Python Data Science Handbook

Essential Tools for Working with Data

Stephen CUI

March 8, 2023

Contents

Part I

Introduction to NumPy

Chapter 1

Understanding Data Types in Python

1.1 Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in array module (available since Python 3.3) can be used to create dense arrays of a uniform type.

While Python's array object provides efficient storage of array-based data, NumPy adds to this efficient operations on that data.

1.2 Creating Arrays from Python Lists

Remember that unlike Python lists, NumPy arrays can only contain data of the same type. If the types do not match, NumPy will upcast them according to its type promotion rules.

1.3 Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy.

1.4 NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. The standard NumPy data types are listed in Table 1.1. Note that when constructing an array, they can be specified using a string or using the associated NumPy object. NumPy also supports compound data types, which will be covered in Chapter 9.

Table 1.1: Standard NumPy data types

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C long; normally either int64 or int32)
`intc`	Identical to C int (normally int32 or int64)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either int32 or int64)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for float64
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for complex128
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Chapter 2

The Basics of NumPy Arrays

We'll cover a few categories of basic array manipulations here:

1. Attributes of arrays: Determining the size, shape, memory consumption, and data types of arrays
2. Indexing of arrays: Getting and setting the values of individual array elements
3. Slicing of arrays: Getting and setting smaller subarrays within a larger array
4. Reshaping of arrays: Changing the shape of a given array
5. Joining and splitting of arrays: Combining multiple arrays into one, and splitting one array into many

2.1 NumPy Array Attributes

Each array has attributes including ndim (the number of dimensions), shape (the size of each dimension), size (the total size of the array), and dtype (the type of each element)

2.2 Array Indexing: Accessing Single Elements

- In a one-dimensional array, the ith value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists
- To index from the end of the array, you can use negative indices
- In a multidimensional array, items can be accessed using a comma-separated (row, column) tuple
- Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value into an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

2.3 Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the slice notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0, stop=<size of dimension>, step=1`.

2.3.1 One-Dimensional Subarrays

A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array. 如果step大于0，则start要小于stop，反之要大于，因为每一步之后要进行step处理。

2.3.2 Multidimensional Subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas.

One commonly needed routine is accessing single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:).

In the case of row access, the empty slice can be omitted for a more compact syntax

2.3.3 Subarrays as No-Copy Views

Unlike Python list slices, NumPy array slices are returned as views rather than copies of the array data. views

Now if we modify this subarray, we'll see that the original array is changed!

Some users may find this surprising, but it can be advantageous: for example, when working with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

2.3.4 Creating Copies of Arrays

2.4 Reshaping of Arrays

Another useful type of operation is reshaping of arrays, which can be done with the `reshape` method. In most cases the reshape method will return a no-copy view of the initial array.

A common reshaping operation is converting a one-dimensional array into a two-dimensional row or column matrix.

A convenient shorthand for this is to use `np.newaxis` in the slicing syntax.

2.5 Array Concatenation and Splitting

2.5.1 Concatenation of Arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument.

For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack) and np.hstack (horizontal stack) functions.

Similarly, for higher-dimensional arrays, np.dstack will stack arrays along the third axis.

2.5.2 Splitting of Arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points.

Notice that N split points leads to $N+1$ subarrays. The related functions np.hsplit and np.vsplit are similar.

Chapter 3

Computation on NumPy Arrays:Universal Functions

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use vectorized operations, generally implemented through NumPy's universal functions (ufuncs).

3.1 The Slowness of Loops

This is partly due to the dynamic, interpreted nature of the language; types are flexible, so sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated.

3.2 Introducing Ufuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a vectorized operation.

Vectorized operations in NumPy are implemented via ufuncs, whose main purpose is to quickly execute repeated operations on values in NumPy arrays.

Any time you see such a loop in a NumPy script, you should consider whether it can be replaced with a vectorized expression.

3.3 Exploring NumPy's Ufuncs

Ufuncs exist in two flavors: **unary ufuncs**, which operate on a single input, and **binary ufuncs**, which operate on two inputs.

Array Arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used.

All of these arithmetic operations are simply convenient wrappers around specific ufuncs built into NumPy.

Table 3.1: Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Table 3.1 lists the arithmetic operators implemented in NumPy.

Additionally, there are Boolean/bitwise operators.

Absolute Value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function.

The corresponding NumPy ufunc is `np.absolute`, which is also available under the alias `np.abs`.

This ufunc can also handle complex data, in which case it returns the magnitude.

Trigonometric Functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. Inverse trigonometric functions are also available. The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero.

Exponents and Logarithms

Other common operations available in NumPy ufuncs are the exponentials.

The inverse of the exponentials, the logarithms, are also available. The basic np.log gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well.

There are also some specialized versions that are useful for maintaining precision with very small input, `np.expm1`, `np.log1p`. When x is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

Specialized Ufuncs

Another excellent source for more specialized ufuncs is the submodule scipy.special. If you want to compute some obscure mathematical function on your data, chances are it is implemented in scipy.special.

In mathematics, the error function (also called the Gauss error function), often denoted by erf, is a complex function of a complex variable defined as:

$$erfz = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

In mathematics, the gamma function is one commonly used extension of the factorial function to complex numbers.

$$\Gamma(z) = \int_0^\infty t^z e^{-t} dt$$

In mathematics, the beta function, also called the Euler integral of the first kind, is a special function that is closely related to the gamma function and to binomial coefficients. It is defined by the integral:

$$B(z_1, z_2) = \int_0^1 t^{z_1-1}(1-t)^{z_2-1} dt$$

3.4 Advanced Ufunc Features

Specifying Output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. For all ufuncs, this can be done using the `out` argument of the function. This can even be used with array views.

```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the y array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the out argument can be significant.

Aggregations

For binary ufuncs, aggregations can be computed directly from the object. For example, if we'd like to

reduce

reduce(规约, fold) an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains. If we'd like

accumulate

to store all the intermediate results of the computation, we can instead use **accumulate**.

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`).

Outer Products

outer

Finally, any ufunc can compute the output of all pairs of two different inputs using the outer method.(这里应该不能翻译为外积，虽然有些类似)

The `ufunc.at` and `ufunc.reduceat` methods are useful as well.

Chapter 4

Aggregations: min, max, and Everything in Between

4.1 Summing the Values in an Array

Python内置的sum函数和Numpy提供的sum函数执行的结果都是一样的，但是`np.sum`执行的效率更快。

Be careful, though: the sum function and the np.sum function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings (sum(x, 1) initializes the sum at 1, while np.sum(x, 1) sums along axis 1), and np.sum is aware of multiple array dimensions.

4.2 Minimum and Maximum

Similarly, Python has built-in min and max functions, used to find the minimum value and maximum value of any given array. NumPy's corresponding functions have similar syntax, and again operate much more quickly.

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself.

4.2.1 Multidimensional Aggregates

One common type of aggregation operation is an aggregate along a row or column.

NumPy aggregations will apply across all elements of a multidimensional array.

Aggregation functions take an additional argument specifying the `axis` along which the aggregate is computed. The axis keyword specifies the dimension of the array that will be collapsed, rather than the dimension that will be returned. So, specifying axis=0 means that axis 0 will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

axis

4.2.2 Other Aggregation Functions

NumPy provides several other aggregation functions with a similar API, and additionally most have a NaN-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value.

Table 4.1: Aggregation functions available in NumPy

Function name	NaN-safe version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute mean of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

Table 4.1 provides a list of useful aggregation functions available in NumPy.

4.3 Example: What Is the Average Height of US Presidents?

Chapter 5

Computation on Arrays: Broadcasting

This chapter discusses **broadcasting**: a set of rules by which NumPy lets you apply binary operations (e.g., addition, subtraction, multiplication, etc.) between arrays of different sizes and shapes. broadcasting

5.1 Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis. Broadcasting allows these types of binary operations to be performed on arrays of dif- ferent sizes.

5.2 Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
2. Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

5.2.1 Broadcasting Example 1

Suppose we want to add a two-dimensional array to a one-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays, which have the following shapes:

- `M.shape` is (2, 3)
- `a.shape` is (3,)

We see by rule 1 that the array a has fewer dimensions, so we pad it on the left with ones:

- `M.shape` is (2, 3)
- `a.shape` is (1, 3)

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape` is (2, 3)
- `a.shape` is (2, 3)

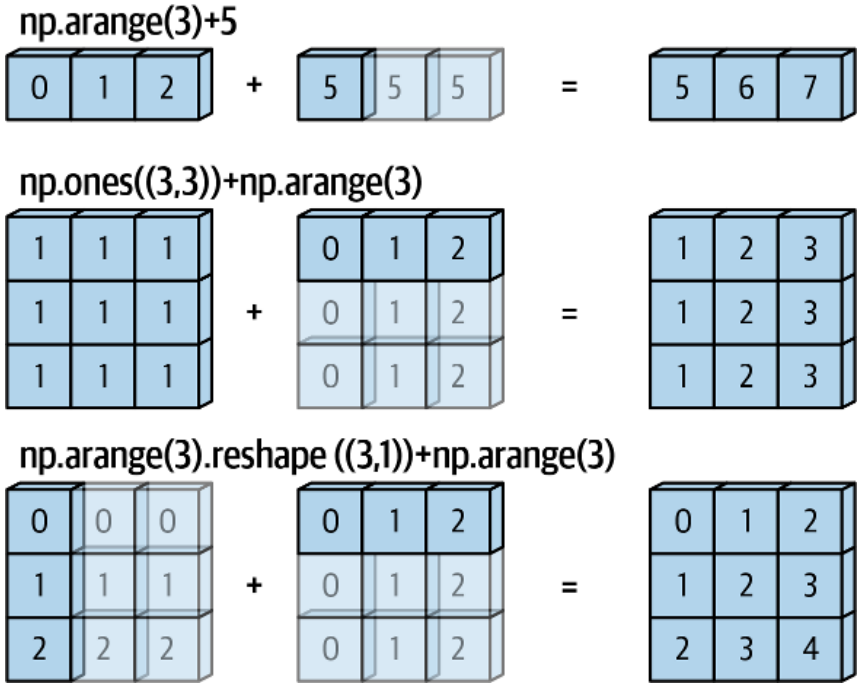


Figure 5.1: Visualization of NumPy broadcasting

5.2.2 Broadcasting Example 2

Now let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
a.shape, b.shape
# ((3, 1), (3,))
```

Again, we'll start by determining the shapes of the arrays:

- `a.shape` is (3, 1)
- `b.shape` is (3,)

Rule 1 says we must pad the shape of b with ones:

- `a.shape` is (3, 1)
- `b.shape` is (1, 3)

And rule 2 tells us that we must upgrade each of these 1s to match the corresponding size of the other array:

- `a.shape` is (3, 3)
- `b.shape` is (3, 3)

5.2.3 Broadcasting Example 3

Next, let's take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))
a = np.arange(3)
M.shape, a.shape
# ((3, 2), (3,))
```

This is just a slightly different situation than in the first example: the matrix M is transposed. How does this affect the calculation? The shapes of the arrays are as follows:

- `M.shape` is (3, 2)
- `a.shape` is (3,)

Again, rule 1 tells us that we must pad the shape of a with ones:

- `M.shape` is (3, 2)
- `a.shape` is (1, 3)

By rule 2, the first dimension of a is then stretched to match that of M:

- `M.shape` is (3, 2)
- `a.shape` is (3, 3)

Now we hit rule 3—the final shapes do not match, so these two arrays are incompatible.

5.3 Broadcasting in Practice

Chapter 6

Comparisons, Masks, and Boolean Logic

Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

6.1 Example: Counting Rainy Days

We saw in Chapter 3 that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise comparisons over arrays

6.2 Comparison Operators as Ufuncs

NumPy implements comparison operators such as $<$ (less than) and $>$ (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available.

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufuncs is shown here:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
$==$	`np.equal`	$!=$	`np.not_equal`
$<$	`np.less`	$<=$	`np.less_equal`
$>$	`np.greater`	$>=$	`np.greater_equal`

6.3 Working with Boolean Arrays

Counting Entries

To count the number of True entries in a Boolean array, `np.count_nonzero` is useful. Another way to get at this information is to use np.sum; in this case, False is interpreted as 0, and True is interpreted as 1.

The benefit of np.sum is that, like with other NumPy aggregation functions, this summation can be done along rows or columns as well.

If we're interested in quickly checking whether any or all the values are True, we can use (you guessed it) `np.any` or `np.all`. `np.all` and `np.any` can be used along particular axes as well.

Boolean Operators

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
&	`np.bitwise_and`	|	`np.bitwise_or`
^	`np.bitwise_xor`	~	`np.bitwise_not`

6.4 Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves.

To select values from array, we can simply index on this Boolean array; this is known as a **masking** operation. What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True.

masking

6.5 Using the Keywords and/or Versus the Operators &/|

One common point of confusion is the difference between the keywords and and or on the one hand, and the operators & and | on the other. When would you use one versus the other?

The difference is this: and and or operate on the object as a whole, while & and | operate on the elements within the object.

When you use and or or, it is equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as True.

When you use & and | on integers, the expression operates on the bitwise representation of the element, applying the and or the or to the individual bits making up the number.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and & and | will operate.

But if you use or on these arrays it will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value.

Similarly, when evaluating a Boolean expression on a given array, you should use | or & rather than or or and.

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError`.

So, remember this: and and or perform a single Boolean evaluation on an entire object, while & and | perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. For Boolean NumPy arrays, the latter is nearly always the desired operation.

Chapter 7

Fancy Indexing

In this chapter, we'll look at another style of array indexing, known as fancy or vectorized indexing, in which we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

7.1 Exploring Fancy Indexing

Fancy Indexing 在概念上非常简单，它意味着传递一个索引数组来一次性获得多个数组元素。

When using arrays of indices, the shape of the result reflects the shape of the index arrays rather than the shape of the array being indexed.

Fancy indexing also works in multiple dimensions. The pairing of indices in fancy indexing follows all the broadcasting rules that were mentioned in Chapter 5.

这里特别需要记住的是，花哨的索引返回的值反映的是广播后的索引数组的形状，而不是被索引的数组的形状。

7.2 Combined Indexing

We can combine fancy and simple indices, combine fancy indexing with slicing, combine fancy indexing with masking.

7.3 Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array.

Notice, though, that repeated indices with these operations can cause some potentially unexpected results.

```
x = np.array([6., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x # array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])
```

分析这段代码，需要将`x[i] += 1`改写成`x[i] = x[i] + 1`，这样先计算`x[i] + 1`，然后将结果赋值给`x[i]`，执行中间过程如下：

```
x = np.zeros(10)
x[[0, 0]] = [4, 6]
temp = x[i] + 1
print(temp) # [1. 1. 1. 1. 1. 1.]
x[i] = temp
x[i] # array([1., 1., 1., 1., 1., 1.])
```

这里的重复赋值会因为重复的索引而出现替换，因此仅会出现最后一个索引的位置被修改。

So what if you want the other behavior where the operation is repeated? For this, you can use the at method of ufuncs.

The at method does an in-place application of the given operator at the specified indices with the specified value . Another method that is similar in spirit is the reduceat method of ufuncs, which you can read about in the NumPy documentation.

in-place

Chapter 8

Sorting Arrays

This chapter covers algorithms related to sorting values in NumPy arrays.

Python has a couple of built-in functions and methods for sorting lists and other iterable objects. The
sorted `sorted` function accepts a list and returns a sorted version of it.
sort By contrast, the `sort` method of lists will sort the list in-place.

Python's sorting methods are quite flexible, and can handle any iterable object.

8.1 Fast Sorting in NumPy: np.sort and np.argsort

The `np.sort` function is analogous to Python's built-in sorted function, and will efficiently return a sorted copy of an array.

Similarly to the `sort` method of Python lists, you can also sort an array in-place using the array `sort` method.

A related function is `argsort`, which instead returns the indices of the sorted elements.

The resulted indices can then be used (via fancy indexing) to construct the sorted array if desired.

8.2 Sorting Along Rows or Columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument.

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

8.3 Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the k smallest values in the array. NumPy enables this with the np.partition function. np.partition takes an array and a number k; the result is a new array with the smallest k values to the left of the partition and the remaining values to the right. Within the two partitions, the elements have arbitrary order.

Finally, just as there is an np.argsort function that computes indices of the sort, there is an `np.argpartition` function that computes indices of the partition.

Chapter 9

Structured Data: NumPy's Structured Arrays

9.1 Exploring Structured Array Creation

We can use dictionary method to create structured array, which key is names and formats.

For clarity, numerical types can be specified using Python types or NumPy dtypes instead.

A compound type can also be specified as a list of tuples. If the names of the types do not matter to you, you can specify the types alone in a comma-separated string.

The shortened string format codes may not be immediately intuitive, but they are built on simple principles. The first (optional) character < or >, means "little endian"(低字节序) or "big endian," respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see Table 9.1). The last character or characters represent the size of the object in bytes.

Table 9.1: NumPy data types

Character	Description	Example
'b'	Byte	`np.dtype('b')`
'i'	Signed integer	`np.dtype('i4') == np.int32`
'u'	Unsigned integer	`np.dtype('u1') == np.uint8`
'f'	Floating point	`np.dtype('f8') == np.int64`
'c'	Complex floating point	`np.dtype('c16') == np.complex128`
'S', 'a'	String	`np.dtype('S5')`
'U'	Unicode string	`np.dtype('U') == np.str_`
'V'	Raw data (void)	`np.dtype('V') == np.void`

9.2 More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values.

为什么我们宁愿用这种方法存储数据，也不用简单的多维数组，或者 Python 字典呢？原因是 NumPy 的 dtype 直接映射到 C 结构的定义，因此包含数组内容的缓存可以直接在 C 程序中使用。如果你想写一个 Python 接口与一个遗留的 C 语言或 Fortran 库交互，从而操作结构化数据，你将会发现结构化数组非常有用!

9.3 Record Arrays: Structured Arrays with a Twist

NumPy also provides record arrays (instances of the np.recarray class), which are almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys.

If we view our data as a record array instead, we can access this with slightly fewer keystrokes. The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax.

Whether the more convenient notation is worth the (slight) overhead will depend on your own application.

9.4 On to Pandas

This chapter on structured and record arrays is purposely located at the end of this part of the book, because it leads so well into the next package we will cover: Pandas. Structured arrays can come in handy in certain situations, like when you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language.

Part II

Data Manipulation with Pandas

正如我们之前看到的那样，NumPy 的 ndarray 数据结构为数值计算任务中常见的干净整齐、组织良好的数据提供了许多不可或缺的功能。虽然它在这方面做得很好，但是当我们需要处理更灵活的数据任务（如为数据添加标签、处理缺失值等），或者需要做一些不是对每个元素都进行广播映射的计算（如分组、透视表等）时，NumPy 的限制就非常明显了，而这些都是分析各种非结构化数据时很重要的一部分。建立在 NumPy 数组结构上的 Pandas，尤其是它的 Series 和 DataFrame 对象，为数据科学家们处理那些消耗大量时间的“数据清理”（data munging）任务提供了捷径。

Chapter 10

Introducing Pandas Objects

At a very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's take a look at these three fundamental Pandas data structures: the `Series`, `DataFrame`, and `Index`.

Series
DataFrame
Index

10.1 The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows.

The Series combines a sequence of values with an explicit sequence of indices, which we can access with the `values` and `index` attributes. The values are simply a familiar NumPy array.

values
index

The index is an array-like object of type pd.Index.

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation.

Series as Generalized NumPy Array

From what we've seen so far, the Series object may appear to be basically interchangeable with a one-dimensional NumPy array. The essential difference is that while the NumPy array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type.

Series as Specialized Dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure that maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code

behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it more efficient than Python dictionaries for certain operations.

The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary.

Unlike a dictionary, though, the Series also supports array-style operations such as slicing.

Constructing Series Objects

We've already seen a few ways of constructing a Pandas Series from scratch. All of them are some version of the following:

```
pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities.

10.2 The Pandas DataFrame Object

Like the Series object discussed in the previous section, the DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

DataFrame as Generalized NumPy Array

Like the Series object, the DataFrame has an index attribute that gives access to the index labels.

Additionally, the DataFrame has a columns attribute, which is an Index object holding the column labels.

DataFrame as Specialized Dictionary

Similarly, we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data.

Notice the potential point of confusion here: in a two-dimensional NumPy array, `data[0]` will return the first row. For a DataFrame, `data['col0']` will return the first column. Because of this, it is probably better to think about DataFrames as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful.

Constructing DataFrame Objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll explore several examples.

From a single Series object

A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series.

From a list of dicts

Any list of dictionaries can be made into a DataFrame. Even if some keys in the dictionary are missing, Pandas will fill them in with NaN values.

From a dictionary of Series objects

A DataFrame can be constructed from a dictionary of Series objects as well.

From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names.

From a NumPy structured array

A Pandas DataFrame operates much like a structured array, and can be created directly from one.

10.3 The Pandas Index Object

As you've seen, the Series and DataFrame objects both contain an explicit index that lets you reference and modify data. This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set (technically a multiset, as Index objects may contain repeated values).

Index as Immutable Array

The Index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices.

Index objects also have many of the attributes familiar from NumPy arrays.

One difference between Index objects and NumPy arrays is that the indices are immutable—that is, they cannot be modified via the normal means. This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.

Index as Ordered Set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way.

Chapter 11

Data Indexing and Selection

If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

11.1 Data Indexing and Selection

As you saw in the previous chapter, a Series object acts in many ways like a one- dimensional NumPy array, and in many ways like a standard Python dictionary. If you keep these two overlapping analogies in mind, it will help you understand the patterns of data indexing and selection in these arrays.

Series as Dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values.

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values.

Series objects can also be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value.

Series as One-Dimensional Array

A Series builds on this dictionary-like interface and provides array-style item selec- tion via the same basic mechanisms as NumPy arrays—that is, slices, masking, and fancy indexing.

> Notice that when slicing with an explicit index (e.g., `data['a':'c']`), the final index is included in the slice, while when slicing with an implicit index (e.g., `data[0:2]`), the final index is excluded from the slice.

Indexers: loc and iloc

If your Series has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style indices.

Because of this potential confusion in the case of integer indexes, Pandas provides some special indexer attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the Series.

loc First, the loc attribute allows indexing and slicing that always references the explicit index.

iloc The iloc attribute allows indexing and slicing that always references the implicit Python-style index.

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of loc and iloc makes them helpful in maintaining clean and readable code; especially in the case of integer indexes, using them consistently can prevent subtle bugs due to the mixed indexing/slicing convention.

11.2 Data Selection in DataFrames

Recall that a DataFrame acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of Series structures sharing the same index.

DataFrame as Dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects.

The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name. Equivalently, we can use attribute-style access with column names that are strings.

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. In particular, you should avoid the temptation to try column assignment via attributes.

DataFrame as Two-Dimensional Array

values We can examine the raw underlying data array using the values attribute.

When it comes to indexing of a DataFrame object, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row and passing a single "index" to a DataFrame accesses a column.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the loc indexer we can combine masking and fancy indexing.

Additional Indexing Conventions

There are a couple of extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be useful in practice.

- First, while indexing refers to columns, slicing refers to rows.
- Similarly, direct masking operations are interpreted row-wise rather than column-wise

Chapter 12

Operating on Data in Pandas

One of the strengths of NumPy is that it allows us to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more complicated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs introduced in Chapter 3 are key to this.

Pandas includes a couple of useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will preserve index and column labels in the output(保留index和column), and for binary operations such as addition and multiplication, Pandas will automatically align indices(自动对齐索引进行运算) when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources—both potentially error-prone tasks with raw NumPy arrays—become essentially foolproof with Pandas.

12.1 Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects.

12.2 Ufuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data.

Index Alignment in Series

The resulting array contains the union of indices of the two input arrays, which could be determined directly from these indices.

Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number," which is how Pandas marks missing data. This index matching is implemented this way for any of Python's built-in arithmetic expressions; any missing values are marked by NaN.

If using NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling `A.add(B)` is equivalent to calling A + B, but allows

Table 12.1: Mapping between Python operators and Pandas methods

Python operator	Pandas method(s)
+	add
-	sub, subtract
*	mul, multiply
/	truediv, div, divide
//	floordiv
%	mod
**	pow

optional explicit specification of the fill value for any elements in A or B that might be missing.(交集执行运算，差集使用fill_value执行运算)

Index Alignment in DataFrames

A similar type of alignment takes place for both columns and indices when performing operations on DataFrame objects.

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted.

Table 12.1 lists Python operators and their equivalent Pandas object methods.

12.3 Ufuncs: Operations Between DataFrames and Series

When performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained, and the result is similar to operations between a two-dimensional and one-dimensional NumPy array.

In Pandas, the convention similarly operates row-wise by default.

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the axis keyword.

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the common errors that might arise when working with heterogeneous and/or misaligned data in raw NumPy arrays.

Chapter 13

Handling Missing Data

In this chapter, we will discuss some general considerations for missing data, look at how Pandas chooses to represent it, and explore some built-in Pandas tools for handling missing data in Python.

13.1 Trade-offs in Missing Data Conventions

A number of approaches have been developed to track the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value(标签值) that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it might involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value that is part of the IEEE floating-point specification.

Neither of these approaches is without trade-offs. Use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often nonoptimized) logic in CPU and GPU arithmetic, because common special values like NaN are not available for all data types.

13.2 Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non- floating-point data types.

Because of these constraints and trade-offs, Pandas has two “modes” of storing and manipulating null values:

- The default mode is to use a sentinel-based missing data scheme, with sentinel values NaN or None depending on the type of the data.
- Alternatively, you can opt in to using the nullable data types (dtypes) Pandas provides (discussed later in this chapter), which results in the creation an accompanying mask array to track missing entries. These missing entries are then presented to the user as the special pd.NA value.

Table 13.1: Pandas handling of NAs by type

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

None as a Sentinel Value

For some data types, Pandas uses None as a sentinel value. None is a Python object, which means that any array containing None must have `dtype=object`—that is, it must be a sequence of Python objects.

This `dtype=object` means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. The downside of using None in this way is that operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types.

Further, because Python does not support arithmetic operations with None, aggregations like sum or min will generally lead to an error.

For this reason, Pandas does not use None as a sentinel in its numerical arrays.

NaN: Missing Numerical Data

The other missing data sentinel, NaN is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation.

这和使用 None 的数组不同，使用 np.nan 的数组会被编译成C代码从而实现快速操作。Keep in mind that NaN is a bit like a data virus—it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN.

This means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful. NumPy does provide NaN-aware versions of aggregations that will ignore these missing values.

The main downside of NaN is that it is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None in Pandas

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate.

For types that don't have an available sentinel value, Pandas automatically typecasts when NA values are present. 比如：如果有一个整数组成的 Series，将 None 赋值给其中一个元素，那么Series的类型将转换为 float.

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the None to a NaN value.

Table 13.1 lists the upcasting conventions in Pandas when NA values are introduced.

Keep in mind that in Pandas, string data is always stored with an object dtype.

13.3 Pandas Nullable Dtypes

In early versions of Pandas, NaN and None as sentinel values were the only missing data representations available. The primary difficulty this introduced was with regard to the implicit type casting. Table 13.1 可以看出来，没有办法真正表示整数的缺失值，因为会被转成浮点型。

To address this difficulty, Pandas later added **nullable dtypes**, which are distinguished from regular dtypes by capitalization of their names (e.g., pd.Int32 versus np.int32). For backward compatibility, these nullable dtypes are only used if specifically requested.

nullable dtypes

13.4 Operating on Null Values

As we have seen, Pandas treats None, NaN, and NA as essentially interchangeable for indicating missing or null values. To facilitate this convention, Pandas provides several methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- isnull: Generates a Boolean mask indicating missing values
- notnull: Opposite of isnull
- dropna: Returns a filtered version of the data
- fillna: Returns a copy of the data with missing values filled or imputed

Detecting Null Values

Pandas data structures have two useful methods for detecting null data: isnull and notnull. Either one will return a Boolean mask over the data.

Dropping Null Values

In addition to these masking methods, there are the convenience methods dropna (which removes NA values) and fillna (which fills in NA values).

For a DataFrame, there are more options. We cannot drop single values from a DataFrame; we can only drop entire rows or columns.

By default, dropna will drop all rows in which any null value is present. But this drops some good data as well; you might rather be interested in dropping rows or columns with all NA values, or a majority of NA values. This can be specified through the how or thresh parameters, which allow fine control of the number of nulls to allow through.(可以这是方式或者阈值来删除行或者列，而不是默认的存在空值即删除)

The default is `how='any'`, such that any row or column containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that contain all null values.

how

For finer-grained control, the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept.

thresh

Alternatively, you can drop NA values along a different axis.

Filling Null Values

Sometimes rather than dropping NA values, you'd like to replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the isnull method as a mask, but because it is such a common operation Pandas provides the fillna method, which returns a copy of the array with the null values replaced.

We can fill NA entries with a single value, such as zero, can specify a forward fill to propagate the previous value forward, or specify a backward fill to propagate the next values backward.

In the case of a DataFrame, the options are similar, but we can also specify an axis along which the fills should take place.

Chapter 14

Hierarchical Indexing

A far more common pattern for handling higher-dimensional data is to make use of hierarchical indexing (also known as multi- indexing) to incorporate multiple index levels within a single index.

14.1 A Multiply Indexed Series

Let’s start by considering how we might represent two-dimensional data within a one-dimensional Series.

Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndex type gives us the types of oper- ations we wish to have. We can create a multi-index from the tuples.

The MultiIndex represents multiple levels of indexing as well as multiple labels for each data point which encode these levels.

In this multi-index representation, any blank entry indicates the same value as the line above it.

MultiIndex as Extra Dimension

The unstack method will quickly convert a multiply indexed Series into a conventionally indexed DataFrame.unstack Naturally, the stack method provides the opposite operation. stack

你可能会纠结于为什么要费时间研究层级索引。其实理由很简单：如果我们可以用含多级索引的一维 Series 数据表示二维数据，那么我们就可以用 Series 或 DataFrame 表示三维甚至更高维度的数据。多级索引每增加一级，就表示数据增加一维，利用这一特点就可以轻松表示任意维度的数据了。

In addition, all the ufuncs and other functionality discussed in Chapter 12 work with hierarchical indices as well.

14.2 Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor.(我个人觉得这种两层索引或者少量层索引还有合适，多层索引的话 List 嵌套根本看不出来)

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default.

Explicit MultiIndex Constructors

For more flexibility in how the index is constructed, you can instead use the constructor methods available in the `pd.MultiIndex` class.

You can construct a MultiIndex from a simple list of arrays giving the index values within each level, or you can construct it from a list of tuples giving the multiple index values of each point.

ou can even construct it from a Cartesian product of single indices.

Similarly, you can construct a MultiIndex directly using its internal encoding by passing levels (a list of lists containing available index values for each level) and codes (a list of lists that reference these labels).

Any of these objects can be passed as the index argument when creating a Series or DataFrame, or be passed to the reindex method of an existing Series or DataFrame.

MultiIndex Level Names

Sometimes it is convenient to name the levels of the MultiIndex. This can be accomplished by passing
names the names argument to any of the previously discussed MultiIndex constructors, or by setting the names attribute of the index after the fact.

MultiIndex for Columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well.

14.3 Indexing and Slicing a MultiIndex

Multiply Indexed Series

We can access single elements by indexing with multiple terms.

The MultiIndex also supports partial indexing, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained.

Partial slicing is available as well, as long as the MultiIndex is sorted.(需要索引是排序的) With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index.

Other types of indexing and selection (discussed in Chapter 11) work as well.

Multiply Indexed DataFrames

Remember that columns are primary in a DataFrame, and the syntax used for multiply indexed Series applies to the columns.

Also, as with the single-index case, we can use the loc, iloc, and ix indexers introduced in Chapter 11.

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in loc or iloc can be passed a tuple of multiple indices.

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error

You could get around this by building the desired slice explicitly using Python’s built- in slice function,
IndexSlice but a better way in this context is to use an IndexSlice object, which Pandas provides for precisely this situation.

14.4 Rearranging Multi-Indexes

One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations.

14.4.1 Sorted and Unsorted Indices

Many of the MultiIndex slicing operations will fail if the index is not sorted.

We’ll start by creating some simple multiply indexed data where the indices are not lexographically sorted(不是按照字典排序的).

For various reasons, partial slices and other similar operations require the levels in the MultiIndex to be in sorted (i.e., lexographical) order. Pandas provides a number of convenience routines to perform this type of sorting, such as the `sort_index` and `sortlevel` methods of the DataFrame.

Stacking and Unstacking Indices

As we saw briefly before, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use.

The opposite of unstack is stack, which here can be used to recover the original series. unstack stack

Index Setting and Resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a DataFrame with state and year columns holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation. reset_index

A common pattern is to build a MultiIndex from the column values. This can be done with the `set_index` method of the DataFrame, which returns a multiply indexed DataFrame set_index

Chapter 15

Combining Datasets: concat and append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets to more complicated database-style joins and merges that correctly handle any overlaps between the datasets.

15.1 Recall: Concatenation of NumPy Arrays

Concatenation of Series and DataFrame objects behaves similarly to concatenation of NumPy arrays,
concatenate which can be done via the `np.concatenate` function.

The first argument is a list or tuple of arrays to concatenate. Additionally, in the case of multidimensional arrays, it takes an axis keyword that allows you to specify the axis along which the result will be concatenated.

15.2 Simple Concatenation with pd.concat

concat The `pd.concat` function provides a similar syntax to np.concatenate but contains a number of options.

pd.concat can be used for a simple concatenation of Series or DataFrame objects, just as np.concatenate can be used for simple concatenations of arrays.

It's default behavior is to concatenate row-wise within the DataFrame (i.e., axis=0). Like np.concatenate, pd.concat allows specification of an axis along which concatenation will take place.

Duplicate Indices

One important difference between np.concatenate and pd.concat is that Pandas concatenation preserves indices, even if the result will have duplicate indices! While this is valid within DataFrames, the outcome is often undesirable. pd.concat gives us a few ways to handle it.

Treating repeated indices as an error

If you'd like to simply verify that the indices in the result of pd.concat do not overlap, you can include the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices.

Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to True, the concatenation will create a new integer index for the resulting DataFrame

Adding MultiIndex keys

Another option is to use the keys option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data

Concatenation with Joins

In practice, data from different sources might have different sets of column names, and pd.concat offers several options in this case.

The default behavior is to fill entries for which no data is available with NA values. To change this, we can adjust the join parameter of the concat function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`.

Another useful pattern is to use the reindex method before concatenation for finer control over which columns are dropped.(这种效果似乎可以实现左连接或者右连接，因为参数 join 只能是内连接和外连接两种)

The append Method

Because direct array concatenation is so common, Series and DataFrame objects have an append method that can accomplish the same thing in fewer keystrokes.

Keep in mind that unlike the append and extend methods of Python lists, the append method in Pandas does not modify the original object; instead it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index and data buffer. Thus, if you plan to do multiple append operations, it is generally better to build a list of DataFrame objects and pass them all at once to the concat function.

FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

Chapter 16

Combining Datasets: merge and join

One important feature offered by Pandas is its high-performance, in-memory join and merge operations, which you may be familiar with if you have ever worked with databases. The main interface for this is the `pd.merge` function.

merge

16.1 Relational Algebra

The behavior implemented in pd.merge is a subset of what is known as relational algebra, which is a formal set of rules for manipulating relational data that forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several fundamental operations, which become the building blocks of more complicated operations on any dataset.

16.2 Categories of Joins

The pd.merge function implements a number of types of joins: one-to-one, many-to-one, and many-to-many. All three types of joins are accessed via an identical call to the pd.merge interface; the type of join performed depends on the form of the input data.

One-to-One Joins

Perhaps the simplest type of merge is the one-to-one join, which is in many ways similar to the column-wise concatenation in Chapter 15.

The pd.merge function recognizes that each DataFrame has an employee column, and automatically joins using this column as a key. The result of the merge is a new Data Frame that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the employee column differs between df1 and df2, and the pd.merge function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index.

Many-to-One Joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries.

Many-to-Many Joins

If the key column in both the left and right arrays contains duplicates, then the result is a many-to-many merge.

These three types of joins can be used with other Pandas tools to implement a wide array of functionality.

16.3 Specification of the Merge Key

We've already seen the default behavior of pd.merge: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and pd.merge provides a variety of options for handling this.

The on Keyword

Most simply, you can explicitly specify the name of the key column using the on keyword, which takes key a column name or a list of column names. This option works only if both the left and right DataFrames have the specified column name.

The left_on and right_on Keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names.(在不同的表中，可能字段名不一样，但表示的意思是一样的)

The result has a redundant column that we can drop if desired—for example, by using the `DataFrame.drop()` method.

The left_index and right_index Keywords

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in pd.merge().

For convenience, Pandas includes the DataFrame.join() method, which performs an index-based merge without extra keyword.

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior.

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the "Merge, Join, and Concatenate" section of the Pandas documentation.

16.4 Specifying Set Arithmetic for Joins

We have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other.

We can specify this explicitly using the how keyword, which defaults to "inner".

Other options for the how keyword are 'outer', 'left', and 'right'. An outer join returns a join over the union of the input columns and fills in missing values with NAs.

The left join and right join return joins over the left entries and right entries, respectively. All of these options can be applied straightforwardly to any of the preceding join types.

16.5 Overlapping Column Names: The suffixes Keyword

当 merge 的数据框存在冲突的命名时，考虑使用前缀/后缀。

Because the output would have two conflicting column, the merge function automatically appends the suffixes `_x` and `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword。

suffixes

16.6 Example: US States Data

Chapter 17

Aggregation and Grouping

A fundamental piece of many data analysis tasks is efficient summarization: computing aggregations like sum, mean, median, min, and max, in which a single number summarizes aspects of a potentially large dataset.

17.1 Planets Data

Here we will use the Planets dataset, available via the Seaborn package. It gives information on planets that astronomers have discovered around other stars (known as extrasolar planets(系外星系), or exoplanets for short).

17.2 Simple Aggregation in Pandas

As with a one-dimensional NumPy array, for a Pandas Series the aggregates return a single value.

For a DataFrame, by default the aggregates return results within each column. By specifying the axis argument, you can instead aggregate within each row.

Pandas Series and DataFrame objects include all of the common aggregates mentioned in Chapter 4; in addition, there is a convenience method, `describe`, that computes several common aggregates for each column and returns the result. describe

Table 17.1 summarizes some other built-in Pandas aggregations.

17.3 groupby: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called groupby operation.

Split, Apply, Combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated Figure 17.1.

Figure 17.1 shows what the groupby operation accomplishes:

- The split step involves breaking up and grouping a DataFrame depending on the value of the specified key.

Table 17.1: Listing of Pandas aggregation methods

Aggregation	Returns
count	Total number of items
first, last	First and last item
mean, median	Mean and median
min, max	Minimum and maximum
std, var	Standard deviation and variance
mad	Mean absolute deviation
prod	Product of all items
sum	Sum of all items

- The apply step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The combine step merges the results of these operations into an output array.

The power of the groupby is that it abstracts away these steps: the user need not think about how the computation is done under the hood, but rather can think about the operation as a whole.

The most basic split-apply-combine operation can be computed with the groupby method of the DataFrame, passing the name of the desired key column.

Notice that what is returned is a DataFrameGroupBy object, not a set of DataFrame objects. This object is where the magic is: you can think of it as a special view of the DataFrame, which is poised to dig into the groups but does no actual computation until the aggregation is applied.

The GroupBy Object

The GroupBy object is a flexible abstraction: in many ways, it can be treated as simply a collection of DataFrames, though it is doing more sophisticated things under the hood.

Perhaps the most important operations made available by a GroupBy are aggregate, filter, transform, and apply.

Column indexing

The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object.

Iteration over groups

The GroupBy object supports direct iteration over the groups, returning each group as a Series or DataFrame. This can be useful for manual inspection of groups for the sake of debugging, but it is often much faster to use the built-in apply functionality

Dispatch methods

Through some Python class magic, any method not explicitly implemented by the GroupBy object will be passed through and called on the groups, whether they are DataFrame or Series objects.

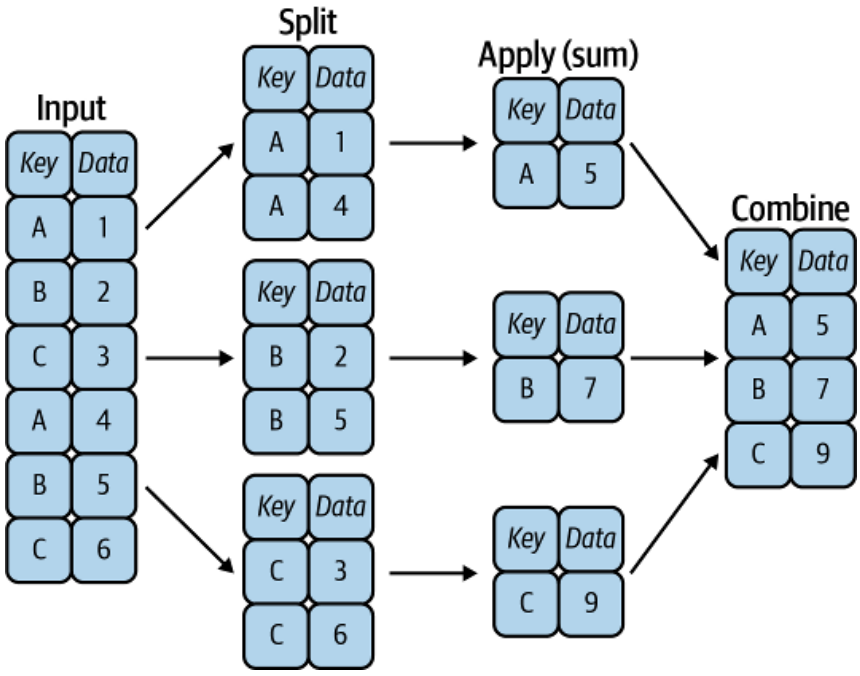


Figure 17.1: A visual representation of a groupby operation

Notice that these dispatch methods are applied to each individual group, and the results are then combined within GroupBy and returned. Again, any valid DataFrame/Series method can be called in a similar manner on the corresponding GroupBy object.

Aggregate, Filter, Transform, Apply

GroupBy objects have aggregate, fil ter, transform, and apply methods that efficiently implement a variety of useful operations before combining the grouped data.

Aggregation

You're now familiar with GroupBy aggregations with sum, median, and the like, but the aggregate method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once.

Another common pattern is to pass a dictionary mapping column names to operations to be applied on that column.

Filtering

A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value.

The filter function should return a Boolean value specifying whether the group passes the filtering.

Transformation

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean

The apply method

The apply method lets you apply an arbitrary function to the group results. The function should take a DataFrame and returns either a Pandas object (e.g., DataFrame, Series) or a scalar; the behavior of the combine step will be tailored to the type of output returned.

apply within a GroupBy is flexible: the only criterion is that the function takes a DataFrame and returns a Pandas object or scalar.

Specifying the Split Key

In the simple examples presented before, we split the DataFrame on a single column name. This is just one of many options by which the groups can be defined.

A list, array, series, or index providing the grouping keys

The key can be any series or list with a length matching that of the DataFrame.

A dictionary or series mapping index to group

Another method is to provide a dictionary that maps index values to the group keys.

Any Python function

Similar to mapping, you can pass any Python function that will input the index value and output the group.

A list of valid keys

Further, any of the preceding key choices can be combined to group on a multi-index.

Grouping Example

Chapter 18

Pivot Tables

We have seen how the groupby abstraction lets us explore relationships within a dataset. A pivot table is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple column- wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and groupby can sometimes cause confusion; it helps me to think of pivot tables as essentially a multidimensional version of groupby aggregation. That is, you split- apply-combine, but both the split and the combine happen across not a one- dimensional index, but across a two-dimensional grid.

18.1 Pivot Tables by Hand

18.2 Pivot Table Syntax

Multilevel Pivot Tables

Just as in a groupby, the grouping in pivot tables can be specified with multiple levels and via a number of options.

We can apply the same strategy when working with the columns as well.

Additional Pivot Table Options

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As with groupby, the aggregation specification can be a string representing one of several common choices ('sum', 'mean', 'count', 'min', 'max', etc.) or a function that implements an aggregation (e.g., np.sum(), min(), sum(), etc.). Additionally, it can be specified as a dictionary mapping a column to any of the desired options. aggfunc

Notice also here that we've omitted the values keyword; when specifying a mapping for aggfunc, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword. The margin label can be specified with the `margins_name` keyword; it defaults to "All". margins

透视表构建的是多层索引的数据框。

Chapter 19

Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of vectorized string operations that are an important part of the type of munging required when working with (read: cleaning up) real-world data.

19.1 Introducing Pandas String Operations

Pandas includes features to address both this need for vectorized string operations as well as the need for correctly handling missing data via the str attribute of Pandas Series and Index objects containing strings.(通过 `str` 属性来实现字符串的向量化操作)

str

19.2 Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of the Pandas string syntax is intuitive enough that it's probably sufficient to just list the available methods.

Methods Similar to Python String Methods

Nearly all of Python's built-in string methods are mirrored by a Pandas vectorized string method. The following Pandas str methods mirror Python string methods:

len	lower	translate	islower	ljust
upper	startswith	isupper	rjust	find
endswith	isnumeric	center	rfind	isalnum
isdecimal	zfill	index	isalpha	split
strip	rindex	isdigit	rsplit	rstrip
capitalize	isspace	partition	lstrip	swapcase

Notice that these have various return values. Some, like lower, return a series of strings. But some others return numbers or Boolean values. Still others return lists or other compound values for each element.

Methods Using Regular Expressions

In addition, there are several methods that accept regular expressions (regexps) to examine the content of each string element, and follow some of the API conventions of Python's built-in re module (see Table 19.1).

Table 19.1: Mapping between Pandas methods and functions in Python's re module

Method	Description
match	Calls re.match on each element, returning a Boolean.
extract	Calls re.match on each element, returning matched groups as strings.
findall	Calls re.findall on each element
replace	Replaces occurrences of pattern with some other string
contains	Calls re.search on each element, returning a boolean
count	Counts occurrences of pattern
split	Equivalent to str.split, but accepts regexps
rsplit	Equivalent to str.rsplit, but accepts regexps

Table 19.2: Other Pandas string methods

Method	Description
`get`	Indexes each element
`slice`	Slices each element
`slice_replace`	Replaces slice in each element with the passed value
`cat`	Concatenates strings
`repeat`	Repeats values
`normalize`	Returns Unicode form of strings
`pad`	Adds whitespace to left, right, or both sides of strings
`wrap`	Splits long strings into lines with length less than a given width
`join`	Joins strings in each element of the Series with the passed separator
`get_dummies`	Extracts dummy variables as a DataFrame

Miscellaneous Methods

Finally, Table 19.2 lists miscellaneous methods that enable other convenient operations.

Vectorized item access and slicing

The get and slice operations, in particular, enable vectorized element access from each array. This behavior is also available through Python's normal indexing syntax.

Indexing via `df.str.get(i)` and `df.str[i]` are likewise similar.

These indexing methods also let you access elements of arrays returned by split. For example, to extract the last name of each entry, combine split with str indexing.

Indicator variables

Another method that requires a bit of extra explanation is the `get_dummies` method. This is useful when your data has a column containing some sort of coded indicator.

19.3 Going Further with Recipes

Going Further with Recipes

Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately, the wide variety of formats used makes this a relatively time-consuming process. This points to the truism that in data science, cleaning and munging of real-world data often comprises the majority of the work.

Chapter 20

Working with Time Series

Pandas was originally developed in the context of financial modeling, so as you might expect, it contains an extensive set of tools for working with dates, times, and time- indexed data. Date and time data comes in a few flavors, which we will discuss here:

- Timestamps(时间戳) Particular moments in time (e.g., July 4, 2021 at 7:00 a.m.).
- Time intervals(时间间隔) and periods(周期) A length of time between a particular beginning and end point; for example, the month of June 2021. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24-hour-long periods comprising days).
- Time deltas(时间增量或者时间差) or durations(持续时间) An exact length of time (e.g., a duration of 22.56 seconds).

20.1 Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and time spans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other tools used in Python.

Native Python Dates and Times: datetime and dateutil

Python's basic objects for working with dates and times reside in the built-in datetime module. Along with the third-party dateutil module, you can use this to quickly perform a host of useful functionalities on dates and times.

The power of datetime and dateutil lies in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times: just as lists of Python numerical variables are suboptimal compared to NumPy-style typed numerical arrays, lists of Python datetime objects are suboptimal compared to typed arrays of encoded dates.

Typed Arrays of Times: NumPy's datetime64

NumPy's datetime64 dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented compactly and operated on in an efficient manner. The datetime64 requires a specific input format.

Table 20.1: Description of date and time codes

Code	Meaning	Time span (relative)	Time span (absolute)
Y	Year	± 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
M	Month	± 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
W	Week	± 1.7e17 years	[1.7e17 BC, 1.7e17 AD]
D	Day	± 2.5e16 years	[2.5e16 BC, 2.5e16 AD]
h	Hour	± 1.0e15 years	[1.0e15 BC, 1.0e15 AD]
m	Minute	± 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
s	Second	± 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
ms	Millisecond	± 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
us	Microsecond	± 2.9e6 years	[290301 BC, 294241 AD]
ns	Nanosecond	± 292 years	[1678 AD, 2262 AD]
ps	Picosecond	± 106 days	[1969 AD, 1970 AD]
fs	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

Once we have dates in this form, we can quickly do vectorized operations on it.

One detail of the datetime64 and related timedelta64 objects is that they are built on a fundamental time unit. Because the datetime64 object is limited to 64-bit precision, the range of encodable times is 2^{64} times this fundamental unit. In other words, datetime64 imposes a trade-off between time resolution and maximum time span.

For example, if you want a time resolution of 1 nanosecond, you only have enough information to encode a range of 2^{64} nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input.

You can force any desired fundamental unit using one of many format codes. Table 20.1, drawn from the NumPy datetime64 documentation, lists the available format codes along with the relative and absolute time spans that they can encode.

For the types of data we see in the real world, a useful default is datetime64[ns], as it can encode a useful range of modern dates with a suitably fine precision.

Finally, note that while the datetime64 data type addresses some of the deficiencies of the built-in Python datetime type, it lacks many of the convenient methods and functions provided by datetime and especially dateutil.

Dates and Times in Pandas: The Best of Both Worlds

Pandas builds upon all the tools just discussed to provide a Timestamp object, which combines the ease of use of datetime and dateutil with the efficient storage and vectorized interface of numpy.datetime64. From a group of these Timestamp objects, Pandas can construct a DatetimeIndex that can be used to index data in a Series or DataFrame.

20.2 Pandas Time Series: Indexing by Time

The Pandas time series tools really become useful when you begin to index data by timestamps.

20.3 Pandas Time Series Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data:

- For timestamps, Pandas provides the Timestamp type. As mentioned before, this is essentially a replacement for Python's native datetime, but it's based on the more efficient numpy.datetime64 data type. The associated Index structure is DatetimeIndex.
- For time periods, Pandas provides the Period type. This encodes a fixed- frequency interval based on numpy.datetime64. The associated index structure is PeriodIndex.
- For time deltas or durations, Pandas provides the Timedelta type. Timedelta is a more efficient replacement for Python's native datetime.timedelta type, and is based on numpy.timedelta64. The associated index structure is TimedeltaIndex.

The most fundamental of these date/time objects are the Timestamp and Datetime Index objects. While these class objects can be invoked directly, it is more common to use the `pd.to_datetime` function, which can parse a wide variety of formats. Passing a single date to `pd.to_datetime` yields a Timestamp; passing a series of dates by default yields a DatetimeIndex.

Any DatetimeIndex can be converted to a PeriodIndex with the `to_period` function, with the addition of a frequency code.

A TimedeltaIndex is created when a date is subtracted from another.

20.4 Regular Sequences: pd.date_range

To make creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: `pd.date_range` for timestamps, `pd.period_range` for periods, and `pd.timedelta_range` for time deltas.

`pd.date_range` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. Alternatively, the date range can be specified not with a start and end point, but with a start point and a number of periods. The spacing can be modified by altering the freq argument, which defaults to D.

To create regular sequences of Period or Timedelta values, the similar `pd.period_range` and `pd.timedelta_range` functions are useful.

20.5 Frequencies and Offsets

Fundamental to these Pandas time series tools is the concept of a frequency or date offset. Table 20.2 summarizes the main codes available.

The monthly, quarterly, and annual frequencies are all marked at the end of the specified period. Adding an S suffix to any of these causes them to instead be marked at the beginning (see Table 20.3).

Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix(比如说美股的财年是任意的):

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, the split point of the weekly frequency can be modified by adding a three-letter weekday code: W-SUN, W-MON, W-TUE, W-WED, etc.

Table 20.2: Listing of Pandas frequency codes

Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseconds		
U	Microseconds		
N	Nanoseconds		

Table 20.3: Listing of start-indexed frequency codes

Code	Description	Code	Description
MS	Month start	BMS	Business month start
QS	Quarter start	BQS	Business quarter start
AS	Year start	BAS	Business year start

All of these short codes refer to specific instances of Pandas time series offsets, which can be found in the pd.tseries.offsets module.

20.6 Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important aspect of the Pandas time series tools. The benefits of indexed data in general (automatic alignment during operations, intuitive data slicing and access, etc.) still apply, and Pandas provides several additional time series–specific operations.

Resampling and Converting Frequencies

resample
asfreq

One common need when dealing with time series data is resampling at a higher or lower frequency. This can be done using the resample method, or the much simpler asfreq method. The primary difference between the two is that resample is fundamentally a data aggregation, while asfreq is fundamentally a data selection.

Let's compare what the two return when we downsample(下采样) the S&P 500 closing price data.Figure 20.1 shows the result. Notice the difference(Figure 20.1): at each point, resample reports the average of the previous year, while asfreq reports the value at the end of the year.

For upsampling, resample and asfreq are largely equivalent, though resample has many more options available. In this case, the default for both methods is to leave the upsampled points empty; that is, filled with NA values. asfreq accepts a method argument to specify how values are imputed. Here, we will resample the

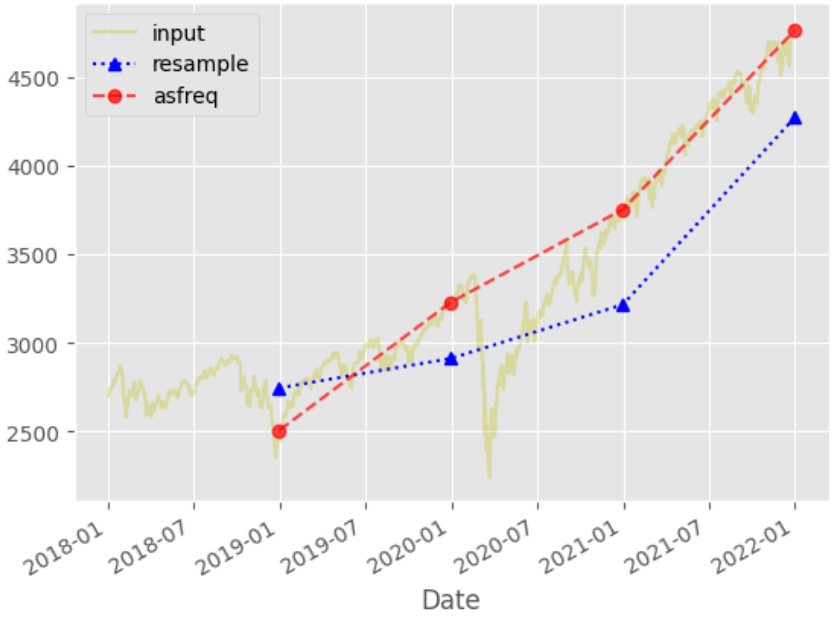


Figure 20.1: Resampling of SP500 closing price

business day data at a daily frequency (i.e., including weekends); Figure 20.2 shows the result. Because the S&P 500 data only exists for business days, the top panel has gaps representing NA values. The bottom panel shows the differences between two strategies for filling the gaps: forward filling and backward filling.

Time Shifts

Another common time series–specific operation is shifting of data in time. For this, Pandas provides the shift method, which can be used to shift data by a given number of entries. With time series data sampled at a regular frequency, this can give us a way to explore trends over time.(比如说要计算资产收益率，一年期的) shift

Rolling Windows

Calculating rolling statistics is a third type of time series–specific operation implemented by Pandas. This can be accomplished via the rolling attribute of Series and DataFrame objects, which returns a view similar to what we saw with the groupby operation (see Chapter 17). This rolling view makes available a number of aggregation operations by default.

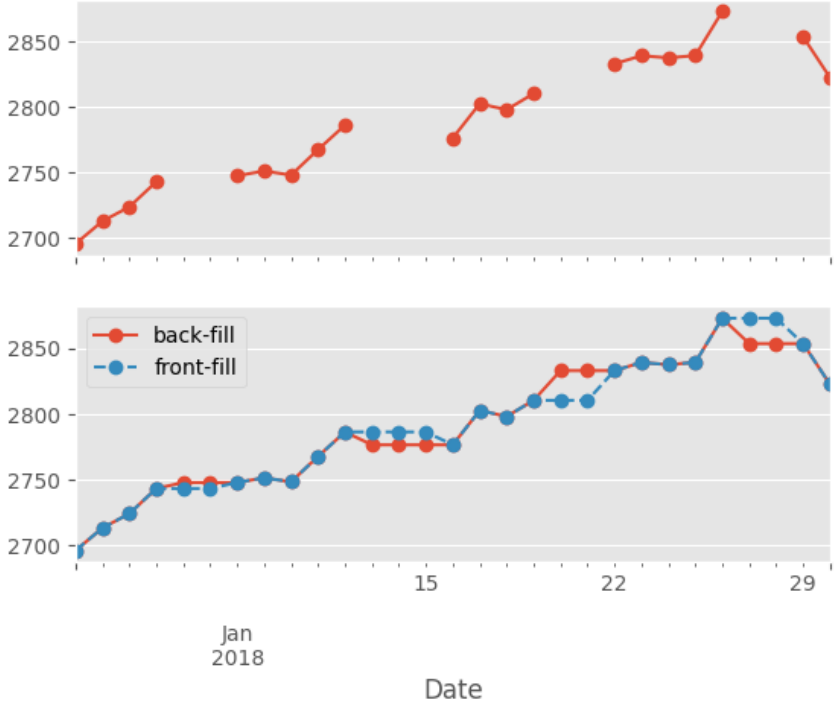


Figure 20.2: Comparison between forward-fill and back-fill interpolation

Chapter 21

High-Performance Pandas: eval and query

As we've already seen in previous chapters, the power of the PyData stack is built upon the ability of NumPy and Pandas to push basic operations into lower-level compiled code via an intuitive higher-level syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use cases, they often rely on the creation of temporary intermediate objects, which can cause undue overhead in computational time and memory use.

To address this, Pandas includes some methods that allow you to directly access C-speed operations without costly allocation of intermediate arrays: `eval` and `query`, which rely on the NumExpr package.

eval
query

21.1 Motivating query and eval: Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations. This is much faster than doing the addition via a Python loop or comprehension. But this abstraction can become less efficient when computing compound expressions(复合代数式). In other words, every intermediate step is explicitly allocated in memory. The NumExpr library gives you the ability to compute this type of compound expression element by element, without the need to allocate full intermediate arrays. The NumExpr documentation has more details, but for the time being it is sufficient to say that the library accepts a string giving the NumPy-style expression you'd like to compute.

NumExpr evaluates the expression in a way that avoids temporary arrays where possible, and thus can be much more efficient than NumPy, especially for long sequences of computations on large arrays. The Pandas eval and query tools that we will discuss here are conceptually similar, and are essentially Pandas-specific wrappers of NumExpr functionality.

21.2 pandas.eval for Efficient Operations

The eval function in Pandas uses string expressions to efficiently compute operations on DataFrame objects.

`pd.eval` supports a wide range of operations. Here's a summary of the operations pd.eval supports:

- **Arithmetic operators**: pd.eval supports all arithmetic operators.
- **Comparison operators**: pd.eval supports all comparison operators, including chained expressions.
- **Bitwise operators**: pd.eval supports the & and | bitwise operators. In addition, it supports the use of the literal and and or in Boolean expressions.
- **Object attributes and indices**: pd.eval supports access to object attributes via the obj.attr syntax and indexes via the obj[index] syntax.
- **Other operations** Other operations, such as function calls, conditional statements, loops, and other more involved constructs are currently **not** implemented in pd.eval. If you'd like to execute these more complicated types of expressions, you can use the NumExpr library itself.

21.3 DataFrame.eval for Column-Wise Operations

Just as Pandas has a top-level pd.eval function, DataFrame objects have an eval method that works in similar ways. The benefit of the eval method is that columns can be referred to by name.

Using pd.eval as in the previous section, we can compute expressions with the three columns like this:

```
df = pd.DataFrame(rng.random(size=(1000, 3)), columns=['A', 'B', 'C'])
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval('(df.A + df.B) / (df.C - 1)')
assert np.allclose(result1, result2)
```

The DataFrame.eval method allows much more succinct evaluation of expressions with the columns:

```
result3 = df.eval('(A + B) / (C - 1)')
assert np.allclose(result1, result3)
```

Notice here that we **treat column names as variables** within the evaluated expression, and the result is what we would wish.

Assignment in DataFrame.eval

In addition to the options just discussed, DataFrame.eval also allows assignment to any column.

In the same way, any existing column can be modified.

Local Variables in DataFrame.eval

The DataFrame.eval method supports an additional syntax that lets it work with local Python variables. 需要在变量前面添加@。

The @ character marks a variable name rather than a column name, and lets you efficiently evaluate expressions involving the two "namespaces": the namespace of columns, and the namespace of Python objects. Notice that this @ character is only supported by the DataFrame.eval method, not by the pandas.eval function, because the pandas.eval function only has access to the one (Python) namespace.

21.4 The DataFrame.query Method

query

The DataFrame has another method based on evaluated strings, called query.

21.5 Performance: When to Use These Functions

When considering whether to use eval and query, there are two considerations: computation time and memory use. Memory use is the most predictable aspect. As already mentioned, every compound expression involving NumPy arrays or Pandas Data Frames will result in implicit creation of temporary arrays.

If the size of the temporary DataFrames is significant compared to your available system memory (typically several gigabytes), then it's a good idea to use an eval or query expression. You can check the approximate size of your array in bytes.

On the performance side, eval can be faster even when you are not maxing out your system memory. The issue is how your temporary objects compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes); if they are much bigger, then eval can avoid some potentially slow movement of values between the different memory caches. In practice, I find that the difference in computation time between the traditional methods and the eval/query method is usually not significant—if anything, the traditional method is faster for smaller arrays! The benefit of eval/query is mainly in the saved memory, and the sometimes cleaner syntax they offer.

Part III

Visualization with Matplotlib

Chapter 22

General Matplotlib Tips

22.1 用不用show()？如何显示图形

如何显示你的图形，就取决于具体的开发环境。Matplotlib 的最佳实践与你使用的开发环境有关。简单来说，就是有三种开发环境，分别是脚本、IPython shell 和 IPython Notebook。

Plotting from a Script

One thing to be aware of: the plt.show command should be used only once per Python session, and is most often seen at the very end of the script. Multiple show commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Plotting from an IPython Shell

Plotting from a Jupyter Notebook

Plotting interactively within a Jupyter notebook can be done with the `%matplotlib` command, and works in a similar way to the IPython shell. You also have the option of embedding graphics directly in the notebook, with two possible options:

- `%matplotlib inline` will lead to static images of your plot embedded in the notebook.
- `%matplotlib notebook` will lead to interactive plots embedded within the note- book.

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig` command. In savefig, the file format is inferred from the extension of the given filename. In savefig, the file format is inferred from the extension of the given filename. Depending on what backends you have installed, many different file formats are available. The list of supported file types can be found for your system by using the following method of the figure canvas object:

savefig

```
fig.canvas.get_supported_filetypes()
```

Two Interfaces for the Price of One

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface.

MATLAB-style Interface

Matplotlib was originally conceived as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface.

It is important to recognize that this interface is stateful: it keeps track of the "current" figure and axes, which are where all plt commands are applied. You can get a reference to these using the plt.gcf (get current figure) and plt.gca (get current axes) routines.

While this stateful interface is fast and convenient for simple plots, it is easy to run into problems. For example, once the second panel is created, how can we go back and add something to the first?

Object-oriented interface

Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are methods of explicit Figure and Axes objects.

For simpler plots, the choice of which style to use is largely a matter of preference, but the object-oriented approach can become a necessity as plots become more complicated

Chapter 23

Simple Line Plots

In Matplotlib, the figure (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The axes (an instance of the class plt.Axes) is what we see above: a bounding box with ticks, grids, and labels, which will eventually contain the plot elements that make up our visualization. Throughout this part of the book, I'll commonly use the variable name fig to refer to a figure instance and ax to refer to an axes instance or group of axes instances.

Once we have created an axes, we can use the ax.plot method to plot some data.

Alternatively, we can use the PyLab interface and let the figure and axes be created for us in the background.

23.1 Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot function takes additional arguments that can be used to specify these. To adjust the color, you can use the `color` keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways. color

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='.75')         # grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # hex code (RRGGBB, 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0, .2, .3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse')  # HTML color names supported
```

If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.

Similarly, the line style can be adjusted using the `linestyle` keyword. linestyle

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
```

```
plt.plot(x, x + 3, linestyle='dotted')

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```

Though it may be less clear to someone reading your code, you can save some key-strokes by combining these linestyle and color codes into a single non-keyword argument to the plt.plot function:

```
plt.plot(x, x + 0, '-g')  # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r')  # dotted red
```

These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/blacK) color systems, commonly used for digital color graphics.

23.2 Adjusting the Plot: Axes Limits

Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust the limits is to use the plt.xlim and plt.ylim functions.

plt.xlim
plt.ylim

If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments.

plt.axis

A useful related method is `plt.axis` (note here the potential confusion between axes with an e, and axis with an i), which allows more qualitative specifications of axis limits.(自动收紧坐标轴，不留空白区域)

Or you can specify that you want an equal axis ratio, such that one unit in x is visually equivalent to one unit in y. Other axis options include 'on', 'off', 'square', 'image', and more. For more information on these, refer to the plt.axis docstring.

23.3 Labeling Plots

We'll briefly look at the labeling of plots: titles, axis labels, and simple legends. Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them. The position, size, and style of these labels can be adjusted using optional arguments to the functions, described in the docstrings.

When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend; it is done via the `plt.legend` method. Though there are several valid ways of using this, I find it easiest to specify the label of each line using the label keyword of the plot function.

23.4 Matplotlib Gotchas

While most plt functions translate directly to ax methods (plt.plot → ax.plot, plt.legend → ax.legend, etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

- `plt.xlabel` → `ax.set_xlabel`
- `plt.ylabel` → `ax.set_ylabel`
- `plt.xlim` → `ax.set_xlim`
- `plt.ylim` → `ax.set_ylim`
- `plt.title` → `ax.set_title`

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the `ax.set` method to set all these properties at once.

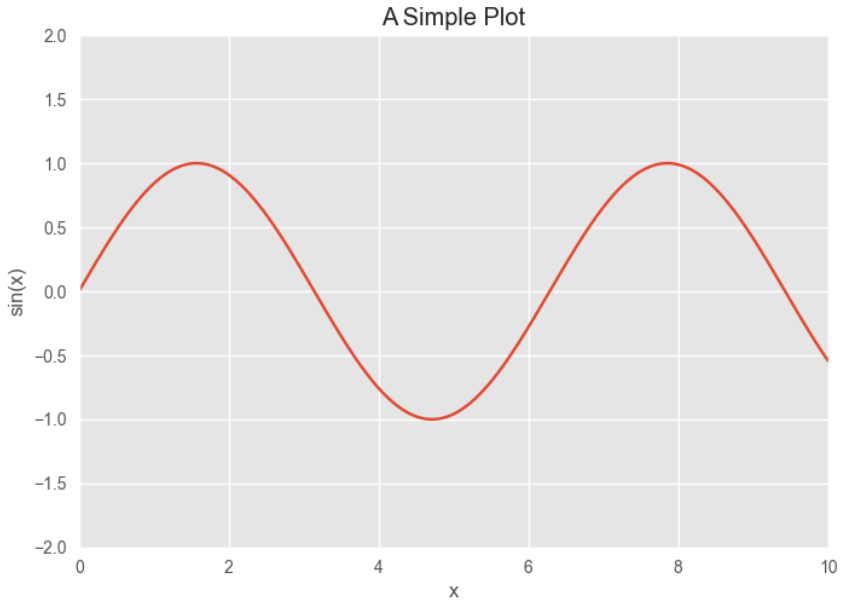


Figure 23.1: Example of using `ax.set` to set multiple properties at once

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2), xlabel='x', ylabel='sin(x)', title='A Simple Plot')
```

Chapter 24

Simple Scatter Plots

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

24.1 Scatter Plots with plt.plot

In the previous chapter we looked at using plt.plot/ax.plot to produce line plots. It turns out that this same function can produce scatter plots as well. The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-' or '–' to control the line style, the marker style has its own set of short string codes.

Most of the possibilities are fairly intuitive, and a number of the more common ones are demonstrated here (see Figure 24.1).

For even more possibilities, these character codes can be used together with line and color codes to plot points along with a line connecting them.

24.2 Scatter Plots with plt.scatter

plt.scatter

A second, more powerful method of creating scatter plots is the `plt.scatter` function, which can be used very similarly to the plt.plot function.

> The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

24.3 plot Versus scatter: A Note on Efficiency

> Aside from the different features available in plt.plot and plt.scatter, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, plt.plot can be noticeably more efficient than plt.scatter. The reason is

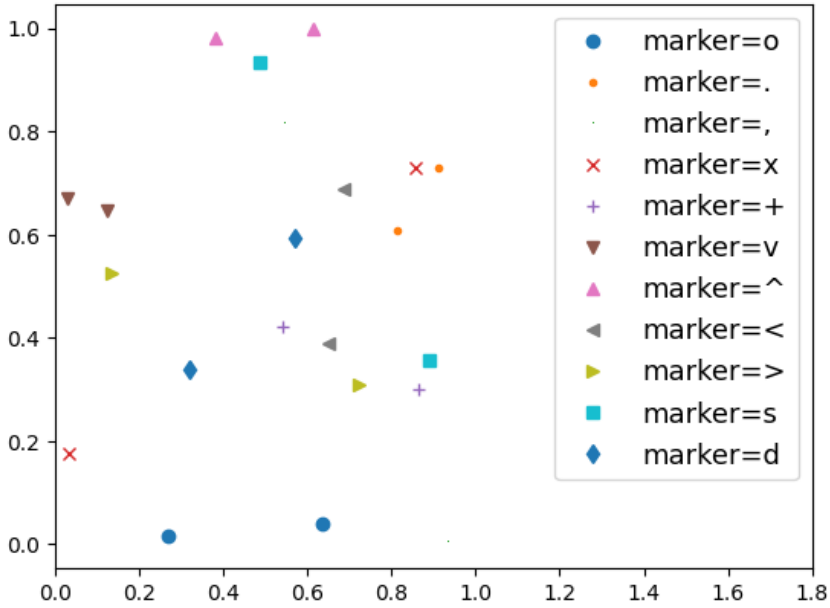


Figure 24.1: Demonstration of point numbers

that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. With plt.plot, on the other hand, the markers for each point are guaranteed to be identical, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, this difference can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

24.4 Visualizing Uncertainties

For any scientific measurement, accurate accounting of uncertainties is nearly as important, if not more so, as accurate reporting of the number itself.(对任何一种科学测量方法来说，准确地衡量数据误差都是无比重要的事情，甚至比数据本身还要重要。)

Basic Errorbars

One standard way to visualize uncertainties is using an errorbar. A basic errorbar can be created with a single Matplotlib function call, as shown in Figure 24.2.

Using these additional options you can easily customize the aesthetics of your errorbar plot. I often find it helpful, especially in crowded plots, to make the errorbars lighter than the points themselves.

Continuous Errors

In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like `plt.plot` and `plt.fill_between` for a useful result.

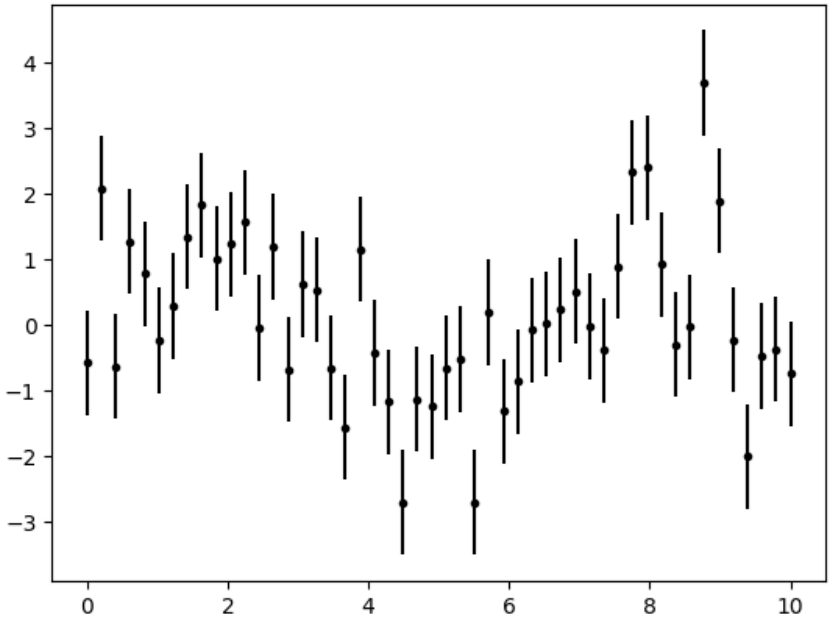


Figure 24.2: An errorbar example

Here we’ ll perform a simple Gaussian process regression which is a method of fitting a very flexible nonparametric function to data with a continuous measure of the uncertainty.

Figure 24.3, the resulting figure gives an intuitive view into what the Gaussian process regression algorithm is doing: in regions near a measured data point, the model is strongly constrained, and this is reflected in the small model uncertainties. In regions far from a measured data point, the model is not strongly constrained, and the model uncertainties increase.

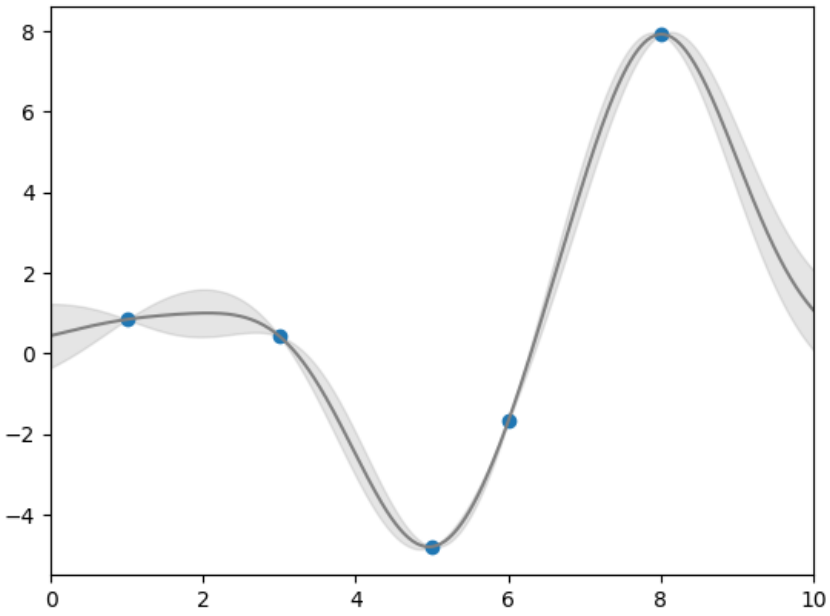


Figure 24.3: Representing continuous uncertainty with filled regions

Chapter 25

Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: plt.contour for contour plots, plt.contourf for filled contour plots, and plt.imshow for showing images.

plt.contour
plt.contourf
plt.imshow

25.1 Visualizing a Three-Dimensional Function

A contour plot can be created with the plt.contour function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays.

np.meshgrid

Notice that when a single color is used, negative values are represented by dashed lines and positive values by solid lines. Alternatively, the lines can be color-coded by specifying a colormap with the cmap argument.

cmap

Here we chose the RdGy (short for Red–Gray) colormap, which is a good choice for divergent data: (i.e., data with positive and negative variation around zero).

Our plot is looking nicer, but the spaces between the lines may be a bit distracting. We can change this by switching to a filled contour plot using the plt.contourf function, which uses largely the same syntax as plt.contour.

Additionally, we'll add a plt.colorbar command, which creates an additional axis with labeled color information for the plot (see Figure 25.1).

One potential issue with this plot is that it is a bit splotchy: the color steps are discrete rather than continuous, which is not always what is desired. This could be remedied by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level. A better way to generate a smooth representation is to use the plt.imshow function, which offers the interpolation argument to generate a smooth two-dimensional representation of the data.

There are a few potential gotchas with plt.imshow, however(以下这几条都很重要):

- It doesn't accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin, ymax] of the image on the plot.

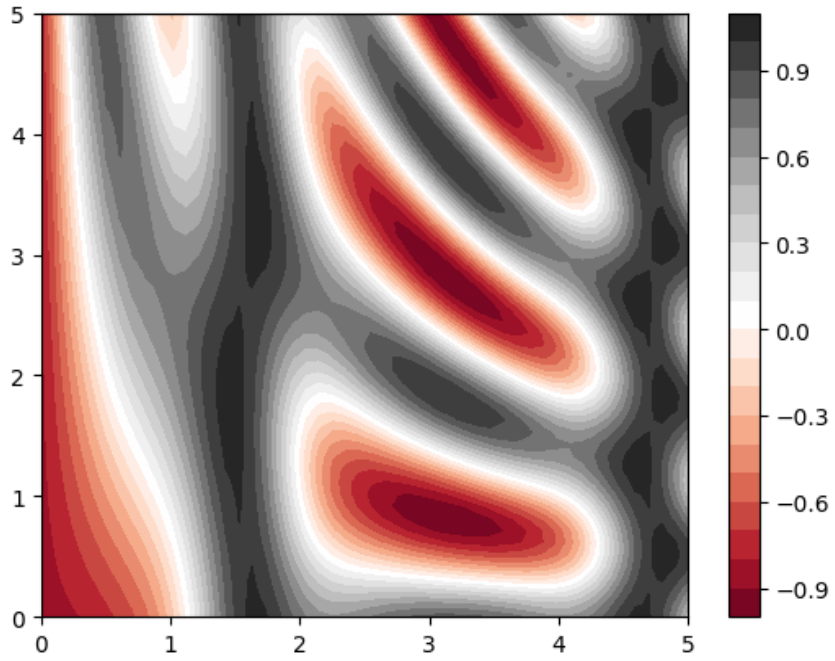


Figure 25.1: Visualizing three-dimensional data with filled contours

- By default it follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
- It will automatically adjust the axis aspect ratio to match the input data; this can be changed with the aspect argument.

Finally, it can sometimes be useful to combine contour plots and image plots. For example, here we'll use a partially transparent background image (with transparency set via the alpha parameter) and overplot contours with labels on the contours themselves, using the plt.clabel function(see Figure 25.2):

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', aspect='equal',
cmap='RdGy', alpha=.5, interpolation='gaussian')
plt.colorbar()
```

The combination of these three functions—plt.contour, plt.contourf, and plt.imshow—gives nearly limitless possibilities for displaying this sort of three- dimensional data within a two-dimensional plot.

25.2 Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset.

The plt.hist docstring has more information on other available customization options. I find this combination of `histtype='stepfilled'` along with some transparency alpha to be helpful when comparing histograms of several distributions(see Figure 25.3).

If you are interested in computing, but not displaying, the histogram (that is, counting the number of points in a given bin), you can use the `np.histogram` function. np.histogram

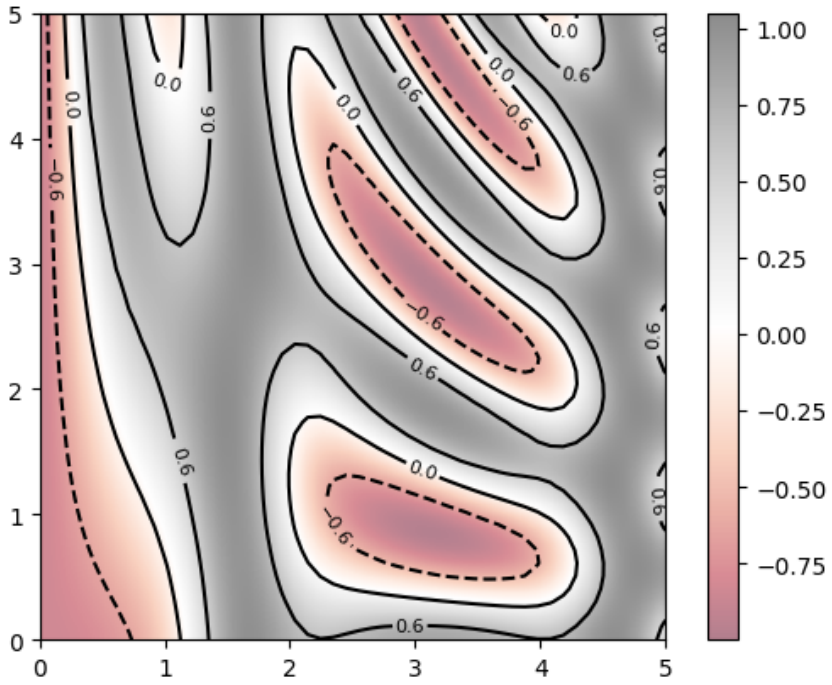


Figure 25.2: Labeled contours on top of an image

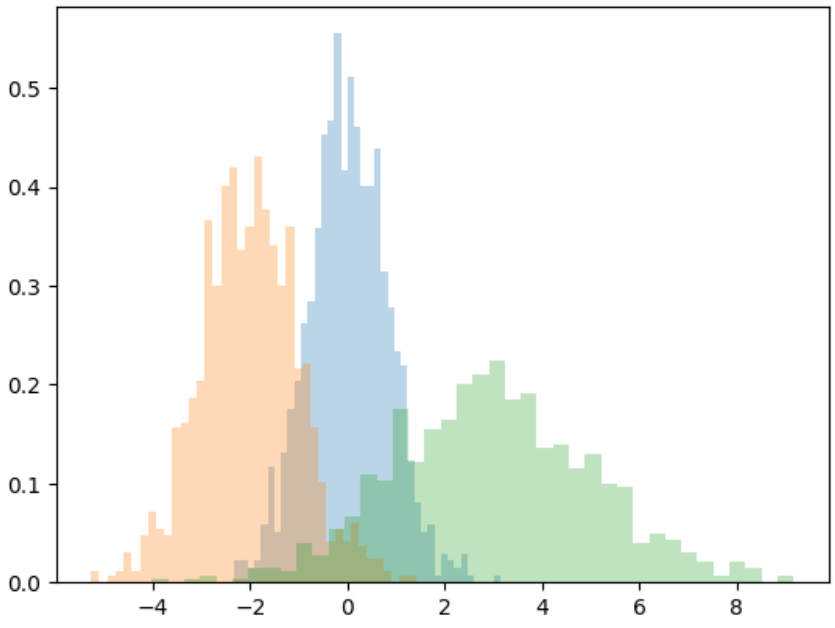


Figure 25.3: Overplotting multiple histograms

25.3 Two-Dimensional Histograms and Binnings

Just as we create histograms in one dimension by dividing the number line into bins, we can also create histograms in two dimensions by dividing points among two-dimensional bins.

plt.hist2d: Two-Dimensional Histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's `plt.hist2d` function. plt.hist2d

Just as plt.hist has a counterpart in np.histogram, plt.hist2d has a counterpart in np.histogram2d. np.histogram2d

plt.hexbin: Hexagonal Binnings

The two-dimensional histogram creates a tesselation of squares across the axes. Another natural shape for such a tesselation is the regular hexagon. For this purpose, Matplotlib provides the plt.hexbin routine, which represents a two-dimensional dataset binned within a grid of hexagons.

plt.hexbin has a number of additional options, including the ability to specify weights for each point and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Kernel Density Estimation

Another common method for estimating and representing densities in multiple dimensions is **kernel density estimation** (KDE). Now I'll simply mention that KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function. kernel density estimation

KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off).

Chapter 26

Customizing Plot Legends

Plot legends give meaning to a visualization, assigning meaning to the various plot elements.

We can specify the location and turn on the frame. We can use the ncol command to specify the number of columns in the legend. And we can use a rounded box (fancybox) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text(see Figure 26.1).

26.1 Choosing Elements for the Legend

As we have already seen, by default the legend includes all labeled elements from the plot. If this is not what is desired, we can fine-tune which elements and labels appear in the legend by using the objects returned by plot commands. plt.plot is able to create multiple lines at once, and returns a list of created line instances. Passing any of these to plt.legend will tell it which to identify, along with the labels we'd like to specify.

Notice that the legend ignores all elements without a label attribute set.

26.2 Legend for Size of Points

Sometimes the legend defaults are not sufficient for the given visualization. For example, perhaps you're using the size of points to mark certain features of the data, and want to create a legend reflecting this. (可视化气泡图，需要将图里的大小也同步调整)

The legend will always reference some object that is on the plot, so if we'd like to display a particular shape we need to plot it. In this case, the objects we want (gray circles) are not on the plot, so we fake them by plotting empty lists. Recall that the legend only lists plot elements that have a label specified.

26.3 Multiple Legends

Sometimes when designing a plot you'd like to add multiple legends to the same axes. Unfortunately, Matplotlib does not make this easy: via the standard legend interface, it is only possible to create a single legend for the entire plot. If you try to create a second legend using plt.legend or ax.legend, it will simply override the first one. We can work around this by creating a new legend artist from scratch (`Artist` is the base class Matplotlib uses for visual attributes), and then using the lower-level `ax.add_artist` method to manually add the second artist to the plot.

Artist

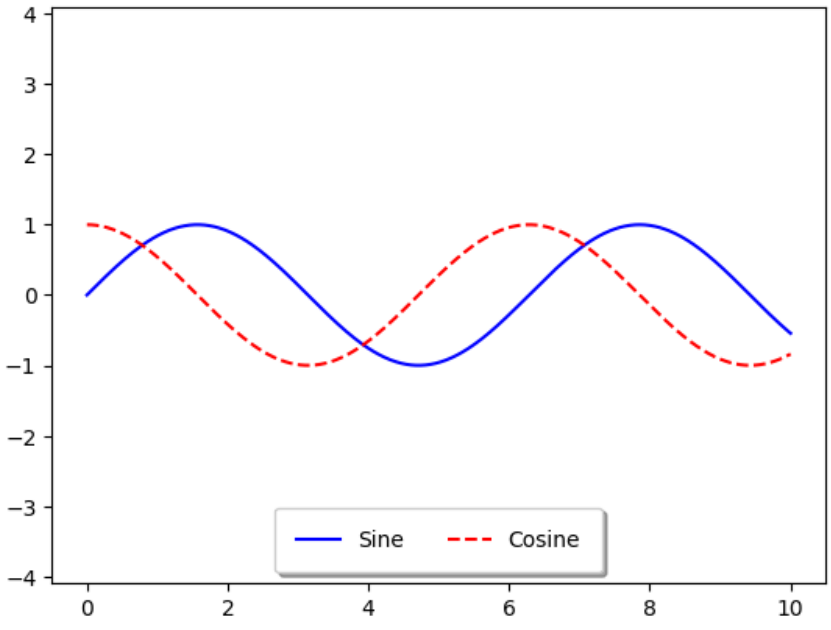


Figure 26.1: A fancybox plot legend

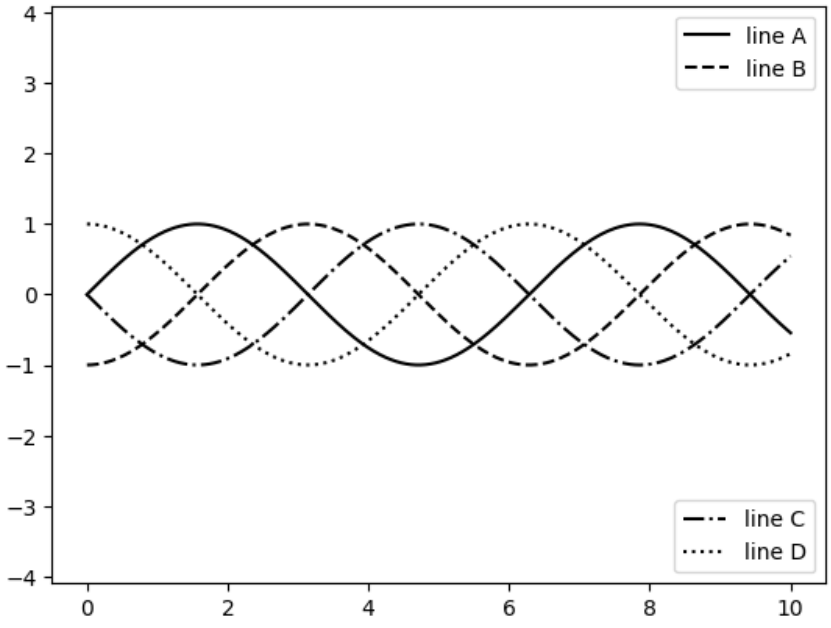


Figure 26.2: A split plot legend

```
fig, ax = plt.subplots()
lines = []
styles = ['-', '--', '-.', ':']
x = np.linspace(0, 10, 1000)
for i in range(4):
lines += ax.plot(x, np.sin(x - i * np.pi / 2),
styles[i], color='black')
ax.legend(lines[:2], ['line A', 'line B'], loc='upper right')

ax.axis('equal')

from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'], loc='lower right')
ax.add_artist(leg)
```

Chapter 27

Customizing Colorbars

Plot legends identify discrete labels of discrete points. For continuous labels based on the color of points, lines, or regions, a labeled colorbar can be a great tool. In Matplotlib, a colorbar is drawn as a separate axes that can provide a key for the meaning of colors in a plot.

As we have seen several times already, the simplest colorbar can be created with the plt.colorbar function. plt.colorbar

27.1 Customizing Colorbars

The colormap can be specified using the cmap argument to the plotting function that is creating the visualization. The names of available colormaps are in the `plt.cm` namespace.

Choosing the Colormap

Broadly, you should be aware of three different categories of colormaps:

- **Sequential colormaps**(顺序配色方案): These are made up of one continuous sequence of colors (e.g., binary or viridis).
- **Divergent colormaps**(互逆配色方法): These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., RdBu or PuOr).
- **Qualitative colormaps**(定性配色方法): These mix colors with no particular sequence (e.g., rainbow or jet).

The jet colormap, which was the default in Matplotlib prior to version 2.0, is an example of a qualitative colormap. Its status as the default was quite unfortunate, because qualitative maps are often a poor choice for representing quantitative data. Among the problems is the fact that qualitative maps usually do not display any uniform progression in brightness as the scale increases. We can see this by converting the jet colorbar into black and white (see Figure 27.1)

Notice the bright stripes in the grayscale image. Even in full color, this uneven brightness means that the eye will be drawn to certain portions of the color range, which will potentially emphasize unimportant parts of the dataset. It’ s better to use a colormap such as viridis (the default as of Matplotlib 2.0), which is specifically constructed to have an even brightness variation across the range; thus, it not only plays well with our color perception, but also will translate well to grayscale printing.

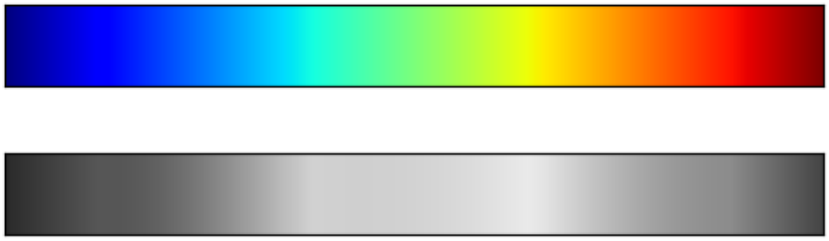


Figure 27.1: The jet colormap and its uneven luminance scale

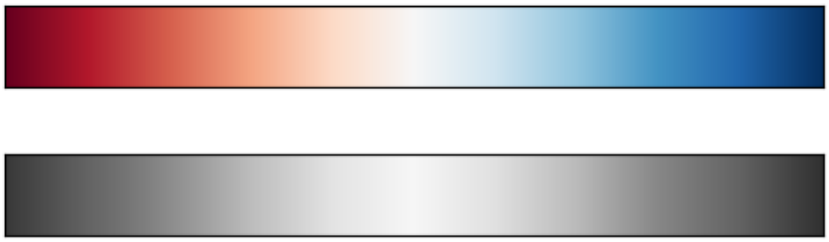


Figure 27.2: The RdBu colormap and its luminance

For other situations, such as showing positive and negative deviations from some mean, dual-color colorbars such as RdBu (Red–Blue) are helpful. However, as you can see in Figure 27.2, it's important to note that the positive/negative information will be lost upon translation to grayscale!

更多关于绘图的可以参考，Ten Simple Rules for Better Figures.

Color Limits and Extensions

Matplotlib allows for a large range of colorbar customization. The colorbar itself is simply an instance of plt.Axes, so all of the axes and tick formatting tricks we've seen so far are applicable. The colorbar has some interesting flexibility: for example, we can narrow the color limits and indicate the out-of-bounds values with a triangular arrow at the top and bottom by setting the extend property.

Discrete Colorbars

Colormaps are by default continuous, but sometimes you'd like to represent discrete values. The easiest way to do this is to use the `plt.cm.get_cmap`[1] function and pass the name of a suitable colormap along with the number of desired bins.

27.2 Example: Handwritten Digits

Because each digit is defined by the hue of its 64 pixels, we can consider each digit to be a point lying in 64-dimensional space: each dimension represents the brightness of one pixel. Visualizing such high-dimensional data can be difficult, but one way to approach this task is to use a dimensionality reduction

[1] MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use "matplotlib.colormaps[name]" or "matplotlib.colormaps.get_cmap(obj)" instead.

technique such as manifold learning to reduce the dimensionality of the data while maintaining the relationships of interest. Dimensionality reduction is an example of unsupervised machine learning, and we will discuss it in more detail in Chapter 34.

Chapter 28

Multiple Subplots

Sometimes it is helpful to compare different views of data side by side. To this end, Matplotlib has the concept of subplots: groups of smaller axes that can exist together within a single figure. These subplots might be insets(画中画), grids of plots, or other more complicated layouts. In this chapter we'll explore four routines for creating subplots in Matplotlib.

28.1 plt.axes: Subplots by Hand

plt.axes

The most basic method of creating an axes is to use the plt.axes function. As we've seen previously, by default this creates a standard axes object that fills the entire figure. plt.axes also takes an optional argument that is a list of four numbers in the figure coordinate system (`[left, bottom, width, height]`), which ranges from 0 at the bottom left of the figure to 1 at the top right of the figure.

For example, we might create an inset axes at the top-right corner of another axes by setting the x and y position to 0.65 (that is, starting at 65% of the width and 65the height of the figure) and the x and y extents to 0.2 (that is, the size of the axes is 20% of the width and 20% of the height of the figure)(see Figure 28.1a).

fig.add_axes

The equivalent of this command within the object-oriented interface is `fig.add_axes`.

28.2 plt.subplot: Simple Grids of Subplots

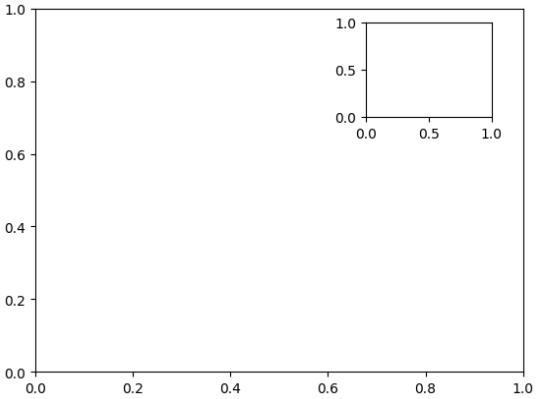
Aligned columns or rows of subplots are a common enough need that Matplotlib has several convenience

plt.subplot

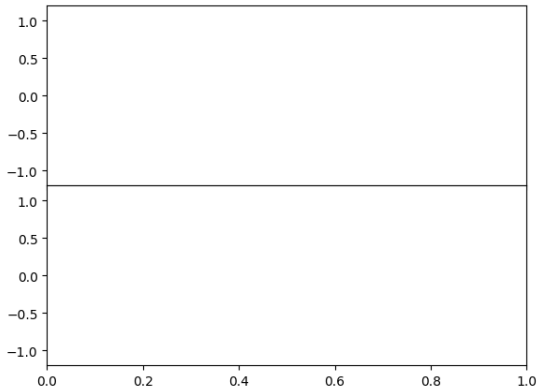
routines that make them easy to create. The lowest level of these is `plt.subplot`, which creates a single subplot within a grid. As you can see, this command takes three integer arguments—the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right.(see Figure 28.2a)

The command `plt.subplots_adjust` can be used to adjust the spacing between these plots. The following code uses the equivalent object-oriented command, `fig.add_subplot`; Figure 28.2b shows the result.

Here we've used the hspace and wspace arguments of `plt.subplots_adjust`, which specify the spacing along the height and width of the figure, in units of the subplot size (in this case, the space is 40% of the subplot width and height).

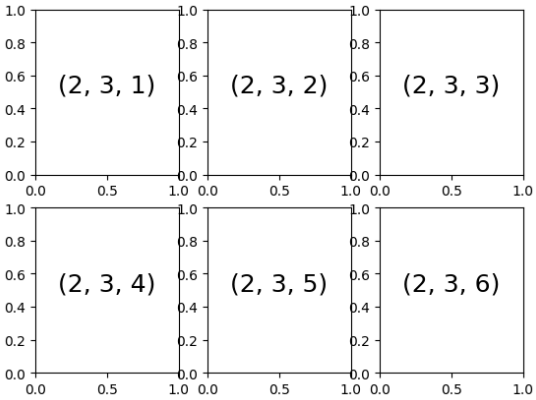


(a) Example of an inset axes

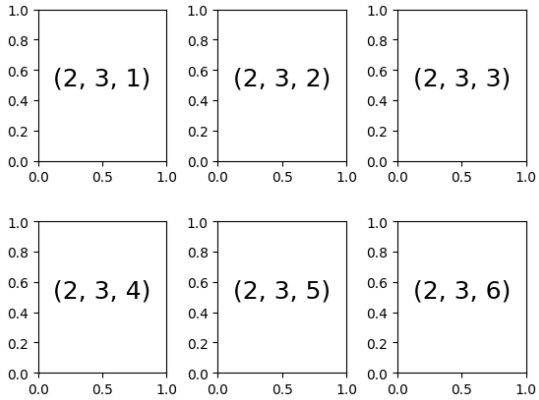


(b) Vertically stacked axes example

Figure 28.1: Subplots by Hand



(a) A plt.subplot example



(b) plt.subplot with adjusted margins

Figure 28.2: plt.subplot: Simple Grids of Subplots

28.3 plt.subplots: The Whole Grid in One Go

plt.subplots

The approach just described quickly becomes tedious when creating a large grid of subplots, especially if you'd like to hide the x- and y-axis labels on the inner plots. For this purpose, `plt.subplots` is the easier tool to use (note the s at the end of subplots). Rather than creating a single subplot, this function creates a full grid of subplots in a single line, returning them in a NumPy array. The arguments are the number of rows and number of columns, along with optional keywords sharex and sharey, which allow you to specify the relationships between different axes.

In comparison to plt.subplot, plt.subplots is more consistent with Python's conventional zero-based indexing, whereas plt.subplot uses MATLAB-style one-based indexing.

28.4 plt.GridSpec: More Complicated Arrangements

plt.GridSpec

To go beyond a regular grid to subplots that span multiple rows and columns, `plt.GridSpec` is the best tool. plt.GridSpec does not create a plot by itself; it is rather a convenient interface that is recognized by the plt.subplot command.

```
grid = plt.GridSpec(2, 3, wspace=.4, hspace=.3)
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2])
plt.show()
```

This type of flexible grid alignment has a wide range of uses. I most often use it when creating multiaxes histogram plots.

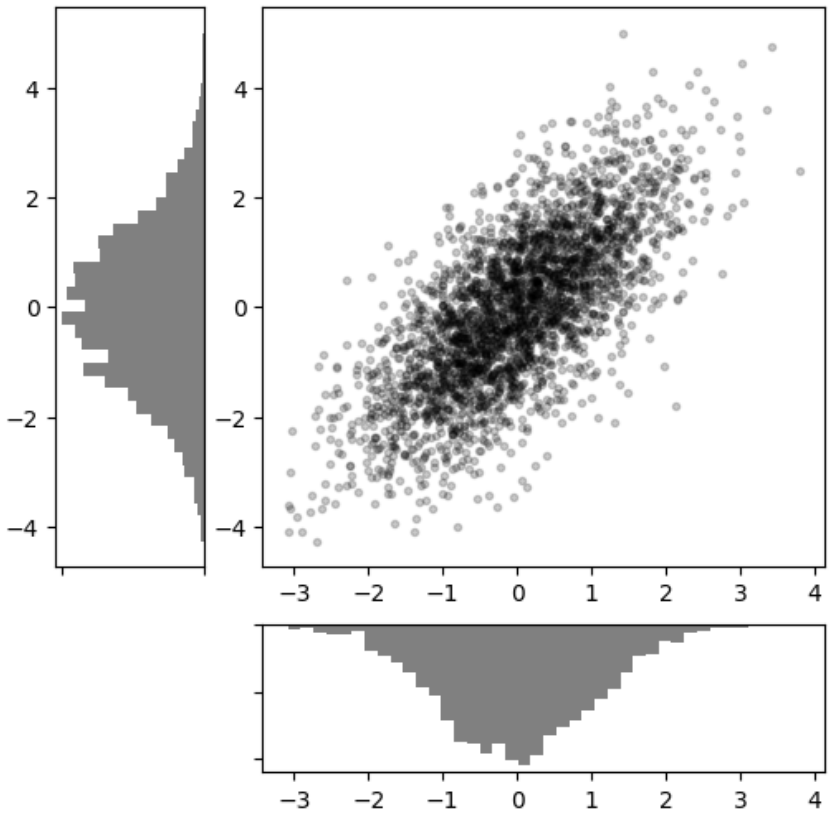


Figure 28.3: Visualizing multidimensional distributions with plt.GridSpec

Chapter 29

Text and Annotation

Creating a good visualization involves guiding the reader so that the figure tells a story. In some cases, this story can be told in an entirely visual manner, without the need for added text, but in others, small textual cues and labels are necessary. Perhaps the most basic types of annotations you will use are axes labels and titles, but the options go beyond this.

When we're visualizing data, it is often useful to annotate certain features of the plot to draw the reader's attention. This can be done manually with the `plt.text`/ `ax.text` functions, which will place text at a particular *x*/*y* value.

plt.text
ax.text

The ax.text method takes an *x* position, a *y* position, a string, and then optional keywords specifying the color, size, style, alignment, and other properties of the text. Here we used `ha='right'` and `ha='center'`, where `ha` is short for horizontal alignment.

29.1 Transforms and Text Position

In the previous example, we anchored our text annotations to data locations. Sometimes it's preferable to anchor the text to a fixed position on the axes or figure, independent of the data. In Matplotlib, this is done by modifying the transform.

Matplotlib makes use of a few different coordinate systems: a data point at $(x, y) = (1, 1)$ corresponds to a certain location on the axes or figure, which in turn corresponds to a particular pixel on the screen. Mathematically, transforming between such coordinate systems is relatively straightforward, and Matplotlib has a well-developed set of tools that it uses internally to perform these transforms.

A typical user rarely needs to worry about the details of the transforms, but it is helpful knowledge to have when considering the placement of text on a figure. There are three predefined transforms that can be useful in this situation:

- **ax.transData**: Transform associated with data coordinates
- **ax.transAxes**: Transform associated with the axes (in units of axes dimensions)
- **fig.transFigure**: Transform associated with the figure (in units of figure dimensions)

The transData coordinates give the usual data coordinates associated with the x- and y-axis labels. The transAxes coordinates give the location from the bottom-left corner of the axes (the white box), as a fraction

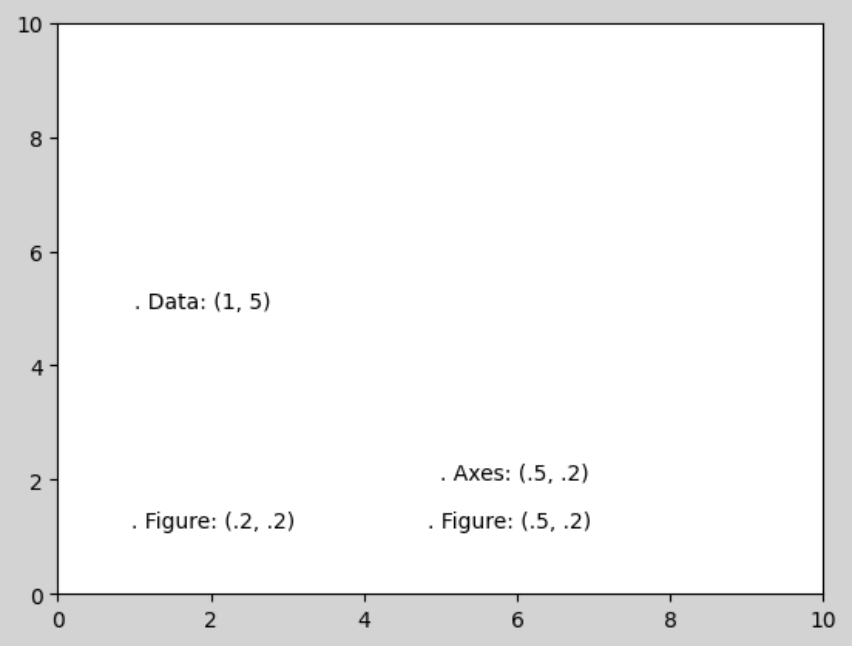


Figure 29.1: Comparing Matplotlib coordinate systems

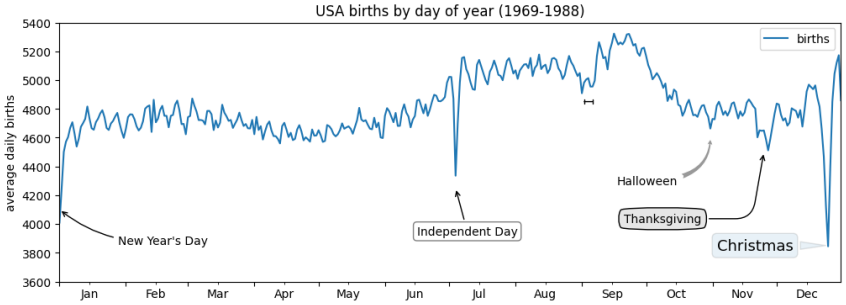


Figure 29.2: Annotated average birth rates by day

of the total axes size. The transFig ure coordinates are similar, but specify the position from the bottom-left corner of the figure (the gray box) as a fraction of the total figure size.

Notice now that if we change the axes limits, it is only the transData coordinates that will be affected, while the others remain stationary.

29.2 Arrows and Annotation

Along with tickmarks and text, another useful annotation mark is the simple arrow.

I'd suggest using the `plt.annotate` function, which creates some text and an arrow and allows the plt.annotate arrows to be very flexibly specified.

More discussion and examples of available arrow and annotation styles can be found in the Matplotlib Annotations tutorial.

Chapter 30

Customizing Ticks

Matplotlib's default tick locators and formatters are designed to be generally sufficient in many common situations, but are in no way optimal for every plot.

Before we go into examples, however, let's talk a bit more about the object hierarchy of Matplotlib plots. Matplotlib aims to have a Python object representing everything that appears on the plot: for example, recall that the Figure is the bounding box within which plot elements appear. Each Matplotlib object can also act as a container of subobjects: for example, each Figure can contain one or more Axes objects, each of which in turn contains other objects representing plot contents.

The tickmarks are no exception. Each axes has attributes xaxis and yaxis, which in turn have attributes that contain all the properties of the lines, ticks, and labels that make up the axes.

30.1 Major and Minor Ticks

Within each axes, there is the concept of a major tickmark, and a minor tickmark. As the names imply, major ticks are usually bigger or more pronounced, while minor ticks are usually smaller. By default, Matplotlib rarely makes use of minor ticks, but one place you can see them is within logarithmic plots.

formatter
locator

These tick properties—locations and labels, that is—can be customized by setting the `formatter` and `locator` objects of each axis.

```
ax.xaxis.get_major_locator()
ax.xaxis.get_minor_locator()
ax.xaxis.get_major_formatter()
ax.xaxis.get_minor_formatter()
```

30.2 Hiding Ticks or Labels

Perhaps the most common tick/label formatting operation is the act of hiding ticks or labels. This can be done using `plt.NullLocator` and `plt.NullFormatter`. Having no ticks at all can be useful in many situations—for example, when you want to show a grid of images.

30.3 Reducing or Increasing the Number of Ticks

One common problem with the default settings is that smaller subplots can end up with crowded labels. Particularly for the x-axis ticks, the numbers nearly overlap, making them quite difficult to decipher. One

Table 30.1: Matplotlib locator options

Locator class	Description
NullLocator	No ticks
FixedLocator	Tick locations are fixed
IndexLocator	Locator for index plots (e.g., where x = range(len(y)))
LinearLocator	Evenly spaced ticks from min to max
LogLocator	Logarithmically spaced ticks from min to max
MultipleLocator	Ticks and range are a multiple of base
MaxNLocator	Finds up to a max number of ticks at nice locations
AutoLocator	(Default) MaxNLocator with simple defaults
AutoMinorLocator	Locator for minor ticks

Table 30.2: Matplotlib formatter options

Formatter class	Description
NullFormatter	No labels on the ticks
IndexFormatter	Set the strings from a list of labels
FixedFormatter	Set the strings manually for the labels
FuncFormatter	User-defined function sets the labels
FormatStrFormatter	Use a format string for each value
ScalarFormatter	Default formatter for scalar values
LogFormatter	Default formatter for log axes

way to adjust this is with `plt.MaxNLocator`, which allows us to specify the maximum number of ticks that will be displayed.

30.4 Fancy Tick Formats

Matplotlib's default tick formatting can leave a lot to be desired: it works well as a broad default, but sometimes you'd like to do something different. 比如说需要绘制 π 作为刻度，而不是整数或者说是小数表示的 π。

We can do this by setting a `MultipleLocator`, which locates ticks at a multiple of the number we provide. MultipleLocato

we can change the tick formatter. There's no built-in formatter for what we want to do, so we'll instead use `plt.FuncFormatter`, which accepts a user-defined function giving fine-grained control over the tick outputs

30.5 Summary of Formatters and Locators

We've seen a couple of the available formatters and locators; I'll conclude this chapter by listing all of the built-in locator options (Table 30.1) and formatter options (Table 30.2).

Chapter 31

Customizing Matplotlib: Configurations and Stylesheets

While many of the topics covered in previous chapters involve adjusting the style of plot elements one by one, Matplotlib also offers mechanisms to adjust the overall style of a chart all at once. In this chapter we'll walk through some of Matplotlib's runtime configuration (**rc**) options, and take a look at the stylesheets feature, which contains some nice sets of default configurations.

31.1 Plot Customization by Hand

31.2 Changing the Defaults: rcParams

Each time Matplotlib loads, it defines a runtime configuration containing the default styles for every plot element you create. This configuration can be adjusted at any time using the `plt.rc` convenience routine.

plt.rc

Optionally, these settings can be saved in a **.matplotlibrc** file.

31.3 Stylesheets

A newer mechanism for adjusting overall chart styles is via Matplotlib's style module, which includes a number of default stylesheets, as well as the ability to create and package your own styles. These stylesheets are formatted similarly to the **.matplotlibrc** files mentioned earlier, but must be named with a **.mplstyle** extension. Even if you don't go as far as creating your own style, you may find what you're looking for in the built-in stylesheets. `plt.style.available` contains a list of the available styles.

The standard way to switch to a stylesheet is to call `style.use`.

But keep in mind that this will change the style for the rest of the Python session! Alternatively, you can use the style context manager, which sets a style temporarily.

Default Style

Matplotlib's default style was updated in the version 2.0 release.

FiveThirtyEight Style

The fivethirtyeight style mimics the graphics found on the popular FiveThirtyEight website. It is typified by bold colors, thick lines, and transparent axes.

ggplot Style

The ggplot package in the R language is a popular visualization tool among data scientists. Matplotlib's ggplot style mimics the default styles from that package.

Bayesian Methods for Hackers Style

There is a neat short online book called Probabilistic Programming and Bayesian Methods for Hackers by Cameron Davidson-Pilon that features figures created with Matplotlib, and uses a nice set of rc parameters to create a consistent and visually appealing style throughout the book. This style is reproduced in the bmh stylesheet.

Dark Background Style

For figures used within presentations, it is often useful to have a dark rather than light background. The `dark_background` style provides this.

Grayscale Style

You might find yourself preparing figures for a print publication that does not accept color figures. For this, the grayscale style can be useful.

Seaborn Style

Matplotlib also has several stylesheets inspired by the Seaborn library. I've found these settings to be very nice, and tend to use them as defaults in my own data exploration

Chapter 32

Three-Dimensional Plotting in Matplotlib

Three- dimensional plots are enabled by importing the mplot3d toolkit, included with the main Matplotlib installation.

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword `projection='3d'` to any of the normal axes creation routines.

Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically, in the notebook; recall that to use interactive figures, you can use `%matplotlib notebook` rather than `%matplotlib inline` when running this code.

32.1 Three-Dimensional Points and Lines

The most basic three-dimensional plot is a line or collection of scatter plots created from sets of (x, y, z) triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the `ax.plot3D` and `ax.scatter3D` functions. The call signature for these is nearly identical to that of their two-dimensional counterparts.

32.2 Three-Dimensional Contour Plots

mplot3d contains tools to create three-dimensional relief(晕渲) plots using the same inputs. Like `ax.contour`, `ax.contour3D` requires all the input data to be in the form of two-dimensional regular grids, with the z data evaluated at each point.

Sometimes the default viewing angle is not optimal, in which case we can use the `view_init` method to set the elevation(角度) and azimuthal angles(方向角).

32.3 Wireframes(线框图) and Surface Plots

Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots. These take a grid of values and project it onto the specified three- dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize.

A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized.

Though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear(直角坐标系).

32.4 Surface Triangulations

For some applications, the evenly sampled grids required by the preceding routines are too restrictive. In these situations, triangulation-based plots can come in handy. What if rather than an even draw from a Cartesian or a polar grid, we instead have a set of random draws?

The function that will help us in this case is `ax.plot_trisurf`, which creates a surface by first finding a set of triangles formed between adjacent points.

Chapter 33

Visualization with Seaborn

33.1 Exploring Seaborn Plots

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Let's take a look at a few of the datasets and plot types available in Seaborn. Note that all of the following could be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood), but the Seaborn API is much more convenient.

Histograms, KDE, and Densities

Often in statistical data visualization, all you want is to plot histograms and joint distributions of variables. We have seen that this is relatively straightforward in Matplotlib.

Rather than just providing a histogram as a visual output, we can get a smooth estimate of the distribution using kernel density estimation, which Seaborn does with `sns.kdeplot`.

If we pass *x* and *y* columns to kdeplot, we instead get a two-dimensional visualization of the joint density.

We can see the joint distribution and the marginal distributions together using `sns.jointplot`.

Pair Plots

When you generalize joint plots to datasets of larger dimensions, you end up with **pair plots**(矩阵图). These are very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

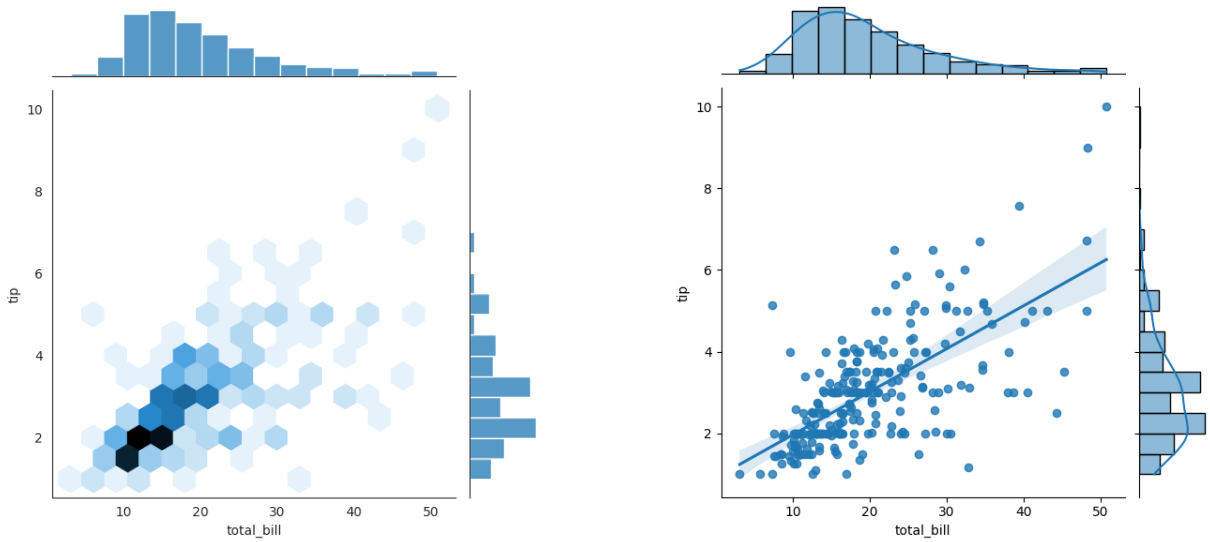
Visualizing the multidimensional relationships among the samples is as easy as calling `sns.pairplot`.

Faceted Histograms

Sometimes the best way to view data is via histograms of subsets. Seaborn's `FacetGrid` makes this simple.

33.2 Categorical Plots

Categorical plots can be useful for this kind of visualization as well. These allow you to view the distribution of a parameter within bins defined by any other parameter.



(a) A joint distribution plot

(b) A joint distribution plot with a regression fit

Figure 33.1: Joint Distributions

Joint Distributions

Similar to the pair plot we saw earlier, we can use sns.jointplot to show the joint distribution between different datasets, along with the associated marginal distributions.(see Figure 33.1a)

The joint plot can even do some automatic kernel density estimation and regression, as shown in Figure 33.1b.

Bar Plots

Time series can be plotted using `sns.catplot`. We can learn more by looking at the factor of data of each of these categories.

For more information on plotting with Seaborn, see the Seaborn documentation, and particularly the example gallery.

33.3 用Basemap可视化地理数据

地理数据可视化是数据科学中一种十分常见的可视化类型。Matplotlib 做此类可视化的主要工具是 Basemap 工具箱。

33.3.1 地图投影

当你想使用地图时，首先要做的就是确定地图的投影类型。你可能已经知道，像地球这样的球体，可以通过球面透视法将三维球面投影成一个二维平面，不会造成变形，也不会破坏其连续性。

圆柱投影

圆柱投影（cylindrical projection）是最简单的地图投影类型，纬度线与经度线分别映射成水平线与竖直线。采用这种投影类型的话，赤道区域的显示效果非常好，但是南北极附近的区域就会严重变形。由于纬度线的间距会因圆柱投影的不同而不同，所以就有了不同的投影属性和南北极附近不

同的变形程度。等距圆柱投影(cyl)，不同纬度在子午线方向的间距保持不变。另外两种圆柱投影是墨卡托（Mercator， projection='merc'）投影和圆柱等积（cylindrical equal-area，projection='cea'）投影。

伪圆柱投影

伪圆柱投影（pseudo-cylindrical projection）的经线不再必须是竖直的，这样可以使南北极附近的区域更加真实。摩尔威德（Mollweide，projection='moll'）投影就是这类投影的典型代表，它所有的经线都是椭圆弧线，如图 4-105 所示。这么做是为了保留地图原貌——虽然南北极附近的区域还有一些变形，但是通过一些区域小图可以反映真实情况。其他伪圆柱投影类型有正弦（sinusoidal，projection='sinu'）投影和罗宾森（Robinson， projection='robin'）投影。

透视投影

透视投影（perspective projection）是从某一个透视点对地球进行透视获得的投影，就好像你站在太空中某一点给地球照相一样（通过技术处理，有些投影类型的透视点可以放在地球上）。一个典型示例是正射（orthographic，projection='ortho'）投影，从无限远处观察地球的一侧。因此，这种投影一次只能显示半个地球。其他的透视投影类型还有球心（gnomonic，projection='gnom'）投影和球极平面（stereographic，projection='stere'）投影。这些投影经常用于显示地图的较小面积区域。

圆锥投影

圆锥投影（conic projection）是先将地图投影成一个圆锥体，然后再将其展开。这样做虽然可以获得非常好的局部效果，但是远离圆锥顶点的区域可能会严重变形。一个典型示例就是兰勃特等角圆锥投影（Lambert conformal conic projection，projection='lcc'），也就是我们之前见到的北美洲地图。这种方法将地图投影成一个由两条标准纬线（用 Basemap 里的 lat_1 与 lat_2 参数设置）构成的圆锥，这两条纬线距离是经过精心挑选的，在两条标准纬线之内比例尺逐渐减小，在两线之外的比例尺逐渐增大。其他常用的圆锥投影还有等距圆锥（equidistant conic，projection='eqdc'）投影和阿尔伯斯等积圆锥（Albers equalarea，projection='aea'）投影。圆锥投影和透视投影一样，适合显示较小与中等区域的地图。

Part IV

Machine Learning

Chapter 34

What Is Machine Learning?

34.1 Qualitative Examples of Machine Learning Applications

Dimensionality Reduction: Inferring Structure of Unlabeled Data

34.2 Summary

In short, we saw the following:

- Supervised learning: Models that can predict labels based on labeled training data
  - Classification: Models that predict labels as two or more discrete categories
  - Regression: Models that predict continuous labels
- Unsupervised learning: Models that identify structure in unlabeled data
  - Clustering: Models that detect and identify distinct groups in the data
  - Dimensionality reduction: Models that detect and identify lower-dimensional structure in higher-dimensional data

Chapter 35

Introducing Scikit-Learn

35.1 Data Representation in Scikit-Learn

Consider the Iris dataset, each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as **samples**, and the number of rows as `n_samples`.

Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as **features**, and the number of columns as `n_features`.

The Features Matrix

The table layout makes clear that the information can be thought of as a two- dimensional numerical array or matrix, which we will call the **features matrix**. By convention, this matrix is often stored in a variable named X. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`, and is most often contained in a NumPy array or a Pandas DataFrame, though some Scikit-Learn models also accept SciPy sparse matrices.

The Target Array

In addition to the feature matrix X, we also generally work with a label or target array, which by convention we will usually call y. The target array is usually one- dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas Series. The target array may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily be working with the common case of a one-dimensional target array.

35.2 The Estimator API

Basics of the API

Most commonly, the steps in using the Scikit-Learn Estimator API are as follows:

1. Choose a class of model by importing the appropriate estimator class from Scikit- Learn.
2. Choose model hyperparameters by instantiating this class with desired values.

3. Arrange data into a features matrix and target vector.
4. Fit the model to your data by calling the fit method of the model instance.
5. Apply the model to new data:
    - For supervised learning, often we predict labels for unknown data using the `predict` method.
    - For unsupervised learning, we often transform or infer properties of the data using the `transform` or `predict` method.

Supervised Learning Example: Simple Linear Regression

Choose model hyperparameters

An important point is that a class of model is not the same as an instance of a model.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., y-intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

These are examples of the important choices that must be made once the model class is selected. These choices are often represented as hyperparameters, or parameters that must be set before the model is fit to data.

Supervised Learning Example: Iris Classification

Gaussian naive Bayes

We will use a simple generative model known as **Gaussian naive Bayes**, which proceeds by assuming each class is drawn from an axis-aligned Gaussian distribution. Because it is so fast and has no hyperparameters to choose, Gaussian naive Bayes is often a good model to use as a baseline classification, before exploring whether improvements can be found through more sophisticated models.

Chapter 36

Hyperparameters and Model Validation

36.1 Thinking About Model Validation

In principle, model validation is very simple: after choosing a model and its hyperparameters, we can estimate how effective it is by applying it to some of the training data and comparing the predictions to the known values.

Model Validation the Right Way: Holdout Sets

A better sense of a model's performance can be found by using what's known as a *holdout set*: that is, we hold back some subset of the data from the training of the model, and then use this holdout set to check the model's performance.

Model Validation via Cross-Validation

One disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training.

One way to address this is to use cross-validation; that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set.

cikit-Learn implements a number of cross-validation schemes that are useful in particular situations; these are implemented via iterators in the `model_selection` module. For example, we might wish to go to the extreme case in which our number of folds is equal to the number of data points: that is, we train on all points but one in each trial. This type of cross-validation is known as *leave-one-out* cross validation. 留一验证

36.2 Selecting the Best Model

The Bias-Variance Trade-off

Fundamentally, finding "the best model" is about finding a sweet spot in the trade-off between bias and variance.

欠拟合的模型会有较高的偏差（bias），过拟合的模型甚至拟合了数据中的随机误差，这样的模型具有较高的方差（variance）。

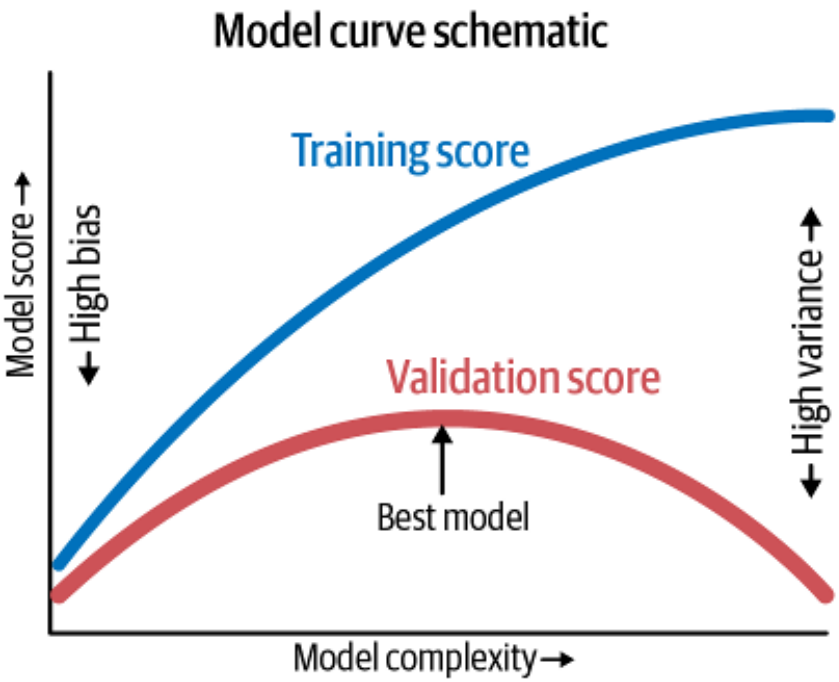


Figure 36.1: A schematic of the relationship between model complexity, training score, and validation score

The R^2 score, or coefficient of determination, measures how well a model performs relative to a simple mean of the target values. $R^2 = 1$ indicates a perfect match, $R^2 = 0$ indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models.

If we imagine that we have some ability to tune the model complexity, we would expect the training score and validation score to behave as illustrated in Figure 36.1, often called a validation curve, and we see the following features:

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is underfit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is overfit, which means that the model predicts the training data very well, but fails for any previously unseen data.

A useful question to answer is this: which model provides a suitable trade-off between bias (underfitting) and variance (overfitting)?

We can make progress in this by visualizing the validation curve for this particular data and model; this can be done straightforwardly using the `validation_curve` convenience routine provided by Scikit-Learn. Given a model, data, parameter name, and a range to explore, this function will automatically compute both the training score and the validation score across the range.

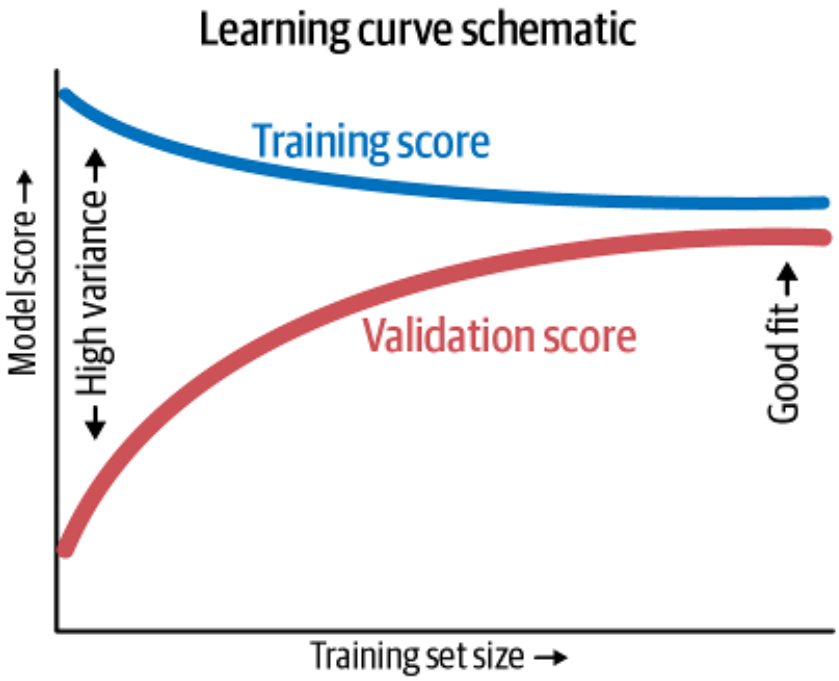


Figure 36.2: Schematic showing the typical interpretation of learning curves

36.3 Learning Curves

One important aspect of model complexity is that the optimal model will generally depend on the size of your training data.

The behavior of the validation curve has not one but two important inputs: the model complexity and the number of training points.

We can gain further insight by exploring the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model. A plot of the training/validation score with respect to the size of the training set is sometimes known as a *learning curve*.

The general behavior we would expect from a learning curve is this:

- A model of a given complexity will overfit a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
- A model of a given complexity will underfit a large dataset: this means that the training score will decrease, but the validation score will increase.
- A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

With these features in mind, we would expect a learning curve to look qualitatively like that shown in Figure 36.2.

The notable feature of the learning curve is the convergence to a particular score as the number of training samples grows. In particular, once you have enough points that a particular model has converged, adding more training data will not help you! The only way to increase model performance in this case is to

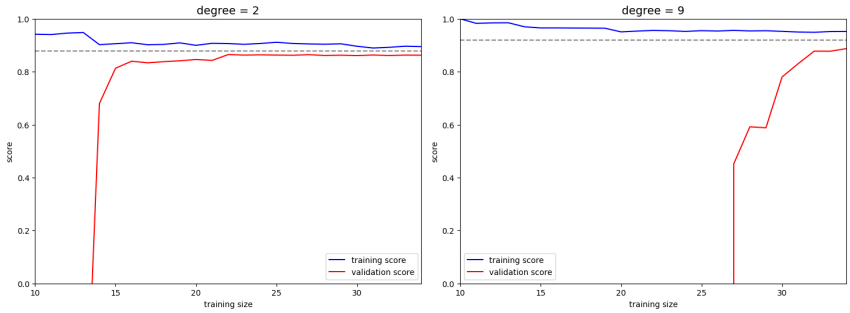


Figure 36.3: Learning curves for a low-complexity model and a high-complexity model

use another (often more complex) model.

Figure 36.3 is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing amounts of training data. In particular, when the learning curve has already converged (i.e., when the training and validation curves are already close to each other) adding more training data will not significantly improve the fit! This situation is seen in the left panel, with the learning curve for the degree-2 model.

The only way to increase the converged score is to use a different (usually more complicated) model. We see this in the right panel: by moving to a much more complicated model, we increase the score of convergence (indicated by the dashed line), but at the expense of higher model variance (indicated by the difference between the training and validation scores). If we were to add even more data points, the learning curve for the more complicated model would eventually converge.

Plotting a learning curve for your particular choice of model and dataset can help you to make this type of decision about how to move forward in improving your analysis.

36.4 Validation in Practice: Grid Search

GridSearchCV include the ability to specify a custom scoring function, to parallelize the computations, to do randomized searches, and more.

Chapter 37

Feature Engineering

In the real world, data rarely comes in such a form. With this in mind, one of the more important steps in using machine learning in practice is feature engineering: that is, taking whatever information you have about your problem and turning it into numbers that you can use to build your feature matrix.

37.1 Categorical Features

One common type of nonnumerical data is categorical data.

The Scikit-Learn's models make the fundamental assumption that numerical features reflect algebraic quantities.

one proven technique is to use one-hot encoding, which effectively creates extra columns indicating the presence or absence of a category with a value of 1 or 0, respectively. When your data takes the form of a list of dictionaries, Scikit-Learn's `DictVectorizer` will do this for you.

To see the meaning of each column, you can inspect the feature names, `get_feature_names_out()`.

There is one clear disadvantage of this approach: if your category has many possible values, this can greatly increase the size of your dataset. However, because the encoded data contains mostly zeros, a sparse output can be a very efficient solution. Nearly all of the Scikit-Learn estimators accept such sparse inputs when fitting and evaluating models. Two additional tools that Scikit-Learn includes to support this type of encoding are `sklearn.preprocessing.OneHotEncoder` and `sklearn.feature_extraction.FeatureHasher`.

37.2 Text Features

Another common need in feature engineering is to convert text to a set of representative numerical values.

One of the simplest methods of encoding this type of data is by *word counts*: you take each snippet of text, count the occurrences of each word within it, and put the results in a table.

There are some issues with using a simple raw word count, however: it can lead to features that put too much weight on words that appear very frequently, and this can be suboptimal in some classification algorithms. One approach to fix this is known as *term frequency–inverse document frequency* (TF–IDF), which weights the word counts by a measure of how often they appear in the documents.

在一份给定的文件里，词频（term frequency, tf）指的是某一个给定的词语在该文件中出现的频率。这个数字是对词数（term count）的标准化，以防止它偏向长的文件。（同一个词语在长文件里

可能会比段文件有更高的次数，而不管该词语重要与否）对于在某一个特定文件里的词语 t_i 来说，它的重要性可表示：

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{37.1}$$

式中假设文件 d_j 中共有 k 个词语， $n_{k,j}$ 是 t_k 在文件 d_j 中出现的次数。分子 $n_{i,j}$ 是该词在文件 d_j 中出现次数，而分母则是在文件 d_j 中所有字词出现的次数之和。

逆向文件频率（inverse document frequency, idf）是一个词语普遍重要性的度量。某一特定词语的 idf，可以有总文件数目除以包含该词语之文件的数目，再将得到的上取以10为底的对数得到：

$$idf_i = \log_{10} \frac{|D|}{|\{j : t_i \in d_j\}|} \tag{37.2}$$

式中，$|D|$表示语料库中的文件总数，$|\{j : t_i \in d_j\}|$ 表示包含词语 t_i 的文件数目，如果词语不再资料中，就导致分母为零，因此一般情况下使用$1 + |\{j : t_i \in d_j\}|$，然后

$$tfidf_{i,j} = tf_{i,j} \times idf_i \tag{37.3}$$

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的 $tf-idf$，因此，$tf-idf$ 倾向于过滤掉常见的词语，保留重要的词语。

例子：假如一篇文件的总词数是100个，而词语母牛出现了3次，那么母牛一次在该文件中的词频就 $3/100 = 0.03$，而计算文件频率的方法就是以文件集的文件总数除以出现母牛一词的文件数，所以，如果母牛一次在1000份文件出现过，而文件总数是10000000份的话，其你想文件频率就是$\log_{10}(10000000/1000) = 4$，最后的 $tf-idf$的分数为 $0.03 * 4 = .012$，更多访问 tf-idf。

37.3 Image Features

Another common need is to suitably encode images for machine learning analysis. A comprehensive summary of feature extraction techniques for images is well beyond the scope of this chapter, but you can find excellent implementations of many of the standard approaches in the Scikit-Image project.

37.4 Derived Features

Another useful type of feature is one that is mathematically derived from some input features. We saw an example of this in Chapter 36 when we constructed *polynomial features* from our input data. We saw that we could convert a linear regression into a polynomial regression not by changing the model, but by transforming the input!

> This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods.
> More generally, this is one motivational path to the powerful set of techniques known as *kernel methods*.

37.5 Imputation of Missing Data

Another common need in feature engineering is handling of missing data. When applying a typical machine learning model to such data, we will need to first replace the missing values with some appropriate

imputation

fill value. This is known as *imputation* of missing values, and strategies range from simple (e.g., replacing missing values with the mean of the column) to sophisticated (e.g., using matrix completion or a robust model to handle such data).

37.6 Feature Pipelines

It can quickly become tedious to do the transformations by hand, especially if you wish to string together multiple steps. Scikit-Learn provides a Pipeline object to streamline type of processing pipeline,

Chapter 38

In Depth: Naive Bayes Classification

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being useful as a quick-and-dirty baseline for a classification problem.

38.1 Bayesian Classification

In Bayesian classification, we're interested in finding the probability of a label L given some observed features, which we can write as $P(L|features)$.

$$P(L|features) = \frac{P(features|L)P(L)}{P(features)}$$

If we are trying to decide between two labels—let's call them L_1 and L_2—then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\frac{P(L_1|features)}{P(L_2|features)} = \frac{P(features|L_1)P(L_1)}{P(features|L_2)P(L_2)}$$

All we need now is some model by which we can compute $P(features|L_i)$ for each label. Such a model is called a *generative model* because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier.

This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of naive Bayes classifiers rest on different naive assumptions about the data.

38.2 Gaussian Naive Bayes

Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. With this classifier, the assumption is that *data from each label is drawn from a simple Gaussian distribution.*

The simplest Gaussian model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by computing the mean and standard deviation of the points within each label, which is all we need to define such a distribution.

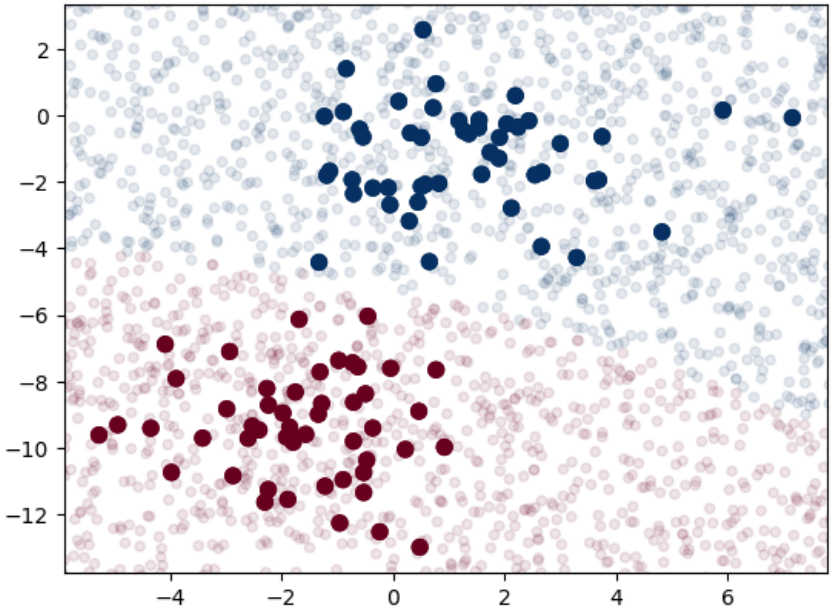


Figure 38.1: Visualization of the Gaussian naive Bayes classification

Figure 38.1, We see a slightly curved boundary in the classifications—in general, the boundary produced by a Gaussian naive Bayes model will be quadratic.

A nice aspect of this Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the `predict_proba` method. If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a good place to start.

Of course, the final classification will only be as good as the model assumptions that lead to it, which is why Gaussian naive Bayes often does not produce very good results. Still, in many cases—especially as the number of features becomes large—this assumption is not detrimental enough to prevent Gaussian naive Bayes from being a reliable method.

38.3 Multinomial Naive Bayes

Another useful example is multinomial naive Bayes, where the features are assumed to be generated from a simple multinomial distribution. The multinomial distribution describes the probability of observing counts among a number of categories, and thus multinomial naive Bayes is most appropriate for features that represent counts or count rates.

Example: Classifying Text

One place where multinomial naive Bayes is often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified.

38.4 When to Use Naive Bayes

Because naive Bayes classifiers make such stringent assumptions about data, they will generally not perform as well as more complicated models. That said, they have several advantages:

- They are fast for both training and prediction.
- They provide straightforward probabilistic prediction.
- They are often easily interpretable.
- They have few (if any) tunable parameters.

Naive Bayes classifiers tend to perform especially well in the following situations:

- When the naive assumptions actually match the data (very rare in practice)
- For very well-separated categories, when model complexity is less important
- For very high-dimensional data, when model complexity is less important

The last two points seem distinct, but they actually are related: as the dimensionality of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in every single dimension to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information.

Chapter 39

In Depth: Linear Regression

Just as naive Bayes (discussed in Chapter 38) is a good starting point for classification tasks, linear regression models are a good starting point for regression tasks. Such models are popular because they can be fit quickly and are straightforward to interpret.

39.1 Simple Linear Regression

We can use the single LinearRegression estimator to fit lines, planes, or hyperplanes to our data. It still appears that this approach would be limited to strictly linear relationships between variables.

39.2 Basis Function Regression

One trick you can use to adapt linear regression to nonlinear relationships between variables is to transform the data according to basis functions. We have seen one version of this before, in the PolynomialRegression pipeline used in Chapter 36 and Chapter 37. The idea is to take our multidimensional linear model:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots$$

and build the x_1, x_2, x_3, and so on from our single-dimensional input x. That is, we let $x_n = f_n(x)$, where f_n is some function that transforms our data.(这个多元回归模型中，特征都是一维输入的函数) For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

Notice that this is still a linear model—the linearity refers to the fact that the coefficients an never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationships between x and y.

Polynomial Basis Functions

This polynomial projection is useful enough that it is built into Scikit-Learn, using the PolynomialFeatures transformer.

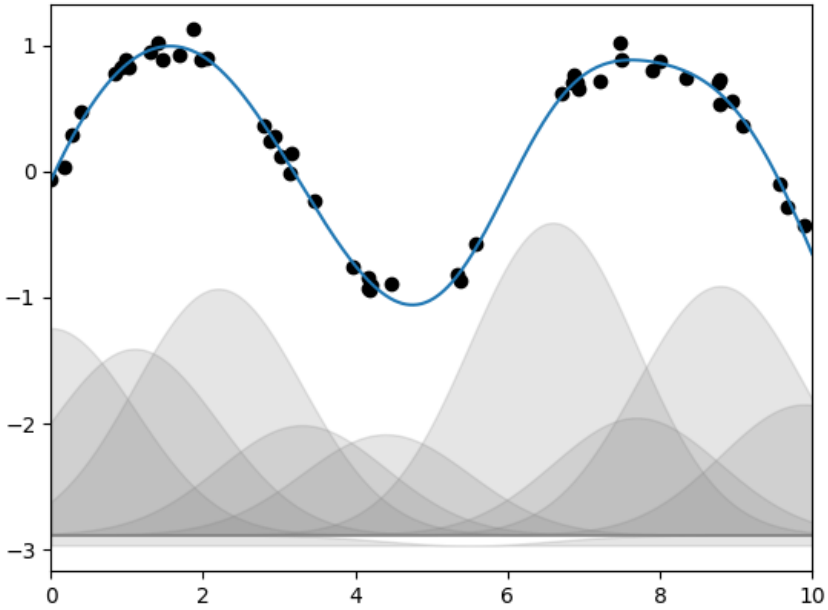


Figure 39.1: A Gaussian basis function fit to nonlinear data

Gaussian Basis Functions

Of course, other basis functions are possible. For example, one useful pattern is to fit a model that is not a sum of polynomial bases, but a sum of Gaussian bases. The result might look something like Figure 39.1.

Algebraically, Gaussian basis functions are defined as follows:

$$\phi_k(t;\mu_k,\sigma_k^2) = \exp\left(-\frac{||t-\mu_k||^2}{2\sigma_k^2}\right), k = 1,\ldots,K \tag{39.1}$$

where$\mu_k$is a parameter determining the center of the basis function,σ_k^2 is a parameter that determines the width and $||.||$ is the Euclidian norm. The basis functions overlap with each other to capture the information about t, and the width parameter play an essential role to capture the structure in the data over the region of input data. The parameters featuring in each basis function are often determined heuristically based on the structure of the observed data.

The shaded regions in the Figure 39.1 are the scaled basis functions, and when added together they reproduce the smooth curve through the data. These Gaussian basis functions are not built into Scikit-Learn, but we can write a custom transformer that will create them.

39.3 Regularization

The introduction of basis functions into our linear regression makes the model much more flexible, but it also can very quickly lead to overfitting.

We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by penalizing large values of the model parameters. Such a penalty is known as regularization, and comes in several forms.

Ridge Regression (L_2 Regularization)

Perhaps the most common form of regularization is known as ridge regression or L_2 regularization (sometimes also called Tikhonov regularization). This proceeds by penalizing the sum of squares (2-norms) of the model coefficients θ n. In this case, the penalty on the model fit would be:

$$P = \alpha \sum_{i=1}^{n} \theta_n^2 \tag{39.2}$$

where α is a free parameter that controls the strength of the penalty. This type of penalized model is built into Scikit-Learn with the `Ridge` estimator. Ridge

Lasso Regression (L_1 Regularization)

Another common type of regularization is known as lasso regression or L_1 regularization and involves penalizing the sum of absolute values (1-norms) of regression coefficients:

$$P = \alpha \sum_{i=1}^{n} |\theta_n| \tag{39.3}$$

Though this is conceptually very similar to ridge regression, the results can differ surprisingly. For example, due to its construction, lasso regression tends to favor sparse models where possible: that is, it preferentially sets many model coefficients to exactly zero.

Chapter 40

In Depth: Support Vector Machines

Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression.

40.1 Motivating Support Vector Machines

As part of our discussion of Bayesian classification (see Chapter 38), we learned about a simple kind of model that describes the distribution of each underlying class, and experimented with using it to probabilistically determine labels for new points. That was an example of generative classification; here we will consider instead discriminative classification. That is, rather than modeling each class, we will simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

These are three very different separators which, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in Figure 40.1) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not good enough, and we need to think a bit more deeply.

40.2 Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a *margin* of some width, up to the nearest point. The line that maximizes this margin is the one we will choose as the optimal model.

margin

Fitting a Support Vector Machine

Let's see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier (SVC) to train an SVM model on this data.

A key to this classifier's success is that for the fit, only the positions of the support vectors matter; any points further from the margin that are on the correct side do not modify the fit. Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

Figure 40.2, in the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model has not changed: the three support

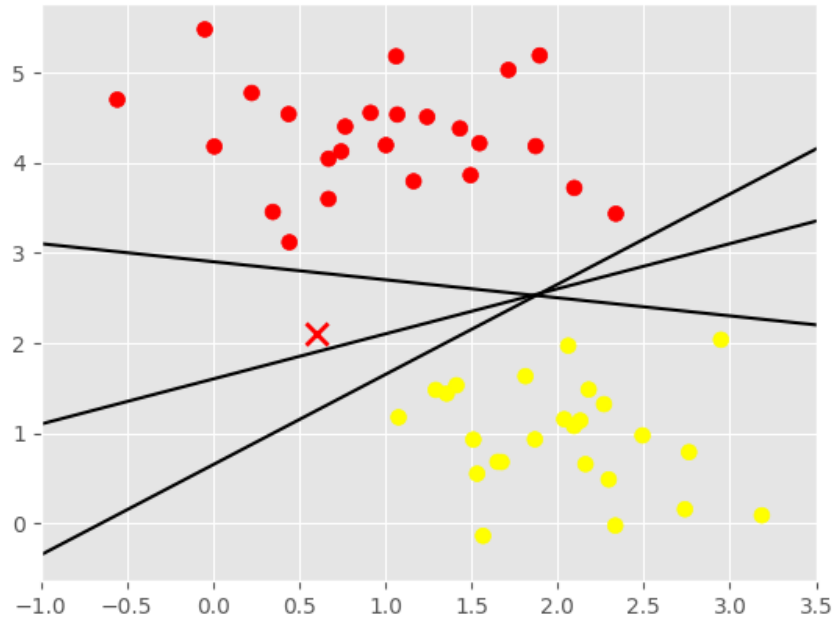


Figure 40.1: Three perfect linear discriminative classifiers for our data

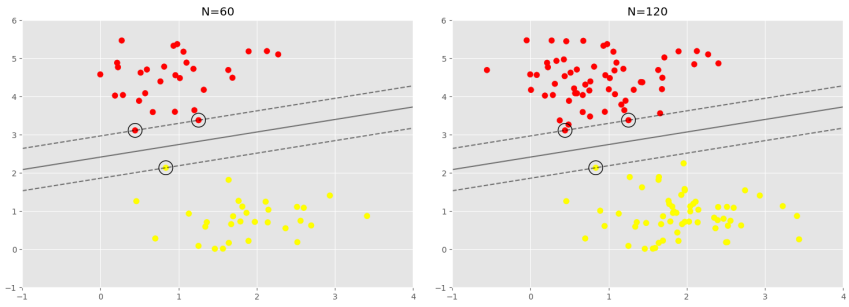


Figure 40.2: The influence of new training points on the SVM model

vectors in the left panel are the same as the support vectors in the right panel. This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

Beyond Linear Boundaries: Kernel SVM

Where SVM can become quite powerful is when it is combined with *kernels*. There we projected our data into a higher-dimensional space defined by polynomials and Gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

Tuning the SVM: Softening Margins

Chapter 41

In Depth: Decision Trees and Random Forests

Here we'll take a look at another powerful algorithm: a nonparametric algorithm called random forests. Random forests are an example of an ensemble method, meaning one that relies on aggregating the results of a set of simpler estimators.

41.1 Summary

A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the meaning of the classification model, random forests may not be the best choice.

Chapter 42

In Depth: Principal Component Analysis

42.1 Introducing Principal Component Analysis

These vectors represent the principal axes of the data, and the length of each vector is an indication of how "important" that axis is in describing the distribution of the data —more precisely, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes are the principal components of the data.

This transformation from data axes to principal axes is an **affine transformation**, which means it is composed of a translation, rotation, and uniform scaling.

42.1.1 PCA as Dimensionality Reduction

Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

42.1.2 What Do the Components Mean?

We can go a bit further here, and begin to ask what the reduced dimensions mean. This meaning can be understood in terms of combinations of basis vectors.

42.1.3 Choosing the Number of Components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. This can be determined by looking at the cumulative **explained variance ratio** as a function of the number of components.

This tells us that our two-dimensional projection loses a lot of information (as measured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help you understand the level of redundancy present in its features.

42.2 PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this: any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So, if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

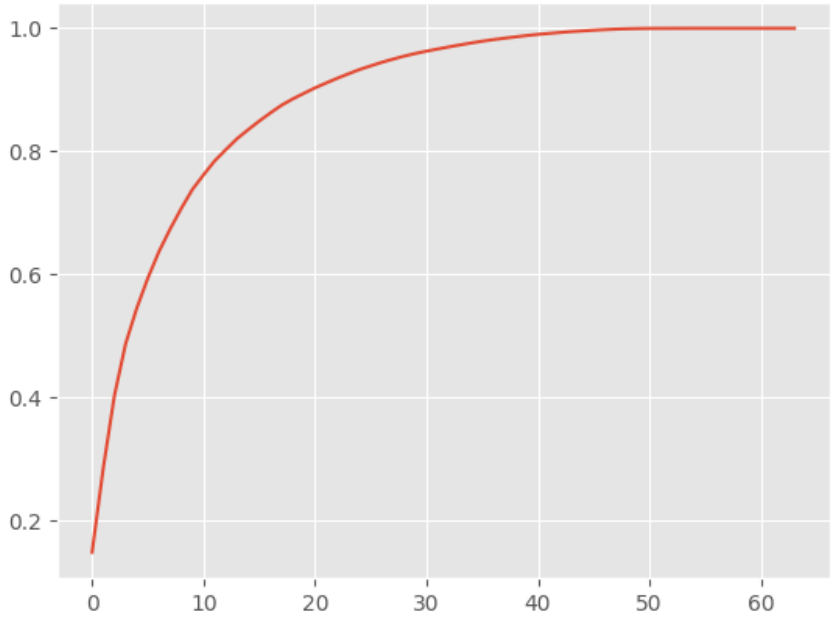


Figure 42.1: The cumulative explained variance, which measures how well PCA preserves the content of the data

42.3 Example: Eigenfaces

In this case, it can be interesting to visualize the images associated with the first several principal components (these components are technically known as **eigenvectors**, so these types of images are often called **eigenfaces**; as you can see in Figure 42.2, they are as creepy as they sound)

The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips.

Figure 42.2: A visualization of eigenfaces learned from the LFW dataset

42.4 Summary

Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationships between points (as we did with the digits data), to understand the main variance in the data (as we did with the eigenfaces), and to understand the intrinsic dimensionality (by plotting the explained variance ratio). Certainly PCA is not useful for every high-dimensional dataset, but it offers a straightforward and efficient path to gaining insight into high-dimensional data.

PCA's main weakness is that it tends to be highly affected by outliers in the data. For this reason, several robust variants of PCA have been developed, many of which act to iteratively discard data points that are poorly described by the initial components.

Chapter 43

In Depth: Manifold Learning

While PCA is flexible, fast, and easily interpretable, it does not perform so well when there are nonlinear relationships within the data.

43.1 Multidimensional Scaling

Given a distance matrix between points, it recovers a D-dimensional coordinate representation of the data, this is exactly what the multidimensional scaling algorithm aims to do.

43.1.1 MDS as Manifold Learning

This is essentially the goal of a manifold learning estimator: given high-dimensional embedded data, it seeks a low-dimensional representation of the data that preserves certain relationships within the data. In the case of MDS, the quantity preserved is the distance between every pair of points.

43.1.2 Nonlinear Embeddings: Where MDS Fails

Where MDS breaks down is when the embedding is nonlinear—that is, when it goes beyond this simple set of operations.

43.2 Nonlinear Manifolds: Locally Linear Embedding

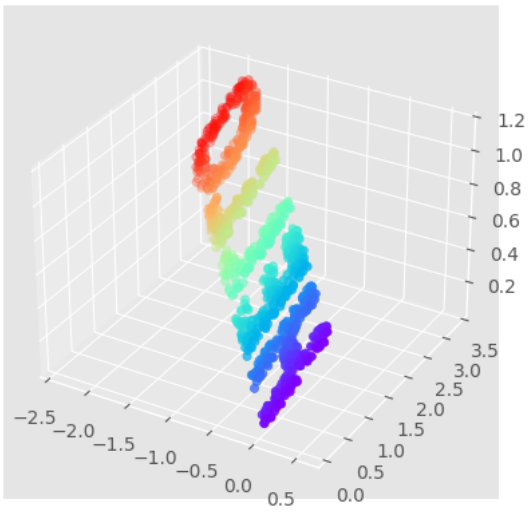
MDS tries to preserve distances between faraway points when constructing the embedding. But what if we instead modified the algorithm such that it only preserves distances between nearby points? The resulting embedding would be closer to what we want.

43.3 Some Thoughts on Manifold Methods

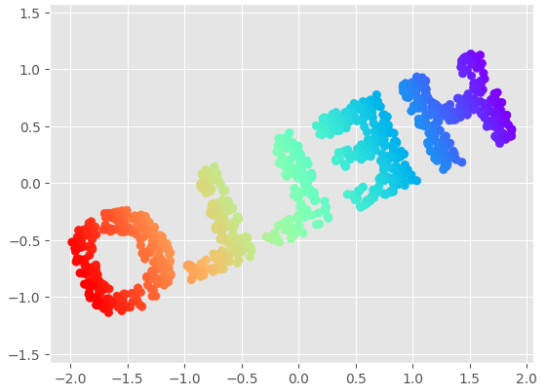
In practice manifold learning techniques tend to be finicky enough that they are rarely used for anything more than simple qualitative visualization of high-dimensional data.

The following are some of the particular challenges of manifold learning, which all contrast poorly with PCA:

- In manifold learning, there is no good framework for handling missing data. In contrast, there are straightforward iterative approaches for dealing with missing data in PCA.



(a) Data embedded linearly into three dimensions



(b) The MDS embedding of the three-dimensional data recovers the input up to a rotation and reflection

Figure 43.1: MDS as Manifold Learning

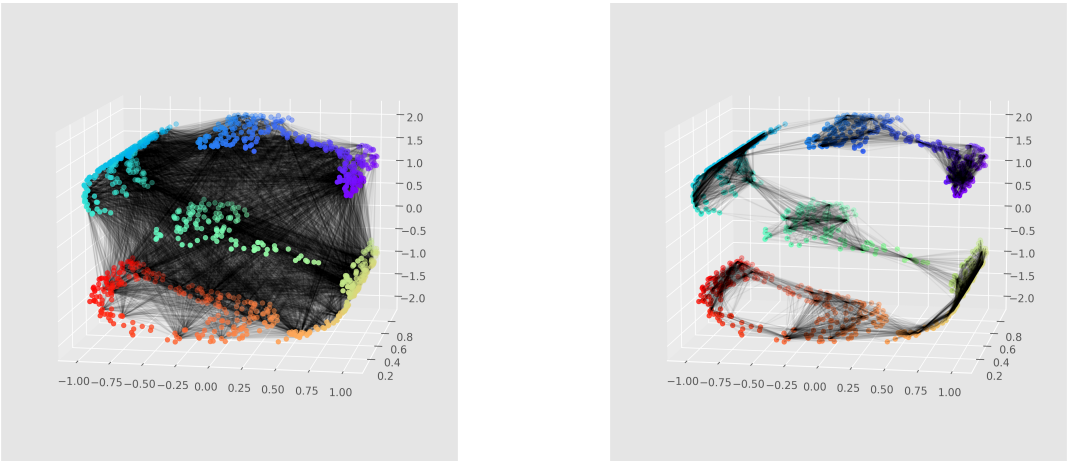


Figure 43.2: Representation of linkages between points within MDS and LLE

- In manifold learning, the presence of noise in the data can "short-circuit" the manifold and drastically change the embedding. In contrast, PCA naturally filters noise from the most important components.
- The manifold embedding result is generally highly dependent on the number of neighbors chosen, and there is generally no solid quantitative way to choose an optimal number of neighbors. In contrast, PCA does not involve such a choice.
- In manifold learning, the globally optimal number of output dimensions is difficult to determine. In contrast, PCA lets you find the number of output dimensions based on the explained variance.
- In manifold learning, the meaning of the embedded dimensions is not always clear. In PCA, the principal components have a very clear meaning.
- In manifold learning, the computational expense of manifold methods scales as $O[N^2]$ or $O[N^3]$. For PCA, there exist randomized approaches that are generally much faster (though see the megaman package for some more scalable implementations of manifold learning).

Based on my own experience, I would give the following recommendations:

- For toy problems such as the S-curve we saw before, LLE and its variants (especially modified LLE) perform very well. This is implemented in sklearn.manifold.LocallyLinearEmbedding.
- For high-dimensional data from real-world sources, LLE often produces poor results, and Isomap seems to generally lead to more meaningful embeddings. This is implemented in sklearn.manifold.Isomap.
- For data that is highly clustered, t-distributed stochastic neighbor embedding (t-SNE) seems to work very well, though it can be very slow compared to other methods. This is implemented in sklearn.manifold.TSNE.

Chapter 44

In Depth: k-Means Clustering

Now we will move on to another class of unsupervised machine learning models: clustering algorithms. Clustering algorithms seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points.

44.1 Introducing k-Means

The k-means algorithm searches for a predetermined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The **cluster center** is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

Those two assumptions are the basis of the k-means model.

44.2 Expectation–Maximization

In short, the expectation–maximization approach in k-means consists of the following procedure:

1. Guess some cluster centers.
2. Repeat until converged:
   (a) E-step: Assign points to the nearest cluster center.
   (b) M-step: Set the cluster centers to the mean of their assigned points.

Here the E-step or expectation step is so named because it involves updating our expectation of which cluster each point belongs to. The M-step or maximization step is so named because it involves maximizing some fitness function that defines the locations of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

There are a few caveats to be aware of when using the expectation–maximization algorithm:

The globally optimal result may not be achieved First, although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the global best solution.

The number of clusters must be selected beforehand Another common challenge with k-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data.

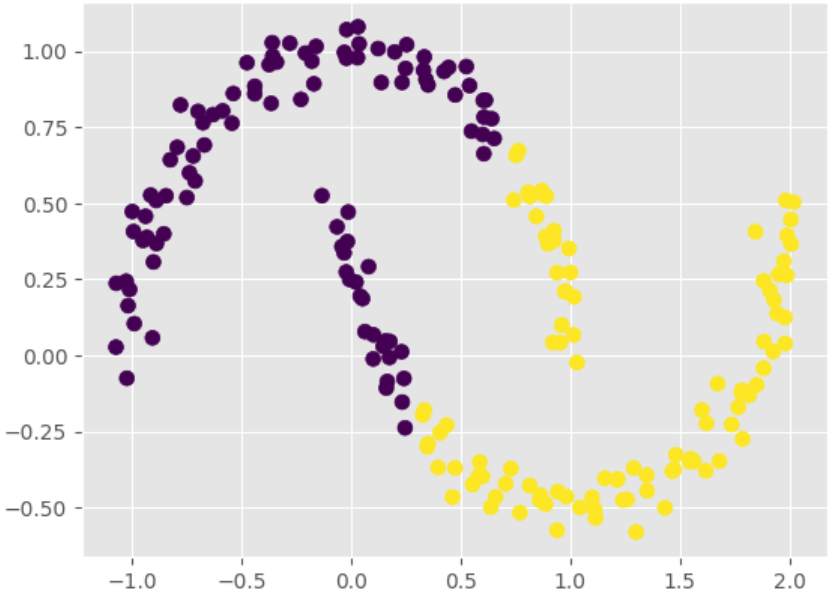


Figure 44.1: Failure of k-means with nonlinear boundaries

Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive, but that we won't discuss further here, is called silhouette analysis.

Alternatively, you might use a more complicated clustering algorithm that has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models;) or which can choose a suitable number of clusters (e.g., DBSCAN, mean-shift, or affinity propagation, all available in the sklearn.cluster submodule).

k-means is limited to linear cluster boundaries The fundamental model assumptions of k-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

k-means can be slow for large numbers of samples Because each iteration of k-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows. You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step. This is the idea behind batch-based k-means algorithms, one form of which is implemented in sklearn.cluster.MiniBatchKMeans.

Chapter 45

In Depth: Gaussian Mixture Models

In particular, the nonprobabilistic nature of k-means and its use of simple distance from cluster center to assign cluster membership leads to poor performance for many real-world situations. Gaussian mixture models can be viewed as an extension of the ideas behind k-means, but can also be a powerful tool for estimation beyond simple clustering.

An important observation for k-means is that these cluster models must be circular: k-means has no built-in way of accounting for oblong or elliptical clusters.

These two disadvantages of k-means—its lack of flexibility in cluster shape and lack of probabilistic cluster assignment—mean that for many datasets (especially low-dimensional datasets) it may not perform as well as you might hope.

45.1 Generalizing E–M: Gaussian Mixture Models

A **Gaussian mixture model** (GMM) attempts to find a mixture of multidimensional Gaussian probability distributions that best model any input dataset.

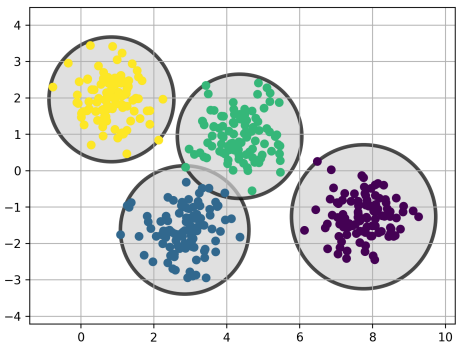
Under the hood, a Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach, which qualitatively does the following:

1. Choose starting guesses for the location and shape.
2. Repeat until converged:
   (a) E-step: For each point, find weights encoding the probability of membership in each cluster.
   (b) M-step: For each cluster, update its location, normalization, and shape based on all data points, making use of the weights.

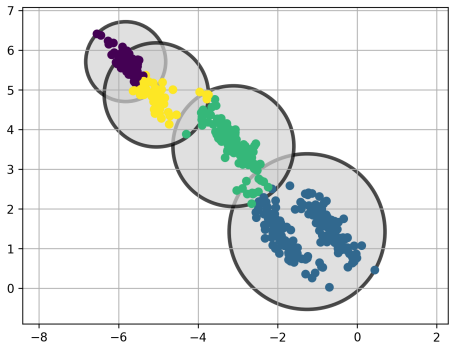
The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the k-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

45.2 Choosing the Covariance Type

If you look at the details of the preceding fits, you will see that the `covariance_type` option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it's essential to set this carefully for any given problem. The default is `covariance_type="diag"`,



(a) The circular clusters implied by the k-means model



(b) Poor performance of k-means for noncircular clusters

Figure 45.1: Motivating Gaussian Mixtures: Weaknesses of k-Means

which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. `covariance_type="spherical"` is a slightly simpler and faster model, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of k-means, though it's not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use `covariance_type="full"`, which allows each cluster to be modeled as an ellipse with arbitrary orientation.

45.3 Gaussian Mixture Models as Density Estimation

Though the GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for density estimation. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

A GMM is convenient as a flexible means of modeling an arbitrary multidimensional distribution of data. The fact that a GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the likelihood of the data under the model, using cross-validation to avoid overfitting. Another means of correcting for overfitting is to adjust the model likelihoods using some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC).