping between the continuous random variable u and the discrete outcomes of the die. This mapping is shown in Fig. 2.13 where the unit interval is broken up into segments, each of length $1/6$. Each individual segment is assigned to one of the die outcomes. For example, if $u \in [1/6, 2/6)$, then the outcome for the die is 2. Because the die is fair, all segments on the unit interval are the same length. Thus, our new random variable v is derived from u by this assignment.

For example, for $v = 2$, we have,

$$\mathbb{P}(v = 2) = \mathbb{P}(u \in [1/6, 2/6)) = 1/6$$

where, in the language of the Eq. 2.8.0.1, $f(x) = 1$ (uniform distribution), $\Delta x = 1/6$, and $N_\Delta(2) = [1/6, 2/6)$. Naturally, this pattern holds for all the other die outcomes in $\{1, 2, 3, \ldots, 6\}$. Let's consider a quick simulation to make this concrete. The following code generates uniform random samples and stacks them in a Pandas dataframe.

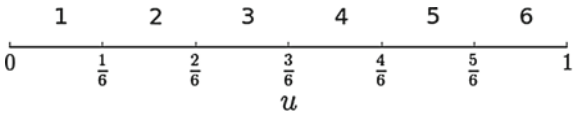

Fig. 2.13 A uniform distribution random variable on the unit interval is assigned to the six outcomes of a fair die using these segements

```
import pandas as pd
import numpy as np
from pandas import DataFrame
u= np.random.rand(100)
df = DataFrame(data=u,columns=['u'])
```

The next block uses `pd.cut` to map the individual samples to the set $\{1, 2, \ldots, 6\}$ labeled `v`.

```
labels = [1,2,3,4,5,6]
df['v']=pd.cut(df.u,np.linspace(0,1,7),
               include_lowest=True,labels=labels)
```

This is what the dataframe contains. The `v` column contains the samples drawn from the fair die.

```
>>> df.head()
          u  v
0  0.876426  6
1  0.360385  3
2  0.427294  3
3  0.833833  6
4  0.112139  1
```

The following is a count of the number of samples in each group. There should be roughly the same number of samples in each group because the die is fair.

```
>>> df.groupby('v').count()
    u
v
1  18
2  12
3  20
4  13
5  15
6  22
```

So far, so good. We now have a way to simulate a fair die from a uniformly distributed random variable.

To extend this to unfair die, we need only make some small adjustments to this code. For example, suppose that we want an unfair die so that $\mathbb{P}(1) = \mathbb{P}(2) = \mathbb{P}(3) = 1/12$ and $\mathbb{P}(4) = \mathbb{P}(5) = \mathbb{P}(6) = 1/4$. The only change we have to make is with `pd.cut` as follows,

```
df['v']=pd.cut(df.u,[0,1/12,2/12,3/12,2/4,3/4,1],
               include_lowest=True,labels=labels)

>>> df.groupby('v').count()/df.shape[0]
      u
v
1  0.08
2  0.10
3  0.09
4  0.23
5  0.19
6  0.31
```

where now these are the individual probabilities of each digit. You can take more than `100` samples to get a clearer view of the individual probabilities but the mechanism for generating them is the same. The method is called the inverse CDF[3] method because the CDF (namely, `[0,1/12,2/12,3/12,2/4,3/4,1]`) in the last example has been inverted (using the `pd.cut` method) to generate the samples. The inversion is easier to see for continuous variables, which we consider next.

2.8.2 Inverse CDF Method for Continuous Variables

The method above applies to continuous random variables, but now we have to use squeeze the intervals down to individual points. In the example above, our inverse function was a piecewise function that operated on uniform random samples. In this case, the piecewise function collapses to a continuous inverse function. We want to generate random samples for a CDF that is invertible. As before, the criterion for generating an appropriate sample v is the following,

$$\mathbb{P}(F(x) < v < F(x+\Delta x)) = F(x+\Delta x) - F(x) = \int_x^{x+\Delta x} f(u)du \approx f(x)\Delta x$$

which is saying that the probability that the sample v is contained in a Δx interval is approximately equal to the density function, $f(x)\Delta x$, at that point. Once again, the trick is to use a uniform random sample u and an invertible CDF $F(x)$ to construct these samples. Note that for a uniform random variable $u \sim \mathcal{U}[0,1]$, we have,

$$\begin{aligned}\mathbb{P}(x < F^{-1}(u) < x+\Delta x) &= \mathbb{P}(F(x) < u < F(x+\Delta x)) \\ &= F(x+\Delta x) - F(x) \\ &= \int_x^{x+\Delta x} f(p)dp \approx f(x)\Delta x\end{aligned}$$

This means that $v = F^{-1}(u)$ is distributed according to $f(x)$, which is what we want.

Let's try this to generate samples from the exponential distribution,

$$f_\alpha(x) = \alpha e^{-\alpha x}$$

which has the following CDF,

$$F(x) = 1 - e^{-\alpha x}$$

[3] Cumulative density function. Namely, $F(x) = \mathbb{P}(X < x)$.

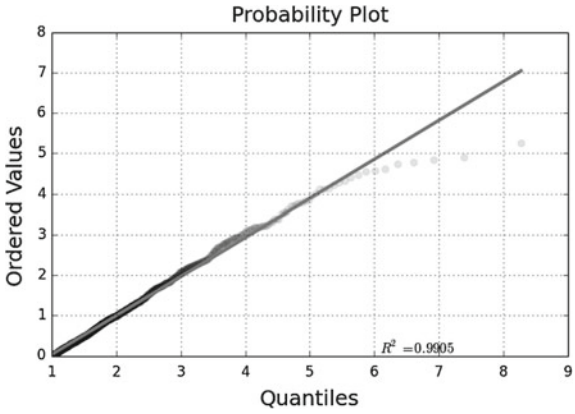

Fig. 2.14 The samples created using the inverse cdf method match the exponential reference distribution

and corresponding inverse,

$$F^{-1}(u) = \frac{1}{\alpha} \ln \frac{1}{(1-u)}$$

Now, all we have to do is generate some uniformly distributed random samples and then feed them into F^{-1}.

```
>>> from numpy import array, log
>>> import scipy.stats
>>> alpha = 1.   # distribution parameter
>>> nsamp = 1000 # num of samples
>>> # define uniform random variable
>>> u=scipy.stats.uniform(0,1)
>>> # define inverse function
>>> Finv=lambda u: 1/alpha*log(1/(1-u))
>>> # apply inverse function to samples
>>> v = array(map(Finv,u.rvs(nsamp)))
```

Now, we have the samples from the exponential distribution, but how do we know the method is correct with samples distributed accordingly? Fortunately, `scipy.stats` already has a exponential distribution, so we can check our work against the reference using a *probability plot* (i.e., also known as a *quantile-quantile* plot). The following code sets up the probability plot from `scipy.stats`.

```
fig,ax=subplots()
scipy.stats.probplot(v,(1,),dist='expon',plot=ax)
```

Note that we have to supply an axes object (`ax`) for it to draw on. The result is Fig. 2.14. The more the samples line match the diagonal line, the more they match the reference distribution (i.e., exponential distribution in this case). You may also want to try `dist=norm` in the code above To see what happens when the normal distribution is the reference distribution.

2.8.3 Rejection Method

In some cases, inverting the CDF may be impossible. The *rejection* method can handle this situation. The idea is to pick two uniform random variables $u_1, u_2 \sim \mathcal{U}[a,b]$ so that

$$\mathbb{P}\left(u_1 \in N_\Delta(x) \bigwedge u_2 < \frac{f(u_1)}{M}\right) \approx \frac{\Delta x}{b-a}\frac{f(u_1)}{M}$$

where we take $x = u_1$ and $f(x) < M$. This is a two-step process. First, draw u_1 uniformly from the interval $[a,b]$. Second, feed it into $f(x)$ and if $u_2 < f(u_1)/M$, then you have a valid sample for $f(x)$. Thus, u_1 is the proposed sample from f that may or may not be rejected depending on u_2. The only job of the M constant is to scale down the $f(x)$ so that the u_2 variable can span the range. The *efficiency* of this method is the probability of accepting u_1 which comes from integrating out the above approximation,

$$\int \frac{f(x)}{M(b-a)}dx = \frac{1}{M(b-a)}\int f(x)dx = \frac{1}{M(b-a)}$$

This means that we don't want an unecessarily large M because that makes it more likely that samples will be discarded.

Let's try this method for a density that does not have a continuous inverse.[4]

$$f(x) = \exp\left(-\frac{(x-1)^2}{2x}\right)(x+1)/12$$

where $x > 0$. The following code implements the rejection plan.

```
>>> import numpy as np
>>> x = np.linspace(0.001,15,100)
>>> f= lambda x: np.exp(-(x-1)**2/2./x)*(x+1)/12.
>>> fx = f(x)
>>> M=0.3                              # scale factor
>>> u1 = np.random.rand(10000)*15  # uniform random samples scaled out
>>> u2 = np.random.rand(10000)     # uniform random samples
>>> idx,= np.where(u2<=f(u1)/M)    # rejection criterion
>>> v = u1[idx]
```

Figure 2.15 shows a histogram of the so-generated samples that nicely fits the probability density function. The title in the figure shows the efficiency, which is poor. It means that we threw away most of the proposed samples. Thus, even though there is nothing conceptually wrong with this result, the low efficiency must be fixed, as a practical matter. Figure 2.16 shows where the proposed samples were rejected. Samples under the curve were retained (i.e., $u_2 < \frac{f(u_1)}{M}$) but the vast majority of the samples are outside this umbrella.

[4]Note that this example density does not *exactly* integrate out to one like a probability density function should, but the normalization constant for this is distracting for our purposes here.

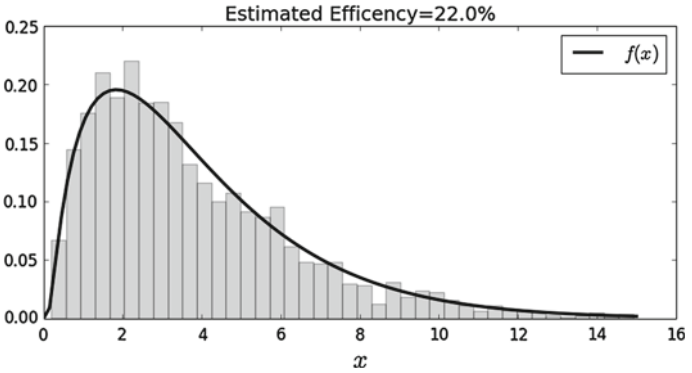

Fig. 2.15 The rejection method generate samples in the histogram that nicely match the target distribution. Unfortunately, the efficiency is not so good

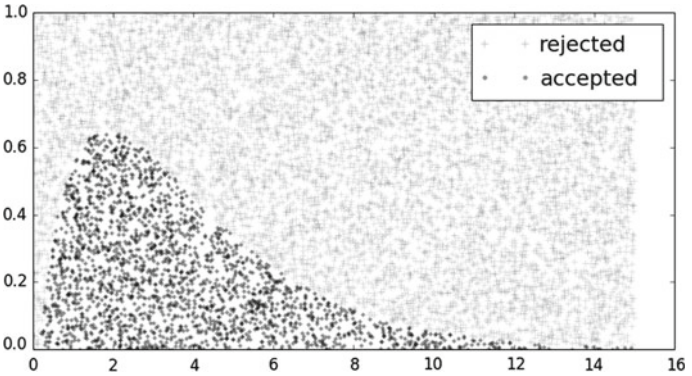

Fig. 2.16 The proposed samples under the *curve* were accepted and the others were not. This shows the majority of samples were rejected

The rejection method uses u_1 to select along the domain of $f(x)$ and the other u_2 uniform random variable decides whether to accept or not. One idea would be to choose u_1 so that x values are coincidentally those that are near the peak of $f(x)$, instead of uniformly anywhere in the domain, especially near the tails, which are low probability anyway. Now, the trick is to find a new density function $g(x)$ to sample from that has a similiar concentration of probability density. One way it to familiarize oneself with the probability density functions that have adjustable parameters and fast random sample generators already. There are lots of places to look and, chances are, there is likely already such a generator for your problem. Otherwise, the family of β densities is a good place to start.

To be explicit, what we want is $u_1 \sim g(x)$ so that, returning to our earlier argument,

$$\mathbb{P}\left(u_1 \in N_\Delta(x) \bigwedge u_2 < \frac{f(u_1)}{M}\right) \approx g(x)\Delta x \frac{f(u_1)}{M}$$

but this is *not* what we need here. The problem is with the second part of the logical $\bigwedge$ conjunction. We need to put something there that will give us something proportional to $f(x)$. Let us define the following,

$$h(x) = \frac{f(x)}{g(x)} \tag{2.8.3.1}$$

with corresponding maximum on the domain as $h_{\max}$ and then go back and construct the second part of the clause as

$$\mathbb{P}\left(u_1 \in N_\Delta(x) \bigwedge u_2 < \frac{h(u_1)}{h_{\max}}\right) \approx g(x)\Delta x \frac{h(u_1)}{h_{\max}} = f(x)/h_{\max}$$

Recall that satisfying this criterion means that $u_1 = x$. As before, we can estimate the probability of acceptance of the u_1 as $1/h_{\max}$.

Now, how to construct the $g(x)$ function in the denominator of Eq. 2.8.3.1? Here's where familarity with some standard probability densities pays off. For this case, we choose the chi-squared distribution. The following plots the $g(x)$ and $f(x)$ (left plot) and the corresponding $h(x) = f(x)/g(x)$ (right plot). Note that $g(x)$ and $f(x)$ have peaks that almost coincide, which is what we are looking for (Fig. 2.17).

```
>>> ch=scipy.stats.chi2(4) # chi-squared
>>> h = lambda x: f(x)/ch.pdf(x) # h-function
```

Now, let's generate some samples from this χ^2 distribution with the rejection method.

```
>>> hmax=h(x).max()
>>> u1 = ch.rvs(5000)          # samples from chi-square distribution
>>> u2 = np.random.rand(5000)# uniform random samples
>>> idx = (u2 <= h(u1)/hmax) # rejection criterion
>>> v = u1[idx]                # keep these only
```

Using the χ^2 distribution with the rejection method results in throwing away less than 10 % of the generated samples compared with our prior example where we threw out at least 80 %. This is dramatically more efficient. Figure 2.18 shows

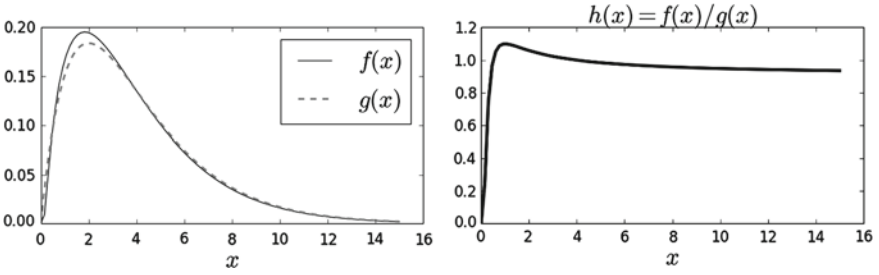

Fig. 2.17 The *plot* on the *right* shows $h(x) = f(x)/g(x)$ and the one on the *left* shows $f(x)$ and $g(x)$ separately

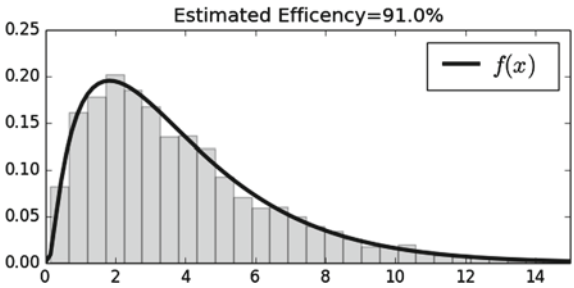

Fig. 2.18 Using the updated method, the histogram matches the target probability density function with high efficiency

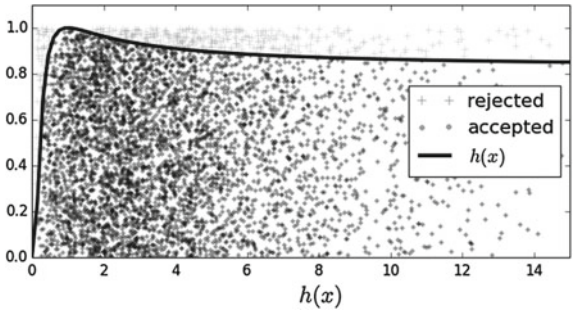

Fig. 2.19 Fewer proposed points were rejected in this case, which means better efficiency

that the histogram and the probability density function match. For completeness, Fig. 2.19 shows the samples with the corresponding threshold $h(x)/h_{\max}$ that was used to select them.

In this section, we investigated how to generate random samples from a given distribution, beit discrete or continuous. For the continuous case, the key issue was whether or not the cumulative density function had a continuous inverse. If not, we had to turn to the rejection method, and find an appropriate related density that we could easily sample from to use as part of a rejection threshold. Finding such a function is an art, but many families of probability densities have been studied over the years that already have fast random number generators.

The rejection method has many complicated extensions that involve careful partitioning of the domains and lots of special methods for corner cases. Nonetheless, all of these advanced techniques are still variations on the same fundamental theme we illustrated here [9, 10].

2.9 Useful Inequalities

In practice, few quantities can be analytically calculated. Some knowledge of bounding inequalities helps find the ballpark for potential solutions. This sections discusses three key inequalities that are important for probability, statistics, and machine learning.

2.9.1 Markov's Inequality

Let X be a non-negative random variable and suppose that $\mathbb{E}(X) < \infty$. Then, for any $t > 0$,

$$\mathbb{P}(X > t) \leq \frac{\mathbb{E}(X)}{t}$$

This is a foundational inequality that is used as a stepping stone to other inequalities. It is easy to prove. Because $X > 0$, we have the following,

$$\mathbb{E}(X) = \int_0^\infty x f_x(x)dx = \underbrace{\int_0^t x f_x(x)dx}_{\text{omit this}} + \int_t^\infty x f_x(x)dx$$
$$\geq \int_t^\infty x f_x(x)dx \geq t \int_t^\infty x f_x(x)dx = t\mathbb{P}(X > t)$$

The step that establishes the inequality is the part where the $\int_0^t x f_x(x)dx$ is omitted. For a particular $f_x(x)$ that my be concentrated around the $[0, t]$ interval, this could be a lot to throw out. For that reason, the Markov Inequality is considered a *loose* inequality, meaning that there is a substantial gap between both sides of the inequality. For example, as shown in Fig. 2.20, the χ^2 distribution has a lot of its mass on the left, which would be omitted in the Markov Inequality. Figure 2.21 shows the two curves established by the Markov Inequality. The gray shaded region is the gap between the two terms and indicates that looseness of the bound (fatter shaded region) for this case.

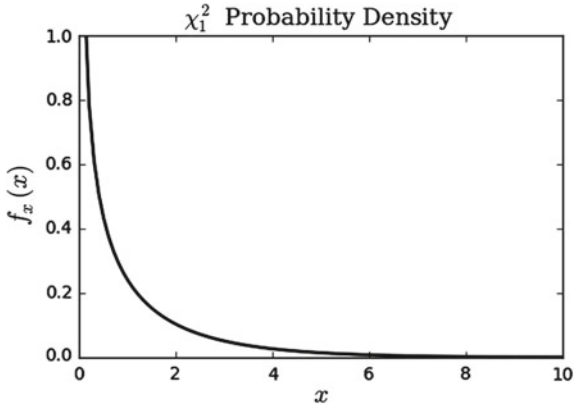

Fig. 2.20 The χ_1^2 density has much of its weight on the *left*, which is excluded in the establishment of the Markov Inequality

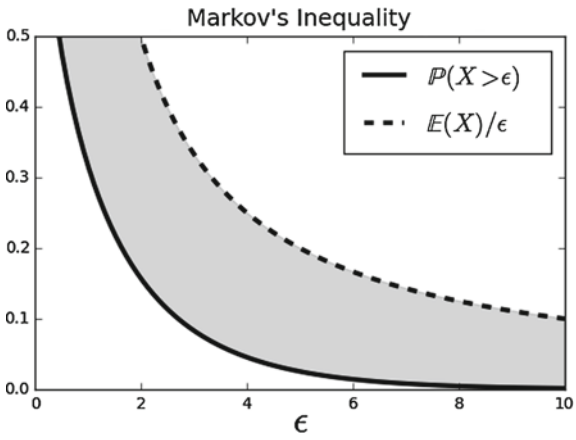

Fig. 2.21 The *shaded area* shows the region between the curves on either side of the Markov Inequality

2.9.2 Chebyshev's Inequality

Chebyshev's Inequality drops out directly from the Markov Inequality. Let $\mu = \mathbb{E}(X)$ and $\sigma^2 = \mathbb{V}(X)$. Then, we have

$$\mathbb{P}(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2}$$

Note that if we normalize so that $Z = (X - \mu)/\sigma$, we have $\mathbb{P}(|Z| \geq k) \leq 1/k^2$. In particular, $\mathbb{P}(|Z| \geq 2) \leq 1/4$. We can illustrate this inequality using Sympy statistics module,

```
>>> import sympy
>>> import sympy.stats as ss
>>> t=sympy.symbols('t',real=True)
>>> x=ss.ChiSquared('x',1)
```

To get the left side of the Chebyshev inequality, we have to write this out as the following conditional probability,

```
>>> r = ss.P((x-1) > t,x>1)+ss.P(-(x-1) > t,x<1)
```

This is because of certain limitations in the statistics module at this point in its development regarding the absolute value function. We could take the above expression, which is a function of t and attempt to compute the integral, but that would take a very long time (the expression is very long and complicated, which is why we did not print it out above). This is because Sympy is a pure-python module that does not utilize any C-level optimizations under the hood. In this situation, it's better to use the built-in cumulative density function as in the following (after some rearrangement of the terms),

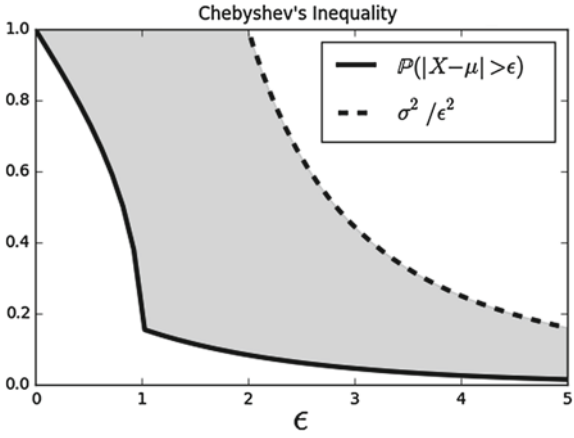

Fig. 2.22 The *shaded area* shows the region between the curves on either side of the Chebyshev Inequality

```
>>> w=(1-ss.cdf(x)(t+1))+ss.cdf(x)(1-t)
```

To plot this, we can evaluate it at a variety of `t` values by using the `.subs` substitution method, but it is more convenient to use the `lambdify` method to convert the expression to a function.

```
>>> fw=sympy.lambdify(t,w)
```

Then, we can evaluate this function using something like

```
>>> map(fw,[0,1,2,3,4])
[1.0,0.157299207050285,0.08326451666355039,0.045500263896358
75,0.0253473186774682]
```

to produce the following Fig. 2.22.

Programming Tip

Note that we cannot use vectorized inputs for the `lambdify` function because it contains embedded functions that are only available in Sympy. Otherwise, we could have used `lambdify(t,fw,numpy)` to specify the corresponding functions in Numpy to use for the expression.

2.9.3 Hoeffding's Inequality

Hoeffding's Inequality is similar, but less loose, than Markov's Inequality. Let $X_1, \ldots, X_n$ be iid observations such that $\mathbb{E}(X_i) = \mu$ and $a \leq X_i \leq b$. Then, for any $\epsilon > 0$, we have

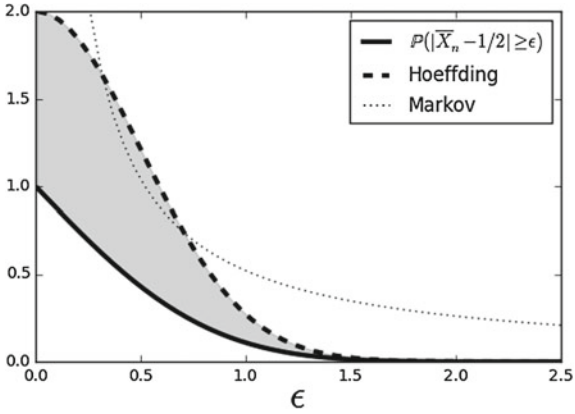

Fig. 2.23 This shows the Markov and Hoeffding bounds for the case of ten identically and uniformly distributed random variables

$$\mathbb{P}(|\overline{X}_n - \mu| \geq \epsilon) \leq 2\exp(-2n\epsilon^2/(b-a)^2)$$

where $\overline{X}_n = \frac{1}{n}\sum_i^n X_i$. Note that we further assume that the individual random variables are bounded.

Corollary *If $X_1, \ldots, X_n$ are independent with $\mathbb{P}(a \leq X_i \leq b) = 1$ and all with $\mathbb{E}(X_i) = \mu$. Then, we have*

$$|\overline{X}_n - \mu| \leq \sqrt{\frac{c}{2n}\log\frac{2}{\delta}}$$

where $c = (b-a)^2$. We will see this inequality again in the machine learning chapter. Figure 2.23 shows the Markov and Hoeffding bounds for the case of ten identically and uniformly distributed random variables, $X_i \sim \mathcal{U}[0, 1]$. The solid line shows $\mathbb{P}(|\overline{X}_n - 1/2| > \epsilon)$. Note that the Hoeffding Inequality is tighter than the Markov Inequality and that both of them merge when ϵ gets big enough.

References

1. F. Jones, *Lebesgue Integration on Euclidean Space*, Jones and Bartlett Books in Mathematics (Jones and Bartlett, London, 2001)
2. G. Strang, *Linear Algebra and Its Applications* (Elsevier Science, 2014). https://books.google.com/books?id=9A7jBQAAQBAJ, ISBN 9781483265117
3. E. Nelson, *Radically Elementary Probability Theory*, Annals of Mathematics Studies (Princeton University Press, Princeton, 1987)
4. T. Mikosch, *Elementary Stochastic Calculus with Finance in View*, Advanced Series on Statistical Science & Applied Probability (World Scientific, Singapore, 1998)
5. H. Kobayashi, B.L. Mark, W. Turin, *Probability, Random Processes, And Statistical Analysis: Applications To Communications, Signal Processing, Queueing Theory And Mathematical Finance*, Engineering Pro Collection (Cambridge University Press, Cambridge, 2011)

6. Z. Brzezniak, T. Zastawniak, *Basic Stochastic Processes: A Course Through Exercises*, Springer Undergraduate Mathematics Series (Springer, London, 1999)
7. D.J.C. MacKay, *Information Theory, Inference and Learning Algorithms* (Cambridge University Press, Cambridge, 2003)
8. T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics (Springer, New York, 2013)
9. W.L. Dunn, J.K. Shultis, *Exploring Monte Carlo Methods* (Elsevier Science, Boston, 2011)
10. N.L. Johnson, S. Kotz, N. Balakrishnan, *Continuous Univariate Distributions*, vol. 2, Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics (Wiley & Sons, New York, 1995)

Chapter 3
Statistics

3.1 Introduction

To get started thinking about statistics, consider the three famous problems

- Suppose you have a bag filled with colored marbles. You close your eyes and reach into it and pull out a handful of marbles, what can you say about what is in the bag?
- You arrive in a strange town and you need a taxicab. You look out the window, and in the dark, you can just barely make out the number on the roof of one of the cabs. In this town, you know they label the cabs sequentially. How many cabs does the town have?
- You have already taken the entrance exam twice and you want to know if it's worth it to take it a third time in the hopes that your score will improve. Because only the last score is reported, you are worried that you may do worse the third time. How do you decide whether or not to take the test again?

Statistics provides a structured way to approach each of these problems. This is important because it is easy to be fooled by your biases and intuitions. Unfortunately, the field does not provide a *single* way to do this, which explains the many library shelves that groan under the weight of statistics texts. This means that although many statistical quantities are easy to *compute*, these are not so easy to justify, explain, or even understand. Fundamentally, when we start with just the data, we lack the underlying probability density that we discussed in the last chapter. This removes key structures that we have to compensate for in however we choose to process the data. In the following, we consider some of the most powerful statistical tools in the Python arsenal and suggest ways to think through them

J. Unpingco, *Python for Probability, Statistics, and Machine Learning*,
DOI 10.1007/978-3-319-30717-6_3

3.2 Python Modules for Statistics

3.2.1 Scipy Statistics Module

Although there are some basic statistical functions in Numpy (e.g., `mean`, `std`, `median`), the real repository for statistical functions is in `scipy.stats`. There are over eighty continuous probability distributions implemented in `scipy.stats` and an additional set of more than ten discrete distributions, along with many other supplementary statistical functions that we will select from in what follows.

To get started with `scipy.stats`, you have to load the module and create an object that has the distribution you're interested in. For example,

```
>>> import scipy.stats # might take awhile
>>> n = scipy.stats.norm(0,10) # create normal distrib
```

The `n` variable is an object that represents a normally distributed random variable with mean zero and standard deviation, $\sigma = 10$. Note that the more general term for these two parameters is *location* and *scale*, respectively. Now that we have this defined, we can compute `mean`, as in the following:

```
>>> n.mean() # we already know this from its definition!
0
```

We can also compute higher order moments as

```
>>> n.moment(4)
30000
```

The main public methods for continuous random variables are

- `rvs`: random variates
- `pdf`: probability density function
- `cdf`: cumulative distribution function
- `sf`: survival Function (1-CDF)
- `ppf`: percent point function (Inverse of CDF)
- `isf`: inverse survival function (Inverse of SF)
- `stats`: mean, variance, (Fisher's) skew, or (Fisher's) kurtosis
- `moment`: non-central moments of the distribution

For example, we can compute the value of the pdf at a specific point.

```
>>> n.pdf(0)
0.039894228040143268
```

or, the `cdf` for the same random variable.

```
>>> n.cdf(0)
0.5
```

You can also create samples from this distribution as in the following:

```
>>> n.rvs(10)
array([ -8.11311677,   1.48034316,   1.0824489 ,  -4.38642452,
        23.69872505, -22.19428082,  -7.19207387,  10.6447697,
         3.4549407 ,   1.67282213])
```

Many common statistical tests are already built-in. For example, Shapiro-Wilks tests the null hypothesis that the data were drawn from a normal distribution,[1] as in the following:

```
>>> scipy.stats.shapiro(n.rvs(100))
(0.9914381504058838, 0.779195249080658)
```

The second value in the tuple is the p-value.

3.2.2 Sympy Statistics Module

Sympy has its own much smaller, but still extremely useful statistics module that enables symbolic manipulation of statistical quantities. For example,

```
>>> from sympy import stats
>>> X = stats.Normal('x',0,10) # create normal random variable
```

We can obtain the probability density function as,

```
>>> from sympy.abc import x
>>> stats.density(X)(x)
sqrt(2)*exp(-x**2/200)/(20*sqrt(pi))
```

and we can evaluate the cumulative density function as the following,

```
>>> stats.cdf(X)(0)
1/2
```

Note that you can evaluate this numerically by using the `evalf()` method on the output. Sympy provides intuitive ways to consider standard probability questions by using the `stats.P` function, as in the following:

```
>>> stats.P(X>0) # prob X >0?
1/2
```

There is also a corresponding expectation function, `stats.E` you can use to compute complicated expectations using all of Sympy's powerful built-in integration machinery. For example we can compute, $\mathbb{E}(\sqrt{|X|})$ in the following,

```
>>> stats.E(abs(X)**(1/2)).evalf()
2.59995815363879
```

Unfortunately, there is very limited support for multivariate distributions at the time of this writing.

[1] We will explain null hypothesis and the rest of it later.

3.2.3 Other Python Modules for Statistics

There are many other important Python modules for statistical work. Two important modules are Seaborn and Statsmodels. As we discussed earlier, Seaborn is library built on top of Matplotlib for very detailed and expressive statistical visualizations, ideally suited for exploratory data analysis. Statsmodels is designed to complement Scipy with descriptive statistics, estimation, and inference for a large variety of statistical models. Statsmodels includes (among many others) generalized linear models, robust linear models, and methods for timeseries analysis, with an emphasis on econometric data and problems. Both these modules are well supported and very well documented and designed to integrate tightly into Matplotlib, Numpy, Scipy, and the rest of the scientific Python stack. Because the focus of this text is more conceptual as opposed to domain-specific, I have chosen not to emphasize either of these, notwithstanding how powerful each is.

3.3 Types of Convergence

The absence of the probability density for the raw data means that we have to argue about sequences of random variables in a structured way. From basic calculus, recall the following convergence notation,

$$x_n \to x_o$$

for the real number sequence x_n. This means that for any given $\epsilon > 0$, no matter how small, we can exhibit a m such that for any $n > m$, we have

$$|x_n - x_o| < \epsilon$$

Intuitively, this means that once we get past m in the sequence, we get as to within ϵ of x_o. This means that nothing surprising happens in the sequence on the long march to infinity, which gives a sense of uniformity to the convergence process. When we argue about convergence for statistics, we want to same look-and-feel as we have here, but because we are now talking about random variables, we need other concepts. There are two moving parts for random variables. Recall that random variables are really functions that map sets into the real line: $X : \Omega \mapsto \mathbb{R}$. Thus, one part to keep track of is the behavior of the subsets of Ω while arguing about convergence. The other part is the sequence of values that the random variable takes on the real line and how those behave in the convergence process.

3.3.1 Almost Sure Convergence

The most straightforward extension into statistics of this convergence concept is *convergence with probability one*, which is also known as *almost sure convergence*, which is the following,

$$P\{\text{for each } \epsilon > 0 \text{ there is } n_\epsilon > 0 \text{ such that for all } n > n_\epsilon,\ |X_n - X| < \epsilon\} = 1 \tag{3.3.1.1}$$

Note the similarity to the prior notion of convergence for real numbers. When this happens, we write this as $X_n \xrightarrow{as} X$. In this context, almost sure convergence means that if we take any particular $\omega \in \Omega$ and then look at the sequence of real numbers that are produced by each of the random variables,

$$(X_1(\omega), X_2(\omega), X_3(\omega), \ldots, X_n(\omega))$$

then this sequence is just a real-valued sequence in the sense of our convergence on the real line and converges in the same way. If we collect all of the ω for which this is true and the measure of that collection equals one, then we have almost sure convergence of the random variable. Notice how the convergence idea applies to both sides of the random variable: the (domain) Ω side and the (co-domain) real-valued side.

An equivalent and more compact way of writing this is the following,

$$P\left(\omega \in \Omega : \lim_{n\to\infty} X_n(\omega) = X(\omega)\right) = 1$$

Example To get some feel for the mechanics of this kind of convergence consider the following sequence of uniformly distributed random variables on the unit interval, $X_n \sim \mathcal{U}[0, 1]$. Now, consider taking the maximum of the set of n such variables as the following,

$$X_{(n)} = \max\{X_1, \ldots, X_n\}$$

In other words, we scan through a list of n uniformly distributed random variables and pick out the maximum over the set. Intuitively, we should expect that $X_{(n)}$ should somehow converge to one. Let's see if we can make this happen almost surely. We want to exhibit m so that the following is true,

$$P(|1 - X_{(n)}|) < \epsilon \text{ when } n > m$$

Because $X_{(n)} < 1$, we can simplify this as the following,

$$1 - P(X_{(n)} < \epsilon) = 1 - (1 - \epsilon)^m \underset{m\to\infty}{\longrightarrow} 1$$

Thus, this sequence converges almost surely. We can work this example out in Python using Scipy to make it concrete with the following code,

```
>>> from scipy import stats
>>> u=stats.uniform()
>>> xn = lambda i: u.rvs(i).max()
>>> xn(5)
0.96671783848200299
```

Thus, the `xn` variable is the same as the $X_{(n)}$ random variable in our example. Figure 3.1 shows a plot of these random variables for different values of n and multiple realizations of each random variable (multiple gray lines). The dark horizontal line is at the `0.95` level. For this example, suppose we are interested in the convergence of the random variable to within `0.05` of one so we are interested in the region between one and `0.95`. Thus, in our Eq. 3.3.1.1, $\epsilon = 0.05$. Now, we have to find n_ϵ to get the almost sure convergence. From Fig. 3.1, as soon as we get past $n > 60$, we can see that all the realizations start to fit in the region above the `0.95` horizontal line. However, there are still some cases where a particular realization will skip below this line. To get the probability guarantee of the definition satisfied, we have to make sure that for whatever n_ϵ we settle on, the probability of this kind of noncompliant behavior should be extremely small, say, less than 1 %. Now, we can compute the following to estimate this probability for $n = 60$ over 1000 realizations,

```
>>> import numpy as np
>>> np.mean([xn(60) > 0.95 for i in range(1000)])
0.96099999999999997
```

So, the probability of having a noncompliant case beyond $n > 60$ is pretty good, but not still what we are after (`0.99`). We can solve for the m in our analytic proof of convergence by plugging in our factors for ϵ and our desired probability constraint,

```
>>> print np.log(1-.99)/np.log(.95)
89.7811349607
```

Now, rounding this up and re-visiting the same estimate as above,

```
>>> import numpy as np
>>> np.mean([xn(90) > 0.95 for i in range(1000)])
0.995
```

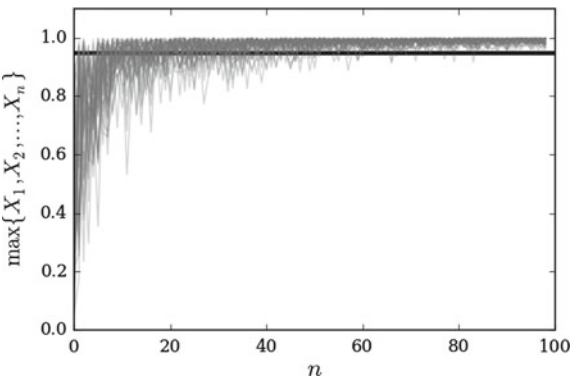

Fig. 3.1 Almost sure convergence example for multiple realizations of the limiting sequence

which is the result we were looking for. The important thing to understand from this example is that we had to choose convergence criteria for *both* the values of the random variable (`0.95`) and for the probability of achieving that level (`0.99`) in order to compute the m. Informally speaking, almost sure convergence means that not only will any particular X_n be close to X for large n, but whole sequence of values will remain close to X with high probability.

3.3.2 Convergence in Probability

A weaker kind of convergence is *convergence in probability* which means the following:

$$\mathbb{P}(\mid X_n - X \mid > \epsilon) \to 0$$

as $n \to \infty$ for each $\epsilon > 0$.

This is notationally shown as $X_n \xrightarrow{P} X$. For example, let's consider the following sequence of random variables where $X_n = 1/2^n$ with probability p_n and where $X_n = c$ with probability $1 - p_n$. Then, we have $X_n \xrightarrow{P} 0$ as $p_n \to 1$. This is allowable under this notion of convergence because a diminishing amount of *non-converging* behavior (namely, when $X_n = c$) is possible. Note that we have said nothing about *how* $p_n \to 1$.

Example To get some sense of the mechanics of this kind of convergence, let $\{X_1, X_2, X_3, \ldots\}$ be the indicators of the corresponding intervals,

$$(0, 1], (0, \tfrac{1}{2}], (\tfrac{1}{2}, 1], (0, \tfrac{1}{3}], (\tfrac{1}{3}, \tfrac{2}{3}], (\tfrac{2}{3}, 1]$$

In other words, just keep splitting the unit interval into equal chunks and enumerate those chunks with X_i. Because each X_i is an indicator function, it takes only two values: zero and one. For example, for $X_2 = 1$ if $0 < x \leq 1/2$ and zero otherwise. Note that $x \sim \mathcal{U}(0, 1)$. This means that $P(X_2 = 1) = 1/2$. Now, we want to compute the sequence of $P(X_n > \epsilon)$ for each n for some $\epsilon \in (0, 1)$. For X_1, we have $P(X_1 > \epsilon) = 1$ because we already chose ϵ in the interval covered by X_1. For X_2, we have $P(X_2 > \epsilon) = 1/2$, for X_3, we have $P(X_3 > \epsilon) = 1/3$, and so on. This produces the following sequence: $(1, \frac{1}{2}, \frac{1}{2}, \frac{1}{3}, \frac{1}{3}, \ldots)$. The limit of the sequence is zero so that $X_n \xrightarrow{P} 0$. However, for every $x \in (0, 1)$, the sequence of function values of $X_n(x)$ consists of infinitely many zeros and ones (remember that indicator functions can evaluate to either zero or one). Thus, the set of x for which the sequence $X_n(x)$ converges is empty because the sequence bounces between zero and one. This means that almost sure convergence fails here even though we have convergence in probability. The key distinction is that convergence in probability considers the

convergence of a sequence of probabilities whereas almost sure convergence is concerned about the sequence of values of the random variables over sets of events that *fill out* the underlying probability space entirely (i.e., with probability one).

This is a good example so let's see if we can make it concrete with some Python. The following is a function to compute the different subintervals,

```
>>> make_interval= lambda n: np.array(zip(range(n+1),range(1,n+1)))/n
```

Now, we can use this function to create a Numpy array of intervals, as in the example,

```
>>> intervals= np.vstack([make_interval(i) for i in range(1,5)])
>>> print intervals
[[ 0.          1.        ]
 [ 0.          0.5       ]
 [ 0.5         1.        ]
 [ 0.          0.33333333]
 [ 0.33333333  0.66666667]
 [ 0.66666667  1.        ]
 [ 0.          0.25      ]
 [ 0.25        0.5       ]
 [ 0.5         0.75      ]
 [ 0.75        1.        ]]
```

The following function computes the bit string in our example, $\{X_1, X_2, \ldots, X_n\}$,

```
>>> bits= lambda u:((intervals[:,0] < u) & (u<=intervals[:,1])).astype(int)
>>> bits(u.rvs())
array([1, 0, 1, 0, 0, 1, 0, 0, 0, 1])
```

Now that we have the individual bit strings, to show convergence we want to show that the probability of each entry goes to a limit. For example, using ten realizations,

```
>>> print np.vstack([bits(u.rvs()) for i in range(10)])
[[1 1 0 1 0 0 0 1 0 0]
 [1 1 0 1 0 0 0 1 0 0]
 [1 1 0 0 1 0 0 1 0 0]
 [1 0 1 0 0 1 0 0 1 0]
 [1 0 1 0 0 1 0 0 1 0]
 [1 1 0 0 1 0 0 1 0 0]
 [1 1 0 1 0 0 1 0 0 0]
 [1 1 0 0 1 0 0 1 0 0]
 [1 1 0 0 1 0 0 1 0 0]
 [1 1 0 1 0 0 1 0 0 0]]
```

We want the limiting probability of a one in each column to convert to a limit. We can estimate this over 1000 realizations using the following code,

```
>>> np.vstack([bits(u.rvs()) for i in range(1000)]).mean(axis=0)
array([ 1.   ,  0.493,  0.507,  0.325,  0.34 ,  0.335,  0.253,  0.24 ,
        0.248,  0.259])
```

Note that these entries should approach the $(1, \frac{1}{2}, \frac{1}{2}, \frac{1}{3}, \frac{1}{3}, \ldots)$ sequence we found earlier. Figure 3.2 shows the convergence of these probabilities for a large number of intervals. Eventually, the probability shown on this graph will decrease to zero with large enough n. Again, note that the individual sequences of zeros and ones do not converge, but the probabilities of these sequences converge. This is the key difference between almost sure convergence and convergence in probability. Thus, convergence in probability does *not* imply almost sure convergence. Conversely, almost sure convergence *does* imply convergence in probability.

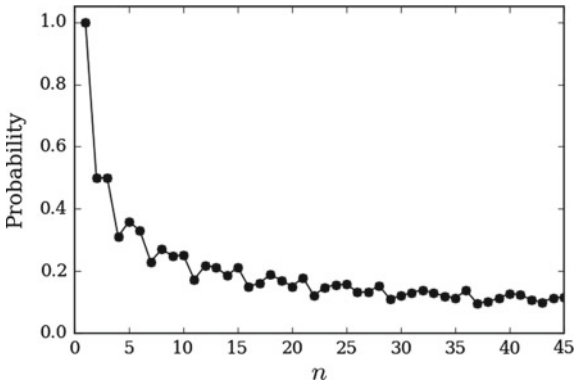

Fig. 3.2 Convergence in probability for the random variable sequence

The following notation should help emphasize the difference between almost sure convergence and convergence in probability, respectively,

$$P\left(\lim_{n\to\infty} |X_n - X| < \epsilon\right) = 1 \text{(almost sure convergence)}$$
$$\lim_{n\to\infty} P(|X_n - X| < \epsilon) = 1 \text{(convergence in probability)}$$

3.3.3 Convergence in Distribution

So far, we have been discussing convergence in terms of sequences of probabilities or sequences of values taken by the random variable. By contrast, the next major kind of convergence is *convergence in distribution* where

$$\lim_{n\to\infty} F_n(t) = F(t)$$

for all t for which F is continuous and F is the cumulative density function. For this case, convergence is only concerned with the cumulative density function, written as $X_n \xrightarrow{d} X$.

Example To develop some intuition about this kind of convergence, consider a sequence of X_n Bernoulli random variables. Furthermore, suppose these are all really just the same random variable X. Trivially, $X_n \xrightarrow{d} X$. Now, suppose we define $Y = 1 - X$, which means that Y has the same distribution as X. Thus, $X_n \xrightarrow{d} Y$. By contrast, because $|X_n - Y| = 1$ for all n, we can never have almost sure convergence or convergence in probability. Thus, convergence in distribution is the weakest of the three forms of convergence in the sense that it is implied by the other two, but implies neither of the two.

As another striking example, we could have $Y_n \xrightarrow{d} Z$ where $Z \sim \mathcal{N}(0, 1)$, but we could also have $Y_n \xrightarrow{d} -Z$. That is, Y_n could converge in distribution to either Z or $-Z$. This may seem ambiguous, but this kind of convergence is practically very useful because it allows for complicated distributions to be approximated by simpler distributions.

3.3.4 Limit Theorems

Now that we have all of these notions of convergence, we can apply them to different situations and see what kinds of claims we can construct from them.

Weak Law of Large Numbers. Let $\{X_1, X_2, \ldots, X_n\}$ be an iid set of random variables with finite mean $\mathbb{E}(X_k) = \mu$ and finite variance. Let $\overline{X}_n = \frac{1}{n}\sum_k X_k$. Then, we have $\overline{X}_n \xrightarrow{P} \mu$. This result is important because we frequently estimate parameters using an averaging process of some kind. This basically justifies this in terms of convergence in probability. Informally, this means that the distribution of $\overline{X}_n$ becomes concentrated around μ as $n \to \infty$.

Strong Law of Large Numbers. Let $\{X_1, X_2, \ldots, \}$ be an iid set of random variables. Suppose that $\mu = \mathbb{E}|X_i| < \infty$, then $\overline{X}_n \xrightarrow{as} \mu$. The reason this is called the strong law is that it implies the weak law because almost sure convergence implies convergence in probability. The so-called Komogorov criterion gives the convergence of the following,

$$\sum_k \frac{\sigma_k^2}{k^2}$$

as a sufficient condition for concluding that the Strong Law applies to the sequence $\{X_k\}$ with corresponding $\{\sigma_k^2\}$.

As an example, consider an infinite sequence of Bernoulli trials with $X_i = 1$ if the ith trial is successful. Then $\overline{X}_n$ is the relative frequency of successes in n trials and $\mathbb{E}(X_i)$ is the probability p of success on the ith trial. With all that established, the Weak Law says only that if we consider a sufficiently large and fixed n, the probability that the relative frequency will converge to p is guaranteed. The Strong Law states that if we regard the observation of all the infinite $\{X_i\}$ as one performance of the experiment, the relative frequency of successes will almost surely converge to p. The difference between the Strong Law and the Weak Law of large numbers is subtle and rarely arises in practical applications of probability theory.

Central Limit Theorem. Although the Weak Law of Large Numbers tells us that the distribution of $\overline{X}_n$ becomes concentrated around μ, it does not tell us what that distribution is. The Central Limit Theorem (CLT) says that $\overline{X}_n$ has a distribution that is approximately Normal with mean μ and variance σ^2/n. Amazingly, nothing is

assumed about the distribution of X_i, except the existence of the mean and variance. The following is the Central Limit Theorem: Let $\{X_1, X_2, \ldots, X_n\}$ be iid with mean μ and variance σ^2. Then,

$$Z_n = \frac{\sqrt{n}(\overline{X}_n - \mu)}{\sigma} \xrightarrow{P} Z \sim \mathcal{N}(0, 1)$$

The loose interpretation of the Central Limit Theorem is that $\overline{X}_n$ can be legitimately approximated by a Normal distribution. Because we are talking about convergence in probability here, claims about probability are legitimized, not claims about the random variable itself. Intuitively, this shows that normality arises from sums of small, independent disturbances of finite variance. Technically, the finite variance assumption is essential for normality. Although the Central Limit Theorem provides a powerful, general approximation, the quality of the approximation for a particular situation still depends on the original (usually unknown) distribution.

3.4 Estimation Using Maximum Likelihood

The estimation problem starts with the desire to infer something meaningful from data. For parametric estimation, the strategy is to postulate a model for the data and then use the data to fit model parameters. This leads to two fundamental questions: where to get the model and how to estimate the parameters? The first question is best answered by the maxim: *all models are wrong, some are useful*. In other words, choosing a model depends as much on the application as on the model itself. Think about models as building different telescopes to view the sky. No one would ever claim that the telescope generates the sky! It is same with data models. Models give us multiple perspectives on the data that themselves are proxies for some deeper underlying phenomenon.

Some categories of data may be more commonly studied using certain types of models, but this is usually very domain-specific and ultimately depends on the aims of the analysis. In some cases, there may be strong physical reasons behind choosing a model. For example, one could postulate that the model is linear with some noise as in the following:

$$Y = aX + \epsilon$$

which basically says that you, as the experimenter, dial in some value for X and then read off something directly proportional to X as the measurement, Y, plus some additive noise that you attribute to jitter in the apparatus. Then, the next step is to estimate the paramater a in the model, given some postulated claim about the nature of ϵ. How to compute the model parameters depends on the particular methodology. The two broad rubrics are parametric and non-parametric estimation. In the former, we assume we know the density function of the data and then try to derive the

embedded parameters for it. In the latter, we claim only to know that the density function is a member of a broad class of density functions and then use the data to characterize a member of that class. Broadly speaking, the former consumes less data than the latter, because there are fewer unknowns to compute from the data.

Let's concentrate on parametric estimation for now. The tradition is to denote the unknown parameter to be estimated as θ which is a member of a large space of alternates, Θ. To judge between potential θ values, we need an objective function, known as a *risk* function, $L(\theta, \hat{\theta})$, where $\hat{\theta}(\mathbf{x})$ is an estimate for the unknown θ that is derived from the available data $\mathbf{x}$. The most common and useful risk function is the squared error loss,

$$L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$$

Although neat, this is not practical because we need to know the unknown θ to compute it. The other problem is because $\hat{\theta}$ is a function of the observed data, it is also a random variable with its own probability density function. This leads to the notion of the *expected risk* function,

$$R(\theta, \hat{\theta}) = \mathbb{E}_\theta(L(\theta, \hat{\theta})) = \int L(\theta, \hat{\theta}(\mathbf{x}))\, f(\mathbf{x}; \theta) d\mathbf{x}$$

In other words, given a fixed θ, integrate over the probability density function of the data, $f(\mathbf{x})$, to compute the risk. Plugging in for the squared error loss, we compute the mean squared error,

$$\mathbb{E}_\theta(\theta - \hat{\theta})^2 = \int (\theta - \hat{\theta})^2 f(\mathbf{x}; \theta) d\mathbf{x}$$

This has the important factorization into the *bias*,

$$\text{bias} = \mathbb{E}_\theta(\hat{\theta}) - \theta$$

with the corresponding variance, $\mathbb{V}_\theta(\hat{\theta})$ as in the following *mean squared error* (MSE):

$$\mathbb{E}_\theta(\theta - \hat{\theta})^2 = \text{bias}^2 + \mathbb{V}_\theta(\hat{\theta})$$

This is an important trade-off that we will return to repeatedly. The idea is the bias is nonzero when the estimator $\hat{\theta}$, integrated over all possible data, $f(\mathbf{x})$, does not equal the underlying target parameter θ. In some sense, the estimator misses the target, no matter how much data is used. When the bias equals zero, the estimated is *unbiased*. For fixed MSE, low bias implies high variance and vice-versa. This trade-off was once not emphasized and instead much attention was paid to the smallest variance of unbiased estimators (see Cramer-Rao bounds). In practice, understanding and exploiting the trade-off between bias and variance and reducing the MSE is more important.

With all this set up, we can now ask how bad can bad get by examining *minimax* risk,

$$R_{\text{mmx}} = \inf_{\hat{\theta}} \sup_{\theta} R(\theta, \hat{\theta})$$

where the inf is take over all estimators. Intuitively, this means if we found the worst possible θ and swept over all possible parameter estimators $\hat{\theta}$, and then took the smallest possible risk we could find, we would have the minimax risk. Thus, an estimator, $\hat{\theta}_{\text{mmx}}$, is a *minimax estimator* if it achieves this feat,

$$\sup_{\theta} R(\theta, \hat{\theta}_{\text{mmx}}) = \inf_{\hat{\theta}} \sup_{\theta} R(\theta, \hat{\theta})$$

In other words, even in the face of the worst θ (i.e., the $\sup_\theta$), $\hat{\theta}_{\text{mmx}}$ still achieves the minimax risk. There is a greater theory that revolves around minimax estimators of various kinds, but this is far beyond our scope here. The main thing to focus on is that under certain technical but easily satisfiable conditions, the maximum likelihood estimator is approximately minimax. Maximum likelihood is the subject of the next section. Let's get started with the simplest application: coin-flipping.

3.4.1 Setting Up the Coin Flipping Experiment

Suppose we have coin and want to estimate the probability of heads (p) for it. We model the distribution of heads and tails as a Bernoulli distribution with the following probability mass function:

$$\phi(x) = p^x(1-p)^{(1-x)}$$

where x is the outcome, *1* for heads and *0* for tails. Note that maximum likelihood is a parametric method that requires the specification of a particular model for which we will compute embedded parameters. For n independent flips, we have the joint density as the product of n of these functions as in,

$$\phi(\mathbf{x}) = \prod_{i=1}^{n} p_i^x(1-p)^{(1-x_i)}$$

The following is the *likelihood function*,

$$\mathcal{L}(p; \mathbf{x}) = \prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i}$$

This is basically notation. We have just renamed the previous equation to emphasize the p parameter, which is what we want to estimate.

The principle of *maximum likelihood* is to maximize the likelihood as the function of p after plugging in all of the x_i data. We then call this maximizer $\hat{p}$ which is a function of the observed x_i data, and as such, is a random variable with its own distribution. This method therefore ingests data and an assumed model for the probability density, and produces a function that estimates the embedded parameter in the assumed probability density. Thus, maximum likelihood generates the *functions* of data that we need in order to get at the underlying parameters of the model. Note that there is no limit to the ways we can functionally manipulate the data we have collected. The maximum likelihood principle gives us a systematic method for constructing these functions subject to the assumed model. This is a point worth emphasizing: the maximum likelihood principle yields functions as solutions the same way solving differential equations yields functions as solutions. It is very, very much harder to produce a function than to produce a value as a solution, even with the assumption of a convenient probability density. Thus, the power of the principle is that you can construct such functions subject to the model assumptions.

Simulating the Experiment. We need the following code to simulate coin flipping.

```
>>> from scipy.stats import bernoulli
>>> p_true=1/2.0         # estimate this!
>>> fp=bernoulli(p_true) # create bernoulli random variate
>>> xs = fp.rvs(100)     # generate some samples
>>> print xs[:30]        # see first 30 samples
[0 1 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0 1]
```

Now, we can write out the likelihood function using Sympy. Note that we give the Sympy variables the `positive=True` attribute upon construction because this eases Sympy's internal simplification algorithms.

```
>>> import sympy
>>> x,p,z=sympy.symbols('x p z', positive=True)
>>> phi=p**x*(1-p)**(1-x) # distribution function
>>> L=np.prod([phi.subs(x,i) for i in xs]) # likelihood function
>>> print L # approx 0.5?
p**57*(-p + 1)**43
```

Note that, once we plug in the data, the likelihood function is solely a function of the unknown parameter (p in this case). The following code uses calculus to find the extrema of the likelihood function. Note that taking the `log` of L makes the maximization problem tractable but doesn't change the extrema.

```
>>> logL=sympy.expand_log(sympy.log(L))
>>> sol,=sympy.solve(sympy.diff(logL,p),p)
>>> print sol
57/100
```

Programming Tip

Note that `sol,=sympy.solve` statement includes a comma after the `sol` variable. This is because the `solve` function returns a list containing a single element. Using this assignment unpacks that single element into the `sol` variable directly. This is another one of the many small elegancies of Python.

The following code generates Fig. 3.3.

```
fig,ax=subplots()
x=np.linspace(0,1,100)
ax.plot(x,map(sympy.lambdify(p,logJ,'numpy'),x),'k-',lw=3)
ax.plot(sol,logJ.subs(p,sol),'o',
        color='gray',ms=15,label='Estimated')
ax.plot(p_true,logJ.subs(p,p_true),'s',
        color='k',ms=15,label='Actual')
ax.set_xlabel('$p$',fontsize=18)
ax.set_ylabel('Likelihood',fontsize=18)
ax.set_title('Estimate not equal to true value',fontsize=18)
ax.legend(loc=0)
```

Programming Tip

In the prior code, we use the `lambdify` function in `lambdify(p,logJ,'numpy')` to take a Sympy expression and convert it into a Numpy version that is easier to compute. The `lambdify` function has an extra argument where you can specify the function space that it should use to convert the expression. In the above this is set to Numpy.

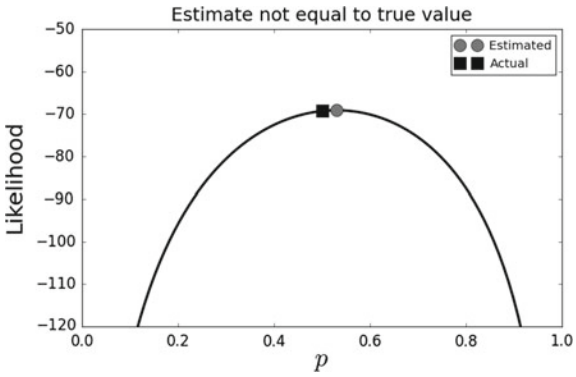

Fig. 3.3 Maximum likelihood estimate versus true parameter. Note that the estimate is slightly off from the true value. This is a consequence of the fact that the estimator is a function of the data and lacks knowledge of the true underlying value

Figure 3.3 shows that our estimator $\hat{p}$ (circle) is not equal to the true value of p (square), despite being the maximum of the likelihood function. This may sound disturbing, but keep in mind this estimate is a function of the random data; and since that data can change, the ultimate estimate can likewise change. I invite you to run this code in the corresponding IPython notebook a few times to observe this. Remember that the estimator is a *function* of the data and is thus also a *random variable*, just like the data is. This means it has its own probability distribution with corresponding mean and variance. So, what we are observing is a consequence of that variance.

Figure 3.4 shows what happens when you run many thousands of coin experiments and compute the maximum likelihood estimate for each experiment, given a particular number of samples per experiment. This simulation gives us a histogram of the maximum likelihood estimates, which is an approximation of the probability distribution of the $\hat{p}$ estimator itself. This figure shows that the sample mean of the estimator ($\mu = \frac{1}{n}\sum \hat{p}_i$) is pretty close to the true value, but looks can be deceiving. The only way to know for sure is to check if the estimator is unbiased, namely, if

$$\mathbb{E}(\hat{p}) = p$$

Because this problem is simple, we can solve for this in general noting that the terms above are either p, if $x_i = 1$ or $1 - p$ if $x_i = 0$. This means that we can write

$$\mathcal{L}(p|\mathbf{x}) = p^{\sum_{i=1}^{n} x_i}(1-p)^{n-\sum_{i=1}^{n} x_i}$$

with corresponding logarithm as

$$J = \log(\mathcal{L}(p|\mathbf{x})) = \log(p)\sum_{i=1}^{n} x_i + \log(1-p)\left(n - \sum_{i=1}^{n} x_i\right)$$

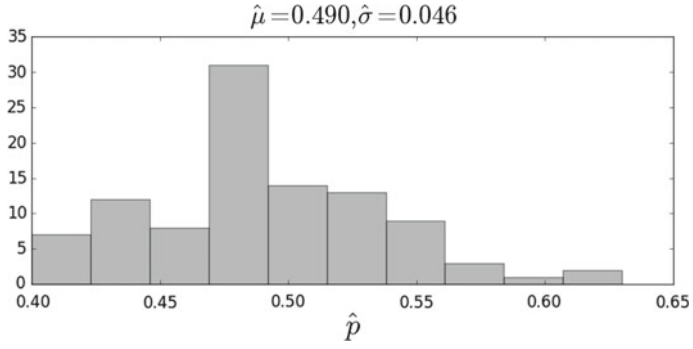

Fig. 3.4 Histogram of maximum likelihood estimates. The title shows the estimated mean and standard deviation of the samples

Taking the derivative of this gives:

$$\frac{dJ}{dp} = \frac{1}{p}\sum_{i=1}^{n} x_i + \frac{(n - \sum_{i=1}^{n} x_i)}{p-1}$$

and solving this for p leads to

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

This is our *estimator* for p. Up until now, we have been using Sympy to solve for this based on the data x_i but now that we have it analytically we don't have to solve for it each time. To check if this estimator is biased, we compute its expectation:

$$\mathbb{E}\left(\hat{p}\right) = \frac{1}{n}\sum_{i}^{n} \mathbb{E}(x_i) = \frac{1}{n} n \mathbb{E}(x_i)$$

by linearity of the expectation and where

$$\mathbb{E}(x_i) = p$$

Therefore,

$$\mathbb{E}\left(\hat{p}\right) = p$$

This means that the estimator is *unbiased*. Similarly,

$$\mathbb{E}\left(\hat{p}^2\right) = \frac{1}{n^2}\mathbb{E}\left[\left(\sum_{i=1}^{n} x_i\right)^2\right]$$

and where

$$\mathbb{E}\left(x_i^2\right) = p$$

and by the independence assumption,

$$\mathbb{E}\left(x_i x_j\right) = \mathbb{E}(x_i)\mathbb{E}(x_j) = p^2$$

Thus,

$$\mathbb{E}\left(\hat{p}^2\right) = \left(\frac{1}{n^2}\right) n \left[p + (n-1)p^2\right]$$

So, the variance of the estimator, $\hat{p}$, is the following:

$$\mathbb{V}(\hat{p}) = \mathbb{E}\left(\hat{p}^2\right) - \mathbb{E}\left(\hat{p}\right)^2 = \frac{p(1-p)}{n}$$

Note that the n in the denominator means that the variance asymptotically goes to zero as n increases (i.e., we consider more and more samples). This is good news because it means that more and more coin flips lead to a better estimate of the underlying p.

Unfortunately, this formula for the variance is practically useless because we need p to compute it and p is the parameter we are trying to estimate in the first place! However, this is where the *plug-in* principle[2] saves the day. It turns out in this situation, you can simply substitute the maximum likelihood estimator, $\hat{p}$, for the p in the above equation to obtain the asymptotic variance for $\mathbb{V}(\hat{p})$. The fact that this works is guaranteed by the asymptotic theory of maximum likelihood estimators.

Nevertheless, looking at $\mathbb{V}(\hat{p})^2$, we can immediately notice that if $p = 0$, then there is no estimator variance because the outcomes are guaranteed to be tails. Also, for any n, the maximum of this variance happens at $p = 1/2$. This is our worst case scenario and the only way to compensate is with larger n.

All we have computed is the mean and variance of the estimator. In general, this is insufficient to characterize the underlying probability density of $\hat{p}$, except if we somehow knew that $\hat{p}$ were normally distributed. This is where the powerful *Central Limit Theorem* we discussed in Sect. 3.3.4 comes in. The form of the estimator, which is just a sample mean, implies that we can apply this theorem and conclude that $\hat{p}$ is asymptotically normally distributed. However, it doesn't quantify how many samples n we need. In our simulation this is no problem because we can generate as much data as we like, but in the real world, with a costly experiment, each sample may be precious.[3]

In the following, we won't apply the Central Limit Theorem and instead proceed analytically.

Probability Density for the Estimator. To write out the full density for $\hat{p}$, we first have to ask what is the probability that the estimator will equal a specific value and the tally up all the ways that could happen with their corresponding probabilities. For example, what is the probability that

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} x_i = 0$$

[2] This is also known as the *invariance property* of maximum likelihood estimators. It basically states that the maximum likelihood estimator of any function, say, $h(\theta)$, is the same h with the maximum likelihood estimator for θ substituted in for θ; namely, $h(\theta_{ML})$.

[3] It turns out that the central limit theorem augmented with an Edgeworth expansion tells us that convergence is regulated by the skewness of the distribution [1]. In other words, the more symmetric the distribution, the faster it converges to the normal distribution according to the central limit theorem.

This can only happen one way: when $x_i = 0\ \forall i$. The probability of this happening can be computed from the density

$$f(\mathbf{x}, p) = \prod_{i=1}^{n} \left(p^{x_i}(1-p)^{1-x_i}\right)$$

$$f\left(\sum_{i=1}^{n} x_i = 0, p\right) = (1-p)^n$$

Likewise, if $\{x_i\}$ has only one nonzero element, then

$$f\left(\sum_{i=1}^{n} x_i = 1, p\right) = np \prod_{i=1}^{n-1} (1-p)$$

where the n comes from the n ways to pick one element from the n elements x_i. Continuing this way, we can construct the entire density as

$$f\left(\sum_{i=1}^{n} x_i = k, p\right) = \binom{n}{k} p^k (1-p)^{n-k}$$

where the first term on the right is the binomial coefficient of n things taken k at a time. This is the binomial distribution and it's not the density for $\hat{p}$, but rather for $n\hat{p}$. We'll leave this as-is because it's easier to work with below. We just have to remember to keep track of the n factor.

Confidence Intervals Now that we have the full density for $\hat{p}$, we are ready to ask some meaningful questions. For example, what is the probability the estimator is within ϵ fraction of the true value of p?

$$\mathbb{P}\left(|\hat{p} - p| \leq \epsilon p\right)$$

More concretely, we want to know how often the estimated $\hat{p}$ is trapped within ϵ of the actual value. That is, suppose we ran the experiment 1000 times to generate 1000 different estimates of $\hat{p}$. What percentage of the 1000 so-computed values are trapped within ϵ of the underlying value. Rewriting the above equation as the following,

$$\mathbb{P}\left(p - \epsilon p < \hat{p} < p + \epsilon p\right) = \mathbb{P}\left(np - n\epsilon p < \sum_{i=1}^{n} x_i < np + n\epsilon p\right)$$

Let's plug in some live numbers here for our worst case scenario (i.e., highest variance scenario) where $p = 1/2$. Then, if $\epsilon = 1/100$, we have

$$\mathbb{P}\left(\frac{99n}{100} < \sum_{i=1}^{n} x_i < \frac{101n}{100}\right)$$

Since the sum in integer-valued, we need $n > 100$ to even compute this. Thus, if $n = 101$ we have,

$$\mathbb{P}\left(\frac{9999}{200} < \sum_{i=1}^{101} x_i < \frac{10201}{200}\right) = f\left(\sum_{i=1}^{101} x_i = 50,\, p\right)\ldots$$
$$= \binom{101}{50}(1/2)^{50}(1-1/2)^{101-50} \;=\; 0.079$$

This means that in the worst-case scenario for $p = 1/2$, given $n = 101$ trials, we will only get within 1 % of the actual $p = 1/2$ about 8 % of the time. If you feel disappointed, that only means you've been paying attention. What if the coin was really heavy and it was hard work to repeat this 101 times?

Let's come at this another way: given I could only flip the coin 100 times, how close could I come to the true underlying value with high probability (say, 95 %)? In this case, instead of picking a value for ϵ, we are solving for ϵ. Plugging in gives,

$$\mathbb{P}\left(50 - 50\epsilon < \sum_{i=1}^{100} x_i < 50 + 50\epsilon\right) = 0.95$$

which we have to solve for ϵ. Fortunately, all the tools we need to solve for this are already in Scipy (Fig. 3.5).

```
>>> from scipy.stats import binom
>>> # n=100, p = 0.5, distribution of the estimator phat
>>> b=binom(100,.5)
>>> # symmetric sum the probability around the mean
>>> g = lambda i:b.pmf(np.arange(-i,i)+50).sum()
>>> print g(10) # approx 0.95
0.953955933071
```

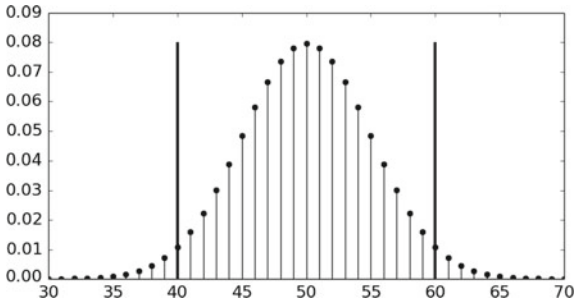

Fig. 3.5 Probability mass function for $\hat{p}$. The two *vertical lines* form the confidence interval

The two vertical lines in the plot show how far out from the mean we have to go to accumulate 95 % of the probability. Now, we can solve this as

$$50 + 50\epsilon = 60$$

which makes $\epsilon = 1/5$ or 20 %. So, flipping 100 times means I can only get within 20 % of the real p 95 % of the time in the worst case scenario (i.e., $p = 1/2$). The following code verifies the situation.

```
>>> from scipy.stats import bernoulli
>>> b=bernoulli(0.5) # coin distribution
>>> xs = b.rvs(100) # flip it 100 times
>>> phat = np.mean(xs) # estimated p
>>> print abs(phat-0.5) < 0.5*0.20 # make it w/in interval?
True
```

Let's keep doing this and see if we can get within this interval 95 % of the time.

```
>>> out=[]
>>> b=bernoulli(0.5) # coin distribution
>>> for i in range(500):   # number of tries
...     xs = b.rvs(100)    # flip it 100 times
...     phat = np.mean(xs) # estimated p
...     out.append(abs(phat-0.5) < 0.5*0.20 ) # within 20% ?
...
>>> # percentage of tries w/in 20\,% interval
>>> print 100*np.mean(out)
97.4
```

Well, that seems to work! Now we have a way to get at the quality of the estimator, $\hat{p}$.

Maximum Likelihood Estimator Without Calculus The prior example showed how we can use calculus to compute the maximum likelihood estimator. It's important to emphasize that the maximum likelihood principle does *not* depend on calculus and extends to more general situations where calculus is impossible. For example, let X be uniformly distributed in the interval $[0, \theta]$. Given n measurements of X, the likelihood function is the following:

$$L(\theta) = \prod_{i=1}^{n} \frac{1}{\theta} = \frac{1}{\theta^n}$$

where each $x_i \in [0, \theta]$. Note that the slope of this function is not zero anywhere so the usual calculus approach is not going to work here. Because the likelihood is the product of the individual uniform densities, if any of the x_i values were outside of the proposed $[0, \theta]$ interval, then the likelihood would go to zero, because the uniform density is zero outside of the $[0, \theta]$. Naturally, this is no good for maximization. Thus, observing that the likelihood function is strictly decreasing with increasing θ, we conclude that the value for θ that maximizes the likelihood is the maximum of the x_i values. To summarize, the maximum likelihood estimator is the following:

$$\theta_{ML} = \max_i x_i$$

As always, we want the distribution of this estimator to judge its performance. In this case, this is pretty straightforward. The cumulative density function for the max function is the following:

$$\mathbb{P}\left(\hat{\theta}_{ML} < v\right) = \mathbb{P}(x_0 \leq v \wedge x_1 \leq v \ldots \wedge x_n \leq v)$$

and since all the x_i are uniformly distributed in $[0, \theta]$, we have

$$\mathbb{P}\left(\hat{\theta}_{ML} < v\right) = \left(\frac{v}{\theta}\right)^n$$

So, the probability density function is then,

$$f_{\hat{\theta}_{ML}}(\theta_{ML}) = n\theta_{ML}^{n-1}\theta^{-n}$$

Then, we can compute the $\mathbb{E}(\theta_{ML}) = (\theta n)/(n+1)$ with corresponding variance as $\mathbb{V}(\theta_{ML}) = (\theta^2 n)/(n+1)^2/(n+2)$.

For a quick sanity check, we can write the following simulation for $\theta = 1$ as in the following:

```
>>> from scipy import stats
>>> rv = stats.uniform(0,1)  # define uniform random variable
>>> mle=rv.rvs((100,500)).max(0) # max along row-dimension
>>> print mean(mle) # approx n/(n+1) = 100/101 ~= 0.99
0.989942138048
>>> print var(mle) #approx n/(n+1)**2/(n+2) ~= 9.61E-5
9.95762009884e-05
```

Programming Tip

The `max(0)` suffix on for the `mle` computation takes the maximum of the so-computed array along the row (`axis=0`) dimension.

You can also plot `hist(mle)` to see the histogram of the simulated maximum likelihood estimates and match it up against the probability density function we derived above.

In this section, we explored the concept of maximum likelihood estimation using a coin flipping experiment both analytically and numerically with the scientific Python stack. We also explored the case when calculus is not workable for maximum likelihood estimation. There are two key points to remember. First, maximum likelihood estimation produces a function of the data that is itself a random variable, with its own probability distribution. We can get at the quality of the so-derived estimators by examining the confidence intervals around the estimated values using the probability distributions associated with the estimators themselves. Second, maximum likelihood estimation applies even in situations where using basic calculus is not applicable [2].

3.4.2 Delta Method

The Central Limit Theorem provides a way to get at the distribution of a random variable. However, sometimes we are more interested in a function of the random variable. In order to extend and generalize the central limit theorem in this way, we need the Taylor series expansion. Recall that the Taylor series expansion is an approximation of a function of the following form,

$$T_r(x) = \sum_{i=0}^{r} \frac{g^{(i)}(a)}{i!}(x-a)^i$$

this basically says that a function g can be adequately approximated about a point a using a polynomial based on its derivatives evaluated at a. Before we state the general theorem, let's examine an example to understand how the mechanics work.

Example Suppose that X is a random variable with $\mathbb{E}(X) = \mu \neq 0$. Furthermore, supposedly have a suitable function g and we want the distribution of $g(X)$. Applying the Taylor series expansion, we obtain the following,

$$g(X) \approx g(\mu) + g'(\mu)(X-\mu)$$

If we use $g(X)$ as an estimator for $g(\mu)$, then we can say that we approximately have the following

$$\begin{aligned}\mathbb{E}(g(X)) &= g(\mu)\\ \mathbb{V}(g(X)) &= (g'(\mu))^2\mathbb{V}(X)\end{aligned}$$

Concretely, suppose we want to estimate the odds, $\frac{p}{1-p}$. For example, if $p = 2/3$, then we say that the odds is `2:1` meaning that the odds of the one outcome are twice as likely as the odds of the other outcome. Thus, we have $g(p) = \frac{p}{1-p}$ and we want to find $\mathbb{V}(g(\hat{p}))$. In our coin-flipping problem, we have the estimator $\hat{p} = \frac{1}{n}\sum X_k$ from the Bernoulli-distributed data X_k individual coin-flips. Thus,

$$\begin{aligned}\mathbb{E}(\hat{p}) &= p\\ \mathbb{V}(\hat{p}) &= \frac{p(1-p)}{n}\end{aligned}$$

Now, $g'(p) = 1/(1-p)^2$, so we have,

$$\begin{aligned}\mathbb{V}(g(\hat{p})) &= (g'(p))^2\mathbb{V}(\hat{p}) \\ &= \left(\frac{1}{(1-p)^2}\right)^2 \frac{p(1-p)}{n} \\ &= \frac{p}{n(1-p)^3}\end{aligned}$$

which is an approximation of the variance of the estimator $g(\hat{p})$. Let's simulate this and see how it agrees.

```
>>> from scipy import stats
>>> # compute MLE estimates
>>> d=stats.bernoulli(0.1).rvs((10,5000)).mean(0)
>>> # avoid divide-by-zero
>>> d=d[np.logical_not(np.isclose(d,1))]
>>> # compute odds ratio
>>> odds = d/(1-d)
>>> print 'odds ratio=',np.mean(odds),'var=',np.var(odds)
odds ratio= 0.122892063492 var= 0.0179795009221
```

The first number above is the mean of the simulated odds ratio and the second is the variance of the estimate. According to the variance estimate above, we have $\mathbb{V}(g(1/10)) \approx 0.0137$, which is not too bad for this approximation. Recall we want to estimate the odds from the $\hat{p}$. The code above takes 5000 estimates of the $\hat{p}$ to estimate $\mathbb{V}(g)$. The odds ratio for $p = 1/10$ is $1/9 \approx 0.111$.

Programming Tip

The code above uses the `np.isclose` function to identify the ones from the simulation and the `np.logical_not` removes these elements from the data because the odds ratio has a zero in the denominator for these values.

Let's try this again with a probability of heads of `0.5` instead of `0.3`.

```
>>> from scipy import stats
>>> d=stats.bernoulli(.5).rvs((10,5000)).mean(0)
>>> d=d[np.logical_not(np.isclose(d,1))]
>>> print 'odds ratio=',np.mean(d),'var=',np.var(d)
odds ratio= 0.499379627777 var= 0.0245123227629
```

The odds ratio is this case is equal to one, which is not close to what was reported. According to our approximation, we have $\mathbb{V}(g) = 0.4$, which does not look like what our simulation just reported. This is because the approximation is best when the odds ratio is nearly linear and worse otherwise (Fig. 3.6).

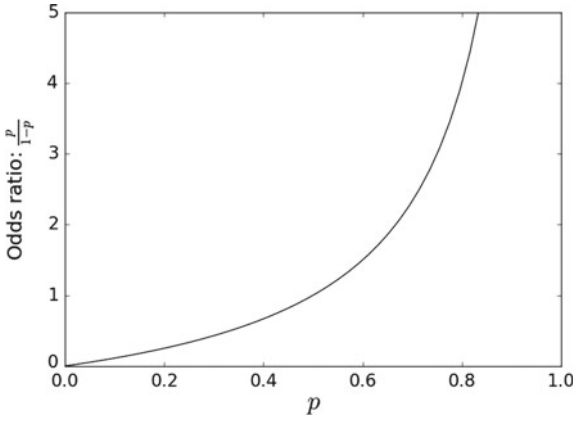

Fig. 3.6 The odds ratio is close to linear for small values but becomes unbounded as p approaches one. The delta method is more effective for small underlying values of p, where the linear approximation is better

3.5 Hypothesis Testing and P-Values

It is sometimes very difficult to unequivocally attribute outcomes to causal factors. For example, did your experiment generate the outcome you were hoping for or not? Maybe something did happen, but the effect is not pronounced enough to separate it from inescapable measurement errors or other factors in the ambient environment? Hypothesis testing is a powerful statistical method to address these questions. Let's begin by again considering our coin-tossing experiment with unknown parameter p. Recall that the individual coin-flips are Bernoulli distributed. The first step is to establish separate hypotheses. First, H_0 is the so-called null hypothesis. In our case this can be

$$H_0 : \theta < \frac{1}{2}$$

and the alternative hypothesis is then

$$H_1 : \theta \geq \frac{1}{2}$$

With this set up, the question now boils down to figuring out which hypothesis the data is most consistent with. To choose between these, we need a statistical test that is a function, G, of the sample set $\mathbf{X}_n = \{X_i\}_n$ into the real line, where X_i is the heads or tails outcome ($X_i \in \{0, 1\}$). In other words, we compute $G(\mathbf{X}_n)$ and check if it exceeds a threshold c. If not, then we declare H_0 (otherwise, declare H_1). Notationally, this is the following:

$$G(\mathbf{X}_n) < c \Rightarrow H_0$$
$$G(\mathbf{X}_n) \geq c \Rightarrow H_1$$

Table 3.1 Truth table for hypotheses testing

	Declare H_0	Declare H_1
H_0 True	Correct	False positive (Type I error)
H_1 True	False negative (Type II error)	Correct (true-detect)

In summary, we have the observed data $\mathbf{X}_n$ and a function G that maps that data onto the real line. Then, using the constant c as a threshold, the inequality effectively divides the real line into two parts, one corresponding to each of the hypotheses.

Whatever this test G is, it will make mistakes of two types—false negatives and false positives. The false positives arise from the case where we declare H_0 when the test says we should declare H_1. This is summarized in the Table 3.1.
For this example, here are the false positives (aka false alarms):

$$P_{FA} = \mathbb{P}\left(G(\mathbf{X}_n) > c \mid \theta \leq \frac{1}{2}\right)$$

Or, equivalently,

$$P_{FA} = \mathbb{P}\left(G(\mathbf{X}_n) > c \mid H_0\right)$$

Likewise, the other error is a false negative, which we can write analogously as

$$P_{FN} = \mathbb{P}\left(G(\mathbf{X}_n) < c | H_1\right)$$

By choosing some acceptable values for either of these errors, we can solve for the other one. The practice is usually to pick a value of P_{FA} and then find the corresponding value of P_{FN}. Note that it is traditional in engineering to speak about *detection probability*, which is defined as

$$P_D = 1 - P_{FN} = \mathbb{P}\left(G(\mathbf{X}_n) > c \mid H_1\right)$$

In other words, this is the probability of declaring H_1 when the test exceeds the threshold. This is otherwise known as the *probability of a true detection* or *true-detect*.

3.5.1 Back to the Coin Flipping Example

In our previous maximum likelihood discussion, we wanted to derive an estimator for the *value* of the probability of heads for the coin flipping experiment. For hypthesis testing, we want to ask a softer question: is the probability of heads greater or less than $1/2$? As we just established, this leads to the two hypotheses:

$$H_0 : \theta < \frac{1}{2}$$

versus,

$$H_1 : \theta > \frac{1}{2}$$

Let's assume we have five observations. Now we need the G function and a threshold c to help pick between the two hypotheses. Let's count the number of heads observed in five observations as our criterion. Thus, we have

$$G(\mathbf{X}_5) := \sum_{i=1}^{5} X_i$$

and, suppose further that we pick H_1 only if exactly five out of five observations are heads. We'll call this the *all-heads* test.

Now, because all of the X_i are random variables, so is G and we must find the corresponding probability mass function for G. Assuming the individual coin tosses are independent, the probability of five heads is θ^5. This means that the probability of rejecting the H_0 hypothesis (and choosing H_1, because there are only two choices here) based on the unknown underlying probability is θ^5. In the parlance, this is known and the *power function* as in denoted by β as in

$$\beta(\theta) = \theta^5$$

Let's get a quick plot this in Fig. 3.7.

Now, we have the following false alarm probability,

$$P_{FA} = \mathbb{P}(G(\mathbf{X}_n) = 5 | H_0) = \mathbb{P}(\theta^5 | H_0)$$

Notice that this is a function of θ, which means there are many false alarm probability values that correspond to this test. To be on the conservative side, we'll pick the supremum (i.e., maximum) of this function, which is known as the *size* of the test, traditionally denoted by α,

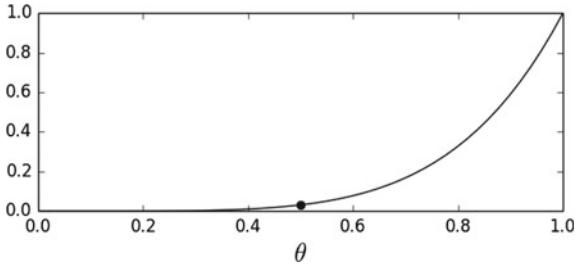

Fig. 3.7 Power function for the all-heads test. The *dark circle* indicates the value of the function indicating α

$$\alpha = \sup_{\theta \in \Theta_0} \beta(\theta)$$

with domain $\Theta_0 = \{\theta < 1/2\}$ which in our case is

$$\alpha = \sup_{\theta < \frac{1}{2}} \theta^5 = \left(\frac{1}{2}\right)^5 = 0.03125$$

Likewise, for the detection probability,

$$\mathbb{P}_D(\theta) = \mathbb{P}(\theta^5 | H_1)$$

which is again a function of the parameter θ. The problem with this test is that the P_D is pretty low for most of the domain of θ. For instance, values in the nineties for P_D only happen when $\theta > 0.98$. In other words, if the coin produces heads 98 times out of 100, then we can detect H_1 reliably. Ideally, we want a test that is zero for the domain corresponding to H_0 (i.e., Θ_0) and equal to one otherwise. Unfortunately, even if we increase the length of the observed sequence, we cannot escape this effect with this test. You can try plotting θ^n for larger and larger values of n to see this.

Majority Vote Test. Due to the problems with the detection probability in the all-heads test, maybe we can think of another test that will have the performance we want? Suppose we reject H_0 if the majority of the observations are heads. Then, using the same reasoning as above, we have

$$\beta(\theta) = \sum_{k=3}^{5} \binom{5}{k} \theta^k (1-\theta)^{5-k}$$

Figure 3.8 shows the power function for both the majority vote and the all-heads tests.

In this case, the new test has *size*

$$\alpha = \sup_{\theta < \frac{1}{2}} \theta^5 + 5\theta^4 (-\theta + 1) + 10\theta^3 (-\theta + 1)^2 = \frac{1}{2}$$

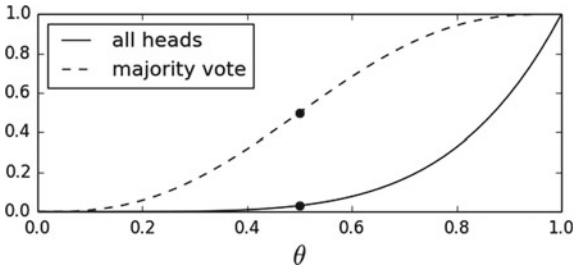

Fig. 3.8 Compares the power function for the all-heads test with that of the majority-vote test

As before we only get to upwards of 90 % for detection probability only when the underlying parameter $\theta > 0.75$. Let's see what happens when we consider more than five samples. For example, let's suppose that we have $n = 100$ samples and we want to vary the threshold for the majority vote test. For example, let's have a new test where we declare H_1 when $k = 60$ out of the 100 trials turns out to be heads. What is the β function in this case?

$$\beta(\theta) = \sum_{k=60}^{100} \binom{100}{k} \theta^k (1-\theta)^{100-k}$$

This is too complicated to write by hand, but the statistics module in Sympy has all the tools we need to compute this.

```
>>> from sympy.stats import P, Binomial
>>> theta = S.symbols('theta',real=True)
>>> X = Binomial('x',100,theta)
>>> beta_function = P(X>60)
>>> print beta_function.subs(theta,0.5) # alpha
0.0176001001088524
>>> print beta_function.subs(theta,0.70)
0.979011423996075
```

These results are much better than before because the β function is much steeper. If we declare H_1 when we observe 60 out of 100 trials are heads, then we wrongly declare heads approximately 1.8 % of the time. Otherwise, if it happens that the true value for $p > 0.7$, we will conclude correctly approximately 97 % of the time. A quick simulation can sanity check these results as shown below:

```
>>> from scipy import stats
>>> rv=stats.bernoulli(0.5) # true p = 0.5
>>> # number of false alarms ~ 0.018
>>> print sum(rv.rvs((1000,100)).sum(axis=1)>60)/1000.
0.025
```

The above code is pretty dense so let's unpack it. In the first line, we use the `scipy.stats` module to define the Bernoulli random variable for the coin flip. Then, we use the `rvs` method of the variable to generate 1000 trials of the experiment where each trial consists of 100 coin flips. This generates a 1000×100 matrix where the rows are the individual trials and the columns are the outcomes of each respective set of 100 coin flips. The `sum(axis=1)` part computes the sum across the columns. Because the values of the embedded matrix are only `1` or `0` this gives us the count of flips that are heads per row. The next `>60` part computes the boolean 1000-long vector of values that are bigger than 60. The final `sum` adds these up. Again, because the entries in the array are `True` or `False` the `sum` computes the count of times the number of heads has exceeded 60 per 100 coin flips in each of 1000 trials. Then, dividing this number by 1000 gives a quick approximation of false alarm probability we computed above for this case where the true value of $p = 0.5$.

3.5.2 Receiver Operating Characteristic

Because the majority vote test is a binary test, we can compute the *Receiver Operating Characteristic* (ROC) which is the graph of the (P_{FA}, P_D). The term comes from radar systems but is a very general method for consolidating all of these issues into a single graph. Let's consider a typical signal processing example with two hypotheses. In H_0, there is noise but no signal present at the receiver,

$$H_0 : X = \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ represents additive noise. In the alternative hypothesis, there is a deterministic signal at the receiver,

$$H_1 : X = \mu + \epsilon$$

Again, the problem is to choose between these two hypotheses. For H_0, we have $X \sim \mathcal{N}(0, \sigma^2)$ and for H_1, we have $X \sim \mathcal{N}(\mu, \sigma^2)$. Recall that we only observe values for x and must pick either H_0 or H_1 from these observations. Thus, we need a threshold, c, to compare x against in order to distinguish the two hypotheses. Figure 3.9 shows the probability density functions under each of the hypotheses. The dark vertical line is the threshold c. The gray shaded area is the probability of detection, P_D and the shaded area is the probability of false alarm, P_{FA}. The test evaluates every observation of x and concludes H_0 if $x < c$ and H_1 otherwise.

Programming Tip

The shading shown in Fig. 3.9 comes from Matplotlib's `fill_between` function. This function has a `where` keyword argument to specify which part of the plot to apply shading with specified `color` keyword argument. Note there is also a `fill_betweenx` function that fills horizontally. The `text` function can place formatted text anywhere in the plot and can utilize basic LaTeX formatting. See the IPython notebook corresponding to this section for the source code.

As we slide the threshold left and right along the horizontal axis, we naturally change the corresponding areas under each of the curves shown in Fig. 3.9 and thereby change the values of P_D and P_{FA}. The contour that emerges from sweeping the threshold this way is the ROC as shown in Fig. 3.10. This figure also shows the diagonal line which corresponds to making decisions based on the flip of a fair coin. Any meaningful test must do better than coin flipping so the more the ROC bows up to the top left corner of the graph, the better. Sometimes ROCs are quantified into a single number called the *area under the curve* (AUC), which varies from 0.5 to 1.0 as shown. In our example, what separates the two probability density functions is the value of μ. In a real situation, this would be determined by signal processing

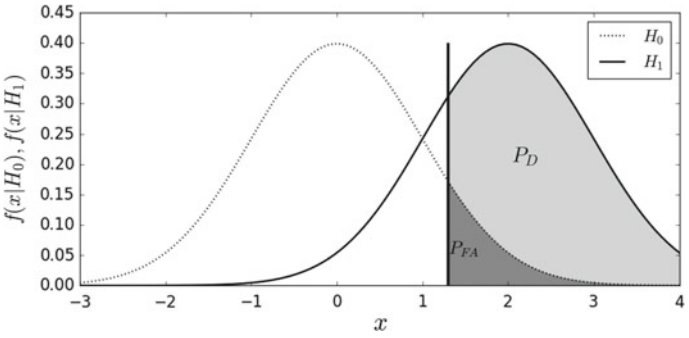

Fig. 3.9 The two density functions for the H_0 and H_1 hypotheses. The *shaded gray* area is the detection probability and the *shaded blue* area is the probability of false alarm. The *vertical line* is the decision threshold

methods that include many complicated trade-offs. The key idea is that whatever those trade-offs are, the test itself boils down to the separation between these two density functions—good tests separate the two density functions and bad tests do not. Indeed, when there is no separation, we arrive at the diagonal-line coin-flipping situation we just discussed.

What values for P_D and P_{FA} are considered *acceptable* depends on the application. For example, suppose you are testing for a fatal disease. It could be that you are willing to except a relatively high P_{FA} value if that corresponds to a good P_D because the test is relatively cheap to administer compared to the alternative of missing a detection. On the other hand, may be a false alarm triggers an expensive response, so

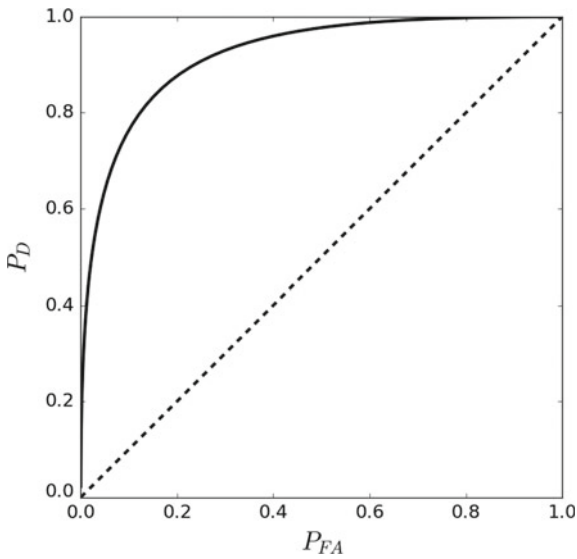

Fig. 3.10 The Receiver Operating Characteristic (ROC) corresponding to Fig. 3.9

that minimizing these alarms is more important than potentially missing a detection. These trade-offs can only be determined by the application and design factors.

3.5.3 P-Values

There are a lot of moving parts in hypothesis testing. What we need is a way to consolidate the findings. The idea is that we want to find the minimum level at which the test rejects H_0. Thus, the p-value is the probability, under H_0, that the test-statistic is at least as extreme as what was actually observed. Informally, this means that smaller values imply that H_0 should be rejected, although this doesn't mean that large values imply that H_0 should be retained. This is because a large p-value can arise from either H_0 being true or the test having low statistical power.

If H_0 is true, the p-value is uniformly distributed in the interval (0, 1). If H_1 is true, the distribution of the p-value will concentrate closer to zero. For continuous distributions, this can be proven rigorously and implies that if we reject H_0 when the corresponding p-value is less than α, then the probability of a false alarm is α. Perhaps it helps to formalize this a bit before computing it. Suppose $\tau(X)$ is a test statistic that rejects H_0 as it gets bigger. Then, for each sample x, corresponding to the data we actually have on-hand, we define

$$p(x) = \sup_{\theta \in \Theta_0} \mathbb{P}_\theta(\tau(X) > \tau(x))$$

This equation states that the supremum (i.e., maximum) probability that the test statistic, $\tau(X)$, exceeds the value for the test statistic on this particular data ($\tau(x)$) over the domain Θ_0 is defined as the p-value. Thus, this embodies a worst-case scenario over all values of θ.

Here's one way to think about this. Suppose you rejected H_0, and someone says that you just got *lucky* and somehow just drew data that happened to correspond to a rejection of H_0. What p-values provide is a way to address this by capturing the odds of just a favorable data-draw. Thus, suppose that your p-value is 0.05. Then, what you are showing is that the odds of just drawing that data sample, given H_0 is in force, is just 5 %. This means that there's a 5 % chance that you somehow lucked out and got a favorable draw of data.

Let's make this concrete with an example. Given, the majority-vote rule above, suppose we actually do observe three of five heads. Given the H_0, the probability of observing this event is the following:

$$p(x) = \sup_{\theta \in \Theta_0} \sum_{k=3}^{5} \binom{5}{k} \theta^k (1-\theta)^{5-k} = \frac{1}{2}$$

For the all-heads test, the corresponding computation is the following:

$$p(x) = \sup_{\theta \in \Theta_0} \theta^5 = \frac{1}{2^5} = 0.03125$$

From just looking at these p-values, you might get the feeling that the second test is better, but we still have the same detection probability issues we discussed above; so, p-values help in summarizing some aspects of our hypothesis testing, but they do *not* summarize all the salient aspects of the *entire* situation.

3.5.4 Test Statistics

As we have seen, it is difficult to derive good test statistics for hypothesis testing without a systematic process. The Neyman-Pearson Test is derived from fixing a false-alarm value (α) and then maximizing the detection probability. This results in the Neyman-Pearson Test,

$$L(\mathbf{x}) = \frac{f_{X|H_1}(\mathbf{x})}{f_{X|H_0}(\mathbf{x})} \underset{H_0}{\overset{H_1}{\gtrless}} \gamma$$

where L is the likelihood ratio and where the threshold γ is chosen such that

$$\int_{x:L(\mathbf{x})>\gamma} f_{X|H_0}(\mathbf{x}) d\mathbf{x} = \alpha$$

The Neyman-Pearson Test is one of a family of tests that use the likelihood ratio.

Example Suppose we have a receiver and we want to distinguish whether just noise (H_0) or signal pluse noise (H_1) is received. For the noise-only case, we have $x \sim \mathcal{N}(0, 1)$ and for the signal pluse noise case we have $x \sim \mathcal{N}(1, 1)$. In other words, the mean of the distribution shifts in the presence of the signal. This is a very common problem in signal processing and communications. The Neyman-Pearson Test then boils down to the following,

$$L(x) = e^{-\frac{1}{2}+x} \underset{H_0}{\overset{H_1}{\gtrless}} \gamma$$

Now we have to find the threshold γ that solves the maximization problem that characterizes the Neyman-Pearson Test. Taking the natural logarithm and re-arranging gives,

$$x \underset{H_0}{\overset{H_1}{\gtrless}} \frac{1}{2} + \log \gamma$$

The next step is find γ corresponding to the desired α by computing it from the following,

$$\int_{1/2+\log\gamma}^{\infty} f_{X|H_0}(x)dx = \alpha$$

For example, taking $\alpha = 1/100$, gives $\gamma \approx 6.21$. To summarize the test in this case, we have,

$$x \underset{H_0}{\overset{H_1}{\gtrless}} 2.32$$

Thus, if we measure X and see that its value exceeds the threshold above, we declare H_1 and otherwise declare H_0. The following code shows how to solve this example using Sympy and Scipy. First, we set up the likelihood ratio,

```
>>> import sympy as S
>>> from sympy import stats
>>> s = stats.Normal('s',1,1) # signal+noise
>>> n = stats.Normal('n',0,1) # noise
>>> x = S.symbols('x',real=True)
>>> L = stats.density(s)(x)/stats.density(n)(x)
```

Next, to find the γ value,

```
>>> g = S.symbols('g',positive=True) # define gamma
>>> v=S.integrate(stats.density(n)(x),
...               (x,S.Rational(1,2)+S.log(g),S.oo))
```

Programming Tip

Providing additional information regarding the Sympy variable by using the keyword argument `positive=True` helps the internal simplification algorithms work faster and better. This is especially useful when dealing with complicated integrals that involve special functions. Furthermore, note that we used the `Rational` function to define the `1/2` fraction, which is another way of providing hints to Sympy. Otherwise, it's possible that the floating-point representation of the fraction could disguise the simple fraction and thereby miss internal simplification opportunities.

We want to solve for `g` in the above expression. Sympy has some built-in numerical solvers as in the following,

```
>>> print S.nsolve(v-0.01,3.0) # approx 6.21
6.21116124253284
```

Note that in this situation it is better to use the numerical solvers because Sympy `solve` may grind along for a long time to resolve this.

Generalized Likelihood Ratio Test. The likelihood ratio test can be generalized using the following statistic,

$$\Lambda(\mathbf{x}) = \frac{\sup_{\theta\in\Theta_0} L(\theta)}{\sup_{\theta\in\Theta} L(\theta)} = \frac{L(\hat{\theta}_0)}{L(\hat{\theta})}$$

where $\hat{\theta}_0$ maximizes $L(\theta)$ subject to $\theta \in \Theta_0$ and $\hat{\theta}$ is the maximum likelihood estimator. The intuition behind this generalization of the Likelihood Ratio Test is that the denomimator is the usual maximum likelihood estimator and the numerator is the maximum likelihood estimator, but over a restricted domain (Θ_0). This means that the ratio is always less than unity because the maximum likelihood estimator over the entire space will always be at least as maximal as that over the more restricted space. When this Λ ratio gets small enough, it means that the maximum likelihood estimator over the entire domain (Θ) is larger which means that it is safe to reject the null hypothesis H_0. The tricky part is that the statistical distribution of Λ is usually eye-wateringly difficult. Fortunately, Wilks Theorem says that with sufficiently large n, the distribution of $-2\log\Lambda$ is approximately chi-square with $r - r_0$ degrees of freedom, where r is the number of free parameters for Θ and r_0 is the number of free parameters in Θ_0. With this result, if we want an approximate test at level α, we can reject H_0 when $-2\log\Lambda \geq \chi^2_{r-r_0}(\alpha)$ where $\chi^2_{r-r_0}(\alpha)$ denotes the $1-\alpha$ quantile of the $\chi^2_{r-r_0}$ chi-square distribution. However, the problem with this result is that there is no definite way of knowing how big n should be. The advantage of this generalized likelihood ratio test is that it can test multiple hypotheses simultaneously, as illustrated in the following example.

Example Let's return to our coin-flipping example, except now we have three different coins. The likelihood function is then,

$$L(p_1, p_2, p_3) = \texttt{binom}(k_1; n_1, p_1)\texttt{binom}(k_2; n_2, p_2)\texttt{binom}(k_3; n_3, p_3)$$

where `binom` is the binomial distribution with the given parameters. For example,

$$\texttt{binom}(k; n, p) = \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k}$$

The null hypothesis is that all three coins have the same probability of heads, $H_0 : p = p_1 = p_2 = p_3$. The alternative hypothesis is that at least one of these probabilites is different. Let's consider the numerator of the Λ first, which will give us the maximum likelihood estimator of p. Because the null hypothesis is that all the p values are equal, we can just treat this as one big binomial distribution with $n = n_1 + n_2 + n_3$ and $k = k_1 + k_2 + k_3$ is the total number of heads observed for any coin. Thus, under the null hypothesis, the distribution of k is binomial with parameters n and p. Now,

what is the maximum likelihood estimator for this distribution? We have worked this problem before and have the following,

$$\hat{p}_0 = \frac{k}{n}$$

In other words, the maximum likelihood estimator under the null hypothesis is the proportion of ones observed in the sequence of n trials total. Now, we have to substitute this in for the likelihood under the null hypothesis to finish the numerator of Λ,

$$L(\hat{p}_0, \hat{p}_0, \hat{p}_0) = \texttt{binom}(k_1; n_1, \hat{p}_0)\texttt{binom}(k_2; n_2, \hat{p}_0)\texttt{binom}(k_3; n_3, \hat{p}_0)$$

For the denomimator of Λ, which represents the case of maximizing over the entire space, the maximum likelihood estimator for each separate binomial distribution is likewise,

$$\hat{p}_i = \frac{k_i}{n_i}$$

which makes the likelihood in the denominator the following,

$$L(\hat{p}_1, \hat{p}_2, \hat{p}_3) = \texttt{binom}(k_1; n_1, \hat{p}_1)\texttt{binom}(k_2; n_2, \hat{p}_2)\texttt{binom}(k_3; n_3, \hat{p}_3)$$

for each of the $i \in \{1, 2, 3\}$ binomial distributions. Then, the Λ statistic is then the following,

$$\Lambda(k_1, k_2, k_3) = \frac{L(\hat{p}_0, \hat{p}_0, \hat{p}_0)}{L(\hat{p}_1, \hat{p}_2, \hat{p}_3)}$$

Wilks theorems states that $-2\log\Lambda$ is chi-square distributed. We can compute this example with the statistics tools in Sympy and Scipy.

```
>>> from scipy.stats import binom, chi2
>>> import numpy as np
>>> # some sample parameters
>>> p0,p1,p2 = 0.3,0.4,0.5
>>> n0,n1,n2 = 50,180,200
>>> brvs= [ binom(i,j) for i,j in zip((n0,n1,n2),(p0,p1,p2))]
>>> def gen_sample(n=1):
...     'generate samples from separate binomial distributions'
...     if n==1:
...         return [i.rvs() for i in brvs]
...     else:
...         return [gen_sample() for k in range(n)]
...
```

Programming Tip

Note the recursion in the definition of the `gen_sample` function where a conditional clause of the function calls itself. This is a quick way to reusing code and generating vectorized output. Using `np.vectorize` is another way, but the code is simple enough in this case to use the conditional clause. In Python, it is generally bad for performance to have code with nested recursion because of how the stack frames are managed. However, here we are only recursing once so this is not an issue.

Next, we compute the logarithm of the numerator of the Λ statistic,

```
>>> k0,k1,k2 = gen_sample()
>>> print k0,k1,k2
12 68 103
>>> pH0 = sum((k0,k1,k2))/sum((n0,n1,n2))
>>> numer = np.sum([np.log(binom(ni,pH0).pmf(ki))
...                     for ni,ki in
...                         zip((n0,n1,n2),(k0,k1,k2))])
>>> print numer
-15.5458638366
```

Note that we used the null hypothesis estimate for the $\hat{p}_0$. Likewise, for the logarithm of the denominator we have the following,

```
>>> denom = np.sum([np.log(binom(ni,pi).pmf(ki))
...                     for ni,ki,pi in
...                        zip((n0,n1,n2),(k0,k1,k2),(p0,p1,p2))])
>>> print denom
-8.42410648079
```

Now, we can compute the logarithm of the Λ statistic as follows and see what the corresponding value is according to Wilks theorem,

```
>>> chsq=chi2(2)
>>> logLambda =-2*(numer-denom)
>>> print logLambda
14.2435147116
>>> print 1- chsq.cdf(logLambda)
0.000807346708329
```

Because the value reported above is less than the 5 % significance level, we reject the null hypothesis that all the coins have the same probability of heads. Note that there are two degrees of freedom because the difference in the number of parameters between the null hypothesis (p) and the alternative (p_1, p_2, p_3) is two. We can build a quick Monte Carlo simulation to check the probability of detection for this example using the following code, which is just a combination of the last few code blocks,

```
>>> c= chsq.isf(.05) # 5% significance level
>>> out = []
>>> for k0,k1,k2 in gen_sample(100):
...     pH0 = sum((k0,k1,k2))/sum((n0,n1,n2))
...     numer = np.sum([np.log(binom(ni,pH0).pmf(ki))
...                         for ni,ki in
...                             zip((n0,n1,n2),(k0,k1,k2))])
```

```
...     denom = np.sum([np.log(binom(ni,pi).pmf(ki))
...                         for ni,ki,pi in
...                             zip((n0,n1,n2),(k0,k1,k2),(p0,p1,p2))])
...     out.append(-2*(numer-denom)>c)
...
>>> print np.mean(out) # estimated probability of detection
0.59
```

The above simulation shows the estimated probability of detection, for this set of example parameters. This relative low probability of detection means that while the test is unlikely (i.e., at the 5 % significance level) to mistakenly pick the null hypothesis, it is likewise missing many of the H_1 cases (i.e., low probability of detection). The trade-off between which is more important is up to the particular context of the problem. In some situations, we may prefer additional false alarms in exchange for missing fewer H_1 cases.

Permutation Test. The Permutation Test is good way to test whether or not samples samples come from the same distribution. For example, suppose that

$$X_1, X_2, \ldots, X_m \sim F$$

and also,

$$Y_1, Y_2, \ldots, Y_n \sim G$$

That is, Y_i and X_i come from different distributions. Suppose we have some test statistic, for example

$$T(X_1, \ldots, X_m, Y_1, \ldots, Y_n) = |\overline{X} - \overline{Y}|$$

Under the null hypothesis for which $F = G$, any of the $(n+m)!$ permutations are equally likely. Thus, suppose for each of the $(n+m)!$ permutations, we have the computed statistic,

$$\{T_1, T_2, \ldots, T_{(n+m)!}\}$$

Then, under the null hypothesis, each of these values is equally likely. The distribution of T under the null hypothesis is the *permutation distribution* that puts weight $1/(n+m)!$ on each T-value. Suppose t_o is the observed value of the test statistic and assume that large T rejects the null hypothesis, then the p-value for the permutation test is the following,

$$P(T > t_o) = \frac{1}{(n+m)!} \sum_{j=1}^{(n+m)!} I(T_j > t_o)$$

where $I()$ is the indicator function. For large $(n+m)!$, we can sample randomly from the set of all permutations to estimate this p-value.

Example Let's return to our coin-flipping example from last time, but now we have only two coins. The hypothesis is that both coins have the same probability of heads. We can use the built-in function in Numpy to compute the random permutations.

```
>>> x=binom(10,0.3).rvs(5) # p=0.3
>>> y=binom(10,0.5).rvs(3) # p=0.5
>>> z = np.hstack([x,y]) # combine into one array
>>> t_o = abs(x.mean()-y.mean())
>>> out = [] # output container
>>> for k in range(1000):
...     perm = np.random.permutation(z)
...     T=abs(perm[:len(x)].mean()-perm[len(x):].mean())
...     out.append((T>t_o))
...
>>> print 'p-value = ', np.mean(out)
p-value =  0.0
```

Note that the size of total permutation space is $8! = 40320$ so we are taking relatively few (i.e., 100) random permutations from this space.

Wald Test. The Wald Test is an asympotic test. Suppose we have $H_0 : \theta = \theta_0$ and otherwise $H_1 : \theta \neq \theta_0$, the corresponding statistic is defined as the following,

$$W = \frac{\hat{\theta}_n - \theta_0}{\mathtt{se}}$$

where $\hat{\theta}$ is the maximum likelihood estimator and $\mathtt{se}$ is the standard error,

$$\mathtt{se} = \sqrt{\mathbb{V}(\hat{\theta}_n)}$$

Under general conditions, $W \xrightarrow{d} \mathcal{N}(0, 1)$. Thus, an asympotic test at level α rejects when $|W| > z_{\alpha/2}$ where $z_{\alpha/2}$ corresponds to $\mathbb{P}(|Z| > z_{\alpha/2}) = \alpha$ with $Z \sim \mathcal{N}(0, 1)$. For our favorite coin-flipping example, if $H_0 : \theta = \theta_0$, then

$$W = \frac{\hat{\theta} - \theta_0}{\sqrt{\hat{\theta}(1 - \hat{\theta})/n}}$$

We can simulate this using the following code at the usual 5 % significance level,

```
>>> from scipy import stats
>>> theta0 = 0.5 # H0
>>> k=np.random.binomial(1000,0.3)
>>> theta_hat = k/1000. # MLE
>>> W = (theta_hat-theta0)/np.sqrt(theta_hat*(1-theta_hat)/1000)
>>> c = stats.norm().isf(0.05/2) # z_{alpha/2}
>>> print abs(W)>c # if true, reject H0
True
```

This rejects H_0 because the true $\theta = 0.3$ and the null hypothesis is that $\theta = 0.5$. Note that $n = 1000$ in this case which puts us well inside the asympotic range of the result. We can re-do this example to estimate the detection probability for this example as in the following code,

```
>>> theta0 = 0.5 # H0
>>> c = stats.norm().isf(0.05/2.) # z_{alpha/2}
>>> out = []
>>> for i in range(100):
...     k=np.random.binomial(1000,0.3)
...     theta_hat = k/1000. # MLE
...     W = (theta_hat-theta0)/np.sqrt(theta_hat*(1-theta_hat)/1000.)
...     out.append(abs(W)>c) # if true, reject H0
...
>>> print np.mean(out) # detection probability
1.0
```

3.5.5 *Testing Multiple Hypotheses*

Thus far, we have focused primarily on two competing hypotheses. Now, we consider multiple comparisons. The general situation is the following. We test the null hypothesis against a sequence of n competing hypotheses H_k. We obtain p-values for each hypothesis so now we have multiple p-values to consider $\{p_k\}$. To boil this sequence down to a single criterion, we can make the following argument. Given n independent hypotheses that are all untrue, the probability of getting at least one false alarm is the following,

$$P_{FA} = 1 - (1 - p_0)^n$$

where p_0 is the individual p-value threshold (say, 0.05). The problem here is that $P_{FA} \to 1$ as $n \to \infty$. If we want to make many comparisons at once and control the overall false alarm rate the overall p-value should be computed under the assumption that none of the competing hypotheses is valid. The most common way to address this is with the Bonferroni correction which says that the individual significance level should be reduced to p/n. Obviously, this makes it much harder to declare significance for any particular hypothesis. The natural consequence of this conservative restriction is to reduce the statistical power of the experiment, thus making it more likely the true effects will be missed.

In 1995, Benjamini and Hochberg devised a simple method that tells which p-values are statistically significant. The procedure is to sort the list of p-values in ascending order, choose a false-discovery rate (say, q), and then find the largest p-value in the sorted list such that $p_k \leq kq/n$, where k is the p-value's position in the sorted list. Finally, declare that p_k value and all the others less than it statistically significant. This procedure guarantees that the proportion of false-positives is less than q (on average). The Benjamini-Hochberg procedure (and its derivatives) is fast and effective and is widely used for testing hundreds of primarily false hypotheses when studying genetics or diseases. Additionally, this procedure provides better statistical power than the Bonferroni correction.

In this section, we discussed the structure of statistical hypothesis testing and defined the various terms that are commonly used for this process, along with the illustrations of what they mean in our running coin-flipping example. From an

engineering standpoint, hypothesis testing is not as common as confidence-intervals and point estimates. On the other hand, hypothesis testing is very common in social and medical science, where one must deal with practical constraints that may limit the sample size or other aspects of the hypothesis testing rubric. In engineering, we can usually have much more control over the samples and models we employ because they are typically inanimate objects that can be measured repeatedly and consistently. This is obviously not so with human studies, which generally have other ethical and legal considerations.

3.6 Confidence Intervals

In a previous coin-flipping discussion, we discussed estimation of the underlying probability of getting a heads. There, we derived the estimator as

$$\hat{p}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

where $X_i \in \{0, 1\}$. Confidence intervals allow us to estimate how close we can get to the true value that we are estimating. Logically, that seems strange, doesn't it? We really don't know the exact value of what we are estimating (otherwise, why estimate it?), and yet, somehow we know how close we can get to something we admit we don't know? Ultimately, we want to make statements like the *probability of the value in a certain interval is 90%*. Unfortunately, that is something we will not be able to say using our methods. Note that Bayesian estimation gets closer to this statement by using *credible intervals*, but that is a story for another day. In our situation, the best we can do is say roughly the following: *if we ran the experiment multiple times, then the confidence interval would trap the true parameter 90% of the time*.

Let's return to our coin-flipping example and see this in action. One way to get at a confidence interval is to use Hoeffding's inequality from Sect. 2.9.3 specialized to our Bernoulli variables as

$$\mathbb{P}(|\ \hat{p}_n - p\ | > \epsilon) \leq 2\exp(-2n\epsilon^2)$$

Now, we can form the interval $\mathbb{I} = [\hat{p}_n - \epsilon_n, \hat{p}_n + \epsilon_n]$, where ϵ_n is carefully constructed as

$$\epsilon_n = \sqrt{\frac{1}{2n}\log\frac{2}{\alpha}}$$

which makes the right-side of the Hoeffding inequality equal to α. Thus, we finally have

$$\mathbb{P}(p \notin \mathbb{I}) = \mathbb{P}\left(|\ \hat{p}_n - p\ | > \epsilon_n\right) \leq \alpha$$

Thus, $\mathbb{P}(p \in \mathbb{I}) \geq 1 - \alpha$. As a numerical example, let's take $n = 100$, $\alpha = 0.05$, then plugging into everything we have gives $\epsilon_n = 0.136$. So, the 95 % confidence interval here is therefore

$$\mathbb{I} = [\hat{p}_n - \epsilon_n, \hat{p}_n + \epsilon_n] = [\hat{p}_n - 0.136, \hat{p}_n + 0.136]$$

The following code sample is a simulation to see if we can really trap the underlying parameter in our confidence interval.

```
>>> from scipy import stats
>>> import numpy as np

>>> b= stats.bernoulli(.5) # fair coin distribution
>>> nsamples = 100
>>> # flip it nsamples times for 200 estimates
>>> xs = b.rvs(nsamples*200).reshape(nsamples,-1)
>>> phat = np.mean(xs,axis=0) # estimated p
>>> # edge of 95\,% confidence interval
>>> epsilon_n=np.sqrt(np.log(2/0.05)/2/nsamples)
>>> pct=np.logical_and(phat-epsilon_n<=0.5,
...                    0.5 <= (epsilon_n +phat)
...                    ).mean()*100
>>> print 'Interval trapped correct value ', pct,'% of the time'
Interval trapped correct value  99.5 % of the time
```

The result shows that the estimator and the corresponding interval was able to trap the true value at least 95 % of the time. This is how to interpret the action of confidence intervals.

However, the usual practice is to not use Hoeffding's inequality and instead use arguments around asymptotic normality. The definition of the standard error is the following:

$$\mathsf{se} = \sqrt{\mathbb{V}(\hat{\theta}_n)}$$

where $\hat{\theta}_n$ is the point-estimator for the parameter θ, given n samples of data X_n, and $\mathbb{V}(\hat{\theta}_n)$ is the variance of $\hat{\theta}_n$. Likewise, the estimated standard error is $\widehat{\mathsf{se}}$. For example, in our coin-flipping example, the estimator was $\hat{p} = \sum X_i/n$ with corresponding variance $\mathbb{V}(\hat{p}_n) = p(1-p)/n$. Plugging in the point estimate gives us the estimated standard error: $\widehat{\mathsf{se}} = \sqrt{\hat{p}(1-\hat{p})/n}$. Because maximum likelihood estimators are asymptotically normal,[4] we know that $\hat{p}_n \sim \mathcal{N}(p, \widehat{\mathsf{se}}^2)$. Thus, if we want a $1 - \alpha$ confidence interval, we can compute

$$\mathbb{P}(\mid \hat{p}_n - p \mid < \xi) > 1 - \alpha$$

but since we know that $(\hat{p}_n - p)$ is asymptotically normal, $\mathcal{N}(0, \widehat{\mathsf{se}}^2)$, we can instead compute

[4] Certain technical regularity conditions must hold for this property of maximum likelihood estimator to work. See [2] for more details.

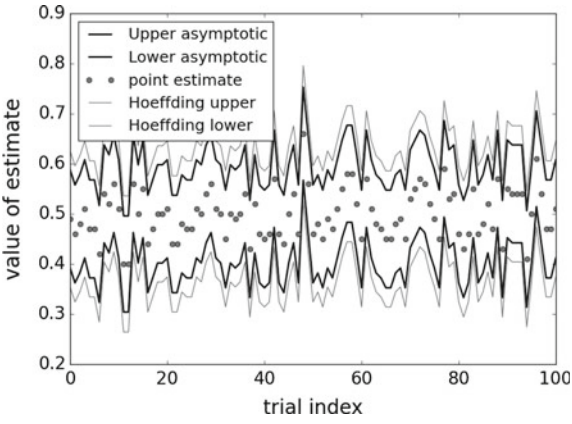

Fig. 3.11 The *gray circles* are the point estimates that are bounded above and below by both asymptotic confidence intervals and Hoeffding intervals. The asymptotic intervals are tighter because the underpinning asymptotic assumptions are valid for these estimates

$$\int_{-\xi}^{\xi} \mathcal{N}(0, \widehat{\mathsf{se}}^2) dx > 1 - \alpha$$

This looks ugly to compute because we need to find ξ, but Scipy has everything we need for this.

```
>>> # compute estimated se for all trials
>>> se=np.sqrt(phat*(1-phat)/xs.shape[0])
>>> # generate random variable for trial 0
>>> rv=stats.norm(0, se[0])
>>> # compute 95\,% confidence interval for that trial 0
>>> np.array(rv.interval(0.95))+phat[0]
array([ 0.42208023,  0.61791977])
>>> def compute_CI(i):
...     return stats.norm.interval(0.95,loc=i,
...                                scale=np.sqrt(i*(1-i)/xs.shape[0]))
...
>>> lower,upper = compute_CI(phat)
```

Figure 3.11 shows the asymptotic confidence intervals and the Hoeffding-derived confidence intervals. As shown, the Hoeffding intervals are a bit more generous than the asymptotic estimates. However, this is only true so long as the asympotic approximation is valid. In other words, there exists some number of n samples for which the asymptotic intervals may not work. So, even though they may be a bit more generous, the Hoeffding intervals do not require arguments about asymptotic convergence. In practice, nonetheless, asymptotic convergence is always in play (even if not explicitly stated).

Confidence Intervals and Hypothesis testing. It turns out that there is a close dual relationship between hypothesis testing and confidence intervals. To see this in action, consider the following hypothesis test for a normal distribution, $H_0 : \mu = \mu_0$ versus $H_1 : \mu \neq \mu_0$. A reasonable test has the following rejection region:

$$\left\{ x :| \bar{x} - \mu_0 | > z_{\alpha/2} \frac{\sigma}{\sqrt{n}} \right\}$$

where $\mathbb{P}(Z > z_{\alpha/2}) = \alpha/2$ and $\mathbb{P}(-z_{\alpha/2} < Z < z_{\alpha/2}) = 1 - \alpha$ and where $Z \sim \mathcal{N}(0, 1)$. This is the same thing as saying that the region corresponding to acceptance of H_0 is then,

$$\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \mu_0 \leq \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \tag{3.6.0.1}$$

Because the test has size α, the false alarm probability, $\mathbb{P}(H_0 \texttt{ rejected } | \ \mu = \mu_0) = \alpha$. Likewise, the $\mathbb{P}(H_0 \texttt{ accepted } | \ \mu = \mu_0) = 1 - \alpha$. Putting this all together with interval defined above means that

$$\mathbb{P}\left(\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \mu_0 \leq \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \Big| H_0\right) = 1 - \alpha$$

Because this is valid for any μ_0, we can drop the H_0 condition and say the following:

$$\mathbb{P}\left(\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \mu_0 \leq \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right) = 1 - \alpha$$

As may be obvious by now, the interval in Eq. 3.6.0.1 above *is* the $1-\alpha$ confidence interval! Thus, we have just obtained the confidence interval by inverting the acceptance region of the level α test. The hypothesis test fixes the *parameter* and then asks what sample values (i.e., the acceptance region) are consistent with that fixed value. Alternatively, the confidence interval fixes the sample value and then asks what parameter values (i.e., the confidence interval) make this sample value most plausible. Note that sometimes this inversion method results in disjoint intervals (known as *confidence sets*).

3.7 Linear Regression

Linear regression gets to the heart of statistics: Given a set of data points, what is the relationship of the data in-hand to data yet seen? How should information from one data set propagate to other data? Linear regression offers the following model to address this question:

$$\mathbb{E}(Y|X = x) \approx ax + b$$

That is, given specific values for X, assume that the conditional expectation is a linear function of those specific values. However, because the observed values are not the expectations themselves, the model accommodates this with an additive noise term. In other words, the observed variable (a.k.a. response, target, dependent variable) is modeled as,

$$\mathbb{E}(Y|X = x_i) + \epsilon_i \approx ax + b + \epsilon_i = y$$

where $\mathbb{E}(\epsilon_i) = 0$ and the ϵ_i are iid and where the distribution function of ϵ_i depends on the problem, even though it is often assumed Gaussian. The $X = x$ values are known as independent variables, covariates, or regressors.

Let's see if we can use all of the methods we have developed so far to understand this form of regression. The first task is to determine how to estimate the unknown linear parameters, a and b. To make this concrete, let's assume that $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Bear in mind that $\mathbb{E}(Y|X = x)$ is a deterministic function of x. In other words, the variable x changes with each draw, but after the data have been collected these are no longer random quantities. Thus, for fixed x, y is a random variable generated by ϵ. Perhaps we should denote ϵ as ϵ_x to emphasize this, but because ϵ is an independent, identically-distributed (iid) random variable at each fixed x, this would be excessive. Because of Gaussian additive noise, the distribution of y is completely characterized by its mean and variance.

$$\begin{aligned}\mathbb{E}(y) &= ax + b \\ \mathbb{V}(y) &= \sigma^2\end{aligned}$$

Using the maximum likelihood procedure, we write out the log-likelihood function as

$$\mathcal{L}(a, b) = \sum_{i=1}^{n} \log \mathcal{N}(ax_i + b, \sigma^2) \propto \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

Note that we suppressed the terms that are irrelevent to the maximum-finding. Taking the derivative of this with respect to a gives the following equation:

$$\frac{\partial \mathcal{L}(a, b)}{\partial a} = 2 \sum_{i=1}^{n} x_i (b + ax_i - y_i) = 0$$

Likewise, we do the same for the b parameter

$$\frac{\partial \mathcal{L}(a, b)}{\partial b} = 2 \sum_{i=1}^{n} (b + ax_i - y_i) = 0$$

The following code simulates some data and uses Numpy tools to compute the parameters as shown,

```
>>> import numpy as np
>>> a = 6;b = 1 # parameters to estimate
>>> x = np.linspace(0,1,100)
>>> y = a*x + np.random.randn(len(x))+b
>>> p,var_=np.polyfit(x,y,1,cov=True) # fit data to line
>>> y_ = np.polyval(p,x) # estimated by linear regression
```

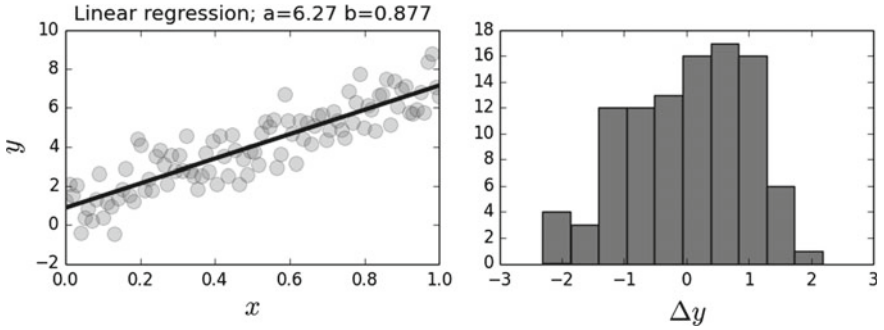

Fig. 3.12 The panel on the *left* shows the data and regression line. The panel on the *right* shows a histogram of the regression errors

The graph on the left of Fig. 3.12 shows the regression line plotted against the data. The estimated parameters are noted in the title. The histogram on the right of Fig. 3.12 shows the residual errors in the model. It is always a good idea to inspect the residuals of any regression for normality. These are the differences between the fitted line for each x_i value and the corresponding y_i value in the data. Note that the x term does not have to be uniformly monotone.

To decouple the deterministic variation from the random variation, we can fix the index and write separate problems of the form

$$y_i = ax_i + b + \epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. What could we do with just this one component of the problem? In other words, suppose we had m-samples of this component as in $\{y_{i,k}\}_{k=1}^m$. Following the usual procedure, we could obtain estimates of the mean of y_i as

$$\hat{y_i} = \frac{1}{m}\sum_{k=1}^{m} y_{i,k}$$

However, this tells us nothing about the individual parameters a and b because they are not separable in the terms that are computed, namely, we may have

$$\mathbb{E}(y_i) = ax_i + b$$

but we still only have one equation and the two unknowns, a and b. How about if we consider and fix another component j as in

$$y_j = ax_j + b + \epsilon_i$$

Then, we have

$$\mathbb{E}(y_j) = ax_j + b$$

so at least now we have two equations and two unknowns and we know how to estimate the left hand sides of these equations from the data using the estimators $\hat{y}_i$ and $\hat{y}_j$. Let's see how this works in the code sample below (Fig. 3.13).

```
>>> x0, xn =x[0],x[80]
>>> # generate synthetic data
>>> y_0 = a*x0 + np.random.randn(20)+b
>>> y_1 = a*xn + np.random.randn(20)+b
>>> # mean along sample dimension
>>> yhat = np.array([y_0,y_1]).mean(axis=1)
>>> a_,b_=np.linalg.solve(np.array([[x0,1],
...                                 [xn,1]]),yhat)
```

Programming Tip

The prior code uses the `solve` function in the Numpy `linalg` module, which contains the core linear algebra codes in Numpy that incorporate the battle-tested LAPACK library.

We can write out the solution for the estimated parameters for this case where $x_0 = 0$

$$\hat{a} = \frac{\hat{y}_i - \hat{y}_0}{x_i}$$
$$\hat{b} = \hat{y}_0$$

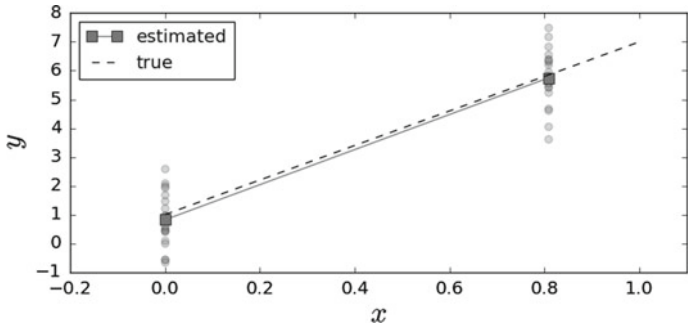

Fig. 3.13 The fitted and true lines are plotted with the data values. The squares at either end of the *solid line* show the mean value for each of the data groups shown

The expectations and variances of these estimators are the following,

$$\mathbb{E}(\hat{a}) = \frac{ax_i}{x_i} = a$$
$$\mathbb{E}(\hat{b}) = b$$
$$\mathbb{V}(\hat{a}) = \frac{2\sigma^2}{x_i^2}$$
$$\mathbb{V}(\hat{b}) = \sigma^2$$

The expectations show that the estimators are unbiased. The estimator $\hat{a}$ has a variance that decreases as larger points x_i are selected. That is, it is better to have samples further out along the horizontal axis for fitting the line. This variance quantifies the *leverage* of those distant points.

Regression From Projection Methods. Let's see if we can apply our knowledge of projection methods to the general case. In vector notation, we can write the following:

$$\mathbf{y} = a\mathbf{x} + b\mathbf{1} + \boldsymbol{\epsilon}$$

where $\mathbf{1}$ is the vector of all ones. Let's use the inner-product notation,

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbb{E}(\mathbf{x}^T \mathbf{y})$$

Then, by taking the inner-product with some $\mathbf{x}_1 \in \mathbf{1}^\perp$ we obtain,[5]

$$\langle \mathbf{y}, \mathbf{x}_1 \rangle = a \langle \mathbf{x}, \mathbf{x}_1 \rangle$$

Recall that $\mathbb{E}(\boldsymbol{\epsilon}) = \mathbf{0}$. We can finally solve for a as

$$\hat{a} = \frac{\langle \mathbf{y}, \mathbf{x}_1 \rangle}{\langle \mathbf{x}, \mathbf{x}_1 \rangle} \tag{3.7.0.2}$$

That was pretty neat but now we have the mysterious $\mathbf{x}_1$ vector. Where does this come from? If we project $\mathbf{x}$ onto the $\mathbf{1}^\perp$, then we get the MMSE approximation to $\mathbf{x}$ in the $\mathbf{1}^\perp$ space. Thus, we take

$$\mathbf{x}_1 = P_{\mathbf{1}^\perp}(\mathbf{x})$$

Remember that $P_{\mathbf{1}^\perp}$ is a projection matrix so the length of $\mathbf{x}_1$ is at most $\mathbf{x}$. This means that the denominator in the $\hat{a}$ equation above is really just the length of the $\mathbf{x}$ vector in the coordinate system of $P_{\mathbf{1}^\perp}$. Because the projection is orthogonal (namely, of minimum length), the Pythagorean theorem gives this length as the following:

[5] The space of all vectors, $\mathbf{a}$ such that $\langle \mathbf{a}, \mathbf{1} \rangle = 0$ is denoted $\mathbf{1}^\perp$.

$$\langle \mathbf{x}, \mathbf{x}_1 \rangle^2 = \langle \mathbf{x}, \mathbf{x} \rangle - \langle \mathbf{1}, \mathbf{x} \rangle^2$$

The first term on the right is the length of the $\mathbf{x}$ vector and last term is the length of $\mathbf{x}$ in the coordinate system orthogonal to $P_{\mathbf{1}^\perp}$, namely that of $\mathbf{1}$. We can use this geometric interpretation to understand what is going on in typical linear regression in much more detail. The fact that the denominator is the orthogonal projection of $\mathbf{x}$ tells us that the choice of $\mathbf{x}_1$ has the strongest effect (i.e., largest value) on reducing the variance of $\hat{a}$. That is, the more $\mathbf{x}$ is aligned with $\mathbf{1}$, the worse the variance of $\hat{a}$. This makes intuitive sense because the closer $\mathbf{x}$ is to $\mathbf{1}$, the more constant it is, and we have already seen from our one-dimensional example that distance between the x terms pays off in reduced variance. We already know that $\hat{a}$ is an unbiased estimator and because we chose $\mathbf{x}_1$ deliberately as a projection, we know that it is also of minimum variance. Such estimators are known as Minimum-Variance Unbiased Estimators (MVUE).

In the same spirit, let's examine the numerator of $\hat{a}$ in Eq. 3.7.0.2. We can write $\mathbf{x}_1$ as the following

$$\mathbf{x}_1 = \mathbf{x} - P_{\mathbf{1}}\mathbf{x}$$

where $P_{\mathbf{1}}$ is projection matrix of $\mathbf{x}$ onto the $\mathbf{1}$ vector. Using this, the numerator of $\hat{a}$ becomes

$$\langle \mathbf{y}, \mathbf{x}_1 \rangle = \langle \mathbf{y}, \mathbf{x} \rangle - \langle \mathbf{y}, P_{\mathbf{1}}\mathbf{x} \rangle$$

Note that,

$$P_{\mathbf{1}} = \mathbf{1}\mathbf{1}^T \frac{1}{n}$$

so that writing this out explicitly gives

$$\langle \mathbf{y}, P_{\mathbf{1}}\mathbf{x} \rangle = \left(\mathbf{y}^T \mathbf{1}\right)\left(\mathbf{1}^T \mathbf{x}\right)/n = \left(\sum y_i\right)\left(\sum x_i\right)/n$$

and similarly, we have the following for the denominator:

$$\langle \mathbf{x}, P_{\mathbf{1}}\mathbf{x} \rangle = \left(\mathbf{x}^T \mathbf{1}\right)\left(\mathbf{1}^T \mathbf{x}\right)/n = \left(\sum x_i\right)\left(\sum x_i\right)/n$$

So, plugging all of this together gives the following,

$$\hat{a} = \frac{\mathbf{x}^T\mathbf{y} - (\sum x_i)(\sum y_i)/n}{\mathbf{x}^T\mathbf{x} - (\sum x_i)^2/n}$$

with corresponding variance,

$$\begin{aligned}\mathbb{V}(\hat{a}) &= \sigma^2 \frac{\|\mathbf{x}_1\|^2}{\langle \mathbf{x}, \mathbf{x}_1 \rangle^2} \\ &= \frac{\sigma^2}{\|\mathbf{x}\|^2 - n(\overline{x^2})}\end{aligned}$$

Using the same approach with $\hat{b}$ gives,

$$\hat{b} = \frac{\langle \mathbf{y}, \mathbf{x}^{\perp} \rangle}{\langle \mathbf{1}, \mathbf{x}^{\perp} \rangle} \tag{3.7.0.3}$$

$$= \frac{\langle \mathbf{y}, \mathbf{1} - P_{\mathbf{x}}(\mathbf{1}) \rangle}{\langle \mathbf{1}, \mathbf{1} - P_{\mathbf{x}}(\mathbf{1}) \rangle} \tag{3.7.0.4}$$

$$= \frac{\mathbf{x}^T \mathbf{x} (\sum y_i)/n - \mathbf{x}^T \mathbf{y} (\sum x_i)/n}{\mathbf{x}^T \mathbf{x} - (\sum x_i)^2/n} \tag{3.7.0.5}$$

where

$$P_{\mathbf{x}} = \frac{\mathbf{x}\mathbf{x}^{\mathbf{T}}}{\|\mathbf{x}\|^2}$$

with variance

$$\begin{aligned}\mathbb{V}(\hat{b}) &= \sigma^2 \frac{\langle \mathbf{1} - \boldsymbol{P}_{\mathbf{x}}(\mathbf{1}), \mathbf{1} - \boldsymbol{P}_{\mathbf{x}}(\mathbf{1}) \rangle}{\langle \mathbf{1}, \mathbf{1} - \boldsymbol{P}_{\mathbf{x}}(\mathbf{1}) \rangle^2} \\ &= \frac{\sigma^2}{n - \frac{(n\overline{x})^2}{\|\mathbf{x}\|^2}}\end{aligned}$$

Qualifying the Estimates. Our formulas for the variance above include the unknown σ^2, which we must estimate from the data itself using our plug-in estimates. We can form the residual sum of squares as

$$\texttt{RSS} = \sum_i (\hat{a} x_i + \hat{b} - y_i)^2$$

Thus, the estimate of σ^2 can be expressed as

$$\hat{\sigma}^2 = \frac{\texttt{RSS}}{n-2}$$

where n is the number of samples. This is also known as the *residual mean square*. The $n-2$ represents the *degrees of freedom* (`df`). Because we estimated two parameters from the same data we have $n-2$ instead of n. Thus, in general, $\texttt{df} = n - p$, where p is the number of estimated parameters. Under the assumption that the noise is Gaussian, the $\texttt{RSS}/\sigma^2$ is chi-squared distributed with $n-2$ degrees of freedom. Another important term is the *sum of squares about the mean*, (a.k.a *corrected* sum of squares),

$$\texttt{SYY} = \sum (y_i - \bar{y})^2$$

The `SYY` captures the idea of not using the x_i data and just using the mean of the y_i data to estimate y. These two terms lead to the R^2 term,

$$R^2 = 1 - \frac{\texttt{RSS}}{\texttt{SYY}}$$

Note that for perfect regression, $R^2 = 1$. That is, if the regression gets each y_i data point exactly right, then $\texttt{RSS} = 0$ this term equals one. Thus, this term is used to measure of goodness-of-fit. The `stats` module in `scipy` computes many of these terms automatically,

```
from scipy import stats
slope,intercept,r_value,p_value,stderr = stats.linregress(x,y)
```

where the square of the `r_value` variable is the R^2 above. The computed p-value is the two-sided hypothesis test with a null hypothesis that the slope of the line is zero. In other words, this tests whether or not the linear regression makes sense for the data for that hypothesis. The Statsmodels module provides a powerful extension to Scipy's stats module by making it easy to do regression and keep track of these parameters. Let's reformulate our problem using the Statsmodels framework by creating a Pandas dataframe for the data,

```
import statsmodels.formula.api as smf
from pandas import DataFrame
import numpy as np
d = DataFrame({'x':np.linspace(0,1,10)}) # create data
d['y'] = a*d.x+ b + np.random.randn(*d.x.shape)
```

Now that we have the input data in the above Pandas dataframe, we can perform the regression as in the following,

```
results = smf.ols('y ~ x', data=d).fit()
```

The $\sim$ symbol is notation for $y = ax + b + \epsilon$, where the constant b is implicit in this usage of Statsmodels. The names in the string are taken from the columns in the dataframe. This makes it very easy to build models with complicated interactions between the named columns in the dataframe. We can examine a report of the model fit by looking at the summary,

```
print results.summary2()
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.808
Dependent Variable: y                AIC:                28.1821
Date:               0000-00-00 00:00 BIC:                00.0000
No. Observations:   10               Log-Likelihood:     -12.091
Df Model:           1                F-statistic:        38.86
Df Residuals:       8                Prob (F-statistic): 0.000250
R-squared:          0.829            Scale:              0.82158
-----------------------------------------------------------------
             Coef.    Std.Err.     t      P>|t|    [0.025   0.975]
-----------------------------------------------------------------
Intercept    1.5352     0.5327   2.8817   0.0205   0.3067   2.7637
x            5.5990     0.8981   6.2340   0.0003   3.5279   7.6701
```

There is a lot more here than we have discussed so far, but the Statsmodels documentation is the best place to go for complete information about this report. The F-statistic attempts to capture the contrast between including the slope parameter or leaving it off. That is, consider two hypotheses:

$$H_0 : \mathbb{E}(Y|X = x) = b$$
$$H_1 : \mathbb{E}(Y|X = x) = b + ax$$

In order to quantify how much better adding the slope term is for the regression, we compute the following:

$$F = \frac{\mathrm{SYY} - \mathrm{RSS}}{\hat{\sigma}^2}$$

The numerator computes the difference in the residual squared errors between including the slope in the regression or just using the mean of the y_i values. Once again, if we assume (or can claim asymptotically) that the ϵ noise term is Gaussian, $\epsilon \sim \mathcal{N}(0, \sigma^2)$, then the H_0 hypothesis will follow an F-distribution[6] with degrees of freedom from the numerator and denominator. In this case, $F \sim F(1, n-2)$. The value of this statistic is reported by Statsmodels above. The corresponding reported probability shows the chance of F exceeding its computed value if H_0 were true. So, the take-home message from all this is that including the slope leads to a much smaller reduction in squared error than could be expected from a favorable draw of n points of this data, under the Gaussian additive noise assumption. This is evidence that including the slope is meaningful for this data.

The Statsmodels report also shows the adjusted R^2 term. This is a correction to the R^2 calculation that accounts for the number of parameters p that the regression is fitting and the sample size n,

$$\texttt{Adjusted}\ \ R^2 = 1 - \frac{\mathrm{RSS}/(n-p)}{\mathrm{SYY}/(n-1)}$$

[6] The $F(m, n)$ F-distribution has two integer degree-of-freedom parameters, m and n.

This is always lower than R^2 except when $p = 1$ (i.e., estimating only b). This becomes a better way to compare regressions when one is attempting to fit many parameters with comparatively small n.

Linear Prediction. Using linear regression for prediction introduces some other issues. Recall the following expectation,

$$\mathbb{E}(Y|X = x) \approx \hat{a}x + \hat{b}$$

where we have determined $\hat{a}$ and $\hat{b}$ from the data. Given a new point of interest, x_p, we would certainly compute

$$\hat{y}_p = \hat{a}x_p + \hat{b}$$

as the predicted value for $\hat{y_p}$. This is the same as saying that our best prediction for y based on x_p is the above conditional expectation. The variance for this is the following,

$$\mathbb{V}(y_p) = x_p^2\mathbb{V}(\hat{a}) + \mathbb{V}(\hat{b}) + 2x_p\text{cov}(\hat{a}\hat{b})$$

Note that we have the covariance above because $\hat{a}$ and $\hat{b}$ are derived from the same data. We can work this out below using our previous notation from Eq. 3.7.0.2,

$$\begin{aligned}
\text{cov}(\hat{a}\hat{b}) =& \frac{\mathbf{x}_1^T\mathbb{V}\{\mathbf{y}\mathbf{y}^T\}\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} = \frac{\mathbf{x}_1^T\sigma^2\mathbf{I}\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} \\
=& \sigma^2\frac{\mathbf{x}_1^T\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} = \sigma^2\frac{(\mathbf{x} - P_1\mathbf{x})^T\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} \\
=& \sigma^2\frac{-\mathbf{x}^T P_1^T\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} = \sigma^2\frac{-\mathbf{x}^T\frac{1}{n}\mathbf{1}\mathbf{1}^T\mathbf{x}^\perp}{(\mathbf{x}_1^T\mathbf{x})(\mathbf{1}^T\mathbf{x}^\perp)} \\
=& \sigma^2\frac{-\mathbf{x}^T\frac{1}{n}\mathbf{1}}{(\mathbf{x}_1^T\mathbf{x})} = \frac{-\sigma^2\overline{x}}{\sum_{i=1}^n(x_i^2 - \overline{x}^2)}
\end{aligned}$$

After plugging all this in, we obtain the following,

$$\mathbb{V}(y_p) = \sigma^2\frac{x_p^2 - 2x_p\overline{x} + \|\mathbf{x}\|^2/n}{\|\mathbf{x}\|^2 - n\overline{x}^2}$$

where, in practice, we use the plug-in estimate for the σ^2.

There is an important consequence for the confidence interval for y_p. We cannot simply use the square root of $\mathbb{V}(y_p)$ to form the confidence interval because the model includes the extra ϵ noise term. In particular, the parameters were computed

using a set of statistics from the data, but now must include different realizations for the noise term for the prediction part. This means we have to compute

$$\eta^2 = \mathbb{V}(y_p) + \sigma^2$$

Then, the 95 % confidence interval $y_p \in (y_p - 2\hat{\eta}, y_p + 2\hat{\eta})$ is the following,

$$\mathbb{P}(y_p - 2\hat{\eta} < y_p < y_p + 2\hat{\eta}) \approx \mathbb{P}(-2 < \mathcal{N}(0, 1) < 2) \approx 0.95$$

where $\hat{\eta}$ comes from substituting the plug-in estimate for σ.

3.7.1 Extensions to Multiple Covariates

With all the machinery we have, it is a short notational hop to consider multiple regressors as in the following,

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

with the usual $\mathbb{E}(\boldsymbol{\epsilon}) = \mathbf{0}$ and $\mathbb{V}(\boldsymbol{\epsilon}) = \sigma^2\mathbf{I}$. Thus, $\mathbf{X}$ is a $n \times p$ full rank matrix of regressors and $\mathbf{Y}$ is the n-vector of observations. Note that the constant term has been incorporated into $\mathbf{X}$ as a column of ones. The corresponding estimated solution for $\boldsymbol{\beta}$ is the following,

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

with corresponding variance,

$$\mathbb{V}(\hat{\boldsymbol{\beta}}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$$

and with the assumption of Gaussian errors, we have

$$\hat{\boldsymbol{\beta}} \sim \mathcal{N}(\boldsymbol{\beta}, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1})$$

The unbiased estimate of σ^2 is the following,

$$\hat{\sigma}^2 = \frac{1}{n-p}\sum \hat{\epsilon}_i^2$$

where $\hat{\boldsymbol{\epsilon}} = \mathbf{X}\hat{\boldsymbol{\beta}} - \mathbf{Y}$ is the vector of residuals. Tukey christened the following matrix as the *hat* matrix (a.k.a. influence matrix),

$$\mathbf{V} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

because it maps $\mathbf{Y}$ into $\hat{\mathbf{Y}}$,

$$\hat{\mathbf{Y}} = \mathbf{V}\mathbf{Y}$$

As an exercise you can check that $\mathbf{V}$ is a projection matrix. Note that that matrix is solely a function of $\mathbf{X}$. The diagonal elements of $\mathbf{V}$ are called the *leverage values* and are contained in the closed interval $[1/n, 1]$. These terms measure of distance between the values of x_i and the mean values over the n observations. Thus, the leverage terms depend only on $\mathbf{X}$. This is the generalization of our initial discussion of leverage where we had multiple samples at only two x_i points. Using the hat matrix, we can compute the variance of each residual, $e_i = \hat{y} - y_i$ as

$$\mathbb{V}(e_i) = \sigma^2(1 - v_i)$$

where $v_i = V_{i,i}$. Given the above-mentioned bounds on v_i, these are always less than σ^2.

Degeneracy in the columns of $\mathbf{X}$ can become a problem. This is when two or more of the columns become co-linear. We have already seen this with our single regressor example wherein $\mathbf{x}$ close to $\mathbf{1}$ was bad news. To compensate for this effect we can load the diagonal elements and solve for the unknown parameters as in the following,

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}$$

where $\alpha > 0$ is a tunable hyper-parameter. This method is known as *ridge regression* and was proposed in 1970 by Hoerl and Kenndard. It can be shown that this is the equivalent to minimizing the following objective,

$$\|\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \alpha\|\boldsymbol{\beta}\|^2$$

In other words, the length of the estimated $\boldsymbol{\beta}$ is penalized with larger α. This has the effect of stabilizing the subsequent inverse calculation and also providing a means to trade bias and variance, which we will discuss at length in Sect. 4.6.

Interpreting Residuals. Our model assumes an additive Gaussian noise term. We can check the voracity of this assumption by examining the residuals after fitting. The residuals are the difference between the fitted values and the original data

$$\hat{\epsilon}_i = \hat{a}x_i + \hat{b} - y_i$$

While the p-value and the F-ratio provide some indication of whether or not computing the slope of the regression makes sense, we can get directly at the key assumption of additive Gaussian noise.

For sufficiently small dimensions, the `scipy.stats.probplot` we discussed in the last chapter provides quick visual evidence one way or another by plotting the standardized residuals,

$$r_i = \frac{e_i}{\hat{\sigma}\sqrt{1 - v_i}}$$

The other part of the iid assumption implies homoscedasticity (all r_i have equal variances). Under the additive Gaussian noise assumption, the e_i should also be distributed according to $\mathcal{N}(0, \sigma^2(1 - v_i))$. The normalized residuals r_i should then be distributed according to $\mathcal{N}(0, 1)$. Thus, the presence of any $r_i \notin [-1.96, 1.96]$ should not be common at the 5 % significance level and is thereby breeds suspicion regarding the homoscedasticity assumption.

The Levene test in `scipy.stats.leven` tests the null hypothesis that all the variances are equal. This basically checks whether or not the standardized residuals vary across x_i more than expected. Under the homoscedasticity assumption, the variance should be independent of x_i. If not, then this is a clue that there is a missing variable in the analysis or that the variables themselves should be transformed (e.g., using the log function) into another format that can reduce this effect. Also, we can use weighted least-squares instead of ordinary least-squares.

Variable Scaling. It is tempting to conclude in a multiple regression that small coefficients in any of the β terms implies that those terms are not important. However, simple unit conversions can cause this effect. For example, if one of the regressors is in units of kilometers and the others are in meters, then just the scale factor can give the impression of outsized or under-sized effects. The common way to account for this is to scale the regressors so that

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

This has the side effect of converting the slope parameters into correlation coefficients, which is bounded by ± 1.

Influential Data. We have already discussed the idea of leverage. The concept of *influence* combines leverage with outliers. To understand influence, consider Fig. 3.14.

The point on the right in Fig. 3.14 is the only one that is contributing to the calculation of the slope for the fitted line. Thus, it is very influential in this sense. Cook's distance is a good way to get at this concept numerically. To compute this, we have to compute the jth component of the estimated target variable with the ith point deleted. We call this $\hat{y}_{j(i)}$. Then, we compute the following,

$$D_i = \frac{\sum_j (\hat{y}_j - \hat{y}_{j(i)})^2}{p/n \sum_j (\hat{y}_j - y_j)^2}$$

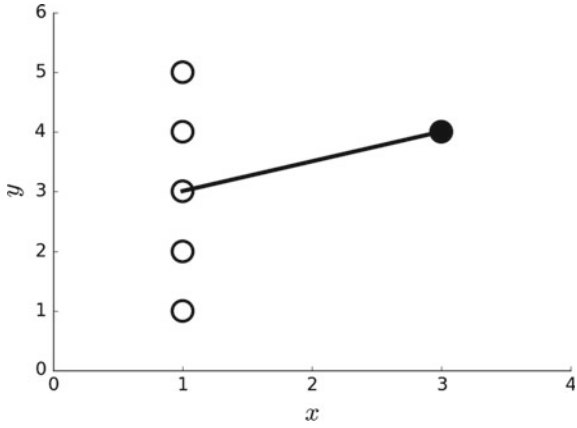

Fig. 3.14 The point on the right has outsized influence in this data because it is the only one used to determine the slope of the fitted line

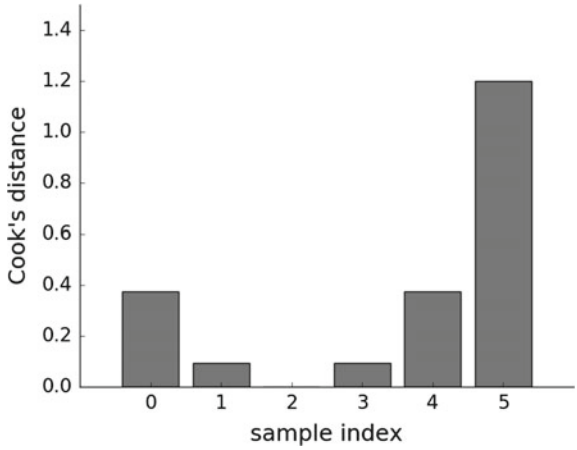

Fig. 3.15 The calculated Cook's distance for the data in Fig. 3.14

where, as before, p is the number of estimated terms (e.g., $p = 2$ in the bivariate case). This calculation emphasizes the effect of the outlier by predicting the target variable with and without each point. In the case of Fig. 3.14, losing any of the points on the left cannot change the estimated target variable much, but losing the single point on the right surely does. The point on the right does not seem to be an outlier (it *is* on the fitted line), but this is because it is influential enough to rotate the line to align with it. Cook's distance helps capture this effect by leaving each sample out and re-fitting the remainder as shown in the last equation. Figure 3.15 shows the calculated Cook's distance for the data in Fig. 3.14, showing that the data point on the right (sample index `5`) has outsized influence on the fitted line. As a rule of thumb, Cook's distance values greater than one are suspect.

As another illustration of influence, consider Fig. 3.16 which shows some data that nicely line up, but with one outlier (filled black circle) in the upper panel. The lower panel shows so-computed Cook's distance for this data. As shown Cook's distance emphasizes the presence of the outlier. Because the calculation involves leaving a single sample out and re-calculating the rest, it can be a time-consuming operation

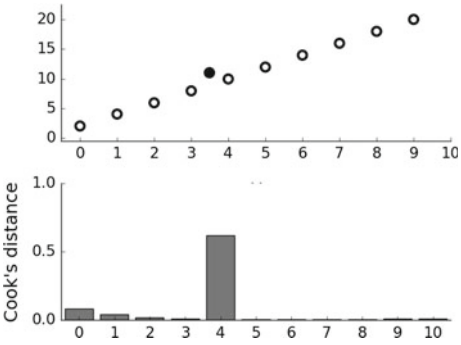

Fig. 3.16 The *upper panel* shows data that fit on a line and an outlier point (*filled black circle*). The *lower panel* shows the calculated Cook's distance for the data in *upper panel* and shows that the tenth point (i.e., the outlier) has disproportionate influence

suitable to relatively small data sets. There is always the temptation to downplay the importance of outliers because they conflict with a favored model, but outliers must be carefully examined to understand why the model is unable to capture them. It could be something as simple as faulty data collection, or it could be an indication of deeper issues that have been overlooked. The following code shows how Cook's distance was compute for Figs. 3.15 and 3.16.

```
>>> fit = lambda i,x,y: np.polyval(np.polyfit(x,y,1),i)
>>> omit = lambda i,x: ([k for j,k in enumerate(x) if j !=i])
>>> def cook_d(k):
...     num = sum((fit(j,omit(k,x),omit(k,y))- fit(j,x,y))**2 for j in x)
...     den = sum((y-np.polyval(np.polyfit(x,y,1),x))**2/len(x)*2)
...     return num/den
...
```

Programming Tip

The function `omit` sweeps through the data and excludes the ith data element. The embedded `enumerate` function associates every element in the iterable with its corresponding index.

3.8 Maximum A-Posteriori

We saw with maximum likelihood estimation how we could use the principle of maximum likelihood to derive a formula of the data that would estimate the underlying parameters (say, θ). Under that method, the parameter was fixed, but unknown. If we change our perspective slightly and consider the underlying parameter as a random variable in its own right, this leads to additional flexibility in estimation. This method is the simplest of the family of Bayesian statistical methods and is most closely related to maximum likelihood estimation. It is very popular in communications and signal processing and is the backbone of many important algorithms in those areas.

Given that the parameter θ is also a random variable, it has a joint distribution with the other random variables, say, $f(x, \theta)$. Bayes' theorem gives the following:

$$\mathbb{P}(\theta|x) = \frac{\mathbb{P}(x|\theta)\mathbb{P}(\theta)}{\mathbb{P}(x)}$$

The $\mathbb{P}(x|\theta)$ term is the usual likelihood term we have seen before. The term in the denominator is *prior* probability of the data x and it explicitly makes a very powerful claim: even before collecting or processing any data, we know what the probability of that data is. The $\mathbb{P}(\theta)$ is the prior probability of the parameter. In other words, regardless of the data that is collected, this is the probability of the parameter itself.

In a particular application, whether or not you feel justified making these claims is something that you have to reconcile for yourself and the problem at hand. There are many persuasive philosophical arguments one way or the other, but the main thing to keep in mind when applying any method is whether or not the assumptions are reasonable for the problem at hand.

However, for now, let's just assume that we somehow have $\mathbb{P}(\theta)$ and the next step is the maximizing of this expression over the θ. Whatever results from that maximization is the maximum a-posteriori (MAP) estimator for θ. Because the maximization takes place with respect to θ and not x, we can ignore the $\mathbb{P}(x)$ part. To make things concrete, let us return to our original coin flipping problem. From our earlier analysis, we know that the likelihood function for this problem is the following:

$$\ell(\theta) := \theta^k(1-\theta)^{(n-k)}$$

where the probability of the coin coming up heads is θ. The next step is the prior probability, $\mathbb{P}(\theta)$. For this example, we will choose the $\beta(6, 6)$ distribution (shown in the top left panel of Fig. 3.17). The β family of distributions is a gold mine because it allows for a wide variety of distributions using few input parameters. Now that we have all the ingredients, we turn to maximizing the posterior function, $\mathbb{P}(\theta|x)$. Because the logarithm is convex, we can use it to make the maximization process easier by converting the product to a sum without changing the extrema that we are looking for. Thus, we prefer the to work with the logarithm of $\mathbb{P}(\theta|x)$ as in the following.

$$\mathcal{L} := \log \mathbb{P}(\theta|x) = \log \ell(\theta) + \log \mathbb{P}(\theta) - \log \mathbb{P}(x)$$

This is tedious to do by hand and therefore an excellent job for Sympy.

```
>>> import sympy
>>> from sympy import stats as st
>>> from sympy.abc import p,k,n
# setup objective function using sympy.log
>>> obj=sympy.expand_log(sympy.log(p**k*(1-p)**(n-k)*
                         st.density(st.Beta('p',6,6))(p)))
# use calculus to maximize objective
>>> sol=sympy.solve(sympy.simplify(sympy.diff(obj,p)),p)[0]
>>> sol
(k + 5)/(n + 10)
```

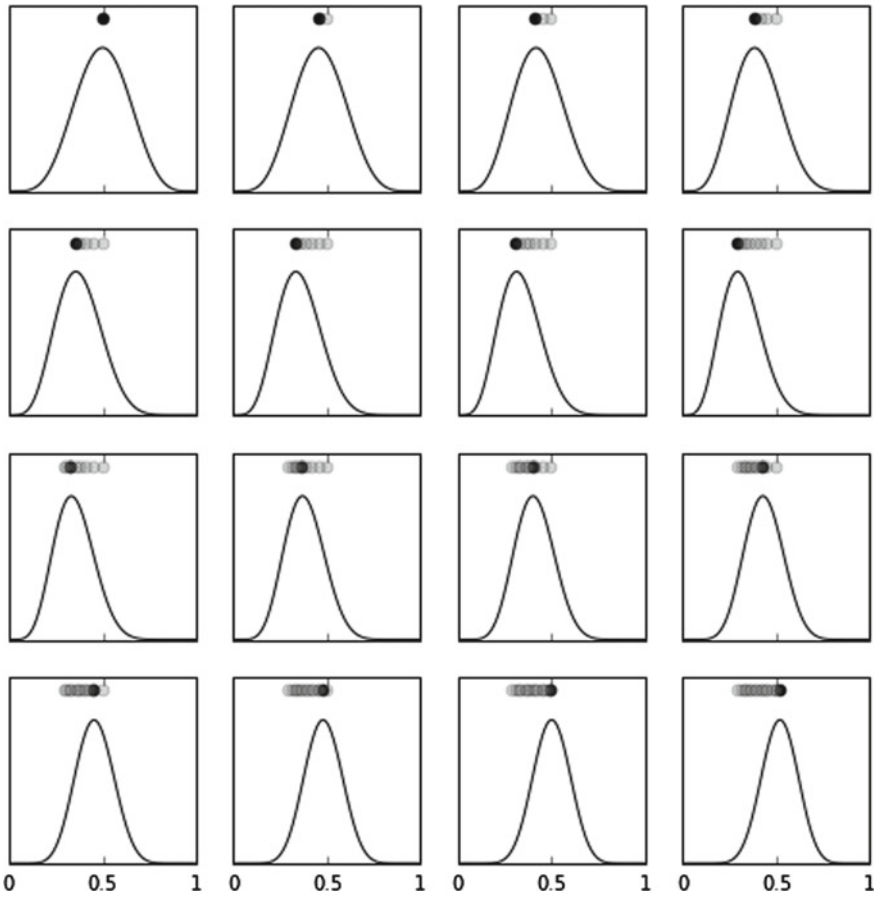

Fig. 3.17 The prior probability is the $\beta(6, 6)$ distribution shown in the *top left panel*. The *dots* near the peaks of each of the subgraphs indicate the MAP estimate at that frame

which means that our MAP estimator of θ is the following:

$$\hat{\theta}_{MAP} = \frac{k+5}{n+10}$$

where k is the number of heads in the sample. This is obviously a biased estimator of θ,

$$\mathbb{E}(\hat{\theta}_{MAP}) = \frac{5+n\theta}{10+n} \neq \theta$$

But is this bias *bad*? Why would anyone want a biased estimator? Remember that we constructed this entire estimator using the idea of the prior probability of $\mathbb{P}(\theta)$ which *favors* (biases!) the estimate according to the prior. For example, if $\theta = 1/2$,

the MAP estimator evaluates to $\hat{\theta}_{MAP} = 1/2$. No bias there! This is because the peak of the prior probability is at $\theta = 1/2$.

To compute the corresponding variance for this estimator, we need this intermediate result,

$$\mathbb{E}(\hat{\theta}^2_{MAP}) = \frac{25 + 10n\theta + n\theta((n-1)p + 1)}{(10+n)^2}$$

which gives the following variance,

$$\mathbb{V}(\hat{\theta}_{MAP}) = \frac{n(1-\theta)\theta}{(n+10)^2}$$

Let's pause and compare this to our previous maximum likelihood (ML) estimator shown below:

$$\hat{\theta}_{ML} = \frac{1}{n}\sum_{i=1}^{n} X_i = \frac{k}{n}$$

As we discussed before, the ML-estimator is unbiased with the following variance.

$$\mathbb{V}(\hat{\theta}_{ML}) = \frac{\theta(1-\theta)}{n}$$

How does this variance compare to that of the MAP? The ratio of the two is the following:

$$\frac{\mathbb{V}(\hat{\theta}_{MAP})}{\mathbb{V}(\hat{\theta}_{ML})} = \frac{n^2}{(n+10)^2}$$

This ratio shows that the variance for the MAP-estimator is smaller than that of the ML-estimator. This is payoff for having a biased MAP-estimator—it requires fewer samples to estimate if the underlying parameter is consistent with the prior probability. If not, then it will take more samples to pull the estimator away from the bias. In the limit as $n \to \infty$, the ratio goes to one. This means that the benefit of the reduced variance vanishes with enough samples.

The above discussion admits a level of arbitrariness via the prior distribution. We don't have to choose just one prior, however. The following shows how we can use the previous posterior distribution as the prior for the next posterior distribution,

$$\mathbb{P}(\theta|x_{k+1}) = \frac{\mathbb{P}(x_{k+1}|\theta)\mathbb{P}(\theta|x_k)}{\mathbb{P}(x_{k+1})}$$

This is a very different strategy because we are using every data sample x_k as a parameter for the posterior distribution instead of lumping all the samples together in a summation (this is where we got the k term in the prior case). This case is much

harder to analyze because now every incremental posterior distribution is itself a random function because of the injection of the x random variable. On the other hand, this is more in line with more general Bayesian methods because it is clear that the output of this estimation process is a posterior distribution function, not just a single parameter estimate.

Figure 3.17 illustrates this method. The graph in the top row, far left shows the prior probability ($\beta(6, 6)$) and the dot on the top shows the most recent MAP-estimate for θ. Thus, before we obtain any data, the peak of the prior probability is the estimate. The next graph to right shows the effect of $x_0 = 0$ on the incremental prior probability. Note that the estimate has barely moved to the left. This is because the influence of the data has not caused the prior probability to drift away from the original $\beta(6, 6)$-distribution. The first two rows of the figure all have $x_k = 0$ just to illustrate how far left the original prior probability can be moved by those data. The dots on the tops of the sub-graphs show how the MAP estimate changes frame-by-frame as more data is incorporated. The remaining graphs, proceeding top-down, left-to-right show the incremental change in the prior probability for $x_k = 1$. Again, this shows how far to the right the estimate can be pulled from where it started. For this example, there are an equal number of $x_k = 0$ and $x_k = 1$ data, which corresponds to $\theta = 1/2$.

Programming Tip

Although the IPython Notebook accompanying this section has the full source code, the following is a quick paraphrase of how Fig. 3.17 was constructed. The first step is to recursively create the posteriors from the data. Note the example data is sorted to make the progression easy to see as a sequence.

```
from sympy.abc import p,x
from scipy.stats import density, Beta, Bernoulli
prior = density(Beta('p',6,6))(p)
likelihood=density(Bernoulli('x',p))(x)
data = (0,0,0,0,0,0,0,1,1,1,1,1,1,1,1)
posteriors = [prior]
for i in data:
    posteriors.append(posteriors[-1]*likelihood.subs(x,i))
```

With the posteriors in hand, the next step is to compute the peak values at each frame using the `fminbound` function from Scipy's `optimize` module.

```
pvals = linspace(0,1,100)
mxvals = []
for i,j in zip(ax.flat,posteriors):
    i.plot(pvals,sympy.lambdify(p,j)(pvals),color='k')
    mxval = fminbound(sympy.lambdify(p,-j),0,1)
    mxvals.append(mxval)
    h = i.axis()[-1]
    i.axis(ymax=h*1.3)
    i.plot(mxvals[-1],h*1.2,'ok')
    i.plot(mxvals[:-1],[h*1.2]*len(mxvals[:-1]),'o')
```

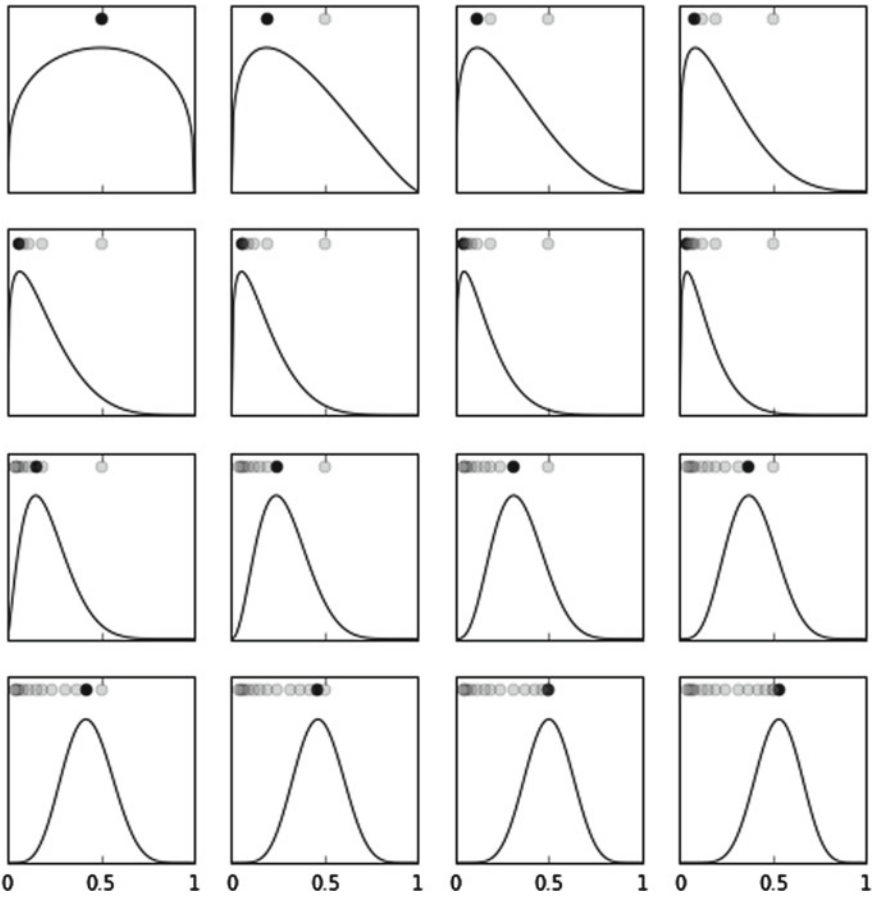

Fig. 3.18 For this example, the prior probability is the $\beta(1.3, 1.3)$ distribution, which has a wider main lobe than the $\beta(6, 6)$ distribution. The *dots* near the peaks of each of the subgraphs indicate the MAP estimate at that frame

The Fig. 3.18 is the same as Fig. 3.17 except that the initial prior probability is the $\beta(1.3, 1.3)$-distribution, which has a wider lobe that the $\beta(6, 6)$-distribution. As shown in the figure, this prior has the ability to be swayed more violently one way or the other based on the x_k data that is incorporated. This means that it can more quickly adapt to data that is not so consistent with the initial prior and thus does not require a large amount of data in order to *unlearn* the prior probability. Depending on the application, the ability to unlearn the prior probability or stick with it is a design problem for the analyst. In this example, because the data are representative of a $\theta = 1/2$ parameter, both priors eventually settle on an estimated posterior that is about the same. However, if this had not been the case ($\theta \neq 1/2$), then the second prior would have produced a better estimate for the same amount of data.[7]

[7] The IPython Notebook corresponding to this chapter contains the source code sot hat you can try different combinations of priors and data values.

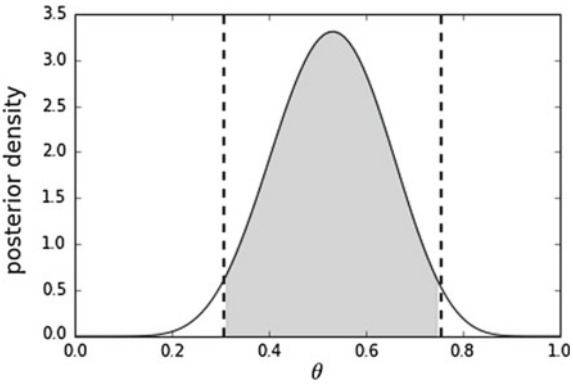

Fig. 3.19 The *credible interval* in Bayesian maximum a-posteriori is the interval corresponding to the *shaded* region in the posterior density

Because we have the entire posterior density available, we can compute something that is closely related to the confidence interval we discussed earlier, except in this situation, given the Bayesian interpretation, it is called a *credible interval* or *credible set*. The idea is that we want to find a symmetric interval around the peak that accounts for 95 % (say) of the posterior density. This means that we can then say the probability that the estimated parameter is within the credible interval is 95 %. The computation requires significant numerical processing because even though we have the posterior density in hand, it is hard to integrate analytically and requires numerical quadrature (see Scipy's `integrate` module). Figure 3.19 shows extent of the interval and the shaded region under the posterior density that accounts for 95 %.

3.9 Robust Statistics

We considered Maximum Likelihood Estimation (MLE) and Maximum A-Posteriori (MAP) estimation and in each case we started out with a probability density function of some kind and we further assumed that the samples were identically distributed and independent (iid). The idea behind robust statistics [3] is to construct estimators that can survive the weakening of either or both of these assumptions. More concretely, suppose you have a model that works great except for a few outliers. The temptation is to just ignore the outliers and proceed. Robust estimation methods provide a disciplined way to handle outliers without cherry-picking data that works for your favored model.

The Notion of Location. The first notion we need is *location*, which is a generalization of the idea of *central value*. Typically, we just use an estimate of the mean for this, but we will see later why this could be a bad idea. The general idea of location satisfies the following requirements Let X be a random variable with distribution F, and let $\theta(X)$ be some descriptive measure of F. Then $\theta(X)$ is said to be a measure of *location* if for any constants a and b, we have the following:

$$\theta(X+b) = \theta(X) + b \tag{3.9.0.1}$$
$$\theta(-X) = -\theta(X) \tag{3.9.0.2}$$
$$X \geq 0 \Rightarrow \theta(X) \geq 0 \tag{3.9.0.3}$$
$$\theta(aX) = a\theta(X) \tag{3.9.0.4}$$

The first condition is called *location equivariance* (or *shift-invariance* in signal processing lingo). The fourth condition is called *scale equivariance*, which means that the units that X is measured in should not effect the value of the location estimator. These requirements capture the intuition of *centrality* of a distribution, or where most of the probability mass is located.

For example, the sample mean estimator is $\hat{\mu} = \frac{1}{n}\sum X_i$. The first requirement is obviously satisfied as $\hat{\mu} = \frac{1}{n}\sum(X_i + b) = b + \frac{1}{n}\sum X_i = b + \hat{\mu}$. Let us consider the second requirement: $\hat{\mu} = \frac{1}{n}\sum -X_i = -\hat{\mu}$. Finally, the last requirement is satisfied with $\hat{\mu} = \frac{1}{n}\sum aX_i = a\hat{\mu}$.

Robust Estimation and Contamination. Now that we have the generalized location of centrality embodied in the *location* parameter, what can we do with it? Previously, we assumed that our samples were all identically distributed. The key idea is that the samples might be actually coming from a *single* distribution that is contaminated by another nearby distribution, as in the following:

$$F(X) = \epsilon G(X) + (1-\epsilon)H(X)$$

where ϵ randomly toggles between zero and one. This means that our data samples $\{X_i\}$ actually derived from two separate distributions, $G(X)$ and $H(X)$. We just don't know how they are mixed together. What we really want is an estimator that captures the location of $G(X)$ in the face of random intermittent contamination by $H(X)$. For example, it may be that this contamination is responsible for the outliers in a model that otherwise works well with the dominant F distribution. It can get even worse than that because we don't know that there is only one contaminating $H(X)$ distribution out there. There may be a whole family of distributions that are contaminating $G(X)$. This means that whatever estimators we construct have to be derived from a more generalized family of distributions instead of from a single distribution, as the maximum-likelihood method assumes. This is what makes robust estimation so difficult—it has to deal with *spaces* of function distributions instead of parameters from a particular probability distribution.

Generalized Maximum Likelihood Estimators. M-estimators are generalized maximum likelihood estimators. Recall that for maximum likelihood, we want to maximize the likelihood function as in the following:

$$L_\mu(x_i) = \prod f_0(x_i - \mu)$$

and then to find the estimator $\hat{\mu}$ so that

$$\hat{\mu} = \arg\max_{\mu} L_{\mu}(x_i)$$

So far, everything is the same as our usual maximum-likelihood derivation except for the fact that we don't assume a specific f_0 as the distribution of the $\{X_i\}$. Making the definition of

$$\rho = -\log f_0$$

we obtain the more convenient form of the likelihood product and the optimal $\hat{\mu}$ as

$$\hat{\mu} = \arg\min_{\mu} \sum \rho(x_i - \mu)$$

If ρ is differentiable, then differentiating this with respect to μ gives

$$\sum \psi(x_i - \hat{\mu}) = 0 \tag{3.9.0.5}$$

with $\psi = \rho'$, the first derivative of ρ, and for technical reasons we will assume that ψ is increasing. So far, it looks like we just pushed some definitions around, but the key idea is we want to consider general ρ functions that may not be maximum likelihood estimators for *any* distribution. Thus, our focus is now on uncovering the nature of $\hat{\mu}$.

Distribution of M-estimates. For a given distribution F, we define $\mu_0 = \mu(F)$ as the solution to the following

$$\mathbb{E}_F(\psi(x - \mu_0)) = 0$$

It is technical to show, but it turns out that $\hat{\mu} \sim \mathcal{N}(\mu_0, \frac{v}{n})$ with

$$v = \frac{\mathbb{E}_F(\psi(x - \mu_0)^2)}{(\mathbb{E}_F(\psi'(x - \mu_0)))^2}$$

Thus, we can say that $\hat{\mu}$ is asymptotically normal with asymptotic value μ_0 and asymptotic variance v. This leads to the efficiency ratio which is defined as the following:

$$\texttt{Eff}(\hat{\mu}) = \frac{v_0}{v}$$

where v_0 is the asymptotic variance of the MLE and measures how near $\hat{\mu}$ is to the optimum. In other words, this provides a sense of how much outlier contamination costs in terms of samples. For example, if for two estimates with asymptotic variances v_1 and v_2, we have $v_1 = 3v_2$, then first estimate requires three times as many observations to obtain the same variance as the second. Furthermore, for the sample

mean (i.e., $\hat{\mu} = \frac{1}{n}\sum X_i$) with $F = \mathcal{N}$, we have $\rho = x^2/2$ and $\psi = x$ and also $\psi' = 1$. Thus, we have $v = \mathbb{V}(x)$. Alternatively, using the sample median as the estimator for the location, we have $v = 1/(4f(\mu_0)^2)$. Thus, if we have $F = \mathcal{N}(0, 1)$, for the sample median, we obtain $v = 2\pi/4 \approx 1.571$. This means that the sample median takes approximately 1.6 times as many samples to obtain the same variance for the location as the sample mean. The sample median is far more immune to the effects of outliers than the sample mean, so this gives a sense of how much this robustness costs in samples.

M-Estimates as Weighted Means. One way to think about M-estimates is a weighted means. Operationally, this means that we want weight functions that can circumscribe the influence of the individual data points, but, when taken as a whole, still provide good estimated parameters. Most of the time, we have $\psi(0) = 0$ and $\psi'(0)$ exists so that ψ is approximately linear at the origin. Using the following definition:

$$W(x) = \begin{cases} \psi(x)/x & \text{if } x \neq 0 \\ \psi'(x) & \text{if } x = 0 \end{cases}$$

We can write our Eq. 3.9.0.5 as follows:

$$\sum W(x_i - \hat{\mu})(x_i - \hat{\mu}) = 0 \tag{3.9.0.6}$$

Solving this for $\hat{\mu}$ yields the following,

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}$$

where $w_i = W(x_i - \hat{\mu})$. This is not practically useful because the w_i contains $\hat{\mu}$, which is what we are trying to solve for. The question that remains is how to pick the ψ functions. This is still an open question, but the Huber functions are a well-studied choice.

Huber Functions. The family of Huber functions is defined by the following:

$$\rho_k(x) = \begin{cases} x^2 & \text{if } |x| \leq k \\ 2k|x| - k^2 & \text{if } |x| > k \end{cases}$$

with corresponding derivatives $2\psi_k(x)$ with

$$\psi_k(x) = \begin{cases} x & \text{if } |x| \leq k \\ \text{sgn}(x)k & \text{if } |x| > k \end{cases}$$

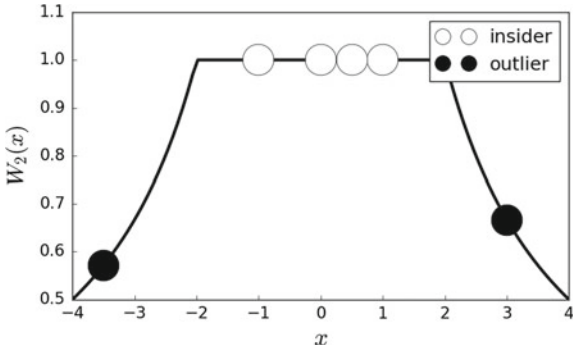

Fig. 3.20 This shows the Huber weight function, $W_2(x)$ and some cartoon data points that are insiders or outsiders as far as the robust location estimate is concerned

where the limiting cases $k \to \infty$ and $k \to 0$ correspond to the mean and median, respectively. To see this, take $\psi_\infty = x$ and therefore $W(x) = 1$ and thus the defining Eq. 3.9.0.6 results in

$$\sum_{i=1}^{n}(x_i - \hat{\mu}) = 0$$

and then solving this leads to $\hat{\mu} = \frac{1}{n}\sum x_i$. Note that choosing $k = 0$ leads to the sample median, but that is not so straightforward to solve for. Nonetheless, Huber functions provide a way to move between two extremes of estimators for location (namely, the mean vs. the median) with a tunable parameter k. The W function corresponding to Huber's ψ is the following:

$$W_k(x) = \min\left\{1, \frac{k}{|x|}\right\}$$

Figure 3.20 shows the Huber weight function for $k = 2$ with some sample points. The idea is that the computed location, $\hat{\mu}$ is computed from Eq. 3.9.0.6 to lie somewhere in the middle of the weight function so that those terms (i.e., *insiders*) have their values fully reflected in the location estimate. The black circles are the *outliers* that have their values attenuated by the weight function so that only a fraction of their presence is represented in the location estimate.

Breakdown Point. So far, our discussion of robustness has been very abstract. A more concrete concept of robustness comes from the breakdown point. In the simplest terms, the breakdown point describes what happens when a single data point in an estimator is changed in the most damaging way possible. For example, suppose we have the sample mean, $\hat{\mu} = \sum x_i/n$, and we take one of the x_i points to be infinite. What happens to this estimator? It also goes infinite. This means that the breakdown point of the estimator is 0 %. On the other hand, the median has a breakdown point of 50 %, meaning that half of the data for computing the median could go infinite without affecting the median value. The median is a *rank* statistic that cares more

about the relative ranking of the data than the values of the data, which explains its robustness.

The simpliest but still formal way to express the breakdown point is to take n data points, $\mathcal{D} = \{(x_i, y_i)\}$. Suppose T is a regression estimator that yields a vector of regression coefficients, $\boldsymbol{\theta}$,

$$T(\mathcal{D}) = \boldsymbol{\theta}$$

Likewise, consider all possible corrupted samples of the data $\mathcal{D}'$. The maximum *bias* caused by this contamination is the following:

$$\texttt{bias}_m = \sup_{\mathcal{D}'} \|T(\mathcal{D}') - T(\mathcal{D})\|$$

where the sup sweeps over all possible sets of m contaminated samples. Using this, the breakdown point is defined as the following:

$$\epsilon_m = \min\left\{\frac{m}{n} : \texttt{bias}_m \to \infty\right\}$$

For example, in our least-squares regression, even one point at infinity causes an infinite T. Thus, for least-squares regression, $\epsilon_m = 1/n$. In the limit $n \to \infty$, we have $\epsilon_m \to 0$.

Estimating Scale. In robust statistics, the concept of *scale* refers to a measure of the dispersion of the data. Usually, we use the estimated standard deviation for this, but this has a terrible breakdown point. Even more troubling, in order to get a good estimate of location, we have to either somehow know the scale ahead of time, or jointly estimate it. None of these methods have easy-to-compute closed form solutions and must be computed numerically.

The most popular method for estimating scale is the *median absolute deviation*

$$\texttt{MAD} = \texttt{Med}(|\mathbf{x} - \texttt{Med}(\mathbf{x})|)$$

In words, take the median of the data $\mathbf{x}$ and then subtract that median from the data itself, and then take the median of the absolute value of the result. Another good dispersion estimate is the *interquartile range*,

$$\texttt{IQR} = x_{(n-m+1)} - x_{(n)}$$

where $m = [n/4]$. The $x_{(n)}$ notation means the nth data element after the data have been sorted. Thus, in this notation, $\texttt{max}(\mathbf{x}) = x_{(n)}$. In the case where $x \sim \mathcal{N}(\mu, \sigma^2)$, then MAD and IQR are constant multiples of σ such that the normalized MAD is the following,

$$\texttt{MADN}(x) = \frac{\texttt{MAD}}{0.675}$$

The number comes from the inverse CDF of the normal distribution corresponding to the 0.75 level. Given the complexity of the calculations, *jointly* estimating both location and scale is a purely numerical matter. Fortunately, the Statsmodels module has many of these ready to use. Let's create some contaminated data in the following code,

```
import statsmodels.api as sm
from scipy import stats
data=np.hstack([stats.norm(10,1).rvs(10),stats.norm(0,1).rvs(100)])
```

These data correspond to our model of contamination that we started this section with. As shown in the histogram in Fig. 3.21, there are two normal distributions, one centered neatly at zero, representing the majority of the samples, and another coming less regularly from the normal distribution on the right. Notice that the group of infrequent samples on the right separates the mean and median estimates (vertical dotted and dashed lines). In the absence of the contaminating distribution on the right, the standard deviation for this data should be close to one. However, the usual non-robust estimate for standard deviation (`np.std`) comes out to approximately three. Using the `MADN` estimator (`sm.robust.scale.mad(data)`) we obtain approximately 1.25. Thus, the robust estimate of dispersion is less moved by the presence of the contaminating distribution.

The generalized maximum likelihood M-estimation extends to joint scale and location estimation using Huber functions. For example,

```
huber = sm.robust.scale.Huber()
loc,scl=huber(data)
```

which implements Huber's *proposal two* method of joint estimation of location and scale. This kind of estimation is the key ingredient to robust regression methods, many of which are implemented in Statsmodels in `statsmodels.formula.api.rlm`. The corresponding documentation has more information.

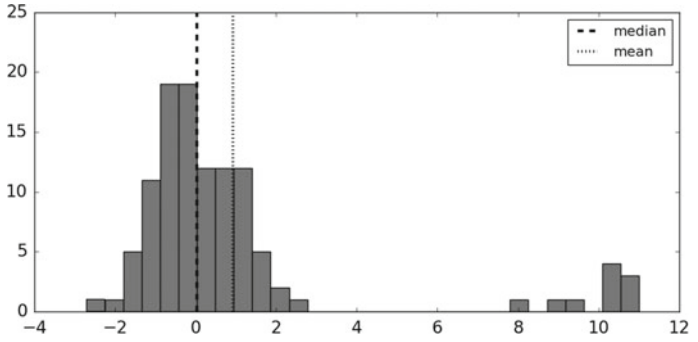

Fig. 3.21 Histogram of sample data. Notice that the group of infrequent samples on the right separates the mean and median estimates indicated by the *vertical lines*

3.10 Bootstrapping

As we have seen, outside of some toy problems, it can be very difficult or impossible to determine the probability density distribution of the estimator of some quantity. The idea behind the bootstrap is that we can use computation to approximate these functions which would otherwise be impossible to solve for analytically.

Let's start with a simple example. Suppose we have the following set of random variables, $\{X_1, X_2, \ldots, X_n\}$ where each $X_k \sim F$. In other words the samples are all drawn from the same unknown distribution F. Having run the experiment, we thereby obtain the following sample set:

$$\{x_1, x_2, \ldots, x_n\}$$

The sample mean is computed from this set as,

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

The next question is how close is the sample mean to the true mean, $\theta = \mathbb{E}_F(X)$. Note that the second central moment of X is as follows:

$$\mu_2(F) := \mathbb{E}_F(X^2) - (\mathbb{E}_F(X))^2$$

The standard deviation of the sample mean, $\bar{x}$, given n samples from an underlying distribution F, is the following:

$$\sigma(F) = (\mu_2(F)/n)^{1/2}$$

Unfortunately, because we have only the set of samples $\{x_1, x_2, \ldots, x_n\}$ and not F itself, we cannot compute this and instead must use the estimated standard error,

$$\bar{\sigma} = (\bar{\mu}_2/n)^{1/2}$$

where $\bar{\mu}_2 = \sum(x_i - \bar{x})^2/(n-1)$, which is the unbiased estimate of $\mu_2(F)$. However, that is not the only way to proceed. Instead, we could replace F by some estimate, $\hat{F}$ obtained as a piecewise function of $\{x_1, x_2, \ldots, x_n\}$ by placing probability mass $1/n$ on each x_i. With that in place, we can compute the estimated standard error as the following:

$$\hat{\sigma}_B = (\mu_2(\hat{F})/n)^{1/2}$$

which is called the *bootstrap estimate* of the standard error. Unfortunately, the story effectively ends here. In even a slightly more general setting, there is no clean formula $\sigma(F)$ within which F can be swapped for $\hat{F}$.

This is where the computer saves the day. We actually do not need to know the formula $\sigma(F)$ because we can compute it using a resampling method. The key idea is to sample with replacement from $\{x_1, x_2, \ldots, x_n\}$. The new set of n independent draws (with replacement) from this set is the *bootstrap sample*,

$$y^* = \{x_1^*, x_2^*, \ldots, x_n^*\}$$

The Monte Carlo algorithm proceeds by first by selecting a large number of bootstrap samples, $\{y_k^*\}$, then computing the statistic on each of these samples, and then computing the sample standard deviation of the results in the usual way. Thus, the bootstrap estimate of the statistic θ is the following,

$$\hat{\theta}_B^* = \frac{1}{B}\sum_k \hat{\theta}^*(k)$$

with the corresponding square of the sample standard deviation as

$$\hat{\sigma}_B^2 = \frac{1}{B-1}\sum_k (\hat{\theta}^*(k) - \hat{\theta}_B^*)^2$$

The process is much simpler than the notation implies. Let's explore this with a simple example using Python. The next block of code sets up some samples from a $\beta(3, 2)$ distribution,

```
>>> import numpy as np
>>> from scipy import stats
>>> rv = stats.beta(3,2)
>>> xsamples = rv.rvs(50)
```

Because this is simulation data, we already know that the mean is $\mu_1 = 3/5$ and the standard deviation of the sample mean for $n = 50$ is $\bar{\sigma} = 1/\sqrt{1250}$, which we will verify this later.

Figure 3.22 shows the $\beta(3, 2)$ distribution and the corresponding histogram of the samples. The histogram represents $\hat{F}$ and is the distribution we sample from to obtain the bootstrap samples. As shown, the $\hat{F}$ is a pretty crude estimate for the F density (smooth solid line), but that's not a serious problem insofar as the following bootstrap estimates are concerned. In fact, the approximation $\hat{F}$ has a naturally tendency to pull towards where most of the probability mass is. This is a feature, not a bug; and is the underlying mechanism for why bootstrapping works, but the formal proofs that exploit this basic idea are far out of our scope here. The next block generates the bootstrap samples

```
>>> yboot = np.random.choice(xsamples,(100,50))
>>> yboot_mn = yboot.mean()
```

and the bootstrap estimate is therefore,

```
>>> np.std(yboot.mean(axis=1)) # approx sqrt(1/1250)
0.025598763883825811
```

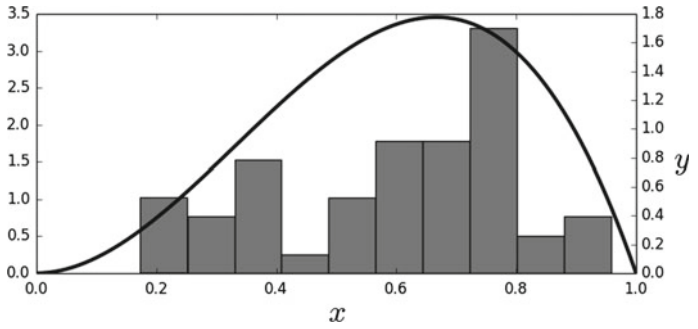

Fig. 3.22 The $\beta(3, 2)$ distribution and the histogram that approximates it

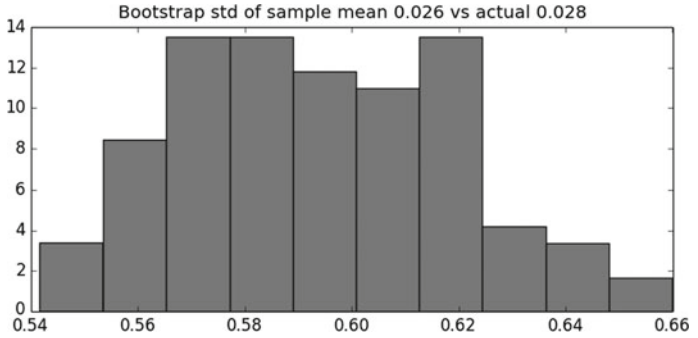

Fig. 3.23 For each bootstrap draw, we compute the sample mean. This is the histogram of those sample means that will be used to compute the bootstrap estimate of the standard deviation

Figure 3.23 shows the distribution of computed sample means from the bootstrap samples. As promised, the next block shows how to use `sympy.stats` to compute the $\beta(3, 2)$ parameters we quoted earlier.

```
>>> import sympy as S
>>> import sympy.stats
>>> for i in range(50): # 50 samples
...     # load sympy.stats Beta random variables
...     # into global namespace using exec
...     execstring = "x%d = S.stats.Beta('x'+str(%d),3,2)"%(i,i)
...     exec(execstring)
...
>>> # populate xlist with the sympy.stats random variables
>>> # from above
>>> xlist = [eval('x%d'%(i)) for i in range(50) ]
>>> # compute sample mean
>>> sample_mean = sum(xlist)/len(xlist)
>>> # compute expectation of sample mean
>>> sample_mean_1 = S.stats.E(sample_mean)
>>> # compute 2nd moment of sample mean
>>> sample_mean_2 = S.stats.E(S.expand(sample_mean**2))
>>> # standard deviation of sample mean
>>> # use sympy sqrt function
>>> sigma_smn = S.sqrt(sample_mean_2-sample_mean_1**2) # 1/sqrt(1250)
>>> print sigma_smn
sqrt(2)/50
```

Programming Tip

Using the `exec` function enables the creation of a sequence of Sympy random variables. Sympy has the `var` function which can automatically create a sequence of Sympy symbols, but there is no corresponding function in the statistics module to do this for random variables.

Example Recall the delta method from Sect. 3.4.2. Suppose we have a set of Bernoulli coin-flips (X_i) with probability of head p. Our maximum likelihood estimator of p is $\hat{p} = \sum X_i/n$ for n flips. We know this estimator is unbiased with $\mathbb{E}(\hat{p}) = p$ and $\mathbb{V}(\hat{p}) = p(1-p)/n$. Suppose we want to use the data to estimate the variance of the Bernoulli trials ($\mathbb{V}(X) = p(1-p)$). By the notation the delta method, $g(x) = x(1-x)$. By the plug-in principle, our maximum likelihood estimator of this variance is then $\hat{p}(1-\hat{p})$. We want the variance of this quantity. Using the results of the delta method, we have

$$\mathbb{V}(g(\hat{p})) = (1-2\hat{p})^2\mathbb{V}(\hat{p})$$
$$\mathbb{V}(g(\hat{p})) = (1-2\hat{p})^2\frac{\hat{p}(1-\hat{p})}{n}$$

Let's see how useful this is with a short simulation.

```
>>> from scipy import stats
>>> import numpy as np
>>> p= 0.25 # true head-up probability
>>> x = stats.bernoulli(p).rvs(10)
>>> print x
[0 0 0 0 0 0 1 0 0 0]
```

The maximum likelihood estimator of p is $\hat{p} = \sum X_i/n$,

```
>>> phat = x.mean()
>>> print phat
0.1
```

Then, plugging this into the delta method approximant above,

```
>>> print (1-2*phat)**2*(phat)**2/10.
0.00064
```

Now, let's try this using the bootstrap estimate of the variance

```
>>> phat_b=np.random.choice(x,(50,10)).mean(1)
>>> print np.var(phat_b*(1-phat_b))
0.005049
```

This shows that the delta method's estimated variance is different from the bootstrap method, but which one is better? For this situation we can solve for this directly using Sympy

```
>>> import sympy as S
>>> from sympy.stats import E, Bernoulli
>>> xdata =[Bernoulli(i,p) for i in S.symbols('x:10')]
>>> ph = sum(xdata)/float(len(xdata))
>>> g = ph*(1-ph)
```

Programming Tip

The argument in the `S.symbols('x:10')` function returns a sequence of Sympy symbols named `x1,x2` and so on. This is shorthand for creating and naming each symbol sequentially.

Note that `g` is the $g(\hat{p}) = \hat{p}(1-\hat{p})$ whose variance we are trying to estimate. Then, we can plug in for the estimated $\hat{p}$ and get the correct value for the variance,

```
>>> print E(g**2) - E(g)**2
0.00442968750000000
```

This case is generally representative—the delta method tends to underestimate the variance and the bootstrap estimate is better here.

3.10.1 Parametric Bootstrap

In the previous example, we used the $\{x_1, x_2, \ldots, x_n\}$ samples themselves as the basis for $\hat{F}$ by weighting each with $1/n$. An alternative is to *assume* that the samples come from a particular distribution, estimate the parameters of that distribution from the sample set, and then use the bootstrap mechanism to draw samples from the assumed distribution, using the so-derived parameters. For example, the next code block does this for a normal distribution.

```
>>> rv = stats.norm(0,2)
>>> xsamples = rv.rvs(45)
>>> # estimate mean and var from xsamples
>>> mn_ = np.mean(xsamples)
>>> std_ = np.std(xsamples)
>>> # bootstrap from assumed normal distribution with
>>> # mn_,std_ as parameters
>>> rvb = stats.norm(mn_,std_) #plug-in distribution
>>> yboot = rvb.rvs(1000)
```

Recall the sample variance estimator is the following:

$$S^2 = \frac{1}{n-1}\sum(X_i - \bar{X})^2$$

Assuming that the samples are normally distributed, this means that $(n-1)S^2/\sigma^2$ has a chi-squared distribution with $n-1$ degrees of freedom. Thus, the variance, $\mathbb{V}(S^2) = 2\sigma^4/(n-1)$. Likewise, the MLE plug-in estimate for this is $\mathbb{V}(S^2) = 2\hat{\sigma}^4/(n-1)$ The following code computes the variance of the sample variance, S^2 using the MLE and bootstrap methods.

```
>>> # MLE-Plugin Variance of the sample mean
>>> print 2*(std_**2)**2/9. # MLE plugin
2.22670148618
>>> # Bootstrap variance of the sample mean
>>> print yboot.var()
3.29467885682
>>> # True variance of sample mean
>>> print 2*(2**2)**2/9.
3.55555555556
```

This shows that the bootstrap estimate is better here than the MLE plugin estimate.

Note that this technique becomes even more powerful with multivariate distributions with many parameters because all the mechanics are the same. Thus, the bootstrap is a great all-purpose method for computing standard errors, but, in the limit, is it converging to the correct value? This is the question of *consistency*. Unfortunately, to answer this question requires more and deeper mathematics than we can get into here. The short answer is that for estimating standard errors, the bootstrap is a consistent estimator in a wide range of cases and so it definitely belongs in your toolkit.

3.11 Gauss Markov

In this section, we consider the famous Gauss-Markov problem which will give us an opportunity to use all the material we have so far developed. The Gauss-Markov model is the fundamental model for noisy parameter estimation because it estimates unobservable parameters given a noisy indirect measurement. Incarnations of the same model appear in all studies of Gaussian models. This case is an excellent opportunity to use everything we have so far learned about projection and conditional expectation.

Following Luenberger [4] let's consider the following problem:

$$\mathbf{y} = \mathbf{W}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where $\mathbf{W}$ is a $n \times m$ matrix, and $\mathbf{y}$ is a $n \times 1$ vector. Also, $\boldsymbol{\epsilon}$ is a n-dimensional normally distributed random vector with zero-mean and covariance,

$$\mathbb{E}(\boldsymbol{\epsilon}\boldsymbol{\epsilon}^T) = \mathbf{Q}$$

Note that engineering systems usually provide a *calibration mode* where you can estimate $\mathbf{Q}$ so it's not fantastical to assume you have some knowledge of the noise statistics. The problem is to find a matrix $\mathbf{K}$ so that $\hat{\boldsymbol{\beta}} = \mathbf{K}\mathbf{y}$ approximates $\boldsymbol{\beta}$. Note that we only have knowledge of $\boldsymbol{\beta}$ via $\mathbf{y}$ so we can't measure it directly. Further, note that $\mathbf{K}$ is a matrix, not a vector, so there are $m \times n$ entries to compute.

We can approach this problem the usual way by trying to solve the MMSE problem:

$$\min_K \mathbb{E}(\|\hat{\boldsymbol{\beta}} - \boldsymbol{\beta}\|^2)$$

which we can write out as

$$\min_K \mathbb{E}(\|\hat{\boldsymbol{\beta}} - \boldsymbol{\beta}\|^2) = \min_K \mathbb{E}(\|\mathbf{K}\mathbf{y} - \boldsymbol{\beta}\|^2) = \min_K \mathbb{E}(\|\mathbf{K}\mathbf{W}\boldsymbol{\beta} + \mathbf{K}\boldsymbol{\epsilon} - \boldsymbol{\beta}\|^2)$$

and since $\boldsymbol{\epsilon}$ is the only random variable here, this simplifies to

$$\min_K \|\mathbf{K}\mathbf{W}\boldsymbol{\beta} - \boldsymbol{\beta}\|^2 + \mathbb{E}(\|\mathbf{K}\boldsymbol{\epsilon}\|^2)$$

The next step is to compute

$$\mathbb{E}(\|\mathbf{K}\boldsymbol{\epsilon}\|^2) = \mathbb{E}(\boldsymbol{\epsilon}^T\mathbf{K}^T\mathbf{K}^T\boldsymbol{\epsilon}) = \mathrm{Tr}(\mathbf{K}\mathbb{E}(\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathbf{T}})\mathbf{K}^T) = \mathrm{Tr}(\mathbf{K}\mathbf{Q}\mathbf{K}^T)$$

using the properties of the trace of a matrix. We can assemble everything as

$$\min_K \|\mathbf{K}\mathbf{W}\boldsymbol{\beta} - \boldsymbol{\beta}\|^2 + \mathrm{Tr}(\mathbf{K}\mathbf{Q}\mathbf{K}^T)$$

Now, if we were to solve this for $\mathbf{K}$, it would be a function of $\boldsymbol{\beta}$, which is the same thing as saying that the estimator, $\hat{\boldsymbol{\beta}}$, is a function of what we are trying to estimate, $\boldsymbol{\beta}$, which makes no sense. However, writing this out tells us that if we had $\mathbf{K}\mathbf{W} = \mathbf{I}$, then the first term vanishes and the problem simplifies to

$$\min_K \mathrm{Tr}(\mathbf{K}\mathbf{Q}\mathbf{K}^T)$$

with the contraint,

$$\mathbf{K}\mathbf{W} = \mathbf{I}$$

This requirement is the same as asserting that the estimator is unbiased,

$$\mathbb{E}(\hat{\boldsymbol{\beta}}) = \mathbf{K}\mathbf{W}\boldsymbol{\beta} = \boldsymbol{\beta}$$

To line this problem up with our earlier work, let's consider the ith column of $\mathbf{K}$, $\mathbf{k}_i$. Now, we can re-write the problem as

$$\min_k(\mathbf{k}_i^T\mathbf{Q}\mathbf{k}_i)$$

with

$$\mathbf{k}_i^T \mathbf{W} = \mathbf{e}_i$$

and from our previous work on contrained optimization, we know the solution to this:

$$\mathbf{k}_i = \mathbf{Q}^{-1}\mathbf{W}(\mathbf{W}^T\mathbf{Q}^{-1}\mathbf{W})^{-1}\mathbf{e}_i$$

Now all we have to do is stack these together for the general solution:

$$\mathbf{K} = (\mathbf{W}^T\mathbf{Q}^{-1}\mathbf{W})^{-1}\mathbf{W}^T\mathbf{Q}^{-1}$$

It's easy when you have all of the concepts lined up! For completeness, the covariance of the error is

$$\mathbb{E}(\hat{\boldsymbol{\beta}} - \boldsymbol{\beta})(\hat{\boldsymbol{\beta}} - \boldsymbol{\beta})^T = \mathbb{E}(\mathbf{K}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^T\mathbf{K}^T) = \mathbf{K}\mathbf{Q}\mathbf{K}^T = (\mathbf{W}^T\mathbf{Q}^{-1}\mathbf{W})^{-1}$$

Figure 3.24 shows the simulated $\mathbf{y}$ data as red circles. The black dots show the corresponding estimates, $\hat{\boldsymbol{\beta}}$ for each sample. The black lines show the true value of $\boldsymbol{\beta}$ versus the average of the estimated $\boldsymbol{\beta}$-values, $\widehat{\boldsymbol{\beta}_m}$. The matrix $\mathbf{K}$ maps the red circles in the corresponding dots. Note there are many possible ways to map the red circles to the plane, but the $\mathbf{K}$ is the one that minimizes the MSE for $\boldsymbol{\beta}$.

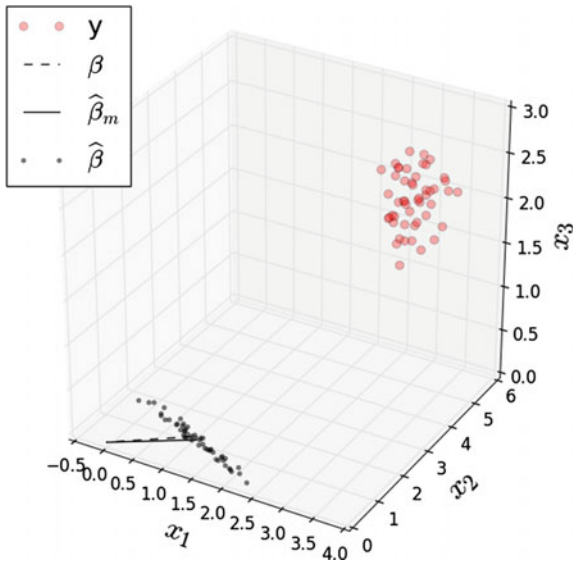

Fig. 3.24 The *red circles* show the points to be estimated in the *xy*-plane by the *black points*

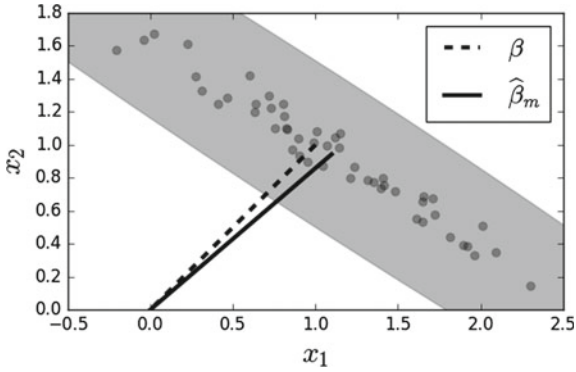

Fig. 3.25 Focusing on the *xy*-plane in Fig. 3.24, the *dashed line* shows the true value for $\boldsymbol{\beta}$ versus the mean of the estimated values $\widehat{\boldsymbol{\beta}}_m$

Programming Tip

Although the full source code is available in the corresponding IPython Notebook, the following snippets provide a quick walkthrough. To simulate the target data, we define the relevant matrices below,

```
Q = np.eye(3)*0.1 # error covariance matrix
# this is what we are trying estimate
beta = matrix(ones((2,1)))
W = matrix([[1,2],
            [2,3],
            [1,1]])
```

Then, we generate the noise terms and create the simulated data, y,

```
ntrials = 50
epsilon = np.random.multivariate_normal((0,0,0),Q,ntrials).T
y=W*beta+epsilon
```

Figure 3.25 shows more detail in the horizontal *xy*-plane of Fig. 3.24. Figure 3.25 shows the dots, which are individual estimates of $\hat{\boldsymbol{\beta}}$ from the corresponding simulated $\mathbf{y}$ data. The dashed line is the true value for $\boldsymbol{\beta}$ and the filled line ($\widehat{\boldsymbol{\beta}}_m$) is the average of all the dots. The gray ellipse provides an error ellipse for the covariance of the estimated $\boldsymbol{\beta}$ values.

Programming Tip

Although the full source code is available in the corresponding IPython Notebook, the following snippets provide a quick walkthrough of the construction of Fig. 3.25. To draw the ellipse, we need to import the patch primitive,

```
from  matplotlib.patches import Ellipse
```

To compute the parameters of the error ellipse based on the covariance matrix of the individual estimates of $\boldsymbol{\beta}$ in the `bm_cov` variable below,

```
U,S,V = linalg.svd(bm_cov)
err = np.sqrt((matrix(bm))*(bm_cov)*(matrix(bm).T))
theta = np.arccos(U[0,1])/np.pi*180
```

Then, we draw the add the scaled ellipse in the following,

```
ax.add_patch(Ellipse(bm,err*2/np.sqrt(S[0]),
                     err*2/np.sqrt(S[1]),
                     angle=theta,color='gray'))
```

3.12 Nonparametric Methods

So far, we have considered parametric methods that reduce inference or prediction to parameter-fitting. However, for these to work, we had to assume a specific functional form for the unknown probability distribution of the data. Nonparametric methods eliminate the need to assume a specific functional form by generalizing to classes of functions.

3.12.1 Kernel Density Estimation

We have already made heavy use of this method with the histogram, which is a special case of kernel density estimation. The histogram can be considered the crudest and most useful nonparametric method, that estimates the underlying probability distribution of the data.

To be formal and place the histogram on the same footing as our earlier estimations, suppose that $\mathcal{X} = [0, 1]^d$ is the d dimensional unit cube and that h is the *bandwidth* or size of a *bin* or sub-cube. Then, there are $N \approx (1/h)^d$ such bins, each with volume h^d, $\{B_1, B_2, \ldots, B_N\}$. With all this in place, we can write the histogram has a probability density estimator of the form,

$$\hat{p}_h(x) = \sum_{k=1}^{N} \frac{\hat{\theta}_k}{h} I(x \in B_k)$$

where

$$\hat{\theta}_k = \frac{1}{n} \sum_{j=1}^{n} I(X_j \in B_k)$$

is the fraction of data points (X_k) in each bin, B_k. We want to bound the bias and variance of $\hat{p}_h(x)$. Keep in mind that we are trying to estimate a function of x, but the set of all possible probability distribution functions is extremely large and hard to

manage. Thus, we need to restrict our attention to the following class of probability distribution of so-called Lipschitz functions,

$$\mathcal{P}(L) = \{p : |p(x) - p(y)| \leq L\|x - y\|, \forall\, x, y\}$$

Roughly speaking, these are the density functions whose slopes (i.e., growth rates) are bounded by L. It turns out that the bias of the histogram estimator is bounded in the following way,

$$\int |p(x) - \mathbb{E}(\hat{p}_h(x))| dx \leq Lh\sqrt{d}$$

Similarly, the variance is bounded by the following,

$$\mathbb{V}(\hat{p}_h(x)) \leq \frac{C}{nh^d}$$

for some constant C. Putting these two facts together means that the risk is bounded by,

$$R(p, \hat{p}) = \int \mathbb{E}(p(x) - \hat{p}_h(x))^2 dx \leq L^2 h^2 d + \frac{C}{nh^d}$$

This upper bound is minimized by choosing

$$h = \left(\frac{C}{L^2 nd}\right)^{\frac{1}{d+2}}$$

In particular, this means that,

$$\sup_{p \in \mathcal{P}(L)} R(p, \hat{p}) \leq C_0 \left(\frac{1}{n}\right)^{\frac{2}{d+2}}$$

where the constant C_0 is a function of L. There is a theorem [2] that shows this bound in tight, which basically means that the histogram is a really powerful probability density estimator for Lipschitz functions with risk that goes as $\left(\frac{1}{n}\right)^{\frac{2}{d+2}}$. Note that this class of functions is not necessarily smooth because the Lipschitz condition admits non-smooth functions. While this is a reassuring result, we typically do not know which function class (Lipschitz or not) a particular probability belongs to ahead of time. Nonetheless, the rate at which the risk changes with both dimension d and n samples would be hard to understand without this result. Figure 3.26 shows the probability distribution function of the $\beta(2, 2)$ distribution compared to computed histograms for different values of n. The box plots on each of the points show how the variation in each bin of the histogram reduces with increasing n. The risk function $R(p, \hat{p})$ above is based upon integrating the squared difference between the histogram (as a piecewise function of x) and the probability distribution function.

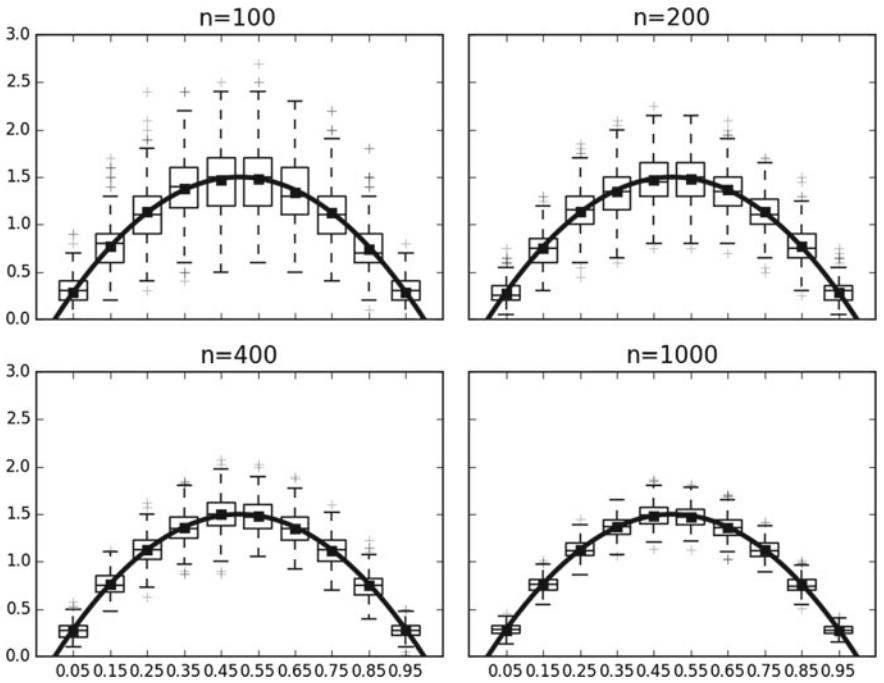

Fig. 3.26 The *box plots* on each of the points show how the variation in each bin of the histogram reduces with increasing n

Programming Tip

The corresponding IPython notebook has the complete source code that generates Fig. 3.26; however, the following snippet is the main element of the code.

```
def generate_samples(n,ntrials=500):
    phat = np.zeros((nbins,ntrials))
    for k in range(ntrials):
        d = rv.rvs(n)
        phat[:,k],_=histogram(d,bins,density=True)
    return phat
```

The code uses the `histogram` function from Numpy. To be consistent with the risk function $R(p,\hat{p})$, we have to make sure the `bins` keyword argument is formatted correctly using a sequence of bin-edges instead of just a single integer. Also, the `density=True` keyword argument normalizes the histogram appropriately so that the comparison between it and the probability distribution function of the simulated beta distribution is correctly scaled.

3.12.2 Kernel Smoothing

We can extend our methods to other function classes using kernel functions. A one-dimensional smoothing kernel is a smooth function K with the following properties,

$$\begin{aligned} \int K(x)dx &= 1 \\ \int xK(x)dx &= 0 \\ 0 < \int x^2 K(x)dx &< \infty \end{aligned}$$

For example, $K(x) = I(x)/2$ is the boxcar kernel, where $I(x) = 1$ when $|x| \leq 1$ and zero otherwise. The kernel density estimator is very similar to the histogram, except now we put a kernel function on every point as in the following,

$$\hat{p}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h^d} K\left(\frac{\|x - X_i\|}{h}\right)$$

where $X \in \mathbb{R}^d$. Figure 3.27 shows an example of a kernel density estimate using a Gaussian kernel function, $K(x) = e^{-x^2/2}/\sqrt{2\pi}$. There are five data points shown by the vertical lines in the upper panel. The dotted lines show the individual $K(x)$ function at each of the data points. The lower panel shows the overall kernel density estimate, which is the scaled sum of the upper panel.

There is an important technical result in [2] that states that kernel density estimators are minimax in the sense we discussed in the maximum likelihood Sect. 3.4. In broad strokes, this means that the analogous risk for the kernel density estimator is approximately bounded by the following factor,

$$R(p, \hat{p}) \lesssim n^{-\frac{2m}{2m+d}}$$

for some constant C where m is a factor related to bounding the derivatives of the probability density function. For example, if the second derivative of the density function is bounded, then $m = 2$. This means that the convergence rate for this estimator decreases with increasing dimension d.

Cross-Validation. As a practical matter, the tricky part of the kernel density estimator (which includes the histogram as a special case) is that we need to somehow compute the bandwidth h term using data. There are several rule-of-thumb methods that for some common kernels, including Silverman's rule and Scott's rule for Gaussian kernels. For example, Scott's factor is to simply compute $h = n^{-1/(d+4)}$ and Silverman's is $h = (n(d+2)/4)^{(-1/(d+4))}$. Rules of this kind are derived by assuming

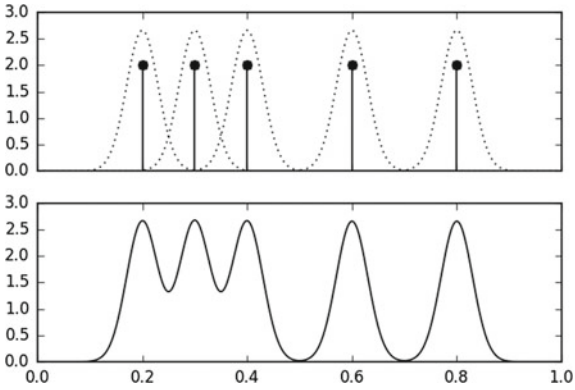

Fig. 3.27 The *upper panel* shows the individual kernel functions placed at each of the data points. The *lower panel* shows the composite kernel density estimate which is the sum of the individual functions in the *upper panel*

the underlying probability density function is of a certain family (e.g., Gaussian), and then deriving the best h for a certain type of kernel density estimator, usually equipped with extra functional properties (say, continuous derivatives of a certain order). In practice, these rules seem to work pretty well, especially for uni-modal probability density functions. Avoiding these kinds of assumptions means computing the bandwidth from data directly and that is where cross validation comes in.

Cross-validation is a method to estimate the bandwidth from the data itself. The idea is to write out the following Integrated Squared Error (ISE),

$$\begin{aligned}\text{ISE}(\hat{p}_h, p) &= \int (p(x) - \hat{p}_h(x))^2 dx \\ &= \int \hat{p}_h(x)^2 dx - 2\int p(x)\hat{p}_h dx + \int p(x)^2 dx\end{aligned}$$

The problem with this expression is the middle term,[8]

$$\int p(x)\hat{p}_h dx$$

where $p(x)$ is what we are trying to estimate with $\hat{p}_h$. The form of the last expression looks like an expectation of $\hat{p}_h$ over the density of $p(x)$, $\mathbb{E}(\hat{p}_h)$. The approach is to approximate this with the mean,

$$\mathbb{E}(\hat{p}_h) \approx \frac{1}{n}\sum_{i=1}^{n} \hat{p}_h(X_i)$$

The problem with this approach is that $\hat{p}_h$ is computed using the same data that the approximation utilizes. The way to get around this is to split the data into two equally sized chunks D_1, D_2; and then compute $\hat{p}_h$ for a sequence of different h values over

[8]The last term is of no interest because we are only interested in relative changes in the ISE.

the D_1 set. Then, when we apply the above approximation for the data (Z_i) in the D_2 set,

$$\mathbb{E}(\hat{p}_h) \approx \frac{1}{|D_2|} \sum_{Z_i \in D_2} \hat{p}_h(Z_i)$$

Plugging this approximation back into the integrated squared error provides the objective function,

$$\text{ISE} \approx \int \hat{p}_h(x)^2 dx - \frac{2}{|D_2|} \sum_{Z_i \in D_2} \hat{p}_h(Z_i)$$

Some code will make these steps concrete. We will need some tools from Scikit-learn.

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.neighbors.kde import KernelDensity
```

The `train_test_split` function makes it easy to split and keep track of the D_1 and D_2 sets we need for cross validation. Scikit-learn already has a powerful and flexible implementation of kernel density estimators. To compute the objective function, we need some basic numerical integration tools from Scipy. For this example, we will generate samples from a $\beta(2, 2)$ distribution, which is implemented in the `stats` submodule in Scipy.

```
>>> from scipy.integrate import quad
>>> from scipy import stats
>>> rv= stats.beta(2,2)
>>> n=100                 # number of samples to generate
>>> d = rv.rvs(n)[:,None] # generate samples as column-vector
```

Programming Tip

The use of the `[:,None]` in the last line formats the Numpy array returned by the `rvs` function into a Numpy vector with a column dimension of one. This is required by the `KernelDensity` constructor because the column dimension is used for different features (in general) for Scikit-learn. Thus, even though we only have one feature, we still need to comply with the structured input that Scikit-learn relies upon. There are many ways to inject the additional dimension other than using `None`. For example, the more cryptic, `np.c_`, or the less cryptic `[:,np.newaxis]` can do the same, as can the `np.reshape` function.

The next step is to split the data into two halves and loop over each of the h_i bandwidths to create a separate kernel density estimator based on the D_1 data,

```
>>> train,test,_,_=train_test_split(d,d,test_size=0.5)
>>> kdes=[KernelDensity(bandwidth=i).fit(train)
...         for i in [.05,0.1,0.2,0.3]]
```

Programming Tip

Note that the single underscore symbol in Python refers to the last evaluated result. the above code unpacks the tuple returned by `train_test_split` into four elements. Because we are only interested in the first two, we assign the last two to the underscore symbol. This is a stylistic usage to make it clear to the reader that the last two elements of the tuple are unused. Alternatively, we could assign the last two elements to a pair of dummy variables that we do not use later, but then the reader skimming the code may think that those dummy variables are relevant.

The last step is to loop over the so-created kernel density estimators and compute the objective function.

```
>>> import numpy as np
>>> for i in kdes:
...     f = lambda x: np.exp(i.score_samples(x))
...     f2 = lambda x: f(x)**2
...     print 'h=%3.2f\t %3.4f'%(i.bandwidth,quad(f2,0,1)[0]
...                               -2*np.mean(f(test)))
...
h=0.05   -1.1323
h=0.10   -1.1336
h=0.20   -1.1330
h=0.30   -1.0810
```

Programming Tip

The lambda functions defined in the last block are necessary because Scikit-learn implements the return value of the kernel density estimator as a logarithm via the `score_samples` function. The numerical quadrature function `quad` from Scipy computes the $\int \hat{p}_h(x)^2 dx$ part of the objective function.

Scikit-learn has many more advanced tools to automate this kind of hyper-parameter (i.e., kernel density bandwidth) search. To utilize these advanced tools, we need to format the current problem slightly differently by defining the following wrapper class (Fig. 3.28).

```
>>> class KernelDensityWrapper(KernelDensity):
...     def predict(self,x):
...         return np.exp(self.score_samples(x))
...     def score(self,test):
...         f = lambda x: self.predict(x)
...         f2 = lambda x: f(x)**2
...         return -(quad(f2,0,1)[0]-2*np.mean(f(test)))
...
```

This is tantamount to reorganizing the above previous code into functions that Scikit-learn requires. Next, we create the dictionary of parameters we want to search over (`params`) below and then start the grid search with the `fit` function,

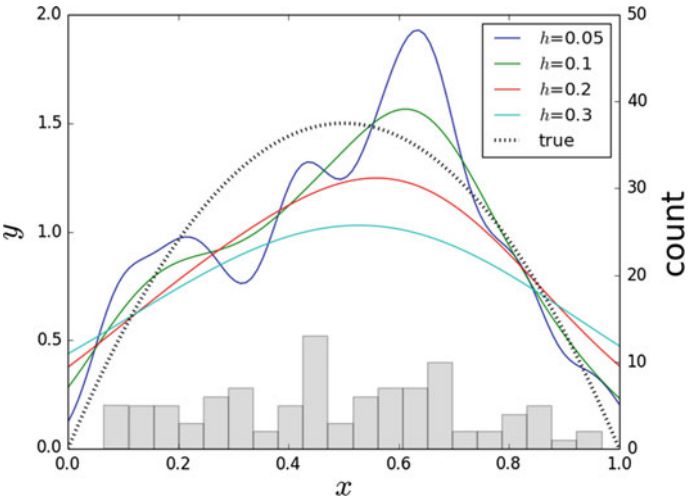

Fig. 3.28 Each *line* above is a different kernel density estimator for the given bandwidth as an approximation to the true density function. A plain histogram is imprinted on the *bottom* for reference

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {'bandwidth':np.linspace(0.01,0.5,10)}
>>> clf = GridSearchCV(KernelDensityWrapper(), param_grid=params,cv=2)
>>> clf.fit(d)
GridSearchCV(cv=2,
estimator=KernelDensityWrapper(algorithm='auto',atol=0,bandwidth=1.0,
breadth_first=True,kernel='gaussian',leaf_size=40,
metric='euclidean',metric_params=None,rtol=0),
fit_params={},iid=True,loss_func=None,n_jobs=1,
param_grid={'bandwidth':array([0.01,0.06444,0.11889,0.17333,0.22778,0.28222,
0.33667,0.39111,0.44556,0.5])},
pre_dispatch='2*n_jobs',refit=True,score_func=None,scoring=None, verbose=0)
>>> print clf.best_params_
{'bandwidth': 0.17333333333333334}
```

The grid search iterates over all the elements in the `params` dictionary and reports the best bandwidth over that list of parameter values. The `cv` keyword argument above specifies that we want to split the data into two equally-sized sets for training and testing. We can also examine the values of the objective function for each point on the grid as follow,

```
>>> from pprint import pprint
>>> pprint(clf.grid_scores_)
[mean: 0.60758, std: 0.07695, params: {'bandwidth': 0.01},
 mean: 1.06325, std: 0.03866, params: {'bandwidth': 0.064444444444444443},
 mean: 1.11859, std: 0.02093, params: {'bandwidth': 0.11888888888888888},
 mean: 1.13187, std: 0.01397, params: {'bandwidth': 0.17333333333333334},
 mean: 1.12007, std: 0.01043, params: {'bandwidth': 0.22777777777777777},
 mean: 1.09186, std: 0.00794, params: {'bandwidth': 0.28222222222222221},
 mean: 1.05391, std: 0.00601, params: {'bandwidth': 0.33666666666666667},
 mean: 1.01126, std: 0.00453, params: {'bandwidth': 0.39111111111111108},
 mean: 0.96717, std: 0.00341, params: {'bandwidth': 0.44555555555555554},
 mean: 0.92355, std: 0.00257, params: {'bandwidth': 0.5}]
```

Programming Tip

The `pprint` function makes the standard output prettier. The only reason for using it here is to get it to look good on the printed page. Otherwise, the IPython notebook handles the visual rendering of output embedded in the notebook via its internal `display` framework.

Keep in mind that the grid search examines multiple folds for cross validation to compute the above means and standard deviations. Note that there is also a `RandomizedSearchCV` in case you would rather specify a distribution of parameters instead of a list. This is particularly useful for searching very large parameter spaces where an exhaustive grid search would be too computationally expensive. Although kernel density estimators are easy to understand and have many attractive analytical properties, they become practically prohibitive for large, high-dimensional data sets.

3.12.3 Nonparametric Regression Estimators

Beyond estimating the underlying probability density, we can use nonparametric methods to compute estimators of the underlying function that is generating the data. Nonparametric regression estimators of the following form are known as linear smoothers,

$$\hat{y}(x) = \sum_{i=1}^{n} \ell_i(x) y_i$$

To understand the performance of these smoothers, we can define the risk as the following,

$$R(\hat{y}, y) = \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n}(\hat{y}(x_i) - y(x_i))^2\right)$$

and find the best $\hat{y}$ that minimizes this. The problem with this metric is that we do not know $y(x)$, which is why we are trying to approximate it with $\hat{y}(x)$. We could construct an estimation by using the data at hand as in the following,

$$\hat{R}(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}(x_i) - Y_i)^2$$

where we have substituted the data Y_i for the unknown function value, $y(x_i)$. The problem with this approach is that we are using the data to estimate the function and

then using the same data to evaluate the risk of doing so. This kind of double-dipping leads to overly optimistic estimators. One way out of this conundrum is to use leave-one-out cross validation, wherein the $\hat{y}$ function is estimated using all but one of the data pairs, (X_i, Y_i). Then, this missing data element is used to estimate the above risk. Notationally, this is written as the following,

$$\hat{R}(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{(-i)}(x_i) - Y_i)^2$$

where $\hat{y}_{(-i)}$ denotes computing the estimator without using the ith data pair. Unfortunately, for anything other than relatively small data sets, it quickly becomes computationally prohibitive to use leave-one-out cross validation in practice. We'll get back to this issue shortly, but let's consider a concrete example of such a nonparametric smoother.

3.12.4 Nearest Neighbors Regression

The simplest possible nonparametric regression method is the k-nearest neighbors regression. This is easier to explain in words than to write out in math. Given an input x, find the closest one of the k clusters that contains it and then return the mean of the data values in that cluster. As a univariate example, let's consider the following *chirp* waveform,

$$y(x) = \cos\left(2\pi\left(f_o x + \frac{BWx^2}{2\tau}\right)\right)$$

This waveform is important in high-resolution radar applications. The f_o is the start frequency and BW/τ is the frequency slope of the signal. For our example, the fact that it is nonuniform over its domain is important. We can easily create some data by sampling the chirp as in the following,

```
>>> import numpy as np
>>> from numpy import cos, pi
>>> xi = np.linspace(0,1,100)[:,None]
>>> xin = np.linspace(0,1,12)[:,None]
>>> f0 = 1 # init frequency
>>> BW = 5
>>> y = cos(2*pi*(f0*xin+(BW/2.0)*xin**2))
```

We can use this data to construct a simple nearest neighbor estimator using Scikit-learn,

```
>>> from sklearn.neighbors import KNeighborsRegressor
>>> knr=KNeighborsRegressor(2)
>>> knr.fit(xin,y)
KNeighborsRegressor(algorithm='auto',leaf_size=30,metric='minkowski',
metric_params=None,n_neighbors=2,p=2,weights='uniform')
```

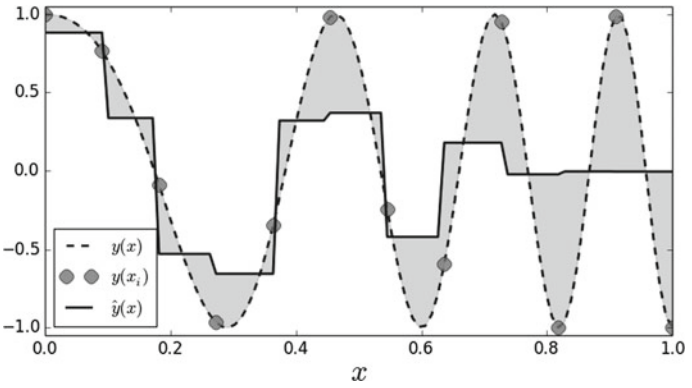

Fig. 3.29 The *dotted line* shows the chirp signal and the *solid line* shows the nearest neighbor estimate. The *gray circles* are the sample points that we used to fit the nearest neighbor estimator. The *shaded area* shows the gaps between the estimator and the unsampled chirp

Programming Tip

Scikit-learn has a fantastically consistent interface. The `fit` function above fits the model parameters to the data. The corresponding `predict` function returns the output of the model given an arbitrary input. We will spend a lot more time on Scikit-learn in the machine learning chapter. The `[:,None]` part at the end is just injecting a column dimension into the array in order to satisfy the dimensional requirements of Scikit-learn.

Figure 3.29 shows the sampled signal (gray circles) against the values generated by the nearest neighbor estimator (solid line). The dotted line is the full unsampled chirp signal, which increases in frequency with x. This is important for our example because it adds a non-stationary aspect to this problem in that the function gets progressively wigglier with increasing x. The area between the estimated curve and the signal is shaded in gray. Because the nearest neighbor estimator uses only two nearest neighbors, for each new x, it finds the two adjacent X_i that bracket the x in the training data and then averages the corresponding Y_i values to compute the estimated value. That is, if you take every adjacent pair of sequential gray circles in the Figure, you find that the horizontal solid line splits the pair on the vertical axis. We can adjust the number of nearest neighbors by changing the constructor,

```
>>> knr=KNeighborsRegressor(3)
>>> knr.fit(xin,y)
KNeighborsRegressor(algorithm='auto',leaf_size=30,metric='minkowski',
metric_params=None,n_neighbors=3,p=2,weights='uniform')
```

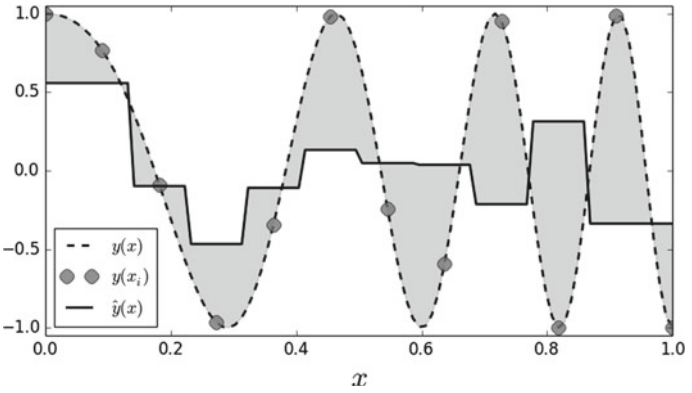

Fig. 3.30 This is the same as Fig. 3.29 except that here there are three nearest neighbors used to build the estimator

which produces the following corresponding Fig. 3.30.

For this example, Fig. 3.30 shows that with more nearest neighbors the fit performs poorly, especially towards the end of the signal, where there is increasing variation, because the chirp is not uniformly continuous.

Scikit-learn provides many tools for cross validation. The following code sets up the tools for leave-one-out cross validation,

```
>>> from sklearn.cross_validation import LeaveOneOut
>>> loo=LeaveOneOut(len(xin))
```

The `LeaveOneOut` object is an iterable that produces a set of disjoint indices of the data—one for fitting the model (training set) and one for evaluating the model (testing set), as shown in the next short sample,

```
>>> pprint(list(LeaveOneOut(3)))
[(array([1, 2]), array([0])),
 (array([0, 2]), array([1])),
 (array([0, 1]), array([2]))]
```

The next block loops over the disjoint sets of training and test indicies iterates provided by the `loo` variable to evaluate the estimated risk, which is accumulated in the `out` list.

```
>>> out=[]
>>> for train_index, test_index in loo:
...     _=knr.fit(xin[train_index],y[train_index])
...     out.append((knr.predict(xi[test_index])-y[test_index])**2)
...
>>> print 'Leave-one-out Estimated Risk: ',np.mean(out),
Leave-one-out Estimated Risk:  1.03517136627
```

The last line in the code above reports leave-one-out's estimated risk.

Linear smoothers of this type can be rewritten in using the following matrix,

$$\mathcal{S} = \left[\ell_i(x_j)\right]_{i,j}$$

so that

$$\hat{\mathbf{y}} = \mathcal{S}\mathbf{y}$$

where $\mathbf{y} = [Y_1, Y_2, \ldots, Y_n] \in \mathbb{R}^n$ and $\hat{\mathbf{y}} = \left[\hat{y}(x_1), \hat{y}(x_2), \ldots, \hat{y}(x_n)\right] \in \mathbb{R}^n$. This leads to a quick way to approximate leave-one-out cross validation as the following,

$$\hat{R} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}(x_i)}{1 - \mathcal{S}_{i,i}}\right)^2$$

However, this does not reproduce the approach in the code above because it assumes that each $\hat{y}_{(-i)}(x_i)$ is consuming one fewer nearest neighbor than $\hat{y}(x)$.

We can get this $\mathcal{S}$ matrix from the `knr` object as in the following,

```
>>> _= knr.fit(xin,y) # fit on all data
>>> S=(knr.kneighbors_graph(xin)).todense()/float(knr.n_neighbors)
```

The `todense` part reformats the sparse matrix that is returned into a regular Numpy `matrix`. The following shows a subsection of this $\mathcal{S}$ matrix,

```
>>> print S[:5,:5]
[[ 0.33333333  0.33333333  0.33333333  0.          0.        ]
 [ 0.33333333  0.33333333  0.33333333  0.          0.        ]
 [ 0.          0.33333333  0.33333333  0.33333333  0.        ]
 [ 0.          0.          0.33333333  0.33333333  0.33333333]
 [ 0.          0.          0.          0.33333333  0.33333333]]
```

The sub-blocks show the windows of the `y` data that are being processed by the nearest neighbor estimator. For example,

```
>>> print np.hstack([knr.predict(xin[:5]),(S*y)[:5]])#columns match
[[ 0.55781314  0.55781314]
 [ 0.55781314  0.55781314]
 [-0.09768138 -0.09768138]
 [-0.46686876 -0.46686876]
 [-0.10877633 -0.10877633]]
```

Or, more concisely checking all entries for approximate equality,

```
>>> print np.allclose(knr.predict(xin),S*y)
True
```

which shows that the results from the nearest neighbor object and the matrix multiply match.

Programming Tip

Note that because we formatted the returned $\mathcal{S}$ as a Numpy matrix, we automatically get the matrix multiplication instead of default element-wise multiplication in the `S*y` term.

3.12.5 Kernel Regression

For estimating the probability density, we started with the histogram and moved to the more general kernel density estimate. Likewise, we can also extend regression from nearest neighbors to kernel-based regression using the *Nadaraya-Watson* kernel regression estimator. Given a bandwidth $h > 0$, the kernel regression estimator is defined as the following,

$$\hat{y}(x) = \frac{\sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right)}$$

Unfortunately, Scikit-learn does not implement this regression estimator; however, Jan Hendrik Metzen makes a compatible version available on `github.com`.

```
>>> from kernel_regression import KernelRegression
```

This code makes it possible to internally optimize over the bandwidth parameter using leave-one-out cross validation by specifying a grid of potential bandwidth values (`gamma`), as in the following,

```
>>> kr = KernelRegression(gamma=np.linspace(6e3,7e3,500))
>>> kr.fit(xin,y)
KernelRegression(gamma=6000.0,kernel='rbf')
```

Figure 3.31 shows the kernel estimator (heavy black line) using the Gaussian kernel compared to the nearest neighbor estimator (solid light black line). As before, the data points are shown as circles. Figure 3.31 shows that the kernel estimator can pick out the sharp peaks that are missed by the nearest neighbor estimator.

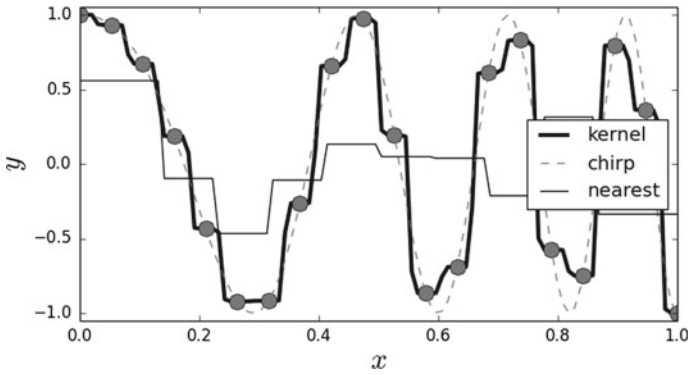

Fig. 3.31 The *heavy black line* is the Gaussian kernel estimator. The *light black line* is the nearest neighbor estimator. The data points are shown as *gray circles*. Note that unlike the nearest neighbor estimator, the Gaussian kernel estimator is able to pick out the sharp peaks in the training data

Thus, the difference between nearest neighbor and kernel estimation is that the latter provides a smooth moving averaging of points whereas the former provides a discontinuous averaging. Note that kernel estimates suffer near the boundaries where there is mismatch between the edges and the kernel function. This problem gets worse in higher dimensions because the data naturally drift towards the boundaries (this is a consequence of the *curse of dimensionality*). Indeed, it is not possible to simultaneously maintain local accuracy (i.e., low bias) and a generous neighborhood (i.e., low variance). One way to address this problem is to create a local polynomial regression using the kernel function as a window to localize a region of interest. For example,

$$\hat{y}(x) = \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)(Y_i - \alpha - \beta x_i)^2$$

and now we have to optimize over the two linear parameters α and β. This method is known as *local linear regression* [5, 6]. Naturally, this can be extended to higher-order polynomials. Note that these methods are not yet implemented in Scikit-learn.

3.12.6 Curse of Dimensionality

The so-called curse of dimensionality occurs as we move into higher and higher dimensions. The term was coined by Bellman in 1961 while he was studying adaptive control processes. Nowadays, the term is vaguely refers to anything that becomes more complicated as the number of dimensions increases substantially. Nevertheless, the concept is useful for recognizing and characterizing the practical difficulties of high-dimensional analysis and estimation.

Consider the volume of an n-dimensional sphere,

$$V_s(n, r) = \begin{cases} \pi^{n/2} \frac{r^n}{(n/2)!} & \text{if } n \text{ is even} \\ 2^n \pi^{(n-1)/2} \frac{r^{(n-1)}}{(n-1)!}((n-1)/2)! & \text{if } n \text{ is odd} \end{cases} \tag{3.12.6.1}$$

Further, consider the sphere $V_s(n, 1/2)$ enclosed by an n dimensional unit cube. The volume of the cube is always equal to one, but $\lim_{n\to\infty} V_s(n, 1/2) = 0$. What does this mean? It means that the volume of the cube is pushed away from its center, where the embedded hypersphere lives. Specifically, the distance from the center of the cube to its vertices in n dimensions is $\sqrt{n}/2$, whereas the distance from the center of the inscribing sphere is $1/2$. This diagonal distance goes to infinity as n does. For a fixed n, the tiny spherical region at the center of the cube has many long spines attached to it, like a hyper-dimensional sea urchin or porcupine.

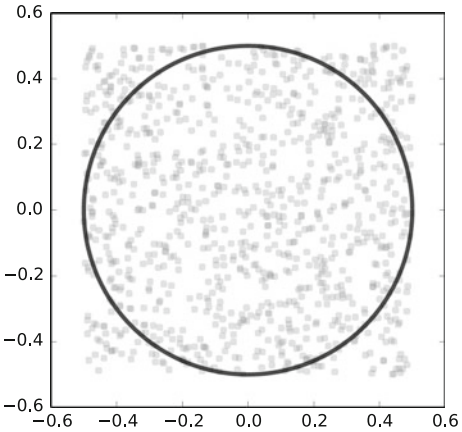

Fig. 3.32 Two dimensional scatter of points randomly and independently uniformly distributed in the unit square. Note that most of the points are contained in the *circle*. Counter to intuition, this does not persist as the number of dimensions increases

What are the consequences of this? For methods that rely on nearest neighbors, exploiting locality to lower bias becomes intractable. For example, suppose we have an n dimensional space and a point near the origin we want to localize around. To estimate behavior around this point, we need to average the unknown function about this point, but in a high-dimensional space, the chances of finding neighbors to average are slim. Looked at from the opposing point of view, suppose we have a binary variable, as in the coin-flipping problem. If we have 1000 trials, then, based on our earlier work, we can be confident about estimating the probability of heads. Now, suppose we have 10 binary variables. Now we have $2^{10} = 1024$ vertices to estimate. If we had the same 1000 points, then at least 24 vertices would not get any data. To keep the same resolution, we would need 1000 samples at each vertex for a grand total of $1000 \times 1024 \approx 10^6$ data points. So, for a ten fold increase in the number of variables, we now have about 1000 more data points to collect to maintain the same statistical resolution. This is the curse of dimensionality.

Perhaps some code will clarify this. The following code generates samples in two dimensions that are plotted as points in Fig. 3.32 with the inscribed circle in two dimensions. Note that for $d = 2$ dimensions, most of the points are contained in the circle.

```
>>> import numpy as np
>>> v=np.random.rand(1000,2)-1/2.
```

The next code block describes the core computation in Fig. 3.33. For each of the dimensions, we create a set of uniformly distributed random variates along each dimension and then compute how close each d dimensional vector is to the origin. Those that measure one half are those contained in the hypersphere. The histogram

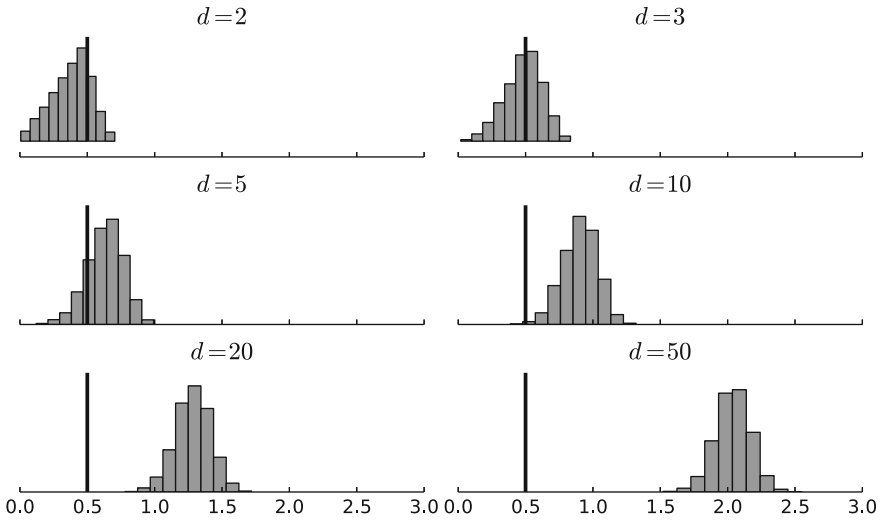

Fig. 3.33 Each *panel* shows the histogram of lengths of uniformly distributed d dimensional random vectors. The population to the *left* of the *dark vertical line* are those that are contained in the inscribed hypersphere. This shows that fewer points are contained in the hypersphere with increasing dimension

of each measurement is shown in the corresponding panel in the Fig. 3.33. The dark vertical line shows the threshold value. Values to the left of this indicate the population that are contained in the hypersphere. Thus, Fig. 3.33 shows that as d increases, fewer points are contained in the inscribed hypersphere. The following code paraphrases the content of Fig. 3.33

```
for d in [2,3,5,10,20,50]:
    v=np.random.rand(5000,d)-1/2.
    hist([np.linalg.norm(i) for i in v])
```

References

1. W. Feller, *An Introduction to Probability Theory and Its Applications: Volume One* (Wiley, 1950)
2. L. Wasserman, *All of Statistics: a Concise Course in Statistical Inference* (Springer, 2004)
3. R.A. Maronna, D.R. Martin, V.J. Yohai, *Robust Statistics: Theory and Methods*, Wiley Series in Probability and Statistics (Wiley, Chichester, 2006)
4. D.G. Luenberger, *Optimization by Vector Space Methods*, Professional Series (Wiley, 1968)
5. C. Loader, *Local Regression and Likelihood* (Springer, New York, 2006)
6. T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics (Springer, New York, 2013)

Chapter 4
Machine Learning

4.1 Introduction

Machine Learning is a huge and growing area. In this chapter, we cannot possibly even survey this area, but we can provide some context and some connections to probability and statistics that should make it easier to think about machine learning and how to apply these methods to real-world problems. The fundamental problem of statistics is basically the same as machine learning: given some data, how to make it actionable? For statistics, the answer is to construct analytic estimators using powerful theory. For machine learning, the answer is algorithmic prediction. Given a data set, what forward-looking inferences can we draw? There is a subtle bit in this description: how can we know the future if all we have is data about the past? This is the crux of the matter for machine learning, as we will explore in the chapter.

4.2 Python Machine Learning Modules

Python provides many bindings for machine learning libraries, some specialized for technologies such as neural networks, and others geared towards novice users. For our discussion, we focus on the powerful and popular Scikit-learn module. Scikit-learn is distinguished by its consistent and sensible API, its wealth of machine learning algorithms, its clear documentation, and its readily available datasets that make it easy to follow along with the online documentation. Like Pandas, Scikit-learn relies on Numpy for numerical arrays. Since its release in 2007, Scikit-learn has become the most widely-used, general-purpose, open-source machine learning modules that is popular in both industry and academia. As with all of the Python modules we use, Scikit-learn is available on all the major platforms.

To get started, let's revisit the familiar ground of linear regression using Scikit-learn. First, let's create some data.

J. Unpingco, *Python for Probability, Statistics, and Machine Learning*,
DOI 10.1007/978-3-319-30717-6_4

```
>>> import numpy as np
>>> from matplotlib.pylab import subplots
>>> from sklearn.linear_model import LinearRegression
>>> X = np.arange(10)          # create some data
>>> Y = X+np.random.randn(10) # linear with noise
```

We next import and create an instance of the `LinearRegression` class from Scikit-learn.

```
>>> from sklearn.linear_model import LinearRegression
>>> lr=LinearRegression() # create model
```

Scikit-learn has a wonderfully consistent API. All Scikit-learn objects use the `fit` method to compute model parameters and the `predict` method to evaluate the model. For the `LinearRegression` instance, the `fit` method computes the coefficients of the linear fit. This method requires a matrix of inputs where the rows are the samples and the columns are the features. The *target* of the regression are the `Y` values, which must be correspondingly shaped, as in the following,

```
>>> X,Y = X.reshape((-1,1)), Y.reshape((-1,1))
>>> lr.fit(X,Y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> lr.coef_
array([[ 0.94211853]])
```

Programming Tip

The negative one in the `reshape((-1,1))` call above is for the truly lazy. Using a negative one tells Numpy to figure out what that dimension should be given the other dimension and number of array elements.

the `coef_` property of the linear regression object shows the estimated parameters for the fit. The convention is to denote estimated parameters with a trailing underscore. The model has a `score` method that computes the R^2 value for the regression. Recall from our statistics chapter Sect. 3.7 that the R^2 value is an indicator of the quality of the fit and varies between zero (bad fit) and one (perfect fit).

```
>>> lr.score(X,Y)
0.9059042979442371
```

Now, that we have this fitted, we can evaluate the fit using the `predict` method,

```
>>> xi = np.linspace(0,10,15) # more points to draw
>>> xi = xi.reshape((-1,1)) # reshape as columns
>>> yp = lr.predict(xi)
```

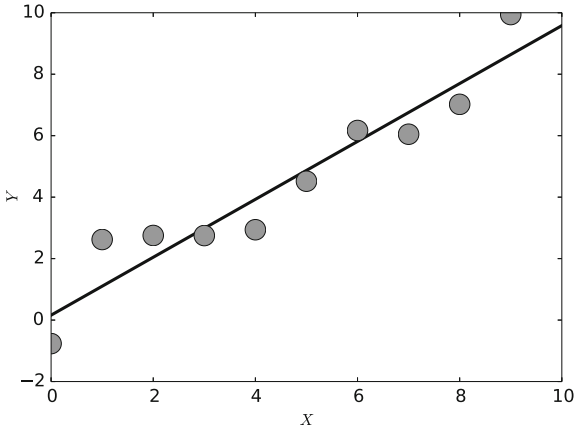

Fig. 4.1 The Scikit-learn module can easily perform basic linear regression. The *circles* show the *training* data and the fitted line is shown in *black*

The resulting fit is shown in Fig. 4.1.

Multilinear Regression. The Scikit-learn module easily extends linear regression to multiple dimensions. For example, for multi-linear regression,

$$y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$$

The problem is to find all of the α terms given the training set $\{x_1, x_2, \ldots, x_n, y\}$. We can create another example data set and see how this works,

```
>>> X=np.random.randint(20,size=(10,2))
>>> Y=X.dot([1, 3])+1 + np.random.randn(X.shape[0])*20
```

Figure 4.2 shows the two dimensional regression example, where the size of the circles is proportional to the targetted Y value. Note that we salted the output with random noise just to keep things interesting. Nonetheless, the interface with Scikit-learn is the same,

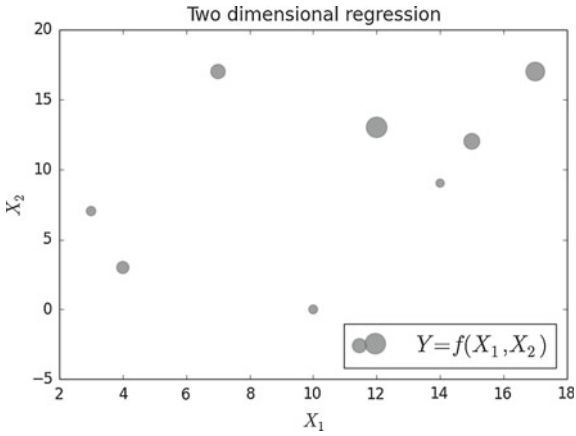

Fig. 4.2 Scikit-learn can easily perform multi-linear regression. The size of the circles indicate the value of the two-dimensional function of (X_1, X_2)

```
>>> lr=LinearRegression()
>>> lr.fit(X,Y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> print lr.coef_
[ 0.35171694  4.04064287]
```

Note that the `coef_` variable now has two terms in it, corresponding to the two input dimensions. Note that the constant offset is already built-in and is an option on the `LinearRegression` constructor. Figure 4.3 shows how the regression performs.

Polynomial Regression. We can extend this to include polynomial regression by using the `PolynomialFeatures` in the `preprocessing` sub-module. To keep it simple, let's go back to our one-dimensional example. First, let's create some synthetic data,

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(10).reshape(-1,1) # create some data
>>> Y = X+X**2+X**3+ np.random.randn(*X.shape)*80
```

Next, we have to create a transformation from `X` to a polynomial of `X`,

```
>>> qfit = PolynomialFeatures(degree=2) # quadratic
>>> Xq = qfit.fit_transform(X)
>>> print Xq
[[ 1  0  0]
 [ 1  1  1]
 [ 1  2  4]
 [ 1  3  9]
 [ 1  4 16]
 [ 1  5 25]
 [ 1  6 36]
 [ 1  7 49]
 [ 1  8 64]
 [ 1  9 81]]
```

Note there is an automatic constant term in the output 0th column where `fit_transform` has mapped the single-column input into a set of columns representing the individual polynomial terms. The middle column has the linear term, and the last

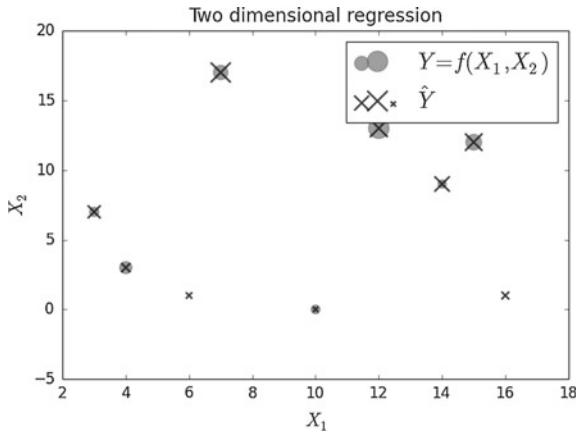

Fig. 4.3 The predicted data is plotted in *black*. It overlays the training data, indicating a good fit

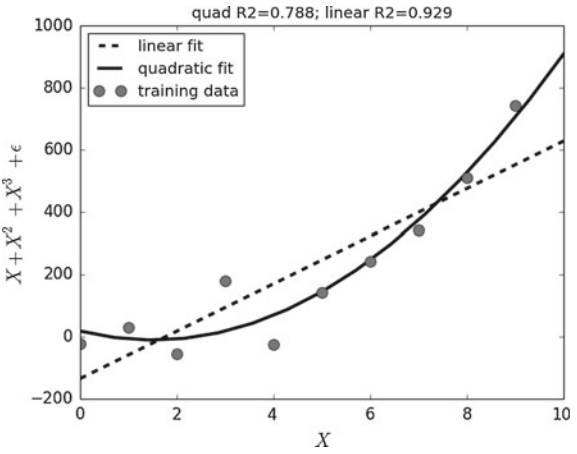

Fig. 4.4 The title shows the R^2 score for the linear and quadratic rogressions

has the quadratic term. With these polynomial features stacked as columns of `Xq`, all we have to do is `fit` and `predict` again. The following draws a comparison between the linear regression and the quadratic repression (Fig. 4.4),

```
>>> lr=LinearRegression() # create linear model
>>> qr=LinearRegression() # create quadratic model
>>> lr.fit(X,Y)  # fit linear model
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> qr.fit(Xq,Y) # fit quadratic model
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> lp = lr.predict(xi)
>>> qp = qr.predict(qfit.fit_transform(xi))
```

This just scratches the surface of Scikit-learn. We will go through many more examples later, but the main thing is to concentrate on the usage (i.e., `fit`, `predict`) which is standardized across all of the machine learning methods that are implemented in Scikit-learn.

4.3 Theory of Learning

There is nothing so practical as a good theory. In this section, we establish the formal framework for thinking about machine learning. This framework will help us think beyond particular methods for machine learning so we can integrate new methods or combine existing methods intelligently.

Both machine learning and statistics share the common goal of trying to derive understanding from data. Some historical perspective helps. Most of the methods in statistics were derived towards the start of the 20th century when data were hard to come by. Society was preoccupied with the potential dangers of human overpopulation and work was focused on studying agriculture and crop yields. At this time, even a dozen data points was considered plenty. Around the same time, the

deep foundations of probability were being established by Kolmogorov. Thus, the lack of data meant that the conclusions had to be buttressed by strong assumptions and solid mathematics provided by the emerging theory of probability. Furthermore, inexpensive powerful computers were not yet widely available. The situation today is much different: there are lots of data collected and powerful and easily programmable computers are available. The important problems no longer revolve around a dozen data points on a farm acre, but rather millions of points on a square millimeter of a DNA microarray. Does this mean that statistics will be superseded by machine learning?

In contrast to classical statistics, which is concerned with developing models that characterize, explain, and describe phenomena, machine learning is primarily concerned with prediction, usually at the expense of all else. Areas like exploratory statistics are very closely related to machine learning, but the degree of emphasis on prediction is still distinguishing. In some sense, this is unavoidable due the size of the data machine learning can reduce. In other words, machine learning can help distill a table of a million columns into one hundred columns, but can we still interpret one hundred columns meaningfully? In classical statistics, this was never an issue because data were of a much smaller scale. Whereas mathematical models, usually normal distributions, fitted with observations are common in statistics, machine learning uses data to construct models that sit on complicated data structures and exploit nonlinear optimizations that lack closed-form solutions. A common maxim is that statistics is data plus analytical theory and machine learning is data plus computable structures. This makes it seem like machine learning is completely ad-hoc and devoid of underlying theory, but this is not the case, and both machine learning and statistics share many important theoretical results. By way of contrast, let us consider a concrete problem.

Let's consider the classic balls in urns problem (see Fig. 4.5): we have an urn containing red and blue balls and we draw five balls from the urn, note the color of each ball, and then try to determine the proportion of red and blue balls in the urn. We have already studied many statistical methods for dealing with this problem in previous sections. Now, let's generalize the problem slightly. Suppose the urn is filled with white balls and there is some target unknown function f that paints each selected ball either red or blue (see Fig. 4.6). The machine learning problem is how to find the f function, given only the observed red/blue balls. So far, this doesn't sound much different from the statistical problem. However, now we want to take our estimated f function, say, $\hat{f}$, and use it to predict the next handful of balls from another urn. Now, here's where the story takes a sharp turn. Suppose the next urn *already* has some red and blue balls in it? Then, applying the function f may result in purple balls which were not seen in the *training* data (see Fig. 4.7). Now, what can we do? We have just scraped the surface of the issues machine learning must confront using methods that are not part of the statistics canon.

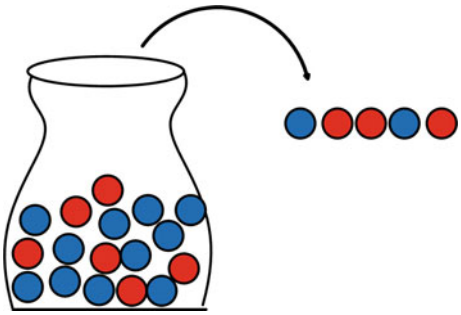

Fig. 4.5 In the classical statistics problem, we observe a sample and model what the urn contains

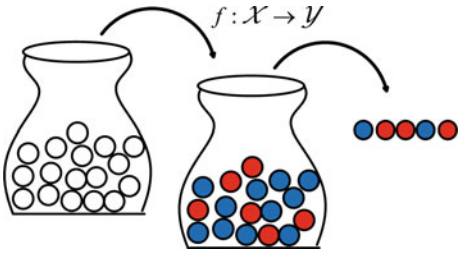

Fig. 4.6 In the machine learning problem, we want the function that colors the marbles

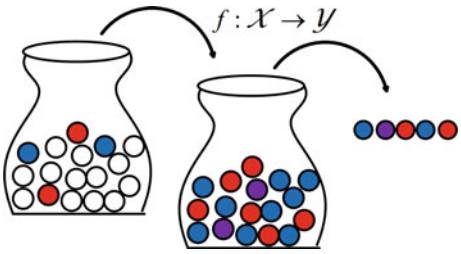

Fig. 4.7 The problem is further complicated because we may see colored marbles that were not present in the original problem

4.3.1 Introduction to Theory of Machine Learning

Some formality and an example can get us going. We define the unknown target function, $f : \mathcal{X} \mapsto \mathcal{Y}$. The training set is $\{(x, y)\}$ which means that we only see the function's inputs/outputs. The hypothesis set $\mathcal{H}$ is the set of all possible guesses at f. This is the set from which we will ultimately draw our final estimate, $\hat{f}$. The machine learning problem is how to derive the best element from the hypothesis set by using the training set. Let's consider a concrete example in the code below. Suppose $\mathcal{X}$ consists of all three-bit vectors (i.e., $\mathcal{X} = \{000, 001, \ldots, 111\}$) as in the code below,

```
>>> import pandas as pd
>>> import numpy as np
>>> from pandas import DataFrame
>>> df=DataFrame(index=pd.Index(['{0:04b}'.format(i) for i in range(2**4)],
...                             dtype='str',
...                             name='x'),columns=['f'])
```

Programming Tip

The string specification above uses Python's advanced string formatting mini-language. In this case, the specification says to convert the integer into a fixed-width, four-character (`04b`) binary representation.

Next, we define the target function f below which just checks if the number of zeros in the binary representation exceeds the number of ones. If so, then the function outputs `1` and `0` otherwise (i.e., $\mathcal{Y} = \{0, 1\}$).

```
>>> df.f=np.array(df.index.map(lambda i:i.count('0'))
...                > df.index.map(lambda i:i.count('1')),dtype=int)
>>> df.head(8) # show top half only
      f
x
0000  1
0001  1
0010  1
0011  0
0100  1
0101  0
0110  0
0111  0
```

The hypothesis set for this problem is the set of *all* possible functions of $\mathcal{X}$. The set $\mathcal{D}$ represents all possible input/output pairs. The corresponding hypothesis set $\mathcal{H}$ has 2^{16} elements, one of which matches f. There are 2^{16} elements in the hypothesis set because for each of sixteen input elements, there are two possible corresponding values zero or one for each input. Thus, the size of the hypothesis set is $2 \times 2 \times \cdots \times 2 = 2^{16}$. Now, presented with a training set consisting of the first eight input/output pairs, our goal is to minimize errors over the training set ($E_{\text{in}}(\hat{f})$). There are 2^8 elements from the hypothesis set that exactly match f over the training set. But how to pick among these 2^8 elements? It seems that we are stuck here. We need another element from the problem in order to proceed. The extra piece we need is to assume that the training set represents a random sampling (*in-sample* data) from a greater population (*out-of-sample* data) that would be consistent with the population that $\hat{f}$ would ultimately predict upon. In other words, we are assuming a stable probability structure for both the in-sample and out-of-sample data. This is a major assumption!

There is a subtle consequence of this assumption—whatever the machine learning method does once deployed, in order for it to continue to work, it cannot disturb the data environment that it was trained on. Said differently, if the method is not to be trained continuously, then it cannot break this assumption by altering the generative environment that produced the data it was trained on. For example, suppose we develop a model that predicts hospital readmissions based on seasonal weather and patient health. Because the model is so effective, in the next six months, the hospital forestalls readmissions by delivering interventions that improve patient health. Clearly using the model cannot change seasonal weather, but because the hospital used the model to change patient health, the training data used to build the model is

no longer consistent with the forward-looking health of the patients. Thus, there is little reason to think that the model will continue to work as well going forward.

Returning to our example, let's suppose that the first eight elements from $\mathcal{X}$ are twice as likely as the last eight. The following code is a function that generates elements from $\mathcal{X}$ according to this distribution.

```
>>> np.random.seed(12)
>>> def get_sample(n=1):
...     if n==1:
...         return '{0:04b}'.format(np.random.choice(range(8)*2+range(8,16)))
...     else:
...         return [get_sample(1) for _ in range(n)]
...
```

Programming Tip

The function that returns random samples uses the `np.random.choice` function from Numpy which takes samples (with replacement) from the given iterable. Because we want the first eight numbers to be twice as frequent as the rest, we simply repeat them in the iterable using `range(8)*2`. Recall that multiplying a Python list by an integer duplicates the entire list by that integer. It does not do element-wise multiplication as with Numpy arrays. If we wanted the first eight to be 10 times more frequent, then we would use `range(8)*10`, for example. This is a simple but powerful technique that requires very little code. Note that the `p` keyword argument in `np.random.choice` also provides an explicit way to specify more complicated distributions.

The next block applies the function definition f to the sampled data to generate the training set consisting of eight elements.

```
>>> train=df.f.ix[get_sample(8)] # 8-element training set
>>> train.index.unique().shape    # how many unique elements?
(6,)
```

Notice that even though there are eight elements, there is redundancy because these are drawn according to an underlying probability. Otherwise, if we just got all sixteen different elements, we would have a training set consisting of the complete specification of f and then we would therefore know what $h \in \mathcal{H}$ to pick! However, this effect gives us a clue as to how this will ultimately work. Given the elements in the training set, consider the set of elements from the hypothesis set that exactly match. How to choose among these? The answer is it does not matter! Why? Because under the assumption that the prediction will be used in an environment that is determined by the same probability, getting something outside of the training set is just as likely as getting something inside the training set. The size of the training set is key here—the bigger the training set, the less likely that there will be real-world data that fall outside of it and the better $\hat{f}$ will perform.[1] The following code shows the elements of the training set in the context of all possible data.

[1] This assumes that the hypothesis set is big enough to capture the entire training set (which it is for this example). We will discuss this trade-off in greater generality shortly.

```
>>> df['fhat']=df.f.ix[train.index.unique()]
>>> df.fhat
x
0000   NaN
0001   NaN
0010     1
0011     0
0100     1
0101   NaN
0110     0
0111   NaN
1000     1
1001     0
1010   NaN
1011   NaN
1100   NaN
1101   NaN
1110   NaN
1111   NaN
Name: fhat, dtype: float64
```

Note that there are NaN symbols where the training set had no values. For definiteness, we fill these in with zeros, although we can fill them with anything we want so long as whatever we do is not determined by the training set.

```
>>> df.fhat.fillna(0,inplace=True) #final specification of fhat
```

Now, let's pretend we have deployed this and generate some test data.

```
>>> test= df.f.ix[get_sample(50)]
>>> (df.ix[test.index]['fhat'] != test).mean()
0.17999999999999999
```

The result shows the error rate, given the probability mechanism that is generating the data. The following Pandas-fu compares the overlap between the training set and the test set in the context of all possible data. The NaN values show the rows where the test data had items absent in the training data. Recall that the method returns zero for these items. As shown, sometimes this works in its favor, and sometimes not.

```
>>> pd.concat([test.groupby(level=0).mean(),
...            train.groupby(level=0).mean()],
...           axis=1,
...           keys=['test','train'])
      test  train
0000     1    NaN
0001     1    NaN
0010     1      1
0011     0      0
0100     1      1
0101     0    NaN
0110     0      0
0111     0    NaN
1000     1      1
1001     0      0
1010     0    NaN
1011     0    NaN
1100     0    NaN
1101     0    NaN
1110     0    NaN
1111     0    NaN
```

Note that where the test data and training data share elements, they agree. When the test set produced an unseen element, it produces a match or not.

Programming Tip

The `pd.concat` function concatenates the two `Series` objects in the list. The `axis=1` means join the two objects along the columns where each newly created column is named according to the given `keys`. The `level=0` in the `groupby` for each of the `Series` objects means group along the index. Because the index corresponds to the 4-bit elements, this accounts for repetition in the elements. The `mean` aggregation function computes the values of the function for each 4-bit element. Because all functions in each respective group have the same value, the `mean` just picks out that value because the average of a list of constants is that constant.

Now, we are in position to ask how big the training set should be to achieve a level of performance. For example, on average, how many in-samples do we need for a given error rate? For this problem, we can ask how large (on average) must the training set be in order to capture *all* of the possibilities and achieve perfect out-of-sample error rates? For this problem, this turns out to be sixty-three.[2] Let's start over and retrain with these many in-samples.

```
>>> train=df.f.ix[get_sample(63)]
>>> del df['fhat']
>>> df['fhat']=df.f.ix[train.index.unique()]
>>> df.fhat.fillna(0,inplace=True) #final specification of fhat
>>> test= df.f.ix[get_sample(50)]
>>> (df.fhat.ix[test] != df.f.ix[test]).mean() # error rate
0.0
```

Notice that this bigger training set has a better error rate because it is able to identify the best element from the hypothesis set because the training set captured more of the complexity of the unknown f. This example shows the trade-offs between the size of the training set, the complexity of the target function, the probability structure of the data, and the size of the hypothesis set.

4.3.2 Theory of Generalization

What we really want to know is how the our method will perform once deployed. It would be nice to have some kind of performance guarantee. In other words, we worked hard to minimize the errors in the training set, but what errors can we expect at deployment? In training, we minimized the in-sample error, $E_{\texttt{in}}(\hat{f})$, but that's not good enough. We want guarantees about the out-of-sample error, $E_{\texttt{out}}(\hat{f})$. This is

[2] This is a slight generalization of the classic coupon collector problem.

what *generalization* means in machine learning. The mathematical statement of this is the following,

$$\mathbb{P}\left(|E_{\text{out}}(\hat{f}) - E_{\text{in}}(\hat{f})| > \epsilon\right) < \delta$$

for a given ϵ and δ. Informally, this says that the probability of the respective errors differing by more than a given ϵ is less than some quantity, δ. This basically means that whatever the performance on the training set, it should probably be pretty close to the corresponding performance once deployed. Note that this does not say that the in-sample errors (E_{in}) are any good in an absolute sense. It just says that we would not expect much different after deployment. Thus, *good* generalization means no surprises after deployment, not necessarily good performance, by any means. There are two main ways to get at this: cross-validation and probability inequalities. Let's consider the latter first. There are two entangled issues: the complexity of the hypothesis set and the probability of the data. It turns out we can separate these two by deriving a separate notion of complexity free from any particular data probability.

VC Dimension. We first need a way to quantify model complexity. Following Wasserman [1], let $\mathcal{A}$ be a class of sets and $F = \{x_1, x_2, \ldots, x_n\}$, a set of n data points. Then, we define

$$N_{\mathcal{A}}(F) = \#\{F \cap A : A \in \mathcal{A}\}$$

This counts the number of subsets of F that can be extracted by the sets of $\mathcal{A}$. The number of items in the set (i.e., cardinality) is noted by the # symbol. For example, suppose $F = \{1\}$ and $\mathcal{A} = \{(x \leq a)\}$. In other words, $\mathcal{A}$ consists of all intervals closed on the right and parameterized by a. In this case we have $N_{\mathcal{A}}(F) = 1$ because all elements can be extracted from F using $\mathcal{A}$.

The *shatter coefficient* is defined as,

$$s(\mathcal{A}, n) = \max_{F \in \mathcal{F}_n} N_{\mathcal{A}}(F)$$

where $\mathcal{F}$ consists of all finite sets of size n. Note that this sweeps over all finite sets so we don't need to worry about any particular data set of finitely many points. The definition is concerned with $\mathcal{A}$ and how its sets can pick off elements from the data set. A set F is *shattered* by $\mathcal{A}$ if it can pick out every element in it. This provides a sense of how the complexity in $\mathcal{A}$ consumes data. In our last example, the set of half-closed intervals shattered every singleton set $\{x_1\}$.

Now, we come to the main definition of the Vapnik-Chervonenkis [2] dimension d_{VC} which defined as the largest k for which $s(\mathcal{A}, n) = 2^k$, except in the case where $s(\mathcal{A}, n) = 2^n$ for which it is defined as infinity. For our example where $F = \{x_1\}$, we already saw that $\mathcal{A}$ shatters F. How about when $F = \{x_1, x_2\}$? Now, we have two points and we have to consider whether all subsets can be extracted by $\mathcal{A}$. In this case, there are four subsets, $\left\{\emptyset, \{x_1\}, \{x_2\}, \{x_1, x_2\}\right\}$. Note that $\emptyset$ denotes the

empty set. The empty set is easily extracted—pick a so that it is smaller than both x_1 and x_2. Assuming that $x_1 < x_2$, we can get the next set by choosing $x_1 < a < x_2$. The last set is likewise do-able by choosing $x_2 < a$. The problem is that we cannot capture the third set, $\{x_2\}$, without capturing x_1 as well. This means that we cannot shatter any finite set with $n = 2$ using $\mathcal{A}$. Thus, $d_{\text{VC}} = 1$.

Here is the climatic result

$$E_{\text{out}}(\hat{f}) \leq E_{\text{in}}(\hat{f}) + \sqrt{\frac{8}{n} \ln \left(\frac{4((2n)^{d_{\text{VC}}} + 1)}{\delta} \right)}$$

with probability at least $1 - \delta$. This basically says that the expected out-of-sample error can be no worse than the in-sample error plus a penalty due to the complexity of the hypothesis set. The expected in-sample error comes from the training set but the complexity penalty comes from just the hypothesis set, so we have disentangled these two issues.

A general result like this, for which we do not worry about the probability of the data, is certain to be pretty generous, but nonetheless, it tells us how the complexity penalty enters into the out-of-sample error. In other words, the bound on $E_{\text{out}}(\hat{f})$ gets worse for a more complex hypothesis set. Thus, this generalization bound is a useful guideline but not very practical if we want to get a good estimate of $E_{\text{out}}(\hat{f})$.

4.3.3 *Worked Example for Generalization/Approximation Complexity*

The stylized curves in Fig. 4.8 illustrate the idea that there is some optimal point of complexity that represents the best generalization given the training set.

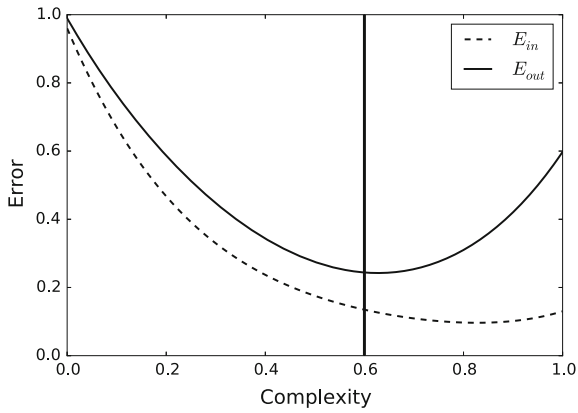

Fig. 4.8 In the ideal situation, there is a *best model* that represents the optimal trade-off between complexity and error. This is shown by the *vertical line*

To get a firm handle on these curves, let's develop a simple one-dimensional machine learning method and go through the steps to create this graph. Let's suppose we have a training set consisting of x-y pairs $\{(x_i, y_i)\}$. Our method groups the x-data into intervals and then averages the y-data in those intervals. Predicting for new x-data means simply identifying the interval containing the new data then reporting the corresponding value. In other words, we are building a simple one-dimensional, nearest neighbor classifier. For example, suppose the training set x-data is the following,

```
>>> train=DataFrame(columns=['x','y'])
>>> train['x']=np.sort(np.random.choice(range(2**10),size=90))
>>> train.x.head(10) # first ten elements
0     15
1     30
2     45
3     65
4     76
5     82
6    115
7    145
8    147
9    158
Name: x, dtype: int32
```

In this example, we took a random set of 10-bit integers. To group these into, say, ten intervals, we simply use Numpy reshape as in the following,

```
>>> train.x.reshape(10,-1)
array([[ 15,  30,  45,  65,  76,  82, 115, 145, 147],
       [158, 165, 174, 175, 181, 209, 215, 217, 232],
       [233, 261, 271, 276, 284, 296, 318, 350, 376],
       [384, 407, 410, 413, 452, 464, 472, 511, 522],
       [525, 527, 531, 534, 544, 545, 548, 567, 567],
       [584, 588, 610, 610, 641, 645, 648, 659, 667],
       [676, 683, 684, 697, 701, 703, 733, 736, 750],
       [754, 755, 772, 776, 790, 794, 798, 804, 830],
       [831, 834, 861, 883, 910, 910, 911, 911, 937],
       [943, 946, 947, 955, 962, 962, 984, 989, 998]])
```

where every row is one of the groups. Note that the range of each group (i.e., length of the interval) is not preassigned, and is learned from the training data. For this example, the y-values correspond to the number of ones in the bit representation of the x-values. The following code defines this target function,

```
>>> f_target=np.vectorize(lambda i:sum(map(int,i)))
```

Programming Tip

The above function uses `np.vectorize` which is a convenience method in Numpy that converts plain Python functions into Numpy versions. This basically saves additional looping semantics and makes it easier to use with other Numpy arrays and functions.

Next, we create the bit representations of all of the x-data below and then complete training set y-values,

```
>>> train['xb']= train.x.map('{0:010b}'.format)
>>> train.y=train.xb.map(f_target)
>>> train.head(5)
    x  y          xb
0  15  4  0000001111
1  30  4  0000011110
2  45  4  0000101101
3  65  2  0001000001
4  76  3  0001001100
```

To train on this data, we just group by the specified amount and then average the y-data over each group.

```
>>> train.y.reshape(10,-1).mean(axis=1)
array([ 3.55555556,  4.88888889,  4.44444444,  4.88888889,  4.11111111,
        4.        ,  6.        ,  5.11111111,  6.44444444,  6.66666667])
```

Note that the `axis=1` keyword argument just means average across the columns. So far, this defines the training. To predict using this method, we have to extract the edges from each of the groups and then fill in with the group-wise mean we just computed for `y`. The following code extracts the edges of each group.

```
>>> le,re=train.x.reshape(10,-1)[:,[0,-1]].T
>>> print le # left edge of group
[ 15 158 233 384 525 584 676 754 831 943]
>>> print re # right edge of group
[147 232 376 522 567 667 750 830 937 998]
```

Next, we compute the group-wise means and assign them to their respective edges.

```
>>> val = train.y.reshape(10,-1).mean(axis=1).round()
>>> func = pd.Series(index=range(1024))
>>> func[le]=val    # assign value to left edge
>>> func[re]=val    # assign value to right edge
>>> func.iloc[0]=0  # default 0 if no data
>>> func.iloc[-1]=0 # default 0 if no data
>>> func.head()
0     0
1   NaN
2   NaN
3   NaN
4   NaN
dtype: float64
```

Note that the Pandas `Series` object automatically fills in unassigned values with `NaN`. We have thus far only filled in values at the edges of the groups. Now, we need to fill in the intermediate values.

```
>>> fi=func.interpolate('nearest')
>>> fi.head()
0    0
1    0
2    0
3    0
4    0
dtype: float64
```

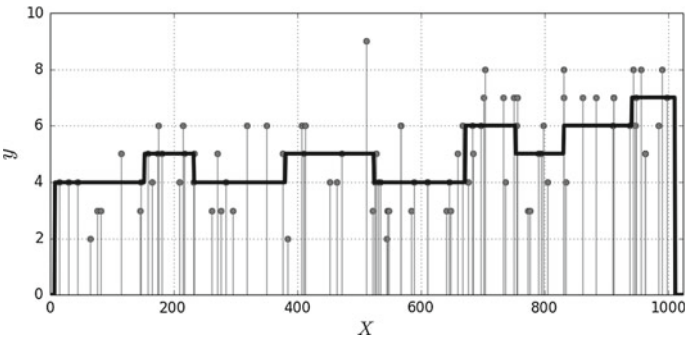

Fig. 4.9 The *vertical lines* show the training data and the *thick black line* is the approximant we have learned from the training data

The `interpolate` method of the `Series` object can apply a wide variety of powerful interpolation methods, but we only need the simple nearest neighbor method to create our piecewise approximant. Figure 4.9 shows how this looks for the training data we have created.

Now, with all that established, we can now draw the curves for this machine learning method. Instead of partitioning the training data for cross-validation (which we'll discuss later), we can simulate test data using the same mechanism as for the training data, as shown next,

```
>>> test=pd.DataFrame(columns=['x','xb','y'])
>>> test['x']=np.random.choice(range(2**10),size=500)
>>> test.xb= test.x.map('{0:010b}'.format)
>>> test.y=test.xb.map(f_target)
>>> test.sort(columns=['x'],inplace=True)
```

The curves are the respective errors for the training data and the testing data. For our error measure, we use the mean-squared-error,

$$E_{\texttt{out}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{f}(x_i) - y_i)^2$$

where $\{(x_i, y_i)\}_{i=1}^{n}$ come from the test data. The in-sample error ($E_{\texttt{in}}$) is defined the same except for the in-sample data. In this example, the size of each group is proportional to d_{VC}, so the more groups we choose, the more complexity in the fitting. Now, we have all the ingredients to understand the trade-offs of complexity versus error.

Figure 4.10 shows the curves for our one-dimensional clustering method. The dotted line shows the mean-squared-error on the training set and the other line shows the same for the test data. The shaded region is the *complexity penalty* of this method. Note that with enough complexity, the method can exactly memorize the testing data, but that only penalizes the testing error ($E_{\texttt{out}}$). This effect is exactly what the Vapnik-Chervonenkis theory expresses. The horizontal axis is proportional to the

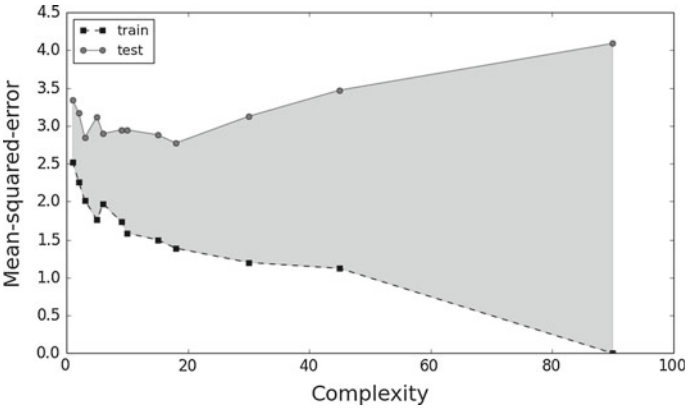

Fig. 4.10 The *dotted line* shows the mean-squared-error on the training set and the *other line* shows the same for the test data. The *shaded region* is the *complexity penalty* of this method. Note that as the complexity of the model increases, the training error decreases, and the method essentially memorizes the data. However, this improvement in training error comes at the cost of larger testing error

VC-dimension. In this case, complexity boils down to the number of intervals used in the sectioning. At the far right, we have as many intervals as there are elements in the data set, meaning that every element is wrapped in its own interval. The average value of the data in that interval is therefore just the corresponding y value because there are no other elements to average over. The Jupyter/IPython notebook corresponding to the section has the code to generate these curves so you can see how these curves change with bigger or smaller data sets.

Before we leave this problem, there is another way to visualize the performance of our learning method. This problem can be thought of as a multi-class identification problem. Given a 10-bit integer, the number of ones in its binary representation is in one of the classes $\{0, 1, \ldots, 10\}$. The output of the model tries to put each integer in its respective class. How well this was done can be visualized using a *confusion matrix* as shown in the next code block,

```
>>> from sklearn.metrics import confusion_matrix
>>> cmx=confusion_matrix(test.y.values,fi[test.x].values)
>>> print cmx
[[ 1  0  0  0  0  0  0  0  0  0]
 [ 1  0  1  0  1  1  0  0  0  0]
 [ 0  0  3  9  7  4  0  0  0  5]
 [ 1  0  3 23 19  6  6  0  2  0]
 [ 0  0  1 26 27 14 27  2  2  0]
 [ 0  0  3 15 31 28 30  8  1  0]
 [ 0  0  1  8 18 20 25 23  2  2]
 [ 1  0  1 10  5 13  7 19  3  6]
 [ 4  0  1  2  0  2  2  7  4  3]
 [ 2  0  0  0  0  1  0  0  0  0]]
```

The rows of this 10×10 matrix show what the true class was and the columns indicate the class that the model predicted. The numbers in the matrix indicate the number of times that association was made. For example, the first row shows that there was one

entry in the test set with no ones in its binary representation (i.e., namely the number zero) and it was correctly classified (namely, it is in the first row, first column of the matrix). The second row shows there were four entries total in the test set with a binary representation containing exactly a single one. This was incorrectly classified as the 0-class (i.e., first column) once, the 2-class (third column) once, the 4-class (fifth column) once, and the 5-class (sixth column) once. It was never classified correctly because the second column is zero for this row. In other words, the diagonal entries show the number of times it was correctly classified.

Using this matrix, we can easily estimate the true-detection probability that we covered earlier in our hypothesis testing section,

```
>>> print cmx.diagonal()/cmx.sum(axis=1)
[ 1.          0.          0.10714286  0.38333333  0.27272727  0.24137931
  0.25252525  0.29230769  0.16        0.        ]
```

In other words, the first element is the probability of detecting `0` when `0` is in force, the second element is the probability of detecting `1` when `1` is in force, and so on. We can likewise compute the false-alarm rate for each of the classes in the following,

```
>>> print (cmx.sum(axis=0) - cmx.diagonal())/(cmx.sum()-cmx.sum(axis=1))
[ 0.01803607  0.          0.02330508  0.15909091  0.20199501  0.15885417
  0.17955112  0.09195402  0.02105263  0.03219316]
```

Programming Tip

The Numpy `sum` function can sum across a particular axis or, if the axis is unspecified, will sum all entries of the array.

In this case, the first element is the probability that `0` is declared when another category is in force, the next element is the probability that `1` is declared when another category is in force, and so on. For a decent classifier, we want a true-detection probability to be greater than the corresponding false-alarm rate, otherwise the classifier is no better than a coin-flip. Note that, at the time of this writing, Scikit-learn has limited tools for this kind of multiple class classification task.

The missing feature of this problem, from the learning algorithm standpoint, is that we did not supply the bit representation of every element which was used to derive the target variable, y. Instead, we just used the integer value of each of the 10-bit numbers, which essentially concealed the mechanism for creating the y values. In other words, there was a unknown transformation from the input space $\mathcal{X}$ to $\mathcal{Y}$ that the learning algorithm had to overcome, but that it could not overcome, at least not without memorizing the training data. This lack of knowledge is a key issue in all machine learning problems, although we have made it explicit here with this stylized example. This means that there may be one or more transformations from $\mathcal{X} \rightarrow \mathcal{X}'$ that can help learning algorithm get traction on the so-transformed space while providing a better trade-off between generalization and approximation than could have been achieved otherwise. Finding such transformations is called *feature engineering*.

4.3.4 Cross-Validation

In the last section, we explored a stylized machine learning example to understand the issues of complexity in machine learning. However, to get an estimate of out-of-sample errors, we simply generated more synthetic data. In practice, this is not an option, so we need to estimate these errors from the training set itself. This is what cross-validation does. The simplest form of cross-validation is k-fold validation. For example, if $K = 3$, then the training data is divided into three sections wherein each of the three sections is used for testing and the remaining two are used for training. This is implemented in Scikit-learn as in the following,

```
>>> import numpy as np
>>> from sklearn.cross_validation import KFold
>>> data =np.array(['a',]*3+['b',]*3+['c',]*3) # example
>>> print data
['a' 'a' 'a' 'b' 'b' 'b' 'c' 'c' 'c']
>>> for train_idx,test_idx in KFold(len(data),3):
...     print train_idx,test_idx
...
[3 4 5 6 7 8] [0 1 2]
[0 1 2 6 7 8] [3 4 5]
[0 1 2 3 4 5] [6 7 8]
```

In the code above, we construct a sample data array and then see how `KFold` splits it up into indicies for training and testing, respectively. Notice that there are no duplicated elements in each row between training and testing indicies. To examine the elements of the data set in each category, we simply use each of the indicies as in the following,

```
>>> for train_idx,test_idx in KFold(len(data),3):
...     print 'training', data[ train_idx ]
...     print 'testing' , data[ test_idx ]
...
training ['b' 'b' 'b' 'c' 'c' 'c']
testing ['a' 'a' 'a']
training ['a' 'a' 'a' 'c' 'c' 'c']
testing ['b' 'b' 'b']
training ['a' 'a' 'a' 'b' 'b' 'b']
testing ['c' 'c' 'c']
```

This shows how each group is used in turn for training/testing. There is no random shuffling of the data unless the `shuffle` keyword argument is given. The error over the test set is th *cross-validation error*. The idea is to postulate models of differing complexity and then pick the one with the best cross-validation error. For example, suppose we had the following sine wave data,

```
>>> xi = np.linspace(0,1,30)
>>> yi = np.sin(2*np.pi*xi)
```

and we want to fit this with polynomials of increasing order.

Figure 4.11 shows the individual folds in each panel. The circles represent the training data. The diagonal line is the fitted polynomial. The gray shaded areas indicate the regions of errors between the fitted polynomial and the held-out testing data. The larger the gray area, the bigger the cross-validation errors, as are reported in the title of each frame.

After reviewing the last three figures and averaging the cross-validation errors, the one with the least average error is declared the winner. Thus, cross-validation provides a method of using a single data set to make claims about unseen out-of-sample data insofar as the model with the best complexity can be determined. The entire process to generate the above figures can be captured using `cross_val_score` as shown for the linear regression (compare the output with the values in the titles in each panel of Fig. 4.11),

```
>>> from sklearn.metrics import  make_scorer, mean_squared_error
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LinearRegression
>>> Xi = xi.reshape(-1,1) # refit column-wise
>>> Yi = yi.reshape(-1,1)
>>> lf = LinearRegression()
>>> scores = cross_val_score(lf,Xi,Yi,cv=4,
...                          scoring=make_scorer(mean_squared_error))
>>> print scores
[ 0.3554451   0.33131438  0.50454257  0.45905672]
```

Programming Tip

The `make_scorer` function is a wrapper that enables `cross_val_score` to compute scores from the given estimator's output.

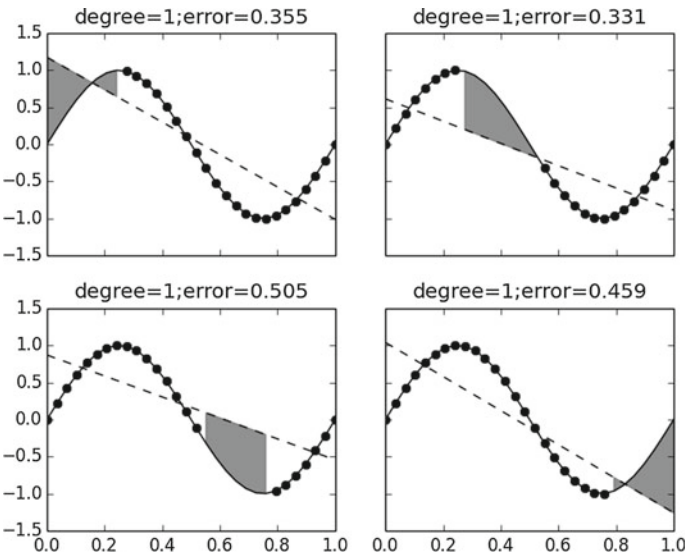

Fig. 4.11 This shows the folds and errors for the linear model. The *shaded areas* show the errors in each respective test set (i.e., *cross-validation scores*) for the linear model

The process can be further automated by using a pipeline as in the following,

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.preprocessing import PolynomialFeatures
>>> polyfitter = Pipeline([('poly', PolynomialFeatures(degree=3)),
...                        ('linear', LinearRegression())])
>>> polyfitter.get_params()
{'linear': LinearRegression(copy_X=True, fit_intercept=True, normalize=False),
 'linear__copy_X': True,
 'linear__fit_intercept': True,
 'linear__normalize': False,
 'poly': PolynomialFeatures(degree=3, include_bias=True, interaction_only=False),
 'poly__degree': 3,
 'poly__include_bias': True,
 'poly__interaction_only': False}
```

The `Pipeline` object is a way of stacking standard steps into one big estimator, while respecting the usual `fit` and `predict` interfaces. The output of the `get_params` function contains the polynomial degrees we previously looped over to create Fig. 4.11, etc. We will use these named parameters in the next code block. To do this automatically using this `polyfitter` estimator, we need the Grid Search Cross Validation object, `GridSearchCV`. The next step is to use this to create the grid of parameters we want to loop over as in the following,

```
>>> from sklearn.grid_search import GridSearchCV
>>> gs=GridSearchCV(polyfitter,{'poly__degree':[1,2,3]},cv=4)
```

The `gs` object will loop over the polynomial degrees up to cubic using four-fold cross validation `cv=4`, like we did manually earlier. The `poly__degree` item comes from the previous `get_params` call. Now, we just apply the usual `fit` method on the training data,

```
>>> _=gs.fit(Xi,Yi)
>>> gs.grid_scores_
[mean: -3.48744, std: 2.51765, params: {'poly__degree': 1},
 mean: -36.06830, std: 28.84096, params: {'poly__degree': 2},
 mean: -0.07906, std: 0.95040, params: {'poly__degree': 3}]
```

the scores shown correspond to the cross validation scores for each of the parameters (e.g., polynomial degrees) using four-fold cross-validation. Note that the higher scores are better here and the cubic polynomial is best, as we observed earlier. The default R^2 metric is used for the scoring in this case as opposed to mean-squared-error. The validation results of this pipeline for the quadratic fit are shown in Fig. 4.12, and for the cubic fit, in Fig. 4.13. This can be changed by passing the `scoring=make_scorer(mean_squared_error)` keyword argument to `GridSearchCV`. There is also `RandomizedSearchCV` that does does not necessarily evaluate every point on the grid and instead randomly samples the grid according to an input probability distribution. This is very useful for a large number of hyper-parameters.

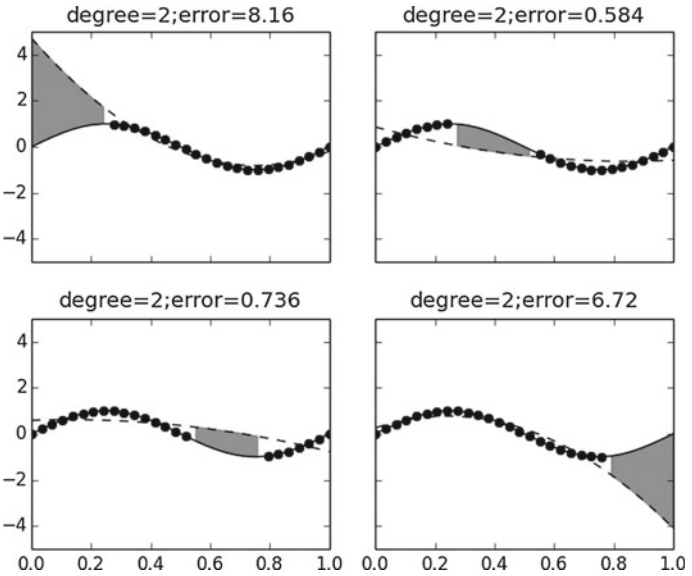

Fig. 4.12 This shows the folds and errors as in Figs. 4.10 and 4.11. The *shaded areas* show the errors in each respective test set for the quadratic model

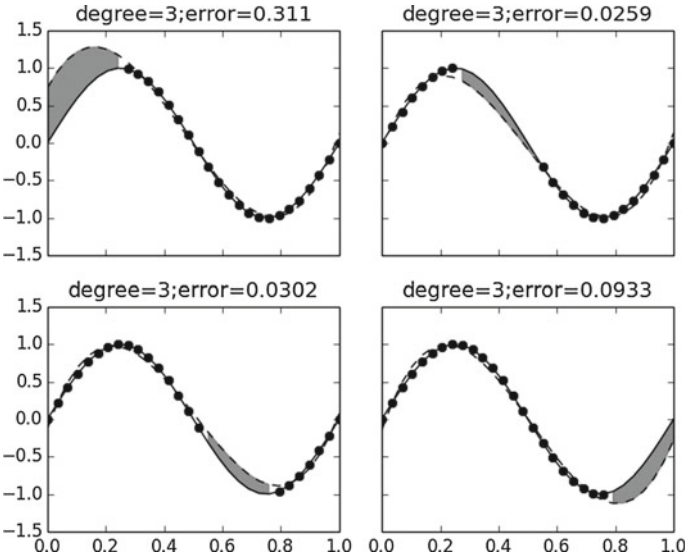

Fig. 4.13 This shows the folds and errors. The *shaded areas* show the errors in each respective test set for the cubic model

4.3.5 Bias and Variance

So far, we have been thinking about the average error in terms of in-samples and out-samples, but this depends on a particular training data set. What we want is a concept that extends to *all* possible training data and captures the performance of the estimator in that setting. For example, our ultimate estimator, $\hat{f}$ is derived from a particular set of training data ($\mathcal{D}$) and is thus denoted, $\hat{f}_{\mathcal{D}}$. This makes the out-of-sample error explicitly, $E_{\texttt{out}}(\hat{f}_{\mathcal{D}})$. To eliminate the dependence on a particular set of training data set, we have to compute the expectation across all training data sets,

$$\mathbb{E}_{\mathcal{D}} E_{\texttt{out}}(\hat{f}_{\mathcal{D}}) = \texttt{bias} + \texttt{var}$$

where

$$\texttt{bias}(x) = (\overline{\hat{f}}(x) - f(x))^2$$

and

$$\texttt{var}(x) = \mathbb{E}_{\mathcal{D}}(\hat{f}_{\mathcal{D}}(x) - \overline{\hat{f}}(x))^2$$

and where $\overline{\hat{f}}$ is the mean of all estimators for all data sets. There is nothing to say that such a mean is an estimator that could have arisen from any *particular* training data, however. It just implies that for any particular point x, the mean of the values of all the estimators is $\overline{\hat{f}}(x)$. Therefore, `bias` captures the sense that, even if all possible data were presented to the learning method, it would still differ from the target function by this amount. On the other hand `var` shows the variation in the final hypothesis, depending on the training data set, notwithstanding the target function. Thus, the tension between approximation and generalization is captured by these two terms. For example, suppose there is only one hypothesis. Then, $\texttt{var} = 0$ because there can be no variation due to a particular set of training data because no matter what that training data is, the learning method always selects the one and only hypothesis. In this case, the bias could be very large, because there is no opportunity for the learning method to alter the hypothesis due to the training data, and the method can only ever pick the single hypothesis!

Let's construct an example to make this concrete. Suppose we have a hypothesis set consisting of all linear regressions without an intercept term, $h(x) = ax$. The training data consists of only two points $\{(x_i, \sin(\pi x_i))\}_{i=1}^{2}$ where x_i is drawn uniformly from the interval $[-1, 1]$. From Sect. 3.7 on linear regression, we know that the solution for a is the following,

$$a = \frac{\mathbf{x}^T \mathbf{y}}{\mathbf{x}^T \mathbf{x}} \tag{4.3.5.1}$$

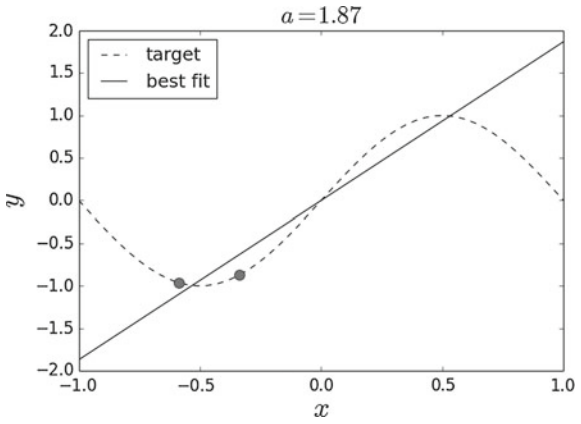

Fig. 4.14 For a two-element training set consisting of the points shown, the line is the best fit over the hypothesis set, $h(x) = ax$

where $\mathbf{x} = [x_1, x_2]$ and $\mathbf{y} = [y_1, y_2]$. The $\overline{\hat{f}}(x)$ represents the solution over all possible sets of training data for a fixed x. The following code shows how to construct the training data,

```
>>> from scipy import stats
>>> def gen_sindata(n=2):
...     x=stats.uniform(-1,1) # define random variable
...     v = x.rvs((n,1)) # generate sample
...     y = np.sin(np.pi*v) # use sample for sine
...     return (v,y)
...
```

Again, using Scikit-learn's `LinearRegression` object, we can compute the a parameter. Note that we have to set `fit_intercept=False` keyword to suppress the default automatic fitting of the intercept (Fig. 4.14).

```
>>> lr = LinearRegression(fit_intercept=False)
>>> lr.fit(*gen_sindata(2))
LinearRegression(copy_X=True, fit_intercept=False, normalize=False)
>>> lr.coef_
array([[ 2.0570357]])
```

Programming Tip

Note that we designed `gen_sindata` to return a tuple to use the automatic unpacking feature of Python functions in `lr.fit(*gen_sindata())`. In other words, using the asterisk notation means we don't have to separately assign the outputs of `gen_sindata` before using them for `lr.fit`.

In this case, $\overline{\hat{f}}(x) = \overline{a}x$, where $\overline{a}$ the expected value of the parameter over *all* possible training data sets. Using our knowledge of probability, we can write this out explicitly as the following,

$$\overline{a} = \mathbb{E}\left(\frac{x_1 \sin(\pi x_1) + x_2 \sin(\pi x_2)}{x_1^2 + x_2^2}\right)$$

where $\mathbf{x} = [x_1, x_2]$ and $\mathbf{y} = [\sin(\pi x_1), \sin(\pi x_2)]$ in Eq. 4.3.5.1. However, computing this expectation analytically is hard, but for this specific situation, $\overline{a} \approx 1.43$. To get this value using simulation, we just loop over the process, collect the outputs, and the average them as in the following,

```
>>> a_out=[] # output container
>>> for i in range(100):
...     _=lr.fit(*gen_sindata(2))
...     a_out.append(lr.coef_[0,0])
...
>>> np.mean(a_out) # approx 1.43
1.3753786877340366
```

Note that you may have to loop over many more iterations to get close to the purported value. The `var` requires the variance of a,

$$\texttt{var}(x) = \mathbb{E}((a - \overline{a})x)^2 = x^2\mathbb{E}(a - \overline{a})^2 \approx 0.71x^2$$

The `bias` is the following,

$$\texttt{bias}(x) = (\sin(\pi x) - \overline{a}x)^2$$

Figure 4.15 shows the `bias`, `var`, and mean-squared-error for this problem. Notice that there is zero bias and zero variance when $x = 0$. This is because the learning method cannot help but get that correct because all the hypotheses happen to match the value of the target function at that point! Likewise, the `var` is zero because all possible pairs, which constitute the training data, are fitted through zero because $h(x) = ax$ has no choice but to go through zero. The errors are worse at the end

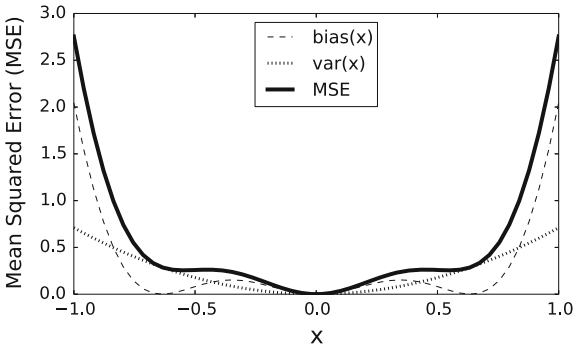

Fig. 4.15 These curves decompose the mean squared error into its constituent bias and variance for this example

points. As we discussed in our statistics chapter, those points have the most leverage against the hypothesized models and result in the worst errors. Notice that reducing the edge-errors depends on getting exactly those points near the edges as training data. The sensitivity to a particular data set is reflected in this behavior.

What if we had more than two points in the training data? What would happen to `var` and `bias`? Certainly, the `var` would decrease because it would be harder and harder to generate training data sets that would be substantially different from each other. The bias would also decrease because more points in the training data means better approximation of the sine function over the interval. What would happen if we changed the hypothesis set to include more complex polynomials? As we have already seen with our polynomial regression earlier in this chapter, we would see the same overall effect as here, but with relatively smaller absolute errors and the same edge effects we noted earlier. The corresponding Jupyter/IPython Notebook has the source code to help you can try these ideas and see for yourself.

4.3.6 Learning Noise

We have thus far not considered the effect of noise in our analysis of learning. The following example should help resolve this. Let's suppose we have the following scalar target function,

$$y(\mathbf{x}) = \mathbf{w}_o^T \mathbf{x} + \eta$$

where $\eta \sim \mathcal{N}(0, \sigma^2)$ is an additive noise term and $\mathbf{w}, \mathbf{x} \in \mathbb{R}^d$. Furthermore, we have n measurements of y. This means the training set consists of $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$. Stacking the measurements together into a vector format,

$$\mathbf{y} = \mathbf{X}\mathbf{w}_o + \boldsymbol{\eta}$$

with $\mathbf{y}, \in \mathbb{R}^n$, $\mathbf{w}_o \in \mathbb{R}^d$ and $\mathbf{X}$ contains $\mathbf{x}_i$ as columns. The hypothesis set consists of all linear models,

$$h(\mathbf{w}, \mathbf{x}) = \mathbf{w}^T \mathbf{x}$$

We need to the learn the correct $\mathbf{w}$ from the hypothesis set given the training data. So far, this is the usual setup for the problem, but how does the noise factor play to this? In our usual situation, the training set consists of randomly chosen elements from a larger space. In this case, that would be the same as getting random sets of $\mathbf{x}_i$ vectors. That still happens in this case, but the problem is that even if the same $\mathbf{x}_i$ appears twice, it will not be associated with the same y value due the additive noise coming from η. To keep this simple, we assume that there is a fixed set of $\mathbf{x}_i$ vectors and that we get all of them in the training set. For every specific training set, we know how to solve for the MMSE from our earlier statistics work,

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Given this setup, what is the in-sample mean-squared-error? Because this is the MMSE solution, we know from our study of the associated orthogonality of such systems that we have,

$$E_{\text{in}} = \|\mathbf{y}\|^2 - \|\mathbf{X}\mathbf{w}\|^2 \tag{4.3.6.1}$$

where our best hypothesis, $\mathbf{h} = \mathbf{X}\mathbf{w}$. Now, we want to compute the expectation of this over the distribution of η. For instance, for the first term, we want to compute,

$$\mathbb{E}|\mathbf{y}|^2 = \frac{1}{n}\mathbb{E}(\mathbf{y}^T\mathbf{y}) = \frac{1}{n}\text{Tr}\,\mathbb{E}(\mathbf{y}\mathbf{y}^T)$$

where Tr is the matrix trace operator (i.e., sum of the diagonal elements). Because each η are independent, we have

$$\text{Tr}\,\mathbb{E}(\mathbf{y}\mathbf{y}^T) = \text{Tr}\,\mathbf{X}\mathbf{w}_o\mathbf{w}_o^T\mathbf{X}^T + \sigma^2\text{Tr}\,\mathbf{I} = \text{Tr}\,\mathbf{X}\mathbf{w}_o\mathbf{w}_o^T\mathbf{X}^T + n\sigma^2 \tag{4.3.6.2}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. For the second term in Eq. (4.3.6.1), we have

$$|\mathbf{X}\mathbf{w}|^2 = \text{Tr}\,\mathbf{X}\mathbf{w}\mathbf{w}^T\mathbf{X}^T = \text{Tr}\,\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}\mathbf{y}^T\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

The expectation of this is the following,

$$\mathbb{E}|\mathbf{X}\mathbf{w}|^2 = \text{Tr}\,\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbb{E}(\mathbf{y}\mathbf{y}^T)\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \tag{4.3.6.3}$$

which, after substituting in Eq. 4.3.6.2, yields,

$$\mathbb{E}|\mathbf{X}\mathbf{w}|^2 = \text{Tr}\,\mathbf{X}\mathbf{w}_o\mathbf{w}_o^T\mathbf{X}^T + \sigma^2 d \tag{4.3.6.4}$$

Next, assembling Eq. 4.3.6.1 from this and Eq. 4.3.6.2 gives,

$$\mathbb{E}(E_{\text{in}}) = \frac{1}{n}E_{\text{in}} = \sigma^2\left(1 - \frac{d}{n}\right) \tag{4.3.6.5}$$

which provides an explicit relationship between the noise power, σ^2, the complexity of the method (d) and the number of training samples (n). This is very illustrative because it reveals the ratio d/n, which is a statement of the trade-off between model complexity and in-sample data size. From our analysis of the VC-dimension, we already know that there is a complicated bound that represents the penalty for complexity, but this problem is unusual in that we can actually derive an expression for this without resorting to bounding arguments. Furthermore, this result shows, that with a very large number of training examples ($n \to \infty$), the expected in-sample

error approaches σ^2. Informally, this means that the learning method cannot *generalize* from noise and thus can only reduce the expected in-sample error by memorizing the data (i.e., $d \approx n$).

The corresponding analysis for the expected out-of-sample error is similar, but more complicated because we don't have the orthogonality condition. Also, the out-of-sample data has different noise from that used to derive the weights, $\mathbf{w}$. This results in extra cross-terms,

$$E_{\text{out}} = \text{Tr}\Big(\mathbf{X}\mathbf{w}_o\mathbf{w}_o^T\mathbf{X}^T + ,,^T + \mathbf{X}\mathbf{w}\mathbf{w}^T\mathbf{X}^T - \mathbf{X}\mathbf{w}\mathbf{w}_o^T\mathbf{X}^T - \mathbf{X}\mathbf{w}_o\mathbf{w}^T\mathbf{X}^T\Big) \tag{4.3.6.6}$$

where we are using the , notation for the noise in the out-of-sample case, which is different from that in the in-sample case. Simplifying this leads to the following,

$$\mathbb{E}(E_{\text{out}}) = \text{Tr}\,\sigma^2\mathbf{I} + \sigma^2\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \tag{4.3.6.7}$$

Then, assembling all of this gives,

$$\mathbb{E}(E_{\text{out}}) = \sigma^2\left(1 + \frac{d}{n}\right) \tag{4.3.6.8}$$

which shows that even in the limit of large n, the expected out-of-sample error also approaches the noise power limit, σ^2. This shows that memorizing the in-sample data (i.e., $d/n \approx 1$) imposes a proportionate penalty on the out-of-sample performance (i.e., $\mathbb{E}E_{\text{out}} \approx 2\sigma^2$ when $\mathbb{E}E_{\text{in}} \approx 0$).

The following code simulates this important example:

```
>>> def est_errors(d=3,n=10,niter=100):
...     assert n>d
...     wo = np.matrix(arange(d)).T
...     Ein = list()
...     Eout = list()
...     # choose any set of vectors
...     X = np.matrix(np.random.rand(n,d))
...     for ni in xrange(niter):
...         y = X*wo + np.random.randn(X.shape[0],1)
...         # training weights
...         w = np.linalg.inv(X.T*X)*X.T*y
...         h = X*w
...         Ein.append(np.linalg.norm(h-y)**2)
...         # out of sample error
...         yp = X*wo + np.random.randn(X.shape[0],1)
...         Eout.append(np.linalg.norm(h-yp)**2)
...     return (np.mean(Ein)/n,np.mean(Eout)/n)
...
```

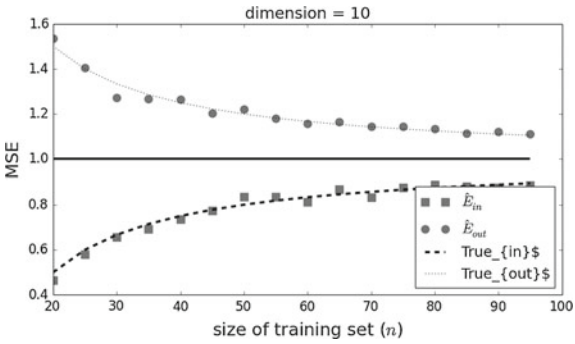

Fig. 4.16 The *dots* show the learning curves estimated from the simulation and the *solid lines* show the corresponding terms for our analytical result. The *horizontal line* shows the variance of the additive noise ($\sigma^2 = 1$ in this case). Both the expected in-sample and out-of-sample errors asymptotically approach this line

Programming Tip

Python has an `assert` statement to make sure that certain entry conditions for the variables in the function are satisfied. It is a good practice to use reasonable assertions at entry and exit to improve the quality of code.

The following runs the simulation for the given value of d.

```
>>> d=10
>>> xi = arange(d*2,d*10,d//2)
>>> ei,eo=np.array([est_errors(d=d,n=n,niter=100) for n in xi]).T
```

which results in Fig. 4.16. This figure shows the estimated expected in-sample and out-of-sample errors from our simulation compared with our corresponding analytical result. The heavy horizontal line shows the variance of the additive noise $\sigma^2 = 1$. Both these curves approach this asymptote because the noise is the ultimate learning limit for this problem. For a given dimension d, even with an infinite amount of training data, the learning method cannot generalize beyond the limit of the noise power. Thus, the expected generalization error is $\mathbb{E}(E_{\text{out}}) - \mathbb{E}(E_{\text{in}}) = 2\sigma^2 \frac{d}{n}$.

4.4 Decision Trees

A decision tree is the easiest classifer to understand, interpret, and explain. A decision tree is constructed by recursively splitting the data set into a sequence of subsets based on if-then questions. The training set consists of pairs $(\mathbf{x}, y)$ where $\mathbf{x} \in \mathbb{R}^d$ where d is the number of features available and where y is the corresponding label. The learning method splits the training set into groups based on $\mathbf{x}$ while attempting to keep the assignments in each group as uniform as possible. In order to do this, the learning method must pick a feature and an associated threshold for that feature upon which to divide the data. This is tricky to explain in words, but easy to see with an example. First, let's set up the Scikit-learn classifer,

```
>>> from sklearn import tree
>>> clf = tree.DecisionTreeClassifier()
```

Let's also create some example data,

```
>>> import numpy as np
>>> M=np.fromfunction(lambda i,j:j>=2,(4,4)).astype(int)
>>> print M
[[0 0 1 1]
 [0 0 1 1]
 [0 0 1 1]
 [0 0 1 1]]
```

Programming Tip

The `fromfunction` creates Numpy arrays using the indicies as inputs to a function whose value is the corresponding array entry.

We want to classify the elements of the matrix based on their respective positions in the matrix. By just looking at the matrix, the classification is pretty simple—classify as `0` for any positions in the first two columns of the matrix, and classify `1` otherwise. Let's walk through this formally and see if this solution emerges from the decision tree. The values of the array are the labels for the training set and the indicies of those values are the elements of $\mathbf{x}$. Specifically, the training set has $\mathcal{X} = \{(i, j)\}$ and $\mathcal{Y} = \{0, 1\}$ Now, let's extract those elements and construct the training set.

```
>>> i,j = np.where(M==0)
>>> x=np.vstack([i,j]).T # build nsamp by nfeatures
>>> y = j.reshape(-1,1)*0 # 0 elements
>>> print x
[[0 0]
 [0 1]
 [1 0]
 [1 1]
 [2 0]
 [2 1]
 [3 0]
 [3 1]]
>>> print y
[[0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]]
```

Thus, the elements of `x` are the two-dimensional indicies of the values of `y`. For example, `M[x[0,0],x[0,1]] = y[0,0]`. Likewise, to complete the training set, we just need to stack the rest of the data to cover all the cases,

```
>>> i,j = np.where(M==1)
>>> x=np.vstack([np.vstack([i,j]).T,x ]) # build nsamp x nfeatures
>>> y=np.vstack([j.reshape(-1,1)*0+1,y]) # 1 elements
```

With all that established, all we have to do is train the classifer,

```
>>> clf.fit(x,y)
DecisionTreeClassifier(compute_importances=None, criterion='gini',
            max_depth=None, max_features=None, max_leaf_nodes=None,
            min_density=None, min_samples_leaf=1, min_samples_split=2,
            random_state=None, splitter='best')
```

To evaluate how the classifer performed, we can report the score,

```
>>> clf.score(x,y)
1.0
```

For this classifer, the *score* is the accuracy, which is defined as the ratio of the sum of the true-positive (TP) and true-negatives (TN) divided by the sum of all the terms, including the false terms,

$$\texttt{accuracy} = \frac{TP + TN}{TP + TN + FN + FP}$$

In this case, the classifier gets every point correctly, so $FN = FP = 0$. On a related note, two other common names from information retrieval theory are *recall* (a.k.a. sensitivity) and *precision* (a.k.a. positive predictive value, $TP/(TP + FP)$). We can visualize this tree in Fig. 4.17. The Gini coefficients (a.k.a. categorical variance) in the figure are a measure of the purity of each so-determined class. This coefficient is defined as,

$$\texttt{Gini}_m = \sum_k p_{m,k}(1 - p_{m,k})$$

where

$$p_{m,k} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

which is the proportion of observations labeled k in the mth node and $I(\cdot)$ is the usual indicator function. Note that the maximum value of the Gini coefficient is $\max \texttt{Gini}_m = 1 - 1/m$. For our simple example, half of the sixteen samples are in

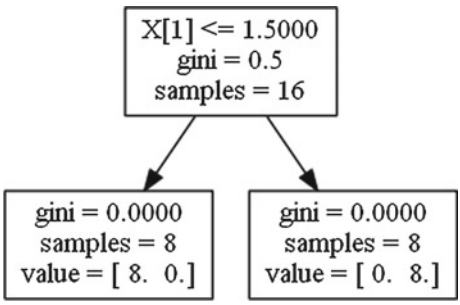

Fig. 4.17 Example decision tree. The Gini coefficient in each branch measures the purity of the partition in each node. The `samples` item in the box shows the number of items in the corresponding node in the decision tree

category `0` and the other half are in the `1` category. Using the notation above, the top box corresponds to the 0th node, so $p_{0,0} = 1/2 = p_{0,1}$. Then, $\texttt{Gini}_0 = 0.5$. The next layer of nodes in Fig. 4.17 is determined by whether or not the second dimension of the **x** data is greater than `1.5`. The Gini coefficients for each of these child nodes is zero because after the prior split, each subsequent category is pure. The `value` list in each of the nodes shows the distribution of elements in each category at each node.

To make this example more interesting, we can contaminate the data slightly,

```
>>> M[1,0]=1 # put in different class
>>> print M # now contaminated
[[0 0 1 1]
 [1 0 1 1]
 [0 0 1 1]
 [0 0 1 1]]
```

Now we have a `1` entry in the previously pure first column's second row.

```
>>> i,j = np.where(M==0)
>>> x=np.vstack([i,j]).T
>>> y = j.reshape(-1,1)*0
>>> i,j = np.where(M==1)
>>> x=np.vstack([np.vstack([i,j]).T,x])
>>> y = np.vstack([j.reshape(-1,1)*0+1,y])
>>> clf.fit(x,y)
DecisionTreeClassifier(compute_importances=None, criterion='gini',
            max_depth=None, max_features=None, max_leaf_nodes=None,
            min_density=None, min_samples_leaf=1, min_samples_split=2,
            random_state=None, splitter='best')
```

The result is shown in Fig. 4.18. Note the tree has grown significantly due to this one change! The 0th node has the following parameters, $p_{0,0} = 7/16$ and $p_{0,1} = 9/16$. This makes the Gini coefficient for the 0th node equal to $\frac{7}{16}\left(1-\frac{7}{16}\right)+\frac{9}{16}(1-\frac{9}{16}) = 0.492$. As before, the root node splits on $X[1] \leq 1.5$. Let's see if we can reconstruct the succeeding layer of nodes manually, as in the following,

```
>>> y[x[:,1]>1.5] # first node on the right
array([[1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1]])
```

This obviously has a zero Gini coefficient. Likewise, the node on the left contains the following,

```
>>> y[x[:,1]<=1.5] # first node on the left
array([[1],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0]])
```

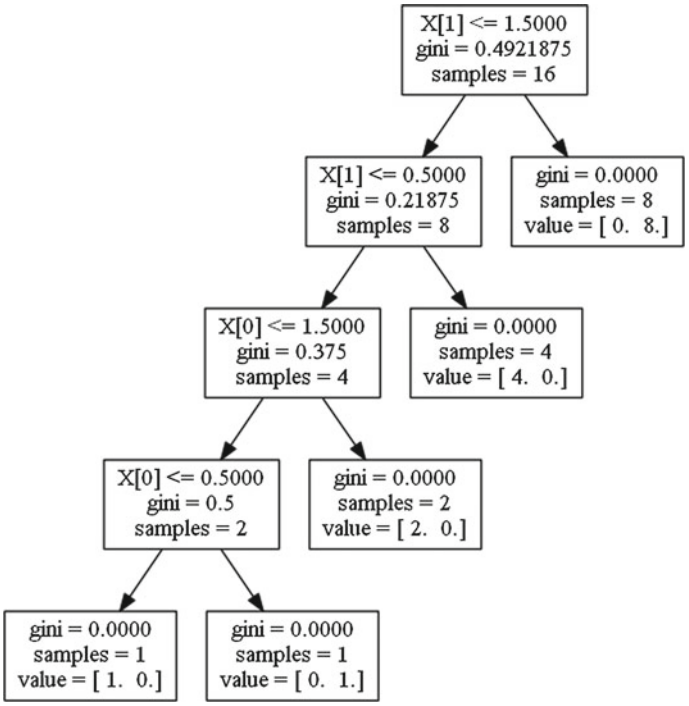

Fig. 4.18 Decision tree for contaminated data. Note that just one change in the training data caused the tree to grow five times as large as before!

The Gini coefficient in this case is computed as `(1/8)*(1-1/8)+(7/8)*(1-7/8)=0.21875`. This node splits based on `X[1]` $<$ `0.5`. The child node to the right derives from the following equivalent logic,

```
>>> np.logical_and(x[:,1]<=1.5,x[:,1]>0.5)
array([False, False, False, False, False, False, False, False, False,
       False,  True,  True, False,  True, False,  True], dtype=bool)
```

with corresponding classes,

```
>>> y[np.logical_and(x[:,1]<=1.5,x[:,1]>0.5)]
array([[0],
       [0],
       [0],
       [0]])
```

Programming Tip

The `logical_and` in Numpy provides element-wise logical conjuction. It is not possible to accomplish this with something like `0.5 < x[:, 1] <= 1.5` because of the way Python parses this syntax.

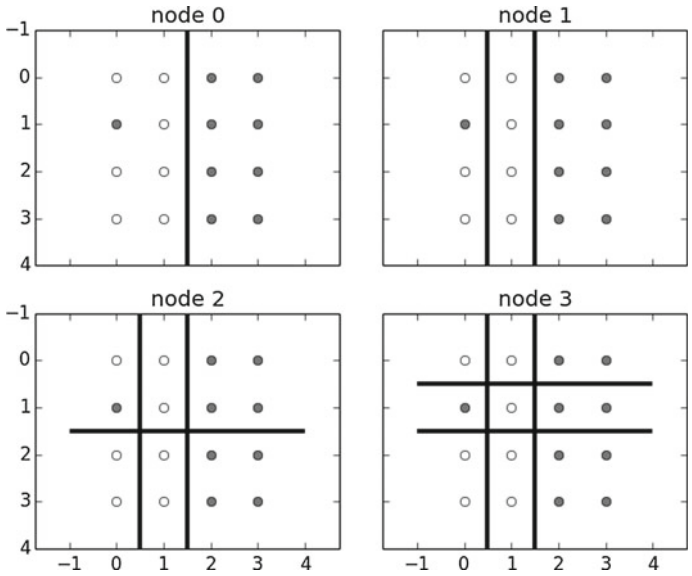

Fig. 4.19 The decision tree divides the training set into regions by splitting successively along each dimension until each region is as pure as possible

Notice that for this example as well as for the previous one, the decision tree was exactly able memorize (overfit) the data with perfect accuracy. From our discussion of machine learning theory, this is an indication potential problems in generalization.

The key step in building the decision tree is to come up with the initial split. There are the number of algorithms that can build decision trees based on different criteria, but the general idea is to control the information *entropy* as the tree is developed. In practical terms, this means that the algorithms attempt to build trees that are not excessively deep. It is well-established that this is a very hard problem to solve completely and there are many approaches to it. This is because the algorithms must make global decisions at each node of the tree using the local data available up to that point.

For this example, the decision tree partitions the $\mathcal{X}$ space into different regions corresponding to different $\mathcal{Y}$ labels as shown in Fig. 4.19. The root node at the top of Fig. 4.18 splits the input data based on $X[1] \leq 1.5$. This corresponds to the top left panel in Fig. 4.19 (i.e., `node 0`) where the vertical line divides the training data shown into two regions, corresponding to the two subsequent child nodes. The next split happens with $X[1] \leq 0.5$ as shown in the next panel of Fig. 4.19 titled `node 1`. This continues until the last panel on the lower right, where the contaminated element we injected has been isolated into its own sub-region. Thus, the last panel is a representation of Fig. 4.18, where the horizontal/vertical lines correspond to successive splits in the decision tree.

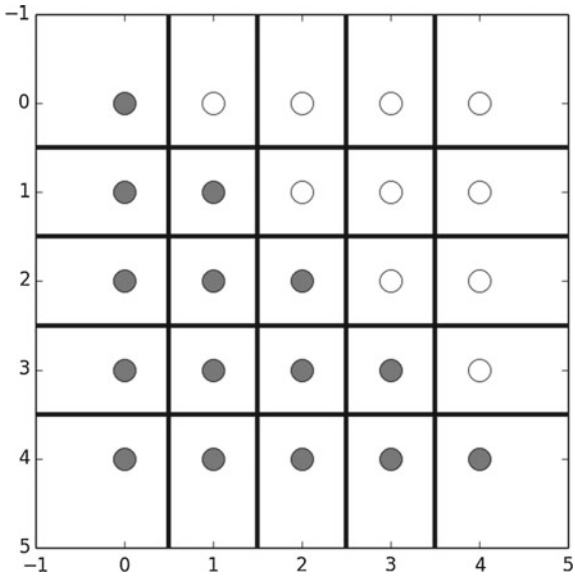

Fig. 4.20 The decision tree fitted to this triangular matrix is very complex, as shown by the number of *horizontal* and *vertical* partitions. Thus, even though the pattern in the training data is visually clear, the decision tree cannot automatically uncover it

Figure 4.20 shows another example, but now using a simple triangular matrix. As shown by the number of vertical and horizontal partitioning lines, the decision tree that corresponds to this figure is tall and complex. Notice that if we apply a simple rotational transform to the training data, we can obtain Fig. 4.21, which requires a trivial decision tree to fit. Thus, there may be transformations of the training data that simplify the decision tree, but these are very difficult to derive in general. Nonetheless, this highlights a key weakness of decision trees wherein they may be easy to understand, to train, and to deploy, but may be completely blind to such time-saving and complexity-saving transformations. Indeed, in higher dimensions, it may be impossible to even visualize the potential of such latent transformations. Thus, the advantages of decision trees can be easily outmatched by other methods that we will study later that *do* have the ability to uncover useful transformations, but which will necessarily be harder to train. Another disadvantage is that because of how decision trees are built, even a single misplaced data point can cause the tree to grow very differently. This is a symptom of high variance.

In all of our examples, the decision tree was able to memorize the training data exactly, as we discussed earlier, this is a sign of potential generalization errors. There are pruning algorithms that strategically remove some of the deepest nodes. but these are not yet fully implemented in Scikit-learn, as of this writing. Alternatively, restricting the maximum depth of the decision tree can have a similar effect. The `DecisionTreeClassifier` and `DecisionTreeRegressor` in Scikit-learn both have keyword arguments that specify maximum depth.

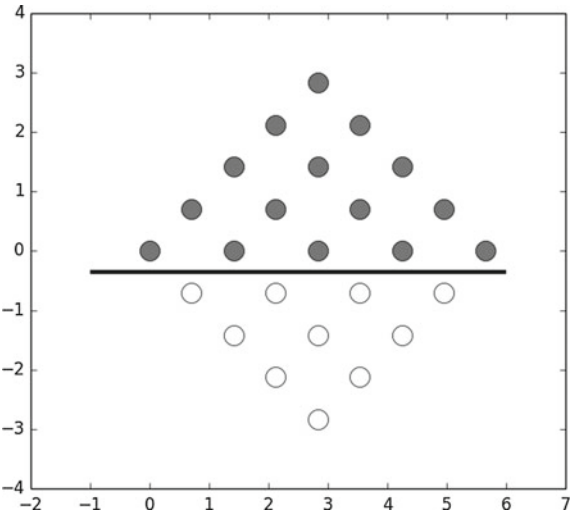

Fig. 4.21 Using a simple rotation on the training data in Fig. 4.20, the decision tree can now easily fit the training data with a single partition

4.4.1 Random Forests

It is possible to combine a set of decision trees into a larger composite tree that has better performance than its individual components by using ensemble learning. This is implemented in Scikit-learn as `RandomForestClassifier`. The composite tree helps mitigate the primary weakness of decision trees—high variance. Random forest classifiers help by averaging out the predictions of many constituent trees to minimize this variance by randomly selecting subsets of the training set to train the embedded trees. On the other hand, this randomization can increase bias because there may be a subset of the training set that yields an excellent decision tree, but the averaging effect over randomized training samples washes this out in the same averaging that reduces the variance. This is a key trade-off. The following code implements a simple random forest classifer from our last example.

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> rfc = RandomForestClassifier(n_estimators=4,max_depth=2)
>>> rfc.fit(X_train,y_train.flat)
RandomForestClassifier(bootstrap=True, compute_importances=None,
            criterion='gini', max_depth=2, max_features='auto',
            max_leaf_nodes=None, min_density=None, min_samples_leaf=1,
            min_samples_split=2, n_estimators=4, n_jobs=1, oob_score=False,
            random_state=None, verbose=0)
```

Note that we have constrained the maximum depth `max_depth=2` to help with generalization. To keep things simple we have only set up a forest with four individual classifers.[3] Figure 4.23 shows the individual classifers in the forest that have been trained above. Even though all the constituent decision trees share the same training data, the random forest algorithm randomly picks feature subsets (with replacement)

[3] We have also set the random seed to a fixed value to make the figures reproducible in the IPython Notebook corresponding to this section.

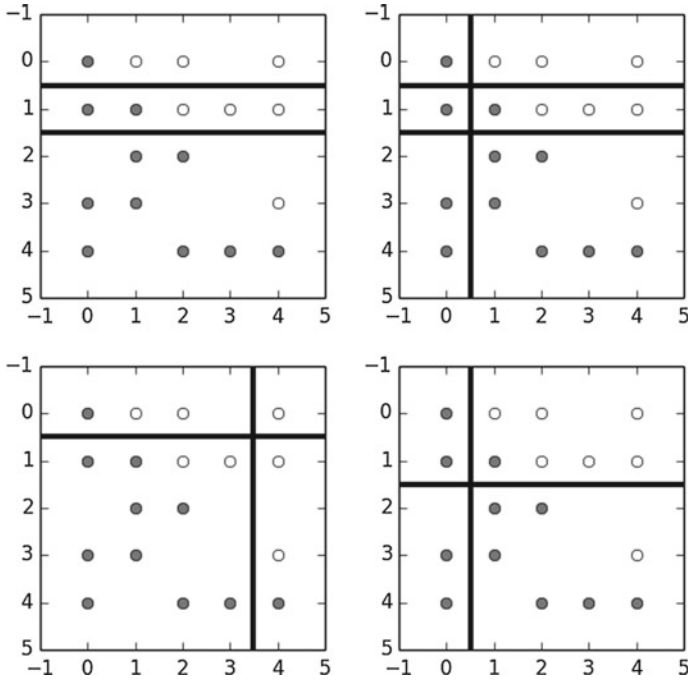

Fig. 4.22 The constituent decision trees of the random forest and how they partitioned the training set are shown in these four panels. The random forest classifier uses the individual outputs of each of the constituent trees to produce a collaborative final estimate

upon which to train individual trees. This helps avoid the tendency of decision trees to become too deep and lopsided, which hurts both performance and generalization. At the prediction step, the individual outputs of each of the constituent decision trees are put to a majority vote for the final classification. To estimate generalization errors without using cross-validation, the training elements *not* used for a particular constituent tree can be used to test that tree and form a collaborative estimate of generalization errors. This is called the *out-of-bag* estimate.

The main advantage of random forest classifiers is that they require very little tuning and provide a way to trade-off bias and variance via averaging and randomization. Furthermore, they are fast and easy to train in parallel (see the `n_jobs` keyword argument) and fast to predict. On the downside, they are less interpretable than simple decision trees. There are many other powerful tree methods in Scikit-learn like `ExtraTrees` and Gradient Boosted Regression Trees `GradientBoostingRegressor` which are discussed in the online documentation.

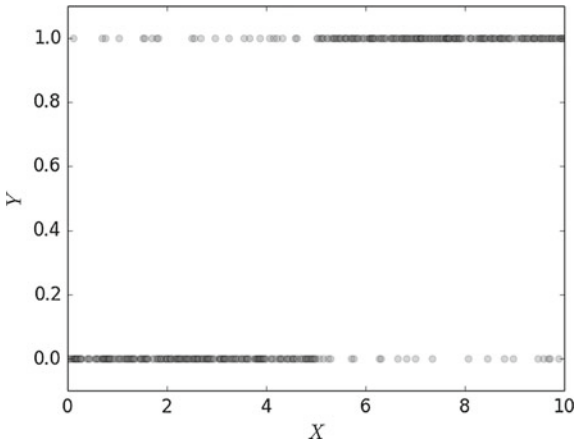

Fig. 4.23 This scatterplot shows the binary Y variables and the corresponding x data for each category

4.5 Logistic Regression

The Bernoulli distribution we studied earlier answers the question of which of two outcomes ($Y \in \{0, 1\}$) would be selected with probability, p.

$$\mathbb{P}(Y) = p^Y (1 - p)^{1-Y}$$

We also know how to solve the corresponding likelihood function for the maximum likelihood estimate of p given observations of the output, $\{Y_i\}_{i=1}^n$. However, now we want to include other factors in our estimate of p. For example, suppose we observe not just the outcomes, but a corresponding continuous variable, x. That is, the observed data is now $\{(x_i, Y_i)\}_{i=1}^n$ How can we incorporate x into our estimation of p?

The most straightforward idea is to model $p = ax + b$ where a, b are parameters of a fitted line. However, because p is a probability with value bounded between zero and one, we need to wrap this estimate in another function that can map the entire real line into the [0, 1] interval. The logistic (a.k.a. sigmoid) function has this property,

$$\theta(s) = \frac{e^s}{1 + e^s}$$

Thus, the new parameterized estimate for p is the following,

$$\hat{p} = \theta(ax + b) = \frac{e^{ax+b}}{1 + e^{ax+b}} \tag{4.5.0.1}$$

This is usually expressed using the *logit* function,

$$\texttt{logit}(t) = \log \frac{t}{1-t}$$

as,

$$\texttt{logit}(p) = b + ax$$

More continuous variables can be accommodated easily as

$$\texttt{logit}(p) = b + \sum_k a_k x_k$$

This can be further extended beyond the binary case to multiple target labels. The maximum likelihood estimate of this uses numerical optimization methods that are implemented in Scikit-learn.

Let's construct some data to see how this works. In the following, we assign class labels to a set of randomly scattered points in the two-dimensional plane,

```
>>> import numpy as np
>>> from matplotlib.pylab import subplots
>>> v = 0.9
>>> @np.vectorize
... def gen_y(x):
...     if x<5: return np.random.choice([0,1],p=[v,1-v])
...     else:   return np.random.choice([0,1],p=[1-v,v])
...
>>> xi = np.sort(np.random.rand(500)*10)
>>> yi = gen_y(xi)
```

Programming Tip

The `np.vectorize` decorator used in the code above makes it easy to avoid looping in code that uses Numpy arrays by embedding the looping semantics inside of the so-decorated function. Note, however, that this does not necessarily accelerate the wrapped function. It's mainly for convenience.

Figure 4.23 shows a scatter plot of the data we constructed in the above code, $\{(x_i, Y_i)\}$. As constructed, it is more likely that large values of x correspond to $Y = 1$. On the other hand, values of $x \in [4, 6]$ of either category are heavily overlapped. This means that x is not a particularly strong indicator of Y in this region. Figure 4.24 shows the fitted logistic regression curve against the same data. The points along the curve are the probabilities that each point lies in either of the two categories. For large values of x the curve is near one, meaning that the probability that the associated Y value is equal to one. On the other extreme, small values of x mean that this probability is close to zero. Because there are only two possible categories, this means that the probability of $Y = 0$ is thereby higher. The region in the middle corresponding to the middle probabilities reflect the ambiguity between the two

catagories because of the overlap in the data for this region. Thus, logistic regression cannot make a strong case for one category here. The following code fits the logistic regression model,

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression()
>>> lr.fit(np.c_[xi],yi)
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, penalty='l2', random_state=None, tol=0.0001)
```

For a deeper understanding of logistic regression, we need to alter our notation slightly and once again use our projection methods. More generally we can rewrite Eq. 4.5.0.1 as the following,

$$p(\mathbf{x}) = \frac{1}{1+\exp(-\boldsymbol{\beta}^T\mathbf{x})} \tag{4.5.0.2}$$

where $\boldsymbol{\beta}, \mathbf{x} \in \mathbb{R}^n$. From our prior work on projection we know that the signed perpendicular distance between $\mathbf{x}$ and the linear boundary described by $\boldsymbol{\beta}$ is $\boldsymbol{\beta}^T\mathbf{x}/\|\boldsymbol{\beta}\|$. This means that the probability that is assigned to any point in $\mathbb{R}^n$ is a function of how close that point is to the linear boundary described by the following equation,

$$\boldsymbol{\beta}^T\mathbf{x} = 0$$

But there is something subtle hiding here. Note that for any $\alpha \in \mathbb{R}$,

$$\alpha\boldsymbol{\beta}^T\mathbf{x} = 0$$

describes the *same* hyperplane. This means that we can multiply $\boldsymbol{\beta}$ by an arbitrary scalar and still get the same geometry. However, because of $\exp(-\alpha\boldsymbol{\beta}^T\mathbf{x})$ in Eq. 4.5.0.2, this scaling determines the intensity of the probability attributed to $\mathbf{x}$. This is illustrated in Fig. 4.25. The panel on the left shows two categories (squares/circles)

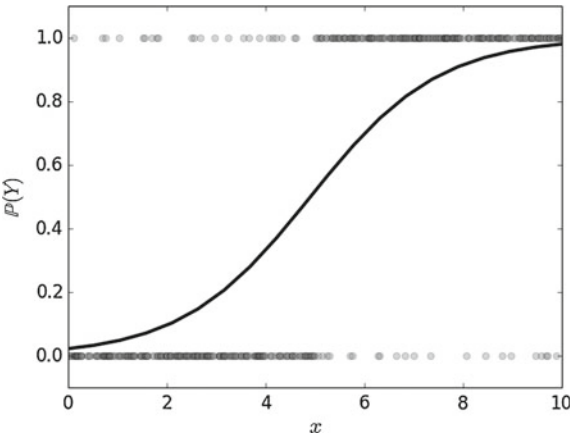

Fig. 4.24 This shows the fitted logistic regression on the data shown in Fig. 4.22. The points along the curve are the probabilities that each point lies in either of the two categories

split by the dotted line that is determined by $\beta^T \mathbf{x} = 0$. The background colors shows the probabilities assigned to points in the plane. The right panel shows that by scaling with α, we can increase the probabilities of class membership for the given points, given the exact same geometry. The points near the boundary have lower probabilities because they could easily be on the opposite side. However, by scaling by α, we can raise those probabilities to any desired level at the cost of driving the points further from the boundary closer to one. Why is this a problem? By driving the probabilities arbitrarily using α, we can overemphasize the training set at the cost of out-of-sample data. That is, we may wind up insisting on emphatic class membership of yet unseen points that are close to the boundary that otherwise would have more equivocal probabilities (say, near $1/2$). Once again, this is another manifestation of bias/variance trade-off.

Regularization is a method that controls this effect by penalizing the size of β as part of its solution. Algorithmically, logistic regression works by iteratively solving a sequence of weighted least squares problems. Regression adds a $\|\beta\|/C$ term to the least squares error. To see this in action, let's create some data from a logistic regression and see if we can recover it using Scikit-learn. Let's start with a scatter of points in the two-dimensional plane,

```
>>> x0,x1=np.random.rand(2,20)*6-3
>>> X = np.c_[x0,x1,x1*0+1] # stack as columns
```

Note that `X` has a third column of all ones. This is a trick to allow the corresponding line to be offset from the origin in the two-dimensional plane. Next, we create a linear boundary and assign the class probabilities according to proximity to the boundary.

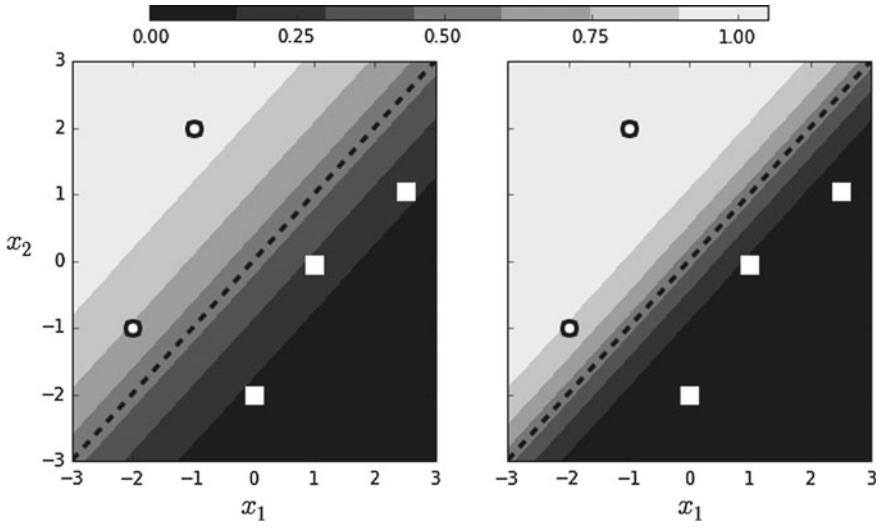

Fig. 4.25 Scaling can arbitrarily increase the probabilities of points near the decision boundary

```
>>> beta = np.array([1,-1,1]) # last coordinate for affine offset
>>> prd = X.dot(beta)
>>> probs = 1/(1+np.exp(-prd/np.linalg.norm(beta)))
>>> c = (prd>0) # boolean array class labels
```

This establishes the training data. The next block creates the logistic regression object and fits the data.

```
>>> lr = LogisticRegression()
>>> _=lr.fit(X[:,:-1],c)
```

Note that we have to omit the third dimension because of how Scikit-learn internally breaks down the components of the boundary. The resulting code extracts the corresponding β from the `LogisticRegression` object.

```
>>> betah = np.r_[lr.coef_.flat,lr.intercept_]
```

Programming Tip

The Numpy `np.r_` object provides a quick way to stack Numpy arrays horizontally instead of using `np.hstack`.

The resulting boundary is shown in the left panel in Fig. 4.26. The crosses and triangles represent the two classes we created above, along with the separating gray line. The logistic regression fit produces the dotted black line. The dark circle is the point that logistic regression categorizes incorrectly. The regularization parameter is

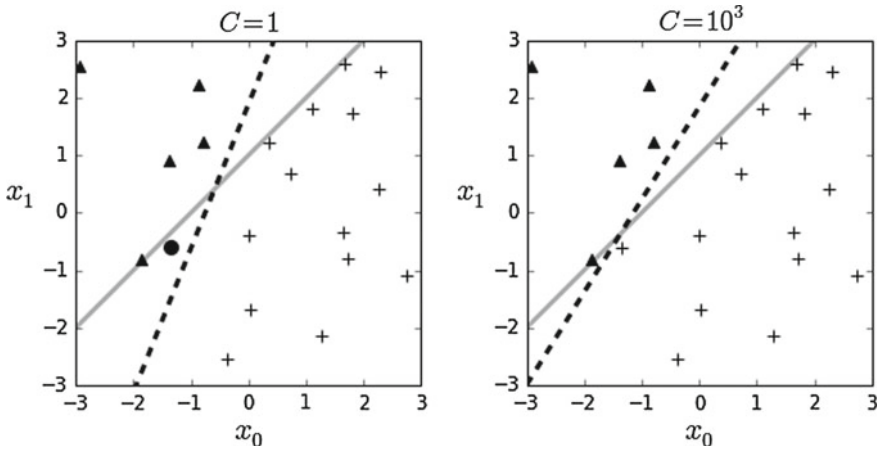

Fig. 4.26 The *left panel* shows the resulting boundary (*dashed line*) with $C = 1$ as the regularization parameter. The *right panel* is for $C = 1000$. The *gray line* is the boundary used to assign the class membership for the synthetic data. The *dark circle* is the point that logistic regression categorizes incorrectly

$C = 1$ by default. Next, we can change the strength of the regularization parameter as in the following,

```
>>> lr = LogisticRegression(C=1000)
```

and the re-fit the data to produce the right panel in Fig. 4.26. By increasing the regularization parameter, we essentially nudged the fitting algorithm to *believe* the data more than the general model. That is, by doing this we accepted more variance in exchange for better bias.

4.5.1 Generalized Linear Models

Logistic regression is one example of a wider class of generalized linear models that embed non-linear transformations in the fitting process. Let's back up and break down logistic regression into smaller parts. As usual, we want to estimate the conditional expectation $\mathbb{E}(Y|X = \mathbf{x})$. For plain linear regression, we have the following approximation,

$$\mathbb{E}(Y|X = \mathbf{x}) \approx \boldsymbol{\beta}^T \mathbf{x}$$

For notation sake, we call $r(x) := \mathbb{E}(Y|X = \mathbf{x})$ the response. For logistic regression, because $Y \in \{0, 1\}$, we have $\mathbb{E}(Y|X = \mathbf{x}) = \mathbb{P}(Y|X = \mathbf{x})$ and the transformation makes $r(\mathbf{x})$ linear.

$$\begin{aligned} \eta(\mathbf{x}) &= \boldsymbol{\beta}^T \mathbf{x} \\ &= \log \frac{r(\mathbf{x})}{1 - r(\mathbf{x})} \\ &= g(r(\mathbf{x})) \end{aligned}$$

where g is defined as the logistic *link* function. The $\eta(x)$ function is the linear predictor. Now that we have transformed the original data space using the logistic function to create the setting for the linear predictor, why don't we just do the same thing for the Y_i data? That is, for plain linear regression, we usually take data, $\{X_i, Y_i\}$ and then use it to fit an approximation to $\mathbb{E}(Y|X = x)$. If we are transforming the conditional expectation using the logarithm, which we are approximating using Y_i, then why don't we correspondingly transform the binary Y_i data? The answer is that if we did so then we would get the logarithm of zero (i.e., infinity) or one (i.e., zero), which is not workable. The alternative is to use a linear Taylor approximation, like we did earlier with the delta method, to expand the g function around $r(x)$, as in the following,

$$
\begin{aligned}
g(Y) &\approx \log \frac{r(x)}{1-r(x)} + \frac{Y - r(x)}{r(x) - r(x)^2} \\
&= \eta(x) + \frac{Y - r(x)}{r(x) - r(x)^2}
\end{aligned}
$$

The interesting part is the $Y - r(x)$ term, because this is where the class label data enters the problem. The expectation $\mathbb{E}(Y - r(x)|X) = 0$ so we can think of this differential as additive noise that dithers $\eta(x)$. The variance of $g(Y)$ is the following,

$$
\begin{aligned}
\mathbb{V}(g(Y)|X) &= \mathbb{V}(\eta(x)|X) + \frac{1}{(r(x)(1-r(x)))^2}\mathbb{V}(Y - r(x)|X) \\
&= \frac{1}{(r(x)(1-r(x)))^2}\mathbb{V}(Y - r(x)|X)
\end{aligned}
$$

Note that $\mathbb{V}(Y|X) = r(x)(1 - r(x))$ because Y is a binary variable. Ultimately, this boils down to the following,

$$
\mathbb{V}(g(Y)|X) = \frac{1}{r(x)(1-r(x))}
$$

Note that the variance is a function of x, which means it is *heteroskedastic*, meaning that the iterative minimum-variance-finding algorithm that computes $\boldsymbol{\beta}$ downplays x where $r(x) \approx 0$ and $r(x) \approx 1$ because the peak of the variance occurs where $r(x) \approx 0.5$, which are those equivocal points close to the boundary.

For generalized linear models, the above sequence is the same and consists of three primary ingredients: the linear predictor ($\eta(x)$), the link function ($g(x)$), and the *dispersion scale function*, V_{ds} such that $\mathbb{V}(Y|X) = \sigma^2 V_{ds}(r(x))$. For logistic regression, we have $V_{ds}(r(x)) = r(x)(1 - r(x))$ and $\sigma^2 = 1$. Note that absolute knowledge of σ^2 is not important because the iterative algorithm needs only a relative proportional scale. To sum up, the iterative algorithm takes a linear prediction for $\eta(x_i)$, computes the transformed responses, $g(Y_i)$, calculates the weights $w_i = \left[(g'(r(x_i))V_{ds}(r(x_i)))\right]^{-1}$, and then does a weighted linear regression of $g(y_i)$ onto x_i with the weights w_i to compute the next $\boldsymbol{\beta}$. More details can be found in the following [3–5].

4.6 Regularization

We have referred to regularization in earlier sections, but we want to develop this important idea more fully. Regularization is the mechanism by which we navigate the bias/variance trade-off. To get started, let's consider a classic constrained least squares problem,

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x}\|_2^2 \\ \text{subject to:} \quad & x_0 + 2x_1 = 1 \end{aligned}$$

where $\|\mathbf{x}\|_2 = \sqrt{x_0^2 + x_1^2}$ is the L_2 norm. Without the constraint, it would be easy to minimize the objective function—just take $\mathbf{x} = 0$. Otherwise, suppose we somehow know that $\|\mathbf{x}\|_2 < c$, then the locus of points defined by this inequality is the circle in Fig. 4.27. The constraint is the line in the same figure. Because every value of c defines a circle, the constraint is satisfied when the circle touches the line. The circle can touch the line at many different points, but we are only interested in the smallest such circle because this is a minimization problem. Intuitively, this means that we *inflate* a L_2 ball at the origin and stop when it just touches the contraint. The point of contact is our L_2 minimization solution.

We can obtain the same result using the method of Lagrange multipliers. We can rewrite the entire L_2 minimization problem as one objective function using the Lagrange multiplier, λ,

$$J(x_0, x_1, \lambda) = x_0^2 + x_1^2 + \lambda(1 - x_0 - x_1)$$

and solve this as an ordinary function using calculus. Let's do this using Sympy.

```
>>> import sympy as S
>>> S.var('x:2 l',real=True)
(x0, x1, l)
>>> J=S.Matrix([x0,x1]).norm()**2 + l*(1-x0-2*x1)
>>> sol=S.solve(map(J.diff,[x0,x1,l]))
>>> print sol
{x0: 1/5, x1: 2/5, l: 2/5}
```

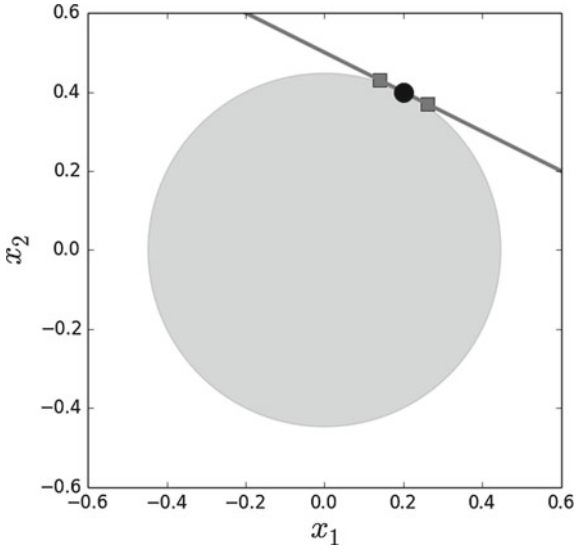

Fig. 4.27 The solution of the constrained L_2 minimization problem is at the point where the constraint (*dark line*) intersects the L_2 ball (*gray circle*) centered at the origin. The point of intersection is indicated by the *dark circle*. The two neighboring *squares* indicate points on the line that are close to the solution

Programming Tip

Using the `Matrix` object is overkill for this problem but it does demonstrate how Sympy's matrix machinery works. In this case, we are using the `norm` method to compute the L_2 norm of the given elements. Using `S.var` defines Sympy variables and injects them into the global namespace. It is more Pythonic to do something like `x0 = S.symbols('x0',real=True)` instead but the other way is quicker, especially for variables with many dimensions.

The solution defines the exact point where the line is tangent to the circle in Fig. 4.27. The Lagrange multiplier has incorporated the constraint into the objective function.

There is something subtle and very important about the nature of the solution, however. Notice that there are other points very close to the solution on the circle, indicated by the squares in Fig. 4.27. This closeness could be a good thing, in case it helps us actually find a solution in the first place, but it may be unhelpful in so far as it creates ambiguity. Let's hold that thought and try the same problem using the L_1 norm instead of the L_2 norm. Recall that

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{d} |x_i|$$

where d is the dimension of the vector $\mathbf{x}$. Thus, we can reformulate the same problem in the L_1 norm as in the following,

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x}\|_1 \\ \text{subject to:} \quad & x_1 + 2x_2 = 1 \end{aligned}$$

It turns out that this problem is somewhat harder to solve using Sympy, but we have convex optimization modules in Python that can help.

```
>>> from cvxpy import Variable, Problem, Minimize, norm1, norm2
>>> x=Variable(2,1,name='x')
>>> constr=[np.matrix([[1,2]])*x==1]
>>> obj=Minimize(norm1(x))
>>> p= Problem(obj,constr)
>>> p.solve()
0.49999999996804073
>>> print x.value
[[  6.20344267e-10]
 [  5.00000000e-01]]
```

Programming Tip

The `cvxy` module provides a unified and accessible interface to the powerful `cvxopt` convex optimization package, as well as other open-source solver packages.

As shown in Fig. 4.28, the constant-norm contour in the L_1 norm is shaped like a diamond instead of a circle. Furthermore, the solutions found in each case are different. Geometrically, this is because inflating the circular L_2 reaches out in all directions whereas the L_1 ball creeps out along the principal axes. This effect is much more pronounced in higher dimensional spaces where L_1-balls get more spikey.[4] Like the L_2 case, there are also neighboring points on the constraint line, but notice that these are not close to the boundary of the corresponding L_1 ball, as they were in the L_2 case. This means that these would be harder to confuse with the optimal solution because they correspond to a substantially different L_1 ball.

To double-check our earlier L_2 result, we can also use the `cvxpy` module to find the L_2 solution as in the following code,

```
>>> constr=[np.matrix([[1,2]])*x==1]
>>> obj=Minimize(norm2(x)) #L2 norm
>>> p= Problem(obj,constr)
>>> p.solve()
0.4472135953578661
>>> print x.value
[[ 0.2]
 [ 0.4]]
```

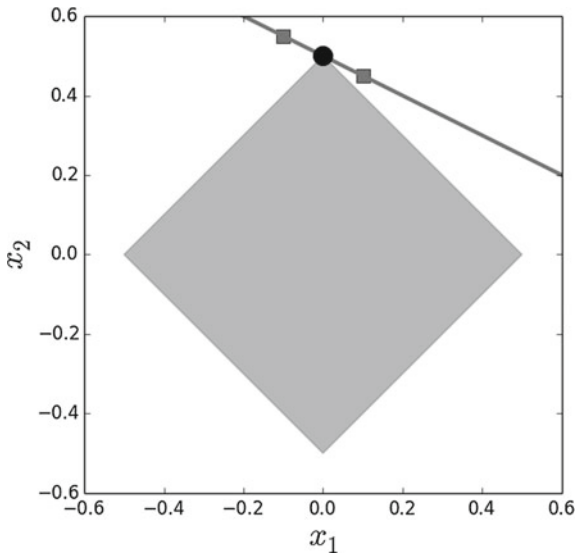

Fig. 4.28 The diamond is the L_1 ball in two dimensions and the line is the constraint. The point of intersection is the solution to the optimization problem. Note that for L_1 optimization, the two nearby points on the constraint (*squares*) do not touch the L_1 ball. Compare this with Fig. 4.27

[4] We discussed the geometry of high dimensional space when we covered the curse of dimensionality in the statistics chapter.

The only change to the code is the L_2 norm and we get the same solution as before.

Let's see what happens in higher dimensions for both L_2 and L_1 as we move from two dimensions to four dimensions.

```
>>> x=Variable(4,1,name='x')
>>> constr=[np.matrix([[1,2,3,4]])*x==1]
>>> obj=Minimize(norm1(x))
>>> p= Problem(obj,constr)
>>> p.solve()
0.24999999913550727
>>> print x.value
[[  3.88487127e-10]
 [  8.33295433e-10]
 [  7.97158525e-10]
 [  2.49999999e-01]]
```

And also in the L_2 case with the following code,

```
>>> constr=[np.matrix([[1,2,3,4]])*x==1]
>>> obj=Minimize(norm2(x))
>>> p= Problem(obj,constr)
>>> p.solve()
0.18257418572129205
>>> print x.value
[[ 0.03333333]
 [ 0.06666667]
 [ 0.1       ]
 [ 0.13333333]]
```

Note that the L_1 solution has selected out only one dimension for the solution, as the other components are effectively zero. This is not so with the L_2 solution, which has meaningful elements in multiple coordinates. This is because the L_1 problem has many pointy corners in the four dimensional space that poke at the hyperplane that is defined by the constraint. This essentially means the subsets (namely, the points at the corners) are found as solutions because these touch the hyperplane. This effect becomes more pronounced in higher dimensions, which is the main benefit of using the L_1 norm as we will see in the next section.

4.6.1 Ridge Regression

Now that we have a sense of the geometry of the situation, let's revisit our classic linear regression probem. To recap, we want to solve the following problem,

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \|y - \mathbf{X}\boldsymbol{\beta}\|$$

where $\mathbf{X} = \left[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_p\right]$ and $\mathbf{x}_i \in \mathbb{R}^n$. Furthermore, we assume that the p column vectors are linearly independent (i.e., $\texttt{rank}(\mathbf{X}) = p$). Linear regression produces the $\boldsymbol{\beta}$ that minimizes the mean squared error above. In the case where $p = n$, there is a unique solution to this problem. However, when $p < n$, then there are infinitely many solutions.

To make this concrete, let's work this out using Sympy. First, let's define an example **X** and **y** matrix,

```
>>> import sympy as S
>>> from sympy import Matrix
>>> X = Matrix([[1,2,3],
...             [3,4,5]])
>>> y = Matrix([[1,2]]).T
```

Now, we can define our coefficient vector β using the following code,

```
>>> b0,b1,b2=S.symbols('b:3',real=True)
>>> beta = Matrix([[b0,b1,b2]]).T # transpose
```

Next, we define the objective function we are trying to minimize

```
>>> obj=(X*beta -y).norm(ord=2)**2
```

Programming Tip

The Sympy `Matrix` class has useful methods like the `norm` function used above to define the objective function. The `ord=2` means we want to use the L_2 norm. The expression in parenthesis evaluates to a `Matrix` object.

Note that it is helpful to define real variables using the keyword argument whenever applicable because it relieves Sympy's internal machinery of dealing with complex numbers. Finally, we can use calculus to solve this by setting the derivatives of the objective function to zero.

```
>>> sol=S.solve([obj.diff(i) for i in beta])
>>> beta.subs(sol)
Matrix([
[         b2],
[-2*b2 + 1/2],
[         b2]])
```

Notice that the solution does not uniquely specify all the components of the `beta` variable. This is a consequence of the $p < n$ nature of this problem where $p = 2$ and $n = 3$. While the existence of this ambiguity does not alter the solution,

```
>>> obj.subs(sol)
0
```

But it does change the length of the solution vector `beta`,

```
>>> beta.subs(sol).norm(2)
sqrt(2*b2**2 + (-2*b2 + 1/2)**2)
```

If we want to minimize this length we can easily use the same calculus as before,

```
>>> S.solve((beta.subs(sol).norm()**2).diff())
[1/6]
```

This provides the solution of minimum length in the L_2 sense,

```
>>> betaL2=beta.subs(sol).subs(b2,S.Rational(1,6))
>>> betaL2
Matrix([
[1/6],
[1/6],
[1/6]])
```

But what is so special about solutions of minimum length? For machine learning, driving the objective function to zero is symptomatic of overfitting the data. Usually, at the zero bound, the machine learning method has essentially memorized the training data, which is bad for generalization. Thus, we can effectively stall this problem by defining a region for the solution that is away from the zero-bound.

$$\begin{aligned} \underset{\boldsymbol{\beta}}{\text{minimize}} \quad & \|y - \mathbf{X}\boldsymbol{\beta}\|_2^2 \\ \text{subject to:} \quad & \|\boldsymbol{\beta}\|_2 < c \end{aligned}$$

where c is the tuning parameter. Using the same process as before, we can re-write this as the following,

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \|y - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \alpha\|\boldsymbol{\beta}\|_2^2$$

where α is the tuning parameter. These are the *penalized* or Lagrange forms of these problems derived from the constrained versions. The objective function is penalized by the $\|\boldsymbol{\beta}\|_2$ term. For L_2 penalization, this is called *ridge* regression. This is implemented in Scikit-learn as `Ridge`. The following code sets this up for our example,

```
>>> from sklearn.linear_model import Ridge
>>> clf = Ridge(alpha=100.0,fit_intercept=False)
>>> clf.fit(np.array(X).astype(float),np.array(y).astype(float))
Ridge(alpha=100.0, copy_X=True, fit_intercept=False, max_iter=None,
   normalize=False, solver='auto', tol=0.001)
```

Note that the `alpha` scales of the penalty for the $\|\boldsymbol{\beta}\|_2$. We set the `fit_intercept=False` argument to omit the extra offset term from our example. The corresponding solution is the following,

```
>>> print clf.coef_
[[ 0.0428641   0.06113005  0.07939601]]
```

To double-check the solution, we can use some optimization tools from Scipy and our previous Sympy analysis, as in the following,

```
>>> from scipy.optimize import minimize
>>> f   = S.lambdify((b0,b1,b2),obj+beta.norm()**2*100.)
>>> g   = lambda x:f(x[0],x[1],x[2])
>>> out = minimize(g,[.1,.2,.3]) # initial guess
>>> out.x
array([ 0.0428641 ,  0.06113005,  0.079396  ])
```

Programming Tip

We had to define the additional `g` function from the lambda function we created from the Sympy expression in `f` because the `minimize` function expects a single object vector as input instead of a three separate arguments.

which produces the same answer as the `Ridge` object. To better understand the meaning of this result, we can re-compute the mean squared error solution to this problem in one step using matrix algebra instead of calculus,

```
>>> betaLS=X.T*(X*X.T).inv()*y
>>> betaLS
Matrix([
[1/6],
[1/6],
[1/6]])
```

Notice that this solves the posited problem exactly,

```
>>> X*betaLS-y
Matrix([
[0],
[0]])
```

This means that the first term in the objective function goes to zero,

$$\|y - \mathbf{X}\boldsymbol{\beta}_{LS}\| = 0$$

But, let's examine the L_2 length of this solution versus the ridge regression solution,

```
>>> print betaLS.norm().evalf(), np.linalg.norm(clf.coef_)
0.288675134594813 0.108985964126
```

Thus, the ridge regression solution is shorter in the L_2 sense, but the first term in the objective function is not zero for ridge regression,

```
>>> print (y-X*clf.coef_.T).norm()**2
1.86870864136429
```

Ridge regression solution trades fitting error ($\|y - \mathbf{X}\boldsymbol{\beta}\|_2$) for solution length ($\|\boldsymbol{\beta}\|_2$).

Let's see this in action with a familiar example from Sect. 3.12.4. Consider Fig. 4.29. For this example, we created our usual chirp signal and attempted to fit it with a high-dimensional polynomial, as we did in Sect. 4.3.4. The lower panel is the same except with ridge regression. The shaded gray area is the space between the true signal and the approximant in both cases. The horizontal hash marks indicate the subset of x_i values that each regressor was trained on. Thus, the training set represents a non-uniform sample of the underlying chirp waveform. The top panel shows the usual polynomial regression. Note that the regressor fits the given points extremely well, but fails at the endpoint. The ridge regressor misses many of the points in the middle, as indicated by the gray area, but does not overshoot at the ends as much as the plain polynomial regression. This is the basic trade-off for ridge regression. The Jupyter/IPython notebook has the code for this graph, but the main steps are shown in the following,

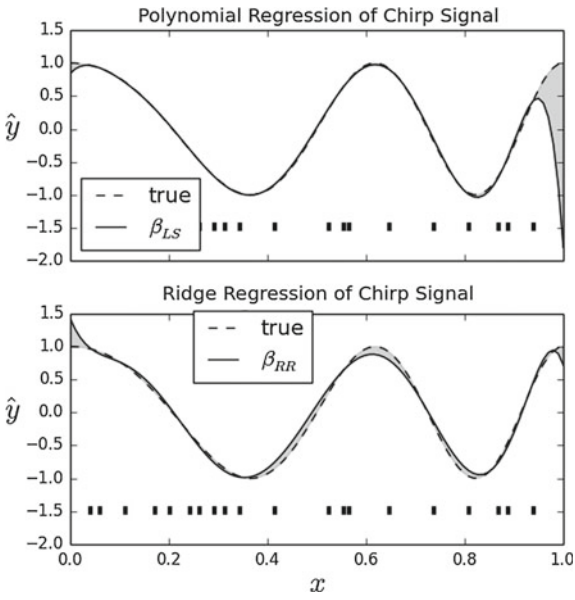

Fig. 4.29 The *top figure* shows polynomial regression and the *lower panel* shows polynomial ridge regression. The ridge regression does not match as well throughout most of the domain, but it does not flare as violently at the ends. This is because the ridge constraint holds the coefficient vector down at the expense of poorer performance along the middle of the domain

```
# create chirp signal
xi = np.linspace(0,1,100)[:,None]
# sample chirp randomly
xin= np.sort(np.random.choice(xi.flatten(),20,replace=False))[:,None]
# create sampled waveform
y = cos(2*pi*(xin+xin**2))
# create full waveform for reference
yi = cos(2*pi*(xi+xi**2))

# create polynomial features
qfit = PolynomialFeatures(degree=8) # quadratic
Xq = qfit.fit_transform(xin)
# reformat input as polynomial
Xiq = qfit.fit_transform(xi)

lr=LinearRegression() # create linear model
lr.fit(Xq,y) # fit linear model

# create ridge regression model and fit
clf = Ridge(alpha=1e-9,fit_intercept=False)
clf.fit(Xq,y)
```

4.6.2 Lasso

Lasso regression follows the same basic pattern as ridge regression, except with the L_1 norm in the objective function.

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \|y - \mathbf{X}\boldsymbol{\beta}\|^2 + \alpha\|\boldsymbol{\beta}\|_1$$

The interface in Scikit-learn is likewise the same. The following is the same problem as before using lasso instead of ridge regression,

```
>>> X = np.matrix([[1,2,3],
...                [3,4,5]])
>>> y = np.matrix([[1,2]]).T
>>> from sklearn.linear_model import Lasso
>>> lr = Lasso(alpha=1.0,fit_intercept=False)
>>> _=lr.fit(X,y)
>>> print lr.coef_
[ 0.          0.          0.32352941]
```

As before, we can use the optimization tools in Scipy to solve this also,

```
>>> from scipy.optimize import fmin
>>> obj = 1/4.*(X*beta-y).norm(2)**2 + beta.norm(1)*l
>>> f = S.lambdify((b0,b1,b2),obj.subs(l,1.0))
>>> g = lambda x:f(x[0],x[1],x[2])
>>> fmin(g,[0.1,0.2,0.3])
Optimization terminated successfully.
         Current function value: 0.360297
         Iterations: 121
         Function evaluations: 221
array([  2.27469304e-06,   4.02831864e-06,   3.23134859e-01])
```

Programming Tip

The `fmin` function from Scipy's optimization module uses an algorithm that does not depend upon derivatives. This is useful because, unlike the L_2 norm, the L_1 norm has sharp corners that make it harder to estimate derivatives.

This result matches the previous one from the Scikit-learn `Lasso` object. Solving it using Scipy is motivating and provides a good sanity check, but specialized algorithms are required in practice. The following code block re-runs the lasso with varying α and plots the coefficients in Fig. 4.30. Notice that as α increases, all but one of the coefficients is driven to zero. Increasing α makes the trade-off between fitting the data in the L_2 sense and wanting to reduce the number of nonzero coefficients (equivalently, the number of features used) in the model. For a given problem, it may be more practical to focus on reducing the number of features in the model (i.e., large α) than the quality of the data fit in the training data. The lasso provides a clean way to navigate this trade-off.

The following code loops over a set of α values and collects the corresponding lasso coefficients to be plotted in Fig. 4.30

```
>>> o=[]
>>> alphas= np.logspace(-3,0,10)
>>> for a in alphas:
...     clf = Lasso(alpha=a,fit_intercept=False)
...     _=clf.fit(X,y)
...     o.append(clf.coef_)
...
```

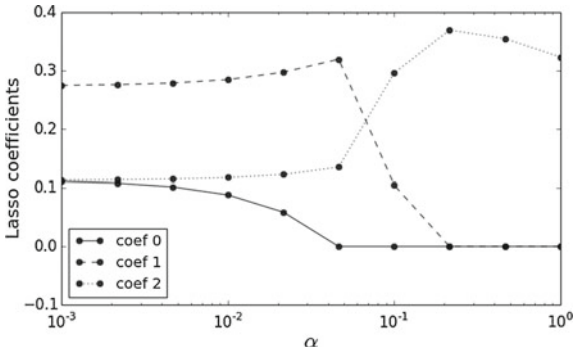

Fig. 4.30 As α increases, more of the model coefficients are driven to zero for lasso regression

4.7 Support Vector Machines

Support Vector Machines (SVM) originated from the statistical learning theory developed by Vapnik-Chervonenkis. As such, it represents a deep application of statistical theory that incorporates the VC dimension concepts we discussed in the first section. Let's start by looking at some pictures. Consider the two-dimensional classification problem shown in Fig. 4.31. Figure 4.31 shows two classes (gray and white circles) that can be separated by any of the lines shown. Specifically, any such separating line can be written as the locus of points ($\mathbf{x}$) in the two-dimensional plane that satisfy the following,

$$\beta_0 + \boldsymbol{\beta}^T \mathbf{x} = 0$$

To classify an arbitrary $\mathbf{x}$ using this line, we just compute the sign of $\beta_0 + \boldsymbol{\beta}^T \mathbf{x}$ and assign one class to the positive sign and the other class to the negative sign. To uniquely specify such a separating line (or, hyperplane in a higher-dimensional space) we need additional criteria.

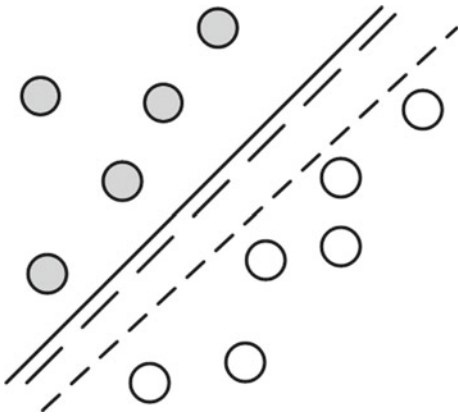

Fig. 4.31 In the two-dimensional plane, the two classes (*gray* and *white circles*) are easily separated by any one of the lines shown

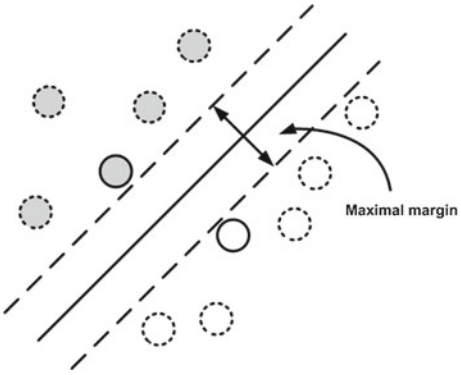

Fig. 4.32 The maximal margin algorithm finds the separating line that maximizes the margin shown. The elements that touch the margins are the support elements. The dotted elements are not relevent to the solution

Figure 4.32 shows the data with two bordering parallel lines that form a margin around the central separating line. The *maximal margin algorithm* finds the widest margin and the unique separating line. As a consequence, the algorithm uncovers the elements in the data that touch the margins. These are the *support* elements. The other elements away from the border are not relevent to the solution. This reduces model variance because the solution is insensitive to the removal of elements other than these supporting elements (usually a small minority).

To see how this works for linearly separable classes, consider a training set consisting of $\{(\mathbf{x}, y)\}$ where $y \in \{-1, 1\}$. For any point $\mathbf{x}_i$, we compute the functional margin as $\hat{\gamma}_i = y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i)$. Thus, $\hat{\gamma}_i > 0$ when $\mathbf{x}_i$ is correctly classified. The geometrical margin is $\gamma = \hat{\gamma}/\|\boldsymbol{\beta}\|$. When $\mathbf{x}_i$ is correctly classified, the geometrical margin is equal to the perpendicular distance from $\mathbf{x}_i$ to the line. Let's look see how the maximal margin algorithm works.

Let M be the width of the margin. The maximal margin algorithm is can be formulated as a quadratic programming problem. We want to simultaneously maximize the margin M while ensuring that all of the data points are correctly classified.

$$\begin{aligned} &\underset{\beta_0, \boldsymbol{\beta}, \|\boldsymbol{\beta}\|=1}{\text{maximize}} \quad M \\ &\text{subject to:} \quad y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i) \geq M, \ i = 1, \ldots, N. \end{aligned}$$

The first line says we want to generate a maximum value for M by adjusting β_0 and $\boldsymbol{\beta}$ while keeping $\|\boldsymbol{\beta}\| = 1$. The functional margins for each ith data element are the constraints to the problem and must be satisfied for every proposed solution. In words, the constraints enforce that the elements have to be correctly classified and outside of the margin around the separating line. With some reformulation, it turns out that $M = 1/\|\boldsymbol{\beta}\|$ and this can be put into the following standard format,

$$\begin{aligned} &\underset{\beta_0, \boldsymbol{\beta}}{\text{minimize}} \quad \|\boldsymbol{\beta}\| \\ &\text{subject to:} \quad y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i) \geq 1, \ i = 1, \ldots, N. \end{aligned}$$

This is a convex optimization problem and can be solved using powerful methods in that area.

The situation becomes more complex when the two classes are not separable and we have to allow some unavoidable mixing between the two classes in the solution. This means that the contraints have to modified as in the following,

$$y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i) \geq M(1 - \xi_i)$$

where the ξ_i are the slack variables and represent the proportional amount that the prediction is on the wrong side of the margin. Thus, elements are misclassified when $\xi_i > 1$. With these additional variables, we have a more general formulation of the convex optimization problem,

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{minimize}} \quad & \|\boldsymbol{\beta}\| \\ \text{subject to:} \quad & y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \sum \xi_i \leq \texttt{constant}, \ i = 1, \ldots, N. \end{aligned}$$

which can be rewritten in the following equivalent form,

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{minimize}} \quad & \frac{1}{2}\|\boldsymbol{\beta}\| + C \sum \xi_i \\ \text{subject to:} \quad & y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i) \geq 1 - \xi_i, \xi_i \geq 0 \ i = 1, \ldots, N. \end{aligned} \tag{4.7.0.1}$$

Because the ξ_i terms are all positive, the objective is to maximize the margin (i.e., minimize $\|\boldsymbol{\beta}\|$) while minimizing the proportional drift of the predictions to the wrong side of the margin (i.e., $C \sum \xi_i$). Thus, large values of C shunt algorithmic focus towards the correctly classified points near the decision boundary and small values focus on further data. The value C is a hyperparameter for the SVM.

The good news is that all of these complicated pieces are handled neatly inside of Scikit-learn. The following sets up the linear *kernel* for the SVM (more on kernels soon),

```
>>> from sklearn.datasets import make_blobs
>>> from sklearn.svm import SVC
>>> sv = SVC(kernel='linear')
```

We can create some synthetic data using `make_blobs` and then fit it to the SVM,

```
>>> X,y=make_blobs(n_samples=200, centers=2, n_features=2,
...                random_state=0,cluster_std=.5)
>>> sv.fit(X,y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='linear', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

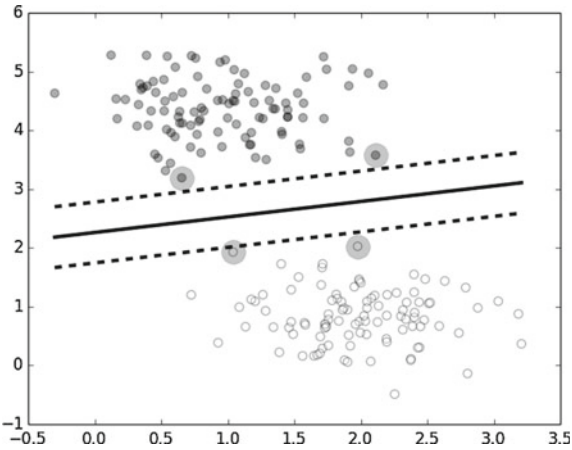

Fig. 4.33 The two class shown (*white* and *gray circles*) are linearly separable. The maximal margin solution is shown by the *dark black line* in the *middle*. The *dotted lines* show the extent of the margin. The *large circles* indicate the support vectors for the maximal margin solution

After fitting, the SVM now has the estimated support vectors and the coefficients of the $\boldsymbol{\beta}$ in the `sv.support_vectors_` and `sv.coef_` attributes, respectively. Figure 4.33 shows the two sample classes (white and gray circles) and the line separating them that was found by the maximal margin algorithm. The two parallel dotted lines show the margin. The large circles enclose the support vectors, which are the data elements that are relevent to the solution. Notice that only these elements can touch the edges of the margins.

Figure 4.34 shows what happens when the value of C changes. Increasing this value emphasizes the ξ part of the objective function in Eq. 4.7.0.1. As shown in the top left panel, a small value for C means that the algorithm is willing to accept many support vectors at the expense of maximizing the margin. That is, the proportional amount that predictions are on the wrong side of the margin is more acceptable with smaller C. As the value of C increases, there are fewer support vectors because the optimization process prefers to eliminate support vectors that are far away from the margins and accept fewer of these that encroach into the margin. Note that as the value of C progresses through this figure, the separating line tilts slightly.

4.7.1 Kernel Tricks

Support Vector Machines provide a powerful method to deal with linear separations, but they can also apply to non-linear boundaries by exploiting the so-called *kernel trick*. The convex optimization formulation of the SVM includes a *dual* formulation that leads to a solution that requires only the inner-products of the features. The kernel trick is to substitute inner-products by nonlinear kernel functions. This can be thought of as mapping the original features onto a possibly infinite dimensional space of new

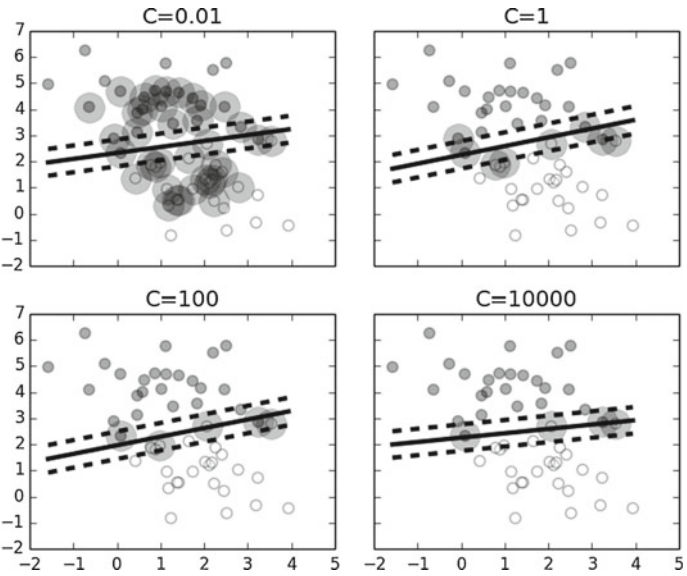

Fig. 4.34 The maximal margin algorithm finds the separating line that maximizes the margin shown. The elements that touch the margins are the support elements. The dotted elements are not relevent to the solution

features. That is, if the data are not linearly separable in two-dimensional space (for example) maybe they are separable in three-dimensional space (or higher)?

To make this concrete, suppose the original input space is $\mathbb{R}^n$ and we want to use a non-linear mapping $\psi : \mathbf{x} \mapsto \mathcal{F}$ where $\mathcal{F}$ is an inner-product space of higher dimension. The kernel trick is to calculate the inner-product in $\mathcal{F}$ using a kernel function, $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}_j) \rangle$. The long way to compute this is to first compute $\psi(\mathbf{x})$ and then do the inner-product. The kernel-trick way to do it is to use the kernel function and avoid computing ψ. In other words, the kernel function returns what the inner-product in $\mathcal{F}$ would have returned if ψ had been applied. For example, to achieve an nth polynomial mapping of the input space, we can use $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + \theta)^n$. For example, suppose the input space is $\mathbb{R}^2$ and $\mathcal{F} = \mathbb{R}^4$ and we have the following mapping,

$$\psi(\mathbf{x}) : (x_0, x_1) \mapsto (x_0^2, x_1^2, x_0 x_1, x_1 x_0)$$

The inner product in $\mathcal{F}$ is then,

$$\langle \psi(\mathbf{x}), \psi(\mathbf{y}) \rangle = \langle \mathbf{x}, \mathbf{y} \rangle^2$$

In other words, the kernel is the square of the inner product in input space. The advantage of using the kernel instead of simply enlarging the feature space is computational because you only need to compute the kernel on all distinct pairs of the

input space. The following example should help make this concrete. First we create some Sympy variables,

```
>>> import sympy as S
>>> x0,x1=S.symbols('x:2',real=True)
>>> y0,y1=S.symbols('y:2',real=True)
```

Next, we create the ψ function that maps into $\mathbb{R}^4$ and the corresponding kernel function,

```
>>> psi = lambda x,y: (x**2,y**2,x*y,x*y)
>>> kern = lambda x,y: S.Matrix(x).dot(y)**2
```

Notice that the inner product in $\mathbb{R}^4$ is equal to the kernel function, which only uses wthe $\mathbb{R}^2$ variables.

```
>>> print S.Matrix(psi(x0,x1)).dot(psi(y0,y1))
x0**2*y0**2 + 2*x0*x1*y0*y1 + x1**2*y1**2
>>> print S.expand(kern((x0,x1),(y0,y1))) # same as above
x0**2*y0**2 + 2*x0*x1*y0*y1 + x1**2*y1**2
```

Polynomial Regression Using Kernels. Recall our favorite linear regression problem from the regularization chapter,

$$\min_{\boldsymbol{\beta}} \|y - \mathbf{X}\boldsymbol{\beta}\|^2$$

where $\mathbf{X}$ is a $n \times m$ matrix with $m > n$. As we discussed, there are multiple solutions to this problem. The least-squares solution is the following:

$$\boldsymbol{\beta}_{LS} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{y}$$

Given a new feature vector $\mathbf{x}$, the corresponding estimator for $\mathbf{y}$ is the following,

$$\hat{\mathbf{y}} = \mathbf{x}^T\boldsymbol{\beta}_{LS} = \mathbf{x}^T\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{y}$$

Using the kernel trick, the solution can be written more generally as the following,

$$\hat{\mathbf{y}} = \mathbf{k}(\mathbf{x})^T\mathbf{K}^{-1}\mathbf{y}$$

where the $n \times n$ kernel matrix $\mathbf{K}$ replaces $\mathbf{X}\mathbf{X}^T$ and where $\mathbf{k}(\mathbf{x})$ is a n-vector of components $\mathbf{k}(\mathbf{x}) = [\kappa(\mathbf{x}_i, \mathbf{x})]$ and where $\mathbf{K}_{i,j} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ for the kernel function κ. With this more general setup, we can substitute $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + \theta)^n$ for nth-order polynomial regression [6]. Note that ridge regression can also be incorporated by inverting $(\mathbf{K} + \alpha\mathbf{I})$, which can help stabilize poorly conditioned $\mathbf{K}$ matrices with a tunable α hyper-parameter [6].

For some kernels, the enlarged $\mathcal{F}$ space is infinite-dimensional. Mercer's conditions provide technical restrictions on the kernel functions. Powerful, well-studied kernels have been implemented in Scikit-learn. The advantage of kernel functions may evaporate for when $n \to m$ in which case using the ψ functions instead can be more practicable.

4.8 Dimensionality Reduction

The features from a particular dataset that will ultimately prove important for machine learning can be difficult to know ahead of time. This is especially true for problems that do not have a strong physical underpinning. The row-dimension of the input matrix (X) for fitting data in Scikit-learn is the number of samples and the column dimension is the number of features. There may be a large number of column dimensions in this matrix, and the purpose of dimensionality reduction is to somehow reduce these to only those columns that are important for the machine learning task.

Fortunately, Scikit-learn provides some powerful tools to help uncover the most relevant features. Principal Component Analysis (PCA) consists of taking the input X matrix and (1) subtracting the mean, (2) computing the covariance matrix, and (3) computing the eigenvalue decomposition of the covariance matrix. For example, if X has more columns than is practicable for a particular learning method, then PCA can reduce the number of columns to a more manageable number. PCA is widely used in statistics and other areas beyond machine learning, so it is worth examining what it does in some detail. First, we need the decomposition module from Scikit-learn.

```
>>> from sklearn import decomposition
>>> import numpy as np
>>> pca = decomposition.PCA()
```

Let's create some very simple data and apply PCA.

```
>>> x = np.linspace(-1,1,30)
>>> X = np.c_[x,x+1,x+2] # stack as columns
>>> pca.fit(X)
PCA(copy=True, n_components=None, whiten=False)
>>> print pca.explained_variance_ratio_
[  1.00000000e+00   4.44023384e-32   6.35796894e-33]
```

Programming Tip

The `np.c_` is a shorcut method for creating stacked column-wise arrays.

In this example, the columns are just constant offsets of the first column. The *explained variance ratio* is the percentage of the variance attributable to the transformed columns of `X`. You can think of this as the information that is relatively concentrated in each column of the transformed matrix `X`. Figure 4.35 shows the graph of this dominant transformed column in the bottom panel.

To make this more interesting, let's change the slope of each of the columns as in the following,

```
>>> X = np.c_[x,2*x+1,3*x+2,x] # change slopes of columns
>>> pca.fit(X)
PCA(copy=True, n_components=None, whiten=False)
>>> print pca.explained_variance_ratio_
[  1.00000000e+00   8.94906713e-33   1.06089989e-33   9.50578639e-35]
```

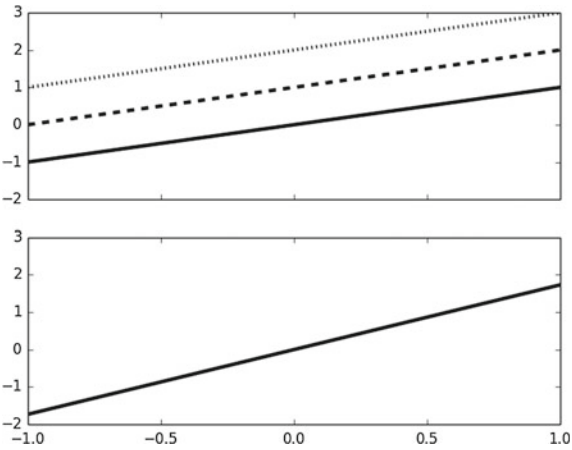

Fig. 4.35 The *top panel* shows the columns of the feature matrix and the *bottom panel* shows the dominant component that PCA has extracted

However, changing the slope did not impact the explained variance ratio. Again, there is still only one dominant column. This means that PCA is invariant to both constant offsets and scale changes. This works for functions as well as simple lines,

```
>>> x = np.linspace(-1,1,30)
>>> X = np.c_[np.sin(2*np.pi*x),
...           2*np.sin(2*np.pi*x)+1,
...           3*np.sin(2*np.pi*x)+2]
>>> pca.fit(X)
PCA(copy=True, n_components=None, whiten=False)
>>> print pca.explained_variance_ratio_
[  1.00000000e+00   3.78264707e-32   4.64963853e-34]
```

Once again, there is only one dominant column, which is shown in the bottom panel of Fig. 4.36. The top panel shows the individual columns of the feature matrix. To sum up, PCA is able to identify and eliminate features that are merely linear

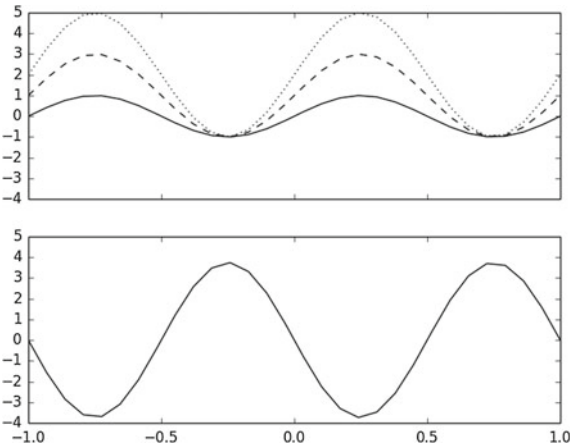

Fig. 4.36 The *top panel* shows the columns of the feature matrix and the *bottom panel* shows the dominant component that PCA has computed

transformations of existing features. This also works when there is additive noise in the features, although more samples are needed to separate the uncorrelated noise from between features.

To see how PCA can simplify machine learning tasks, consider Fig. 4.37 wherein the two classes are separated along the diagonal. After PCA, the transformed data lie along a single axis where the two classes can be split using a one-dimensional interval, which greatly simplifies the classification task. The class identities are preserved under PCA because the principal component is along the same direction that the classes are separated. On the other hand, if the classes are separated along the direction *orthogonal* to the principal component, then the two classes become mixed under PCA and the classification task becomes much harder. Note that in both cases, the `explained_variance_ratio_` is the same because the explained variance ratio does not account for class membership.

PCA works by decomposing the covariance matrix of the data using the Singular Value Decomposition (SVD). This decomposition exists for all matrices and returns the following factorization for an arbitrary matrix $\mathbf{A}$ (Fig. 4.38),

$$\mathbf{A} = \mathbf{USV}^T$$

Because of the symmetry of the covariance matrix, $\mathbf{U} = \mathbf{V}$. The elements of the diagonal matrix $\mathbf{S}$ are the singular values of $\mathbf{A}$ whose squares are the eigenvalues of $\mathbf{A}^T\mathbf{A}$. The eigenvector matrix $\mathbf{U}$ is orthogonal: $\mathbf{U}^T\mathbf{U} = \mathbf{I}$. The singular values are in decreasing order so that the first column of $\mathbf{U}$ is the axis corresponding to the

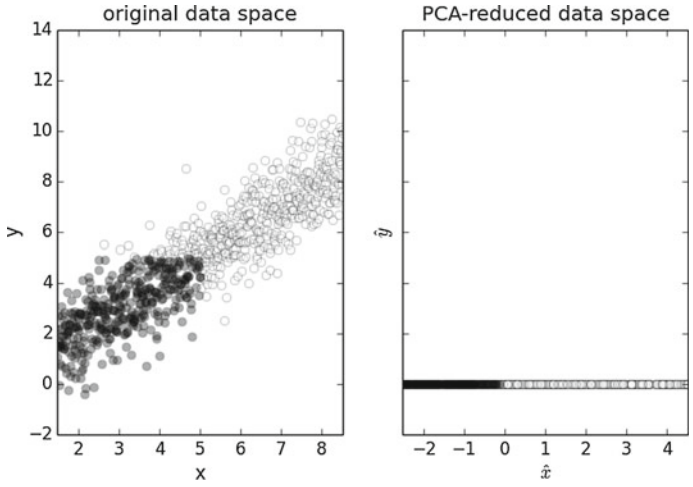

Fig. 4.37 The *left panel* shows the original two-dimensional data space of two easily distinguishable classes and the *right panel* shows the reduced the data space transformed using PCA. Because the two classes are separated along the principal component discovered by PCA, the classes are preserved under the transformation

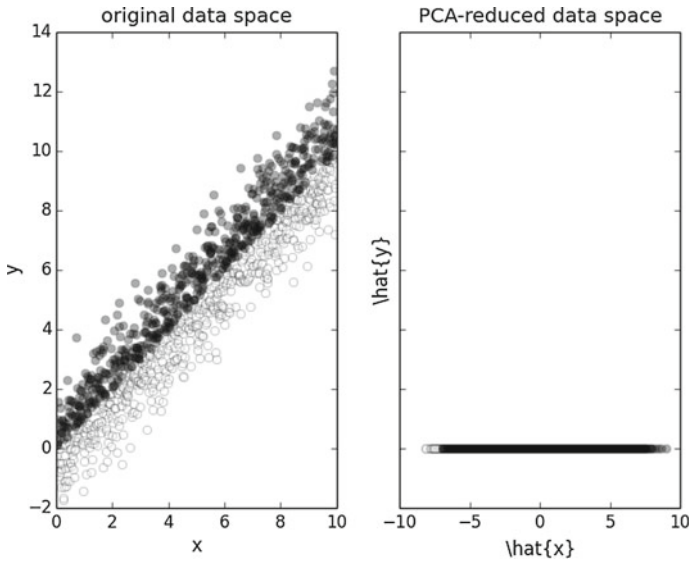

Fig. 4.38 As compared with Fig. 4.37, the two classes differ along the coordinate direction that is orthogonal to the principal component. As a result, the two classes are no longer distinguishable after transformation

largest singular value. This is the first dominant column that PCA identifies. The entries of the covariance matrix are of the form $\mathbb{E}(x_i x_j)$ where x_i and x_j are different features.[5] This means that the covariance matrix is filled with entries that attempt to uncover mutually correlated relationships between all pairs of columns of the feature matrix. Once these have been tabulated in the covariance matrix, the SVD finds optimal orthogonal transformations to align the components along the directions most strongly associated with these correlated relationships. Simultaneously, because orthogonal matrices have columns of unit-length, the SVD collects the absolute squared lengths of these components into the **S** matrix. In our example above in Fig. 4.37, the two feature vectors were obviously correlated along the diagonal, meaning that PCA selected that diagonal direction as the principal component.

We have seen that PCA is a powerful dimension reduction method that is invariant to linear transformations of the original feature space. However, this method performs poorly with transformations that are nonlinear. In that case, there are a wide range of extensions to PCA, such as Kernel PCA, that are available in Scikit-learn, which allow for embedding parameterized non-linearities into the PCA at the risk of overfitting.

[5] Note that these entries are constructed from the data using an estimator of the covariance matrix because we do not have the full probability densities at hand.

4.8.1 Independent Component Analysis

Independent Component Analysis (ICA) via the `FastICA` algorithm is also available in Scikit-learn. This method is fundamentally different from PCA in that it is the small differences between components that are emphasized, not the large principal components. This method is adopted from signal processing. Consider a matrix of signals ($\mathbf{X}$) where the rows are the samples and the columns are the different signals. For example, these could be EKG signals from multiple leads on a single patient. The analysis starts with the following model,

$$\mathbf{X} = \mathbf{S}\mathbf{A}^T \tag{4.8.1.1}$$

In other words, the observed signal matrix is an unknown mixture ($\mathbf{A}$) of some set of conformable, independent random sources $\mathbf{S}$,

$$\mathbf{S} = [\mathbf{s}_1(t), \mathbf{s}_2(t), \ldots, \mathbf{s}_n(t)]$$

The distribution on the random sources is otherwise unknown, except there can be at most one Gaussian source, otherwise, the mixing matrix $\mathbf{A}$ cannot be identified because of technical reasons. The problem in ICA is to find $\mathbf{A}$ in Eq. 4.8.1.1 and thereby un-mix the $s_i(t)$ signals, but this cannot be solved without a strategy to reduce the inherent arbitrariness in this formulation.

To make this concrete, let us simulate the situation with the following code,

```
>>> from numpy import matrix, c_, sin, cos, pi
>>> t = np.linspace(0,1,250)
>>> s1 = sin(2*pi*t*6)
>>> s2 =np.maximum(cos(2*pi*t*3),0.3)
>>> s2 = s2 - s2.mean()
>>> s3 = np.random.randn(len(t))*.1

>>> # normalize columns
>>> s1=s1/np.linalg.norm(s1)
>>> s2=s2/np.linalg.norm(s2)
>>> s3=s3/np.linalg.norm(s3)
>>> S =c_[s1,s2,s3] # stack as columns

>>> # mixing matrix
>>> A = matrix([[  1,  1,1],
...             [0.5, -1,3],
...             [0.1, -2,8]])
>>> X= S*A.T # do mixing
```

The individual signals ($s_i(t)$) and their mixtures ($X_i(t)$) are shown in Fig. 4.39. To recover the individual signals using ICA, we use the `FastICA` object and fit the parameters on the `X` matrix,

```
>>> from sklearn.decomposition import FastICA
>>> ica = FastICA()
>>> # estimate unknown S matrix
>>> S_=ica.fit_transform(X)
```

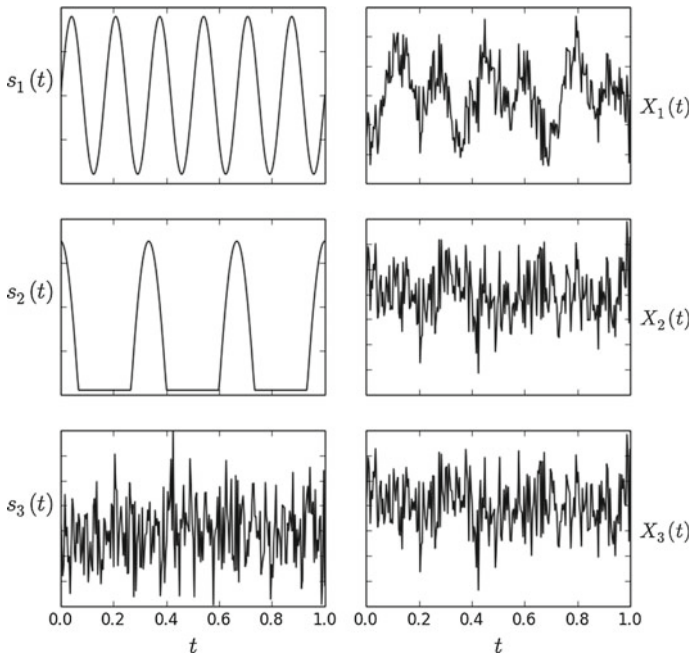

Fig. 4.39 The *left column* shows the original signals and the *right column* shows the mixed signals. The object of ICA is to recover the *left column* from the *right*

The results of this estimation are shown in Fig. 4.40, showing that ICA is able to recover the original signals from the observed mixture. Note that ICA is unable to distinguish the signs of the recovered signals or preserve the order of the input signals.

To develop some intuition as to how ICA accomplishes this feat, consider the following two-dimensional situation with two uniformly distributed independent variables, $u_x, u_y \sim \mathcal{U}[0, 1]$. Suppose we apply the following orthogonal rotation matrix to these variables,

$$\begin{bmatrix} u'_x \\ u'_y \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}$$

The so-rotated variables u'_x, u'_y are no longer independent, as shown in Fig. 4.41. Thus, one way to think about ICA is as a search through orthogonal matrices so that the independence is restored. This is where the prohibition against Gaussian distributions arises. The two dimensional Gaussian distribution of independent variables is proportional the following,

$$f(\mathbf{x}) \propto \exp(-\frac{1}{2}\mathbf{x}^T\mathbf{x})$$

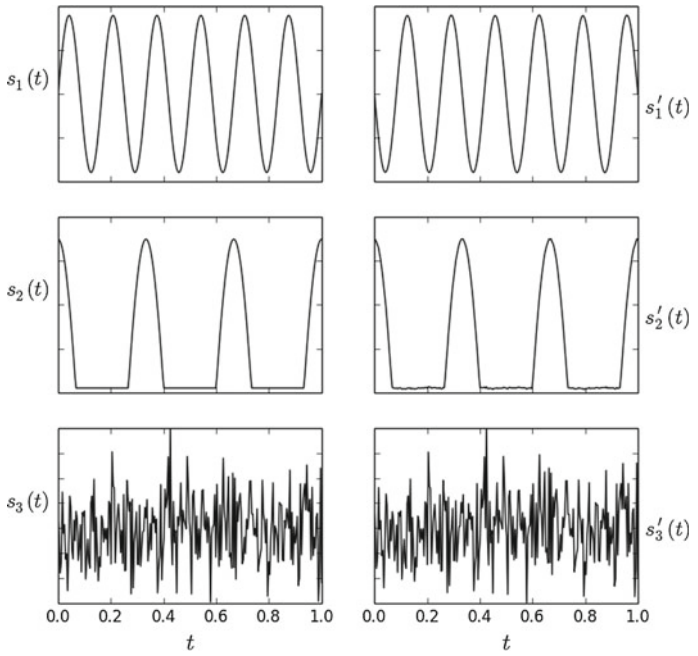

Fig. 4.40 The *left column* shows the original signals and the *right column* shows the signals that ICA was able to recover. They match exactly, outside of a possible sign change

Now, if we similarly rotated the $\mathbf{x}$ vector as,

$$\mathbf{y} = \mathbf{Q}\mathbf{x}$$

the resulting density for $\mathbf{y}$ is obtained by plugging in the following,

$$\mathbf{x} = \mathbf{Q}^T\mathbf{y}$$

because the inverse of an orthogonal matrix is its transpose, we obtain

$$f(\mathbf{y}) \propto \exp(-\frac{1}{2}\mathbf{y}^T\mathbf{Q}\mathbf{Q}^T\mathbf{y}) = \exp(-\frac{1}{2}\mathbf{y}^T\mathbf{y})$$

In other words, the transformation is lost on the $\mathbf{y}$ variable. This means that ICA cannot search over orthogonal transformations if it is blind to them, which explains the restriction of Gaussian random variables. Thus, ICA is a method that seeks to maximize the non-Gaussian-ness of the transformed random variables. There are many methods to doing this, some of which involve cumulants and others that use the *negentropy*,

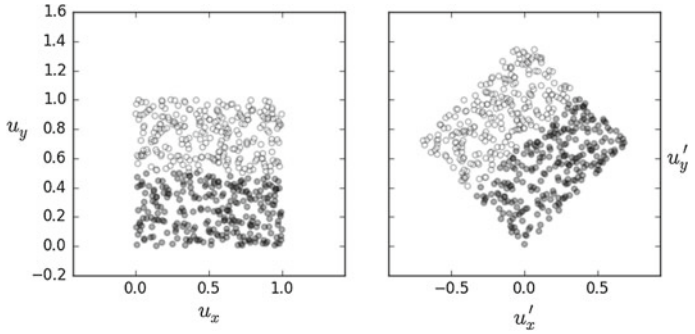

Fig. 4.41 The *left panel* shows two classes labeled on the u_x, u_y uniformly independent random variables. The *right panel* shows these random variables after a rotation, which removes their mutual independence and makes it hard to separate the two classes along the coordinate directions

$$\mathcal{J}(Y) = \mathcal{H}(Z) - \mathcal{H}(Y)$$

where $\mathcal{H}(Z)$ is the information entropy of the Gaussian random variable Z that has the same variance as Y. Further details would take us beyond our scope, but that is the outline of how the FastICA algorithm works.

The implementation of this method in Scikit-learn includes two different ways of extracting more than one independent source component. The *deflation* method iteratively extracts one component at a time using a incremental normalization step. The *parallel* method also uses the single-component method but carries out normalization of all the components simultaneously, instead of for just the newly computed component. Because ICA extracts independent components, a whitening step is used beforehand to balance the correlated components from the data matrix. Whereas PCA returns uncorrelated components along dimensions optimal for Gaussian random variables, ICA returns components that are as far from the Gaussian density as possible.

The left panel on Fig. 4.41 shows the orignal uniform random sources. The white and black colors distinguish between two classes. The right panel shows the mixture of these sources, which is what we observe as input features. The top row of Fig. 4.42 shows the PCA (left) and ICA (right) transformed data spaces. Notice that ICA is able to un-mix the two random sources whereas PCA transforms along the dominant diagonal. Because ICA is able to preserve the class membership, the data space can be reduced to two non-overlapping sections, as shown. However, PCA cannot achieve a similiar separation because the classes are mixed along the dominant diagonal that PCA favors as the main component in the decomposition.

For a good principal component analysis treatment, see [7–9], and [10]. Independent Component Analysis is discussed in more detail in [11].

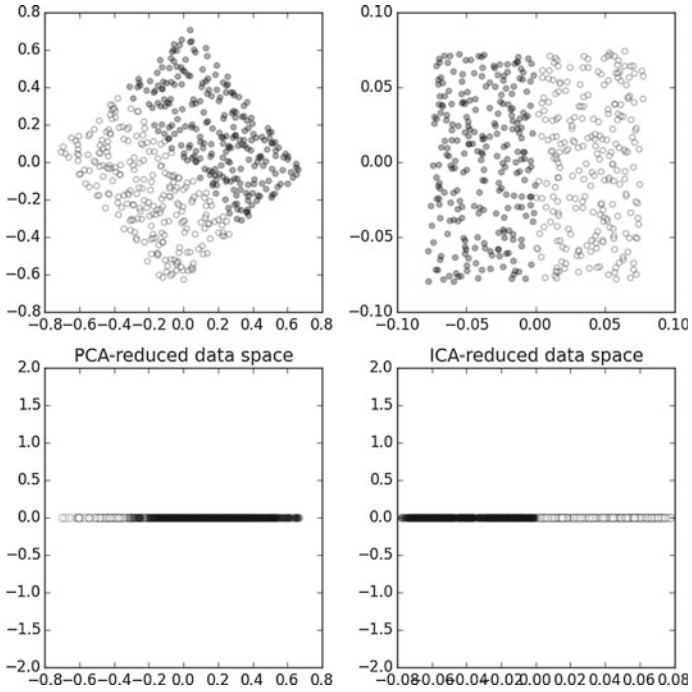

Fig. 4.42 The panel on the *top left* shows two classes in a plane after a rotation. The *bottom left* panel shows the result of dimensionality reduction using PCA, which causes mixing between the two classes. The *top right* panel shows the ICA transformed output and the *lower right panel* shows that, because ICA was able to un-rotate the data, the lower dimensional data maintains the separation between the classes

4.9 Clustering

Clustering is the simplest member of a family of machine learning methods that do not require supervision to learn from data. Unsupervised methods have training sets that do not have a target variable. These unsupervised learning methods rely upon a meaningful metric to group data into clusters. This makes it an excellent exploratory data analysis method because there are very few assumptions built into the method itself. In this section, we focus on the popular K-means clustering method that is available in Scikit-learn.

Let's manufacture some data to get going with `make_blobs` from Scikit-learn. Figure 4.43 shows some example clusters in two dimensions. Clustering methods work by minimizing the following objective function,

$$J = \sum_k \sum_i \|\mathbf{x}_i - \mu_k\|^2$$

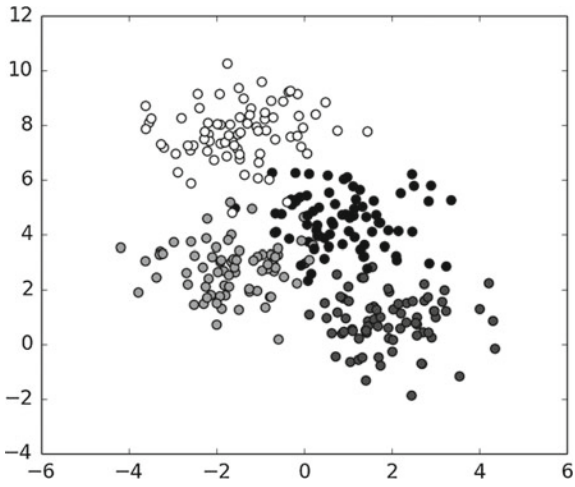

Fig. 4.43 The four clusters are pretty easy to see in this example and we want clustering methods to determine the extent and number of such clusters automatically

The *distortion* for the kth cluster is the summand,

$$\|x_i - \mu_k\|^2$$

Thus, clustering algorithms work to minimize this by adjusting the centers of the individual clusters, μ_k. Intuitively, each μ_k is the *center of mass* of the points in the cloud. The Euclidean distance is the typical metric used for this,

$$\|\mathbf{x}\|^2 = \sum x_i^2$$

There are many clever algorithms that can solve this problem for the best μ_k cluster-centers. The K-means algorithm starts with a user-specified number of K clusters to optimize over. This is implemented in Scikit-learn with the `KMeans` object that follows the usual fitting conventions in Scikit-learn,

```
>>> from sklearn.cluster import KMeans
>>> kmeans = KMeans(n_clusters=4)
>>> kmeans.fit(X)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=4, n_init=10,
    n_jobs=1, precompute_distances=True, random_state=None, tol=0.0001,
    verbose=0)
```

where we have chosen $K = 4$. How do we choose the value of K? This is the eternal question of generalization versus approximation—too many clusters provide great approximation but bad generalization. One way to approach this problem is to compute the mean distortion for increasingly larger values of K until it no longer makes sense. To do this, we want to take every data point and compare it to the centers of all the clusters. Then, take the smallest value of this across all clusters and average those. This gives us an idea of the overall mean performance for the K clusters. The following code computes this explicitly.

Programming Tip

The `cdist` function from Scipy computes all the pairwise differences between the two input collections according to the specified metric.

```
>>> from scipy.spatial.distance import cdist
>>> m_distortions=[]
>>> for k in range(1,7):
...     kmeans = KMeans(n_clusters=k)
...     _=kmeans.fit(X)
...     tmp=cdist(X,kmeans.cluster_centers_,'euclidean')
...     m_distortions.append(sum(np.min(tmp,axis=1))/X.shape[0])
...
```

Note that code above uses the `cluster_centers_`, which are estimated from K-means algorithm. The resulting Fig. 4.44 shows the point of diminishing returns for added additional clusters.

Another figure-of-merit is the silhouette coefficient, which measures how compact and separated the individual clusters are. To compute the silhouette coefficient, we need to compute the mean intra-cluster distance for each sample (a_i) and the mean distance to the next nearest cluster (b_i). Then, the silhouette coefficient for the ith sample is

$$\mathrm{sc}_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

The mean silhouette coefficient is just the mean of all these values over all the samples. The best value is one and the worst is negative one, with values near zero indicating overlapping clusters and negative values showing that samples have been incorrectly assigned to the wrong cluster. This figure-of-merit is implemented in Scikit-learn as in the following,

```
>>> from sklearn.metrics import silhouette_score
```

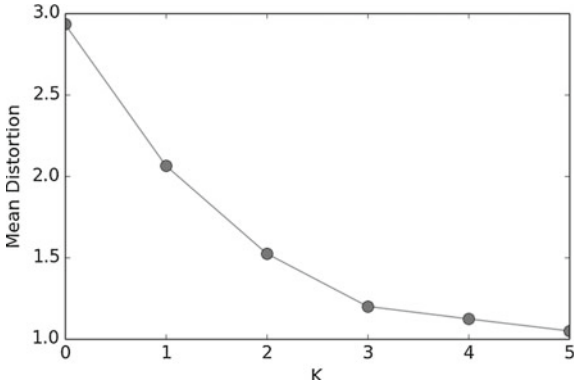

Fig. 4.44 The mean distortion shows that there is a diminishing value in using more clusters

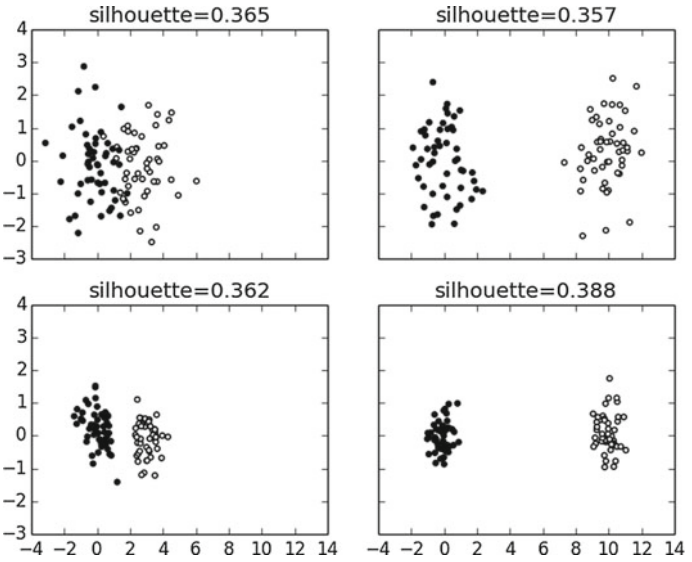

Fig. 4.45 The shows how the silhouette coefficient varies as the clusters move closer and become more compact

Figure 4.46 shows how the silhouette coefficient varies as the clusters become more dispersed and/or closer together.

K-means is easy to understand and to implement, but can be sensitive to the initial choice of cluster-centers. The default initialization method in Scikit-learn uses a very effective and clever randomization to come up with the initial cluster-centers. Nonetheless, to see why initialization can cause instability with K-means, consider the following Fig. 4.46. In Fig. 4.46, there are two large clusters on the left and a very sparse cluster on the far right. The large circles at the centers are the cluster-centers that K-means found. Given $K = 2$, how should the cluster-centers be chosen? Intuitively, the first two clusters should have their own cluster-center somewhere between them and the sparse cluster on the right should have its own cluster-center.[6] Why isn't this happening (Fig. 4.45)?

The problem is that the objective function for K-means is trading the distance of the far-off sparse cluster with its small size. If we keep increasing the number of samples in the sparse cluster on the right, then K-means will move the cluster centers out to meet them, as shown in Fig. 4.46. That is, if one of the initial cluster-centers was right in the middle of the sparse cluster, the algorithm would have immediately captured it and then moved the next cluster-center to the middle of the other two clusters (bottom panel of Fig. 4.46). Without some thoughtful initialization, this may not happen and the sparse cluster would have been merged

[6]Note that we are using the `init=random` keyword argument for this example in order to illustrate this.

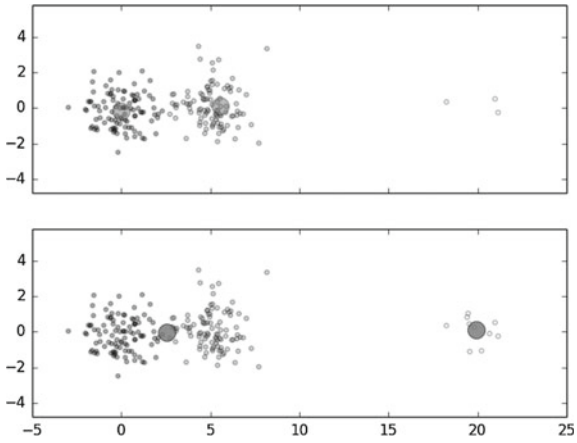

Fig. 4.46 The *large circles* indicate the cluster-centers found by the K-means algorithm

into the middle cluster (top panel of Fig. 4.46). Furthermore, such problems are hard to visualize with high-dimensional clusters. Nonetheless, K-means is generally very fast, easy-to-interpret, and easy to understand. It is straightforward to parallelize using the `n_jobs` keyword argument so that many initial cluster-centers can be easily evaluated. Many extensions of K-means use different metrics beyond Euclidean and incorporate adaptive weighting of features. This enables the clusters to have ellipsoidal instead of spherical shapes.

4.10 Ensemble Methods

With the exception of the random forest, we have so far considered machine learning models as stand-alone entities. Combinations of models that jointly produce a classification are known as *ensembles*. There are two main methodologies that create ensembles: *bagging* and *boosting*.

4.10.1 Bagging

Bagging refers to bootstrap aggregating, where bootstrap here is the same as we discussed in Sect. 3.10. Basically, we resample the data with replacement and then train a classifier on the newly sampled data. Then, we combine the outputs of each of the individual classifiers using a majority-voting scheme (for discrete outputs) or a weighted average (for continuous outputs). This combination is particularly effective for models that are easily influenced by a single data element. The resampling process means that these elements cannot appear in every bootstrapped training set so that some of the models will not suffer these effects. This makes the so-computed combination of outputs less volatile. Thus, bagging helps reduce the collective variance of individual high-variance models.

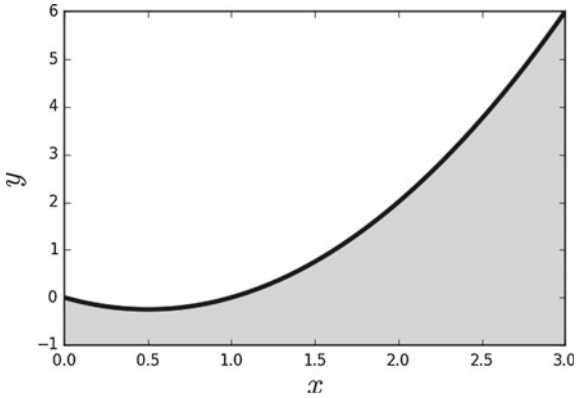

Fig. 4.47 Two regions in the plane are separated by a nonlinear boundary. The training data is sampled from this plane. The objective is to correctly classify the so-sampled data

To get a sense of bagging, let's suppose we have a two-dimensional plane that is partitioned into two regions with the following boundary: $y = -x + x^2$. Pairs of (x_i, y_i) points above this boundary are labeled one and points below are labeled zero. Figure 4.47 shows the two regions with the nonlinear separating boundary as the black curved line.

The problem is to take samples from each of these regions and classify them correctly using a perceptron. A perceptron is the simplest possible linear classifier that finds a line in the plane to separate two purported categories. Because the separating boundary is nonlinear, there is no way that the perceptron can completely solve this problem. The following code sets up the perceptron available in Scikit-learn.

```
>>> from sklearn.linear_model import Perceptron
>>> p=Perceptron()
>>> p
Perceptron(alpha=0.0001,class_weight=None,eta0=1.0,fit_intercept=True,
n_iter=5,n_jobs=1,penalty=None,random_state=0,shuffle=False,
verbose=0,warm_start=False)
```

The training data and the resulting perceptron separating boundary are shown in Fig. 4.48. The circles and crosses are the sampled training data and the gray separating line is the perceptron's separating boundary between the two categories. The black squares are those elements in the training data that the perceptron misclassified. Because the perceptron can only produce linear separating boundaries, and the boundary in this case is non-linear, the perceptron makes mistakes near where the boundary curves. The next step is to see how bagging can improve upon this by using multiple perceptrons.

The following code sets up the bagging classifier in Scikit-learn. Here we select only three perceptrons. Figure 4.49 shows each of the three individual classifiers and the final bagged classifer in the panel on the bottom right. As before, the black circles indicate misclassifications in the training data. Joint classifications are determined by majority voting.

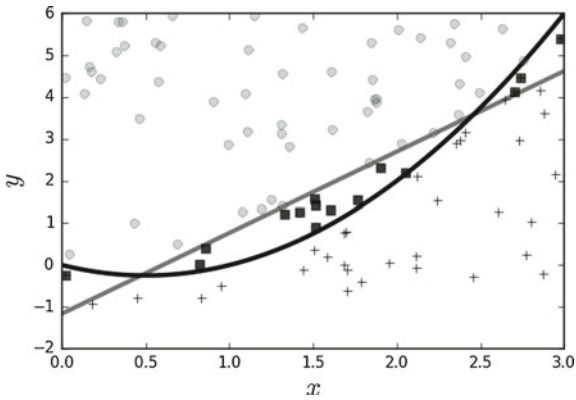

Fig. 4.48 The perceptron finds the best linear boundary between the two classes

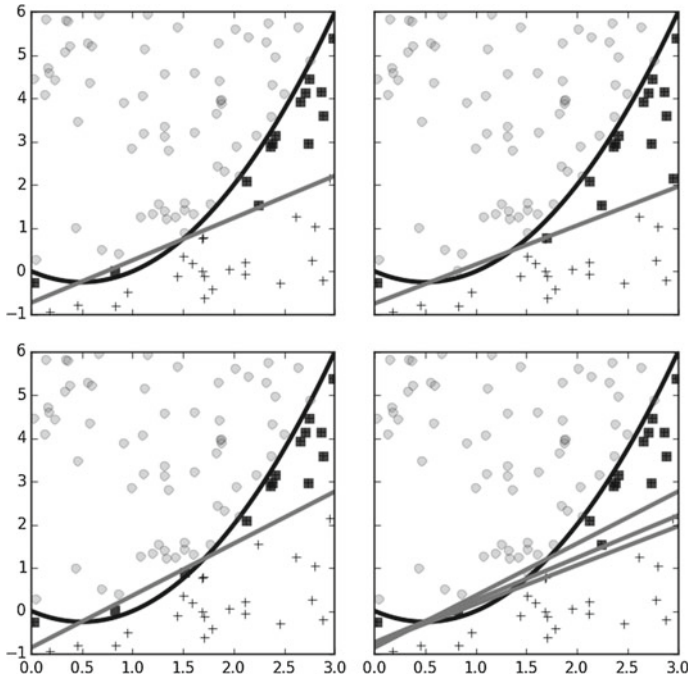

Fig. 4.49 Each panel with the *single gray line* is one of the perceptrons used for the ensemble bagging classifier on the *lower right*

```
>>> from sklearn.ensemble import BaggingClassifier
>>> bp = BaggingClassifier(Perceptron(),max_samples=0.50,n_estimators=3)
>>> bp
BaggingClassifier(base_estimator=Perceptron(alpha=0.0001,class_weight=None,eta0=
1.0,fit_intercept=True,
n_iter=5,n_jobs=1,penalty=None,random_state=0,shuffle=False,
verbose=0,warm_start=False),
bootstrap=True,bootstrap_features=False,max_features=1.0,
max_samples=0.5,n_estimators=3,n_jobs=1,oob_score=False,
random_state=None,verbose=0)
```

The `BaggingClassifier` can estimate its own out-of-sample error if passed the `oob_score=True` flag upon construction. This keeps track of which samples were used for training and which were not, and then estimates the out-of-sample error using those samples that were unused in training. The `max_samples` keyword argument specifies the number of items from the training set to use for the base classifier. The smaller the `max_samples` used in the bagging classifier, the better the out-of-sample error estimate, but at the cost of worse in-sample performance. Of course, this depends on the overall number of samples and the degrees-of-freedom in each individual classifier. The VC-dimension surfaces again!

4.10.2 Boosting

As we discussed, bagging is particularly effective for individual high-variance classifiers because the final majority-vote tends to smooth out the individual classifiers and produce a more stable collaborative solution. On the other hand, boosting is particularly effective for high-bias classifiers that are slow to adjust to new data. On the one hand, boosting is similiar to bagging in that it uses a majority-voting (or averaging for numeric prediction) process at the end; and it also combines individual classifiers of the same type. On the other hand, boosting is serially iterative, whereas the individual classifiers in bagging can be trained in parallel. Boosting uses the misclassifications of prior iterations to influence the training of the next iterative classifier by weighting those misclassifications more heavily in subsequent steps. This means that, at every step, boosting focuses more and more on specific misclassifications up to that point, letting the prior classifications be carried by earlier iterations.

The primary implementation for boosting in Scikit-learn is the Adaptive Boosting (*AdaBoost*) algorithm, which does classification (`AdaBoostClassifier`) and regression (`AdaBoostRegressor`). The first step in the basic AdaBoost algorithm is to initialize the weights over each of the training set indicies, $D_0(i) = 1/n$ where there are n elements in the training set. Note that this creates a discrete uniform distribution over the *indicies*, not over the training data $\{(x_i, y_i)\}$ itself. In other words, if there are repeated elements in the training data, then each gets its own

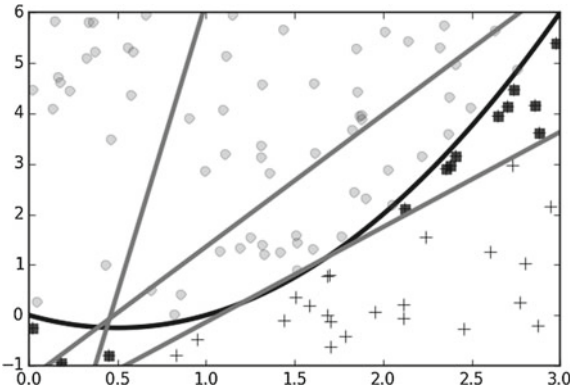

Fig. 4.50 The individual perceptron classifiers embedded in the AdaBoost classifier are shown along with the mis-classified points (in *black*). Compare this to the *lower right panel* of Fig. 4.49

weight. The next step is to train the base classifer h_k and record the classification error at the kth iteration, ϵ_k. Two factors can next be calculated using ϵ_k,

$$\alpha_k = \frac{1}{2} \log \frac{1 - \epsilon_k}{\epsilon_k}$$

and the normalization factor,

$$Z_k = 2\sqrt{\epsilon_k(1 - \epsilon_k)}$$

For the next step, the weights over the training data are updated as in the following,

$$D_{k+1}(i) = \frac{1}{Z_k} D_k(i) \exp\left(-\alpha_k y_i h_k(x_i)\right)$$

The final classification result is assembled using the α_k factors, $g = \text{sgn}(\sum_k \alpha_k h_k)$.

To re-do the problem above using boosting with perceptrons, we set up the AdaBoost classifier in the following,

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> clf=AdaBoostClassifier(Perceptron(),n_estimators=3,
...                        algorithm='SAMME',
...                        learning_rate=0.5)
>>> clf
AdaBoostClassifier(algorithm='SAMME', base_estimator=Perceptron(alpha=0.0001,cla
ss_weight=None,eta0=1.0,fit_intercept=True,
n_iter=5,n_jobs=1,penalty=None,random_state=0,shuffle=False,
verbose=0,warm_start=False), learning_rate=0.5,n_estimators=3,random_state=None)
```

The `learning_rate` above controls how aggressively the weights are updated. The resulting classification boundaries for the embedded perceptrons are shown in Fig. 4.50. Compare this to the lower right panel in Fig. 4.49. The performance for both cases is about the same. The IPython notebook corresponding to this section has more details and the full listing of code used to produce all these figures.

References

1. L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference* (Springer, 2004)
2. V. Vapnik, *The Nature of Statistical Learning Theory* (Springer, Information Science and Statistics, 2000)
3. J. Fox, *Applied Regression Analysis and Generalized Linear Models* (Sage Publications, 2015)
4. K.J. Lindsey, *Applying Generalized Linear Models* (Springer, 1997)
5. S.L. Campbell, C.D. Meyer, *Generalized Inverses of Linear Transformations*, vol. 56 (SIAM, 2009)
6. C. Bauckhage, *Numpy/scipy Recipes for Data Science: Kernel Least Squares Optimization* (1) (researchgate.net, March 2015)
7. W. Richert, *Building Machine Learning Systems With Python* (Packt Publishing Ltd, 2013)
8. E. Alpaydin, *Introduction to Machine Learning* (Wiley Press, 2014)
9. H. Cuesta, *Practical Data Analysis* (Packt Publishing Ltd, 2013)
10. A.J. Izenman, *Modern Multivariate Statistical Techniques*, vol. 1 (Springer, 2008)
11. A. Hyvarinen, J. Karhunen, E. Oja, *Independent Component Analysis*, vol. 46 (John Wiley & Sons, 2004)

Index

J. Unpingco, *Python for Probability, Statistics, and Machine Learning*,
DOI 10.1007/978-3-319-30717-6